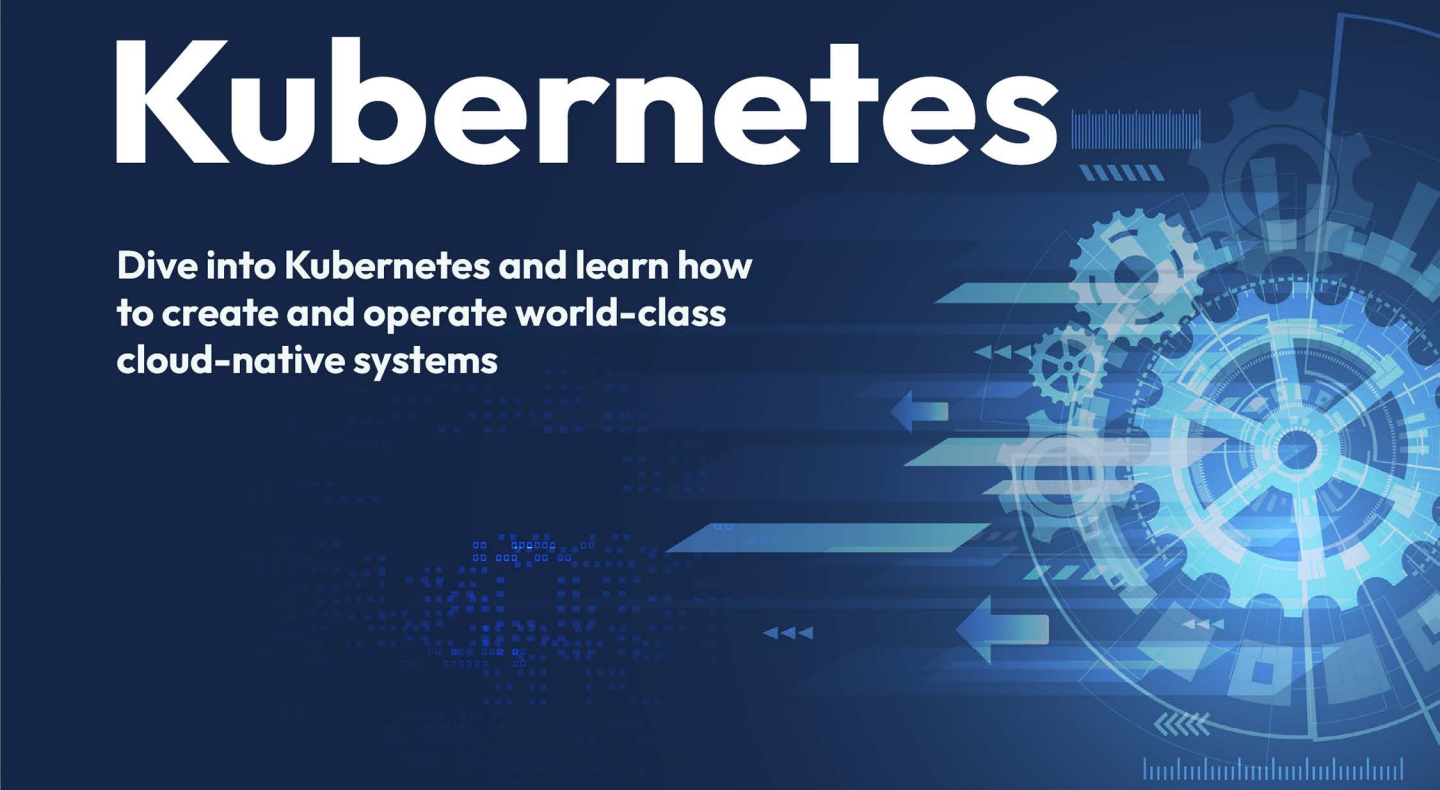
EXPERT INSIGHT

Mastering Kubernetes

Dive into Kubernetes and learn how to create and operate world-class cloud-native systems



Foreword by:
Bilgin Ibryam
Coauthor of Kubernetes Patterns



Fourth Edition

Gigi Sayfan

‹packt›

Mastering Kubernetes

Fourth Edition

Dive into Kubernetes and learn how to create and operate world-class cloud-native systems

Gigi Sayfan

‹packt›

BIRMINGHAM—MUMBAI

Mastering Kubernetes

Fourth Edition

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Rahul Nair
Acquisition Editor – Peer Reviews: Gaurav Gavas
Project Editor: Amisha Vathare
Content Development Editor: Georgia Daisy van der Post
Copy Editor: Safis Editing
Technical Editor: Aniket Shetty
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Presentation Designer: Pranit Padwal
Developer Relations Marketing Executive: Priyadarshini Sharma

First published: May 2017
Second edition: April 2018
Third edition: June 2020
Fourth edition: June 2023

Production reference: 2030723

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80461-139-5

www.packt.com

Foreword

In the realm of distributed computing, it's hard to understate the seismic shift that Kubernetes has brought forth. As the coauthor of *Kubernetes Patterns*, I have had the opportunity to observe this transformation closely.

With the introduction of Kubernetes, we found ourselves at a turning point in the way we manage distributed applications. Instead of focusing on bespoke components, Kubernetes introduced distributed primitives like Pods, Deployments, StatefulSets, and Jobs. These primitives have become the basic building blocks of deployment, offering a unified way to manage applications across various cloud environments. For example, in the Kubernetes model, a Pod is the smallest deployable unit that can be created and managed. It encapsulates an application's containers, providing an isolated environment for running applications. Similarly, a Deployment is a declaration of intent, a way of telling Kubernetes: "Here's my application, keep it running as I've specified." StatefulSets, Jobs, and others each contribute their unique capabilities to the Kubernetes ecosystem, together forming a coherent and holistic approach to managing distributed applications.

As Kubernetes continued to evolve, it also spurred the development of new design patterns. The Sidecar pattern, for instance, allows developers to augment or extend the functionality of an application by running additional utility containers alongside the primary application container. Health checks provide an automated way for Kubernetes to monitor the state of applications and respond to any issues that may arise. These patterns, along with principles like declarative deployment and automated health checks, have changed the way we implement and operate applications.

Beyond its impact on application orchestration, Kubernetes has also standardized the way we interact with our infrastructure through its APIs. The Kubernetes APIs have become the industry standard for managing distributed systems, whether these are on the cluster or remote cloud resources, due to their efficiency and broad ecosystem of tools and services. In this new reality, Kubernetes has become more than a tool and it's not only essential for developers to master it, but also for operations teams to do the same. With Kubernetes becoming the ubiquitous cloud abstraction layer, it's the lingua franca for deploying all workloads, be it microservices, monoliths, databases, machine learning jobs, or any other workload. However, the larger picture is not just about writing code; it's about understanding how applications live and breathe in the Kubernetes world.

The journey with Kubernetes is pushing the boundaries of distributed systems and redefining our understanding of what is possible. *Mastering Kubernetes* by Gigi Sayfan is your companion on this journey. A deep dive into the vast world of Kubernetes, this book will serve as an invaluable resource for anyone looking to navigate the complexities of managing distributed applications with Kubernetes. Enjoy!

Bilgin Ibryam

Coauthor of Kubernetes Patterns

Contributors

About the author

Gigi Sayfan is a principal software engineer that currently works at a stealth start-up and manages dozens of Kubernetes clusters across multiple cloud providers. He has been developing software professionally for more than 25 years in domains as diverse as instant messaging, morphing, the chip fabrication process, control, embedded multimedia applications for game consoles, brain-inspired machine learning, custom browser development, web services for 3D distributed game platforms, IoT sensors, virtual reality, and genomics. He has written production code in many programming languages such as Go, Python, C, C++, C#, Java, Delphi, JavaScript, Cobol, and PowerBuilder for operating systems such as Windows (3.11 through 7), Linux, macOS, Lynx (embedded), and Sony PlayStation. His technical expertise includes cloud-native technologies, DevOps, databases, low-level networking, distributed systems, unorthodox user interfaces, and general software development lifecycle.

Gigi has written several books on Kubernetes and microservices, and hundreds of technical articles about a variety of topics.

About the reviewers

Onur Yilmaz is a senior software engineer at a multinational enterprise software company. He is a Certified Kubernetes Administrator (CKA) and works on Kubernetes and cloud management systems as a keen supporter of cutting-edge technologies. Furthermore, he is the author of multiple books on Kubernetes, Docker, serverless architectures, and cloud-native continuous integration and delivery. In addition, he has one master's degree and two bachelors' degrees in the engineering field.

Sergei Bulavintsev is a senior DevOps engineer at GlobalDots. He is passionate about open source, cloud-native infrastructure, and tools that increase developers' productivity. He has successfully migrated multiple customers to the cloud and Kubernetes, advocating for and implementing the GitOps approach. Sergei is also an active member of his local cloud-native community and holds industry certifications such as CKA, CKAD, CKS, and RHCA level 2.

I would like to thank my wife, Elena, and our two children, Maria and Maxim, for their support and patience. I would also like to thank my family, friends, and colleagues who have helped me become who I am today.

Table of Contents

Preface xxxiii

Chapter 1: Understanding Kubernetes Architecture 1

What is Kubernetes? 2
What Kubernetes is not 2
Understanding container orchestration 3
Physical machines, virtual machines, and containers • 3
The benefits of containers • 3
Containers in the cloud • 3
Cattle versus pets • 4
Kubernetes concepts 5
Node • 5
Cluster • 5
Control plane • 6
Pod • 6
Label • 6
Label selector • 7
Annotation • 7
Service • 7
Volume • 8
Replication controller and replica set • 8
StatefulSet • 8
Secret • 9
Name • 9
Namespace • 9
Diving into Kubernetes architecture in depth 9
Distributed systems design patterns • 10

Sidecar pattern • 10
Ambassador pattern • 10
Adapter pattern • 11
Multi-node patterns • 11
Level-triggered infrastructure and reconciliation • 11
The Kubernetes APIs • 11
Resource categories • 12
Kubernetes components • 15
Control plane components • 15
Node components • 18
Kubernetes container runtimes 18
The Container Runtime Interface (CRI) • 18
Docker • 20
containerd • 22
CRI-O • 22
Lightweight VMs • 22
Summary 23

Chapter 2: Creating Kubernetes Clusters 25

Getting ready for your first cluster 25
Installing Rancher Desktop • 26
Installation on macOS • 26
Installation on Windows • 26
Additional installation methods • 26
Meet kubectl • 27
Kubectl alternatives – K9S, KUI, and Lens • 28
K9S • 28
KUI • 29
Lens • 29
Creating a single-node cluster with Minikube 30
Quick introduction to Minikube • 31
Installing Minikube • 31
Installing Minikube on Windows • 31
Installing Minikube on macOS • 35
Troubleshooting the Minikube installation • 37
Checking out the cluster • 38
Doing work • 40

Examining the cluster with the dashboard • 42
Creating a multi-node cluster with KinD 43
Quick introduction to KinD • 44
Installing KinD • 44
Dealing with Docker contexts • 44
Creating a cluster with KinD • 45
Doing work with KinD • 48
Accessing Kubernetes services locally through a proxy • 49
Creating a multi-node cluster with k3d 49
Quick introduction to k3s and k3d • 50
Installing k3d • 50
Creating the cluster with k3d • 51
Comparing Minikube, KinD, and k3d 54
Honorable mention – Rancher Desktop Kubernetes cluster • 54
Creating clusters in the cloud (GCP, AWS, Azure, and Digital Ocean) 55
The cloud-provider interface • 56
Creating Kubernetes clusters in the cloud • 56
GCP • 57
GKE Autopilot • 57
AWS • 57
Kubernetes on EC2 • 57
Amazon EKS • 58
Fargate • 59
Azure • 59
Digital Ocean • 59
Other cloud providers • 60
Once upon a time in China • 60
IBM Kubernetes service • 60
Oracle Container Service • 61
Creating a bare-metal cluster from scratch 61
Use cases for bare metal • 61
When should you consider creating a bare-metal cluster? • 62
Understanding the process • 62
Using the Cluster API for managing bare-metal clusters • 62
Using virtual private cloud infrastructure • 63
Building your own cluster with Kubespray • 63
Building your cluster with Rancher RKE • 63

Running managed Kubernetes on bare metal or VMs • 63
GKE Anthos • 64
EKS Anywhere • 64
AKS Arc • 64
Summary 64

Chapter 3: High Availability and Reliability 67

High availability concepts 68
Redundancy • 68
Hot swapping • 68
Leader election • 68
Smart load balancing • 69
Idempotency • 69
Self-healing • 69
High availability best practices 69
Creating highly available clusters • 70
Making your nodes reliable • 71
Protecting your cluster state • 72
Clustering etcd • 72
Protecting your data • 72
Running redundant API servers • 73
Running leader election with Kubernetes • 73
Making your staging environment highly available • 74
Testing high availability • 74
High availability, scalability, and capacity planning 76
Installing the cluster autoscaler • 77
Considering the vertical pod autoscaler • 79
Autoscaling based on custom metrics • 80
Large cluster performance, cost, and design trade-offs 80
Best effort • 81
Maintenance windows • 81
Quick recovery • 82
Zero downtime • 82
Site reliability engineering • 84
Performance and data consistency • 84
Choosing and managing the cluster capacity 85
Choosing your node types • 85

Choosing your storage solutions • 85
Trading off cost and response time • 86
Using multiple node configurations effectively • 86
Benefiting from elastic cloud resources • 87
Autoscaling instances • 87
Mind your cloud quotas • 87
Manage regions carefully • 87
Considering container-native solutions • 88
Pushing the envelope with Kubernetes 89
Improving the performance and scalability of Kubernetes • 90
Caching reads in the API server • 90
The pod lifecycle event generator • 91
Serializing API objects with protocol buffers • 92
etcd3 • 92
gRPC instead of REST • 92
Leases instead of TTLs • 92
Watch implementation • 92
State storage • 92
Other optimizations • 93
Measuring the performance and scalability of Kubernetes • 93
The Kubernetes SLOs • 93
Measuring API responsiveness • 93
Measuring end-to-end pod startup time • 95
Testing Kubernetes at scale 97
Introducing the Kubemark tool • 97
Setting up a Kubemark cluster • 97
Comparing a Kubemark cluster to a real-world cluster • 98
Summary 98

Chapter 4: Securing Kubernetes 99

Understanding Kubernetes security challenges 99
Node challenges • 100
Network challenges • 101
Image challenges • 103
Configuration and deployment challenges • 104
Pod and container challenges • 105
Organizational, cultural, and process challenges • 105

Hardening Kubernetes 106
Understanding service accounts in Kubernetes • 107
How does Kubernetes manage service accounts? • 108
Accessing the API server • 108
Authenticating users • 109
Impersonation • 111
Authorizing requests • 112
Using admission control plugins • 114
Securing pods • 115
Using a private image repository • 115
ImagePullSecrets • 115
Specifying a security context for pods and containers • 116
Pod security standards • 117
Protecting your cluster with AppArmor • 117
Writing AppArmor profiles • 119
Pod Security Admission • 121
Managing network policies • 121
Choosing a supported networking solution • 122
Defining a network policy • 122
Limiting egress to external networks • 124
Cross-namespace policies • 124
The costs of network policies • 124
Using secrets • 125
Storing secrets in Kubernetes • 125
Configuring encryption at rest • 125
Creating secrets • 126
Decoding secrets • 126
Using secrets in a container • 127
Managing secrets with Vault • 128
Running a multi-tenant cluster 128
The case for multi-tenant clusters • 129
Using namespaces for safe multi-tenancy • 129
Avoiding namespace pitfalls • 130
Using virtual clusters for strong multi-tenancy • 131
Summary 134

Chapter 5: Using Kubernetes Resources in Practice 137

Designing the Hue platform 138
Defining the scope of Hue • 138
Smart reminders and notifications • 138
Security, identity, and privacy • 138
Hue components • 139
User profile • 139
User graph • 139
Identity • 139
Authorizer • 140
External services • 140
Generic sensor • 140
Generic actuator • 140
User learner • 140
Hue microservices • 140
Plugins • 141
Data stores • 141
Stateless microservices • 141
Serverless functions • 141
Event-driven interactions • 141
Planning workflows • 142
Automatic workflows • 142
Human workflows • 142
Budget-aware workflows • 142
Using Kubernetes to build the Hue platform 143
Using kubectl effectively • 143
Understanding kubectl manifest files • 144
apiVersion • 144
kind • 144
metadata • 145
spec • 145
Deploying long-running microservices in pods • 146
Creating pods • 146
Decorating pods with labels • 147
Deploying long-running processes with deployments • 148
Updating a deployment • 151

Separating internal and external services 152
Deploying an internal service • 152
Creating the hue-reminders service • 154
Exposing a service externally • 155
Ingress • 156
Advanced scheduling 157
Node selector • 158
Taints and tolerations • 159
Node affinity and anti-affinity • 160
Pod affinity and anti-affinity • 161
Pod topology spread constraints • 162
The descheduler • 163
Using namespaces to limit access 164
Using Kustomization for hierarchical cluster structures 166
Understanding the basics of Kustomize • 166
Configuring the directory structure • 167
Applying Kustomizations • 168
Patching • 169
Kustomizing the entire staging namespace • 170
Launching jobs 171
Running jobs in parallel • 172
Cleaning up completed jobs • 173
Scheduling cron jobs • 174
Mixing non-cluster components 176
Outside-the-cluster-network components • 176
Inside-the-cluster-network components • 176
Managing the Hue platform with Kubernetes • 176
Using liveness probes to ensure your containers are alive • 177
Using readiness probes to manage dependencies • 178
Using startup probes • 178
Employing init containers for orderly pod bring-up • 179
Pod readiness and readiness gates • 180
Sharing with DaemonSet pods • 180
Evolving the Hue platform with Kubernetes 182
Utilizing Hue in an enterprise • 182
Advancing science with Hue • 182

Educating the kids of the future with Hue • 182
Summary 183

Chapter 6: Managing Storage 185

Persistent volumes walk-through 186
Understanding volumes • 186
Using emptyDir for intra-pod communication • 186
Using HostPath for intra-node communication • 188
Using local volumes for durable node storage • 190
Provisioning persistent volumes • 191
Creating persistent volumes • 192
Capacity • 193
Volume mode • 193
Access modes • 193
Reclaim policy • 194
Storage class • 194
Volume type • 194
Mount options • 194
Projected volumes • 195
serviceAccountToken projected volumes • 196
Creating a local volume • 196
Making persistent volume claims • 197
Mounting claims as volumes • 199
Raw block volumes • 200
CSI ephemeral volumes • 202
Generic ephemeral volumes • 202
Storage classes • 204
Default storage class • 205
Demonstrating persistent volume storage end to end 205
Public cloud storage volume types – GCE, AWS, and Azure 210
AWS Elastic Block Store (EBS) • 210
AWS Elastic File System (EFS) • 212
GCE persistent disk • 215
Google Cloud Filestore • 216
Azure data disk • 217
Azure file • 218

GlusterFS and Ceph volumes in Kubernetes 219
Using GlusterFS • 219
Creating endpoints • 220
Adding a GlusterFS Kubernetes service • 220
Creating pods • 220
Using Ceph • 221
Connecting to Ceph using RBD • 221
Rook • 223
Integrating enterprise storage into Kubernetes 227
Other storage providers • 228
The Container Storage Interface 228
Advanced storage features • 229
Volume snapshots • 229
CSI volume cloning • 230
Storage capacity tracking • 231
Volume health monitoring • 231
Summary 232

Chapter 7: Running Stateful Applications with Kubernetes 235

Stateful versus stateless applications in Kubernetes 235
Understanding the nature of distributed data-intensive apps • 236
Why manage the state in Kubernetes? • 236
Why manage the state outside of Kubernetes? • 236
Shared environment variables versus DNS records for discovery • 236
Accessing external data stores via DNS • 237
Accessing external data stores via environment variables • 237
Consuming a ConfigMap as an environment variable • 238
Using a redundant in-memory state • 238
Using DaemonSet for redundant persistent storage • 239
Applying persistent volume claims • 239
Utilizing StatefulSet • 239
Working with StatefulSets • 241
Running a Cassandra cluster in Kubernetes 243
A quick introduction to Cassandra • 243
The Cassandra Docker image • 244
Exploring the build.sh script • 246
Exploring the run.sh script • 247

Hooking up Kubernetes and Cassandra • 252
Digging into the Cassandra configuration file • 252
The custom seed provider • 253
Creating a Cassandra headless service • 254
Using StatefulSet to create the Cassandra cluster • 255
Dissecting the StatefulSet YAML file • 255
Summary 259

Chapter 8: Deploying and Updating Applications 261

Live cluster updates 261
Rolling updates • 262
Complex deployments • 264
Blue-green deployments • 265
Canary deployments • 266
Managing data-contract changes • 267
Migrating data • 267
Deprecating APIs • 267
Horizontal pod autoscaling 268
Creating a horizontal pod autoscaler • 269
Custom metrics • 272
Keda • 273
Autoscaling with kubectl • 274
Performing rolling updates with autoscaling • 276
Handling scarce resources with limits and quotas 279
Enabling resource quotas • 279
Resource quota types • 279
Compute resource quota • 280
Storage resource quota • 280
Object count quota • 281
Quota scopes • 282
Resource quotas and priority classes • 282
Requests and limits • 283
Working with quotas • 283
Using namespace-specific context • 283
Creating quotas • 283
Using limit ranges for default compute quotas • 287

Continuous integration and deployment 289
What is a CI/CD pipeline? • 289
Designing a CI/CD pipeline for Kubernetes • 290
Provisioning infrastructure for your applications 290
Cloud provider APIs and tooling • 291
Terraform • 291
Pulumi • 292
Custom operators • 294
Using Crossplane • 294
Summary 295

Chapter 9: Packaging Applications 297

Understanding Helm 297
The motivation for Helm • 298
The Helm 3 architecture • 298
Helm release secrets • 298
The Helm client • 299
The Helm library • 299
Helm 2 vs Helm 3 • 299
Using Helm 300
Installing Helm • 300
Installing the Helm client • 300
Finding charts • 300
Adding repositories • 301
Installing packages • 304
Checking the installation status • 305
Customizing a chart • 307
Additional installation options • 308
Upgrading and rolling back a release • 308
Deleting a release • 310
Working with repositories • 311
Managing charts with Helm • 311
Taking advantage of starter packs • 312
Creating your own charts 313
The Chart.yaml file • 313
Versioning charts • 313
The appVersion field • 313

Deprecating charts • 314
Chart metadata files • 314
Managing chart dependencies • 315
Utilizing additional subfields of the dependencies field • 316
Using templates and values • 317
Writing template files • 318
Testing and troubleshooting your charts • 320
Embedding built-in objects • 322
Feeding values from a file • 322
Helm alternatives 323
Kustomize • 324
Cue • 324
kapp-controller • 324
Summary 324

Chapter 10: Exploring Kubernetes Networking 325

Understanding the Kubernetes networking model 325
Intra-pod communication (container to container) • 326
Inter-pod communication (pod to pod) • 326
Pod-to-service communication • 326
Lookup and discovery • 328
Self-registration • 328
Services and endpoints • 328
Loosely coupled connectivity with queues • 328
Loosely coupled connectivity with data stores • 329
Kubernetes ingress • 329
DNS in Kubernetes • 330
CoreDNS • 332
Kubernetes network plugins 333
Basic Linux networking • 334
IP addresses and ports • 334
Network namespaces • 334
Subnets, netmasks, and CIDRs • 334
Virtual Ethernet devices • 334
Bridges • 334
Routing • 334
Maximum transmission unit • 335

Pod networking • 335
Kubenet • 335
Requirements • 335
Setting the MTU • 336
The CNI • 336
The container runtime • 337
The CNI plugin • 337
Kubernetes and eBPF 340
Kubernetes networking solutions 341
Bridging on bare-metal clusters • 341
The Calico project • 342
Weave Net • 342
Cilium • 342
Efficient IP allocation and routing • 342
Identity-based service-to-service communication • 343
Load balancing • 343
Bandwidth management • 343
Observability • 343
Using network policies effectively 344
Understanding the Kubernetes network policy design • 344
Network policies and CNI plugins • 344
Configuring network policies • 344
Implementing network policies • 345
Load balancing options 346
External load balancers • 347
Configuring an external load balancer • 347
Finding the load balancer IP addresses • 348
Preserving client IP addresses • 349
Understanding even external load balancing • 349
Service load balancers • 350
Ingress • 350
HAProxy • 352
MetalLB • 354
Traefik • 354
Kubernetes Gateway API • 355
Gateway API resources • 355
Attaching routes to gateways • 356

Gateway API in action • 356
Writing your own CNI plugin 357
First look at the loopback plugin • 358
Building on the CNI plugin skeleton • 362
Reviewing the bridge plugin • 364
Summary 368

Chapter 11: Running Kubernetes on Multiple Clusters 371

Stretched Kubernetes clusters versus multi-cluster Kubernetes 372
Understanding stretched Kubernetes clusters • 372
Pros of a stretched cluster • 372
Cons of a stretched cluster • 372
Understanding multi-cluster Kubernetes • 372
Pros of multi-cluster Kubernetes • 373
Cons of multi-cluster Kubernetes • 373
The history of cluster federation in Kubernetes 373
Cluster API 374
Cluster API architecture • 374
Management cluster • 375
Work cluster • 375
Bootstrap provider • 376
Infrastructure provider • 376
Control plane • 376
Custom resources • 376
Karmada 377
Karmada architecture • 378
Karmada concepts • 378
ResourceTemplate • 379
PropagationPolicy • 379
OverridePolicy • 379
Additional capabilities • 379
Clusternet 380
Clusternet architecture • 380
Clusternet hub • 381
Clusternet scheduler • 381
Clusternet agent • 381
Multi-cluster deployment • 381

Clusterpedia 382
Clusterpedia architecture • 382
Clusterpedia API server • 383
ClusterSynchro manager • 383
Storage layer • 384
Storage component • 384
Importing clusters • 384
Advanced multi-cluster search • 384
Resource collections • 385
Open Cluster Management 385
OCM architecture • 386
OCM cluster lifecycle • 387
OCM application lifecycle • 387
OCM governance, risk, and compliance • 388
Virtual Kubelet 389
Tensile-kube • 390
Admiralty • 391
Liqo • 392
Introducing the Gardener project 393
Understanding the terminology of Gardener • 393
Understanding the conceptual model of Gardener • 394
Diving into the Gardener architecture • 395
Managing the cluster state • 395
Managing the control plane • 395
Preparing the infrastructure • 395
Using the Machine controller manager • 395
Networking across clusters • 396
Monitoring clusters • 396
The gardenctl CLI • 396
Extending Gardener • 397
Summary 401

Chapter 12: Serverless Computing on Kubernetes 403

Understanding serverless computing 403
Running long-running services on “serverless” infrastructure • 404
Running functions as a service on “serverless” infrastructure • 404

Serverless Kubernetes in the cloud 405
Azure AKS and Azure Container Instances • 406
AWS EKS and Fargate • 408
Google Cloud Run • 409
Knative 410
Knative serving • 410
Install a quickstart environment • 411
The Knative Service object • 413
Creating new revisions • 415
The Knative Route object • 415
The Knative Configuration object • 417
Knative eventing • 419
Getting familiar with Knative eventing terminology • 419
The architecture of Knative eventing • 421
Checking the scale to zero option of Knative • 422
Kubernetes Function-as-a-Service frameworks 423
OpenFaaS • 423
Delivery pipeline • 423
OpenFaaS features • 424
OpenFaaS architecture • 425
Taking OpenFaaS for a ride • 426
Fission • 436
Fission executor • 437
Fission workflows • 438
Experimenting with Fission • 441
Summary 442

Chapter 13: Monitoring Kubernetes Clusters 445

Understanding observability 446
Logging • 446
Log format • 446
Log storage • 446
Log aggregation • 446
Metrics • 447
Distributed tracing • 447
Application error reporting • 448

Dashboards and visualization • 448
Alerting • 448
Logging with Kubernetes 448
Container logs • 449
Kubernetes component logs • 450
Centralized logging • 450
Choosing a log collection strategy • 450
Cluster-level central logging • 452
Remote central logging • 452
Dealing with sensitive log information • 453
Using Fluentd for log collection • 453
Collecting metrics with Kubernetes 454
Monitoring with the Metrics Server • 455
The rise of Prometheus • 457
Installing Prometheus • 458
Interacting with Prometheus • 460
Incorporating kube-state-metrics • 461
Utilizing the node exporter • 462
Incorporating custom metrics • 463
Alerting with Alertmanager • 463
Visualizing your metrics with Grafana • 465
Considering Loki • 467
Distributed tracing with Kubernetes 468
What is OpenTelemetry? • 468
OpenTelemetry tracing concepts • 469
Introducing Jaeger • 469
Jaeger architecture • 470
Installing Jaeger • 471
Troubleshooting problems 474
Taking advantage of staging environments • 474
Detecting problems at the node level • 475
Problem daemons • 476
Dashboards vs. alerts • 476
Logs vs metrics vs. error reports • 477
Detecting performance and root cause with distributed tracing • 478
Summary 478

Chapter 14: Utilizing Service Meshes 479

What is a service mesh? **479**
Choosing a service mesh **483**
Envoy • 484
Linkerd 2 • 484
Kuma • 484
AWS App Mesh • 484
Mæsh • 484
Istio • 485
OSM (Open Service Mesh) • 485
Cilium Service Mesh • 485
Understanding the Istio architecture **485**
Envoy • 486
Pilot • 487
Citadel • 487
Galley • 488
Incorporating Istio into your Kubernetes cluster **488**
Preparing a minikube cluster for Istio • 488
Installing Istio • 488
Installing BookInfo • 491
Working with Istio **494**
Traffic management • 494
Security • 497
Istio identity • 498
Istio certificate management • 499
Istio authentication • 499
Istio authorization • 500
Monitoring and observability • 503
Istio access logs • 503
Metrics • 506
Distributed tracing • 509
Visualizing your service mesh with Kiali • 512
Summary **513**

Chapter 15: Extending Kubernetes 515

Working with the Kubernetes API 515
Understanding OpenAPI • 515
Setting up a proxy • 516
Exploring the Kubernetes API directly • 516
Using Postman to explore the Kubernetes API • 518
Filtering the output with HTTPie and jq • 519
Accessing the Kubernetes API via the Python client • 520
Dissecting the CoreV1Api group • 520
Listing objects • 523
Creating objects • 523
Watching objects • 525
Creating a pod via the Kubernetes API • 526
Controlling Kubernetes using Go and controller-runtime • 527
Using controller-runtime via go-k8s • 527
Invoking kubectl programmatically from Python and Go • 529
Using Python subprocess to run kubectl • 529
Extending the Kubernetes API 533
Understanding Kubernetes extension points and patterns • 533
Extending Kubernetes with plugins • 534
Extending Kubernetes with the cloud controller manager • 534
Extending Kubernetes with webhooks • 535
Extending Kubernetes with controllers and operators • 535
Extending Kubernetes scheduling • 536
Extending Kubernetes with custom container runtimes • 536
Introducing custom resources • 537
Developing custom resource definitions • 538
Integrating custom resources • 539
Dealing with unknown fields • 540
Finalizing custom resources • 542
Adding custom printer columns • 543
Understanding API server aggregation • 544
Building Kubernetes-like control planes • 545
Writing Kubernetes plugins 545
Writing a custom scheduler • 545
Understanding the design of the Kubernetes scheduler • 545

Scheduling pods manually • 548
Preparing our own scheduler • 549
Assigning pods to the custom scheduler • 550
Writing kubectl plugins • 551
Understanding kubectl plugins • 552
Managing kubectl plugins with Krew • 552
Creating your own kubectl plugin • 553
Employing access control webhooks 555
Using an authentication webhook • 555
Using an authorization webhook • 557
Using an admission control webhook • 559
Configuring a webhook admission controller on the fly • 560
Additional extension points 562
Providing custom metrics for horizontal pod autoscaling • 562
Extending Kubernetes with custom storage • 563
Summary 564

Chapter 16: Governing Kubernetes 565

Kubernetes in the enterprise 565
Requirements of enterprise software • 566
Kubernetes and enterprise software • 566
What is Kubernetes governance? 567
Image management • 567
Pod security • 567
Network policy • 567
Configuration constraints • 568
RBAC and admission control • 568
Policy management • 568
Policy validation and enforcement • 568
Reporting • 569
Audit • 569
Policy engines 569
Admission control as the foundation of policy engines • 569
Responsibilities of a policy engine • 570
Quick review of open source policy engines • 570
OPA/Gatekeeper • 570
Kyverno • 571

jsPolicy • 572
Kubewarden • 572
Kyverno deep dive 573
Quick intro to Kyverno • 574
Installing and configuring Kyverno • 575
Installing pod security policies • 578
Configuring Kyverno • 579
Applying Kyverno policies • 580
Kyverno policies in depth • 583
Understanding policy settings • 583
Understanding Kyverno policy rules • 584
Validating requests • 587
Mutating resources • 588
Generating resources • 590
Advanced policy rules • 591
Writing and testing Kyverno policies • 592
Writing validating policies • 592
Writing mutating policies • 595
Writing generating policies • 597
Testing policies • 598
The Kyverno CLI • 598
Understanding Kyverno tests • 601
Writing Kyverno tests • 603
Running Kyverno tests • 605
Viewing Kyverno reports • 606
Summary 611

Chapter 17: Running Kubernetes in Production 613

Understanding Managed Kubernetes in the cloud 613
Deep integration • 614
Quotas and limits • 615
Real-world examples of quotas and limits • 615
Capacity planning • 616
When should you not use Managed Kubernetes? • 616
Managing multiple clusters 617
Geo-distributed clusters • 617
Multi-cloud • 617

Hybrid • 617
Kubernetes on the edge • 618
Building effective processes for large-scale Kubernetes deployments 618
The development lifecycle • 618
Environments • 619
Separated environments • 619
Staging environment fidelity • 619
Resource quotas • 620
Promotion process • 620
Permissions and access control • 620
The principle of least privilege • 620
Assign permissions to groups • 620
Fine-tune your permission model • 620
Break glass • 621
Observability • 621
One-stop shop observability • 621
Troubleshooting your observability stack • 621
Handling infrastructure at scale 622
Cloud-level considerations • 622
Compute • 622
Design your cluster breakdown • 623
Design your node pool breakdown • 623
Networking • 623
IP address space management • 623
Network topology • 624
Network segmentation • 624
Cross-cluster communication • 624
Cross-cloud communication • 625
Cross-cluster service meshes • 625
Managing egress at scale • 626
Managing the DNS at the cluster level • 627
Storage • 627
Choose the right storage solutions • 627
Data backup and recovery • 627
Storage monitoring • 628
Data security • 628
Optimize storage usage • 628

Test and validate storage performance • 628
Managing clusters and node pools 628
Provisioning managed clusters and node pools • 628
The Cluster API • 629
Terraform/Pulumi • 629
Kubernetes operators • 629
Utilizing managed nodes • 629
Bin packing and utilization 629
Understanding workload shape • 630
Setting requests and limits • 630
Utilizing init containers • 631
Shared nodes vs. dedicated nodes • 631
Large nodes vs, small nodes • 632
Small and short-lived workloads • 634
Pod density • 635
Fallback node pools • 635
Upgrading Kubernetes 635
Know the lifecycle of your cloud provider • 636
Upgrading clusters • 637
Planning an upgrade • 637
Detecting incompatible resources • 637
Updating incompatible resources • 640
Dealing with removed features • 641
Upgrading node pools • 642
Syncing with control plane upgrades • 642
How to perform node pool upgrades • 642
Concurrent draining • 643
Dealing with workloads with no PDB • 643
Dealing with workloads with PDB zero • 643
Troubleshooting 644
Handling pending pods • 644
Handling scheduled pods that are not running • 650
Handling running pods that are not ready • 652
Cost management 655
Cost mindset • 655
Cost observability • 656
Tagging • 656

Policies and budgets • 656
Alerting • 657
Tools • 657
The smart selection of resources • 657
Efficient usage of resources • 658
Discounts, reserved instances, and spot instances • 658
Discounts and credits • 658
Reserved instances • 658
Spot instances • 659
Invest in local environments • 660
Summary 660

Chapter 18: The Future of Kubernetes 661

The Kubernetes momentum 661
The importance of the CNCF • 662
Project curation • 663
Certification • 663
Training • 664
Community and education • 664
Tooling • 664
The rise of managed Kubernetes platforms 664
Public cloud Kubernetes platforms • 665
Bare metal, private clouds, and Kubernetes on the edge • 665
Kubernetes PaaS (Platform as a Service) • 665
Upcoming trends 666
Security • 666
Networking • 666
Custom hardware and devices • 667
Service mesh • 667
Serverless computing • 668
Kubernetes on the edge • 668
Native CI/CD • 668
Operators • 669
Kubernetes and AI 669
Kubernetes and AI synergy • 670
Training AI models on Kubernetes • 670
Running AI-based systems on Kubernetes • 670

Kubernetes and AIOps • 671
Kubernetes challenges **671**
Kubernetes complexity • 671
Serverless function platforms • 671
Summary **672**

Other Books You May Enjoy **677**

Index **681**

Preface

Kubernetes is an open source system that automates the deployment, scaling, and management of containerized applications. If you are running more than just a few containers or want to automate the management of your containers, you need Kubernetes. This book focuses on guiding you through the advanced management of Kubernetes clusters.

The book begins by explaining the fundamentals behind Kubernetes' architecture and covers Kubernetes' design in detail. You will discover how to run complex stateful microservices on Kubernetes, including advanced features such as horizontal pod autoscaling, rolling updates, resource quotas, and persistent storage backends. Using real-world use cases, you will explore the options for network configuration and understand how to set up, operate, secure, and troubleshoot Kubernetes clusters. Finally, you will learn about advanced topics such as custom resources, API aggregation, service meshes, and serverless computing. All the content is up to date and complies with Kubernetes 1.26. By the end of this book, you'll know everything you need to know to go from intermediate to advanced level.

Who this book is for

The book is for system administrators and developers who have intermediate-level knowledge of Kubernetes and now want to master the advanced features. You should also have basic networking knowledge. This advanced-level book provides a pathway to master Kubernetes.

What this book covers

Chapter 1, *Understanding Kubernetes Architecture*, in this chapter, we will build together the foundation necessary to utilize Kubernetes to its full potential. We will start by understanding what Kubernetes is, what Kubernetes isn't, and what container orchestration means exactly. Then we will cover important Kubernetes concepts that will form the vocabulary we will use throughout the book.

Chapter 2, *Creating Kubernetes Clusters*, in this chapter, we will roll up our sleeves and build some Kubernetes clusters using minikube, KinD, and k3d. We will discuss and evaluate other tools such as kubeadm and Kubespray. We will also look into deployment environments such as local, cloud, and bare metal.

Chapter 3, *High Availability and Reliability*, in this chapter, we will dive into the topic of highly available clusters. This is a complicated topic. The Kubernetes project and the community haven't settled on one true way to achieve high-availability nirvana.

There are many aspects to highly available Kubernetes clusters, such as ensuring that the control plane can keep functioning in the face of failures, protecting the cluster state in etcd, protecting the system's data, and recovering capacity and/or performance quickly. Different systems will have different reliability and availability requirements.

Chapter 4, *Securing Kubernetes*, in this chapter, we will explore the important topic of security. Kubernetes clusters are complicated systems composed of multiple layers of interacting components. The isolation and compartmentalization of different layers is very important when running critical applications. To secure the system and ensure proper access to resources, capabilities, and data, we must first understand the unique challenges facing Kubernetes as a general-purpose orchestration platform that runs unknown workloads. Then we can take advantage of various securities, isolation, and access control mechanisms to make sure the cluster, the applications running on it, and the data are all safe. We will discuss various best practices and when it is appropriate to use each mechanism.

Chapter 5, *Using Kubernetes Resources in Practice*, in this chapter, we will design a fictional massive-scale platform that will challenge Kubernetes' capabilities and scalability. The Hue platform is all about creating an omniscient and omnipotent digital assistant. Hue is a digital extension of you. Hue will help you do anything, find anything, and, in many cases, will do a lot on your behalf. It will obviously need to store a lot information, integrate with many external services, respond to notifications and events, and be smart about interacting with you.

Chapter 6, *Managing Storage*, in this chapter, we'll look at how Kubernetes manages storage. Storage is very different from computation, but at a high level, they are both resources. Kubernetes as a generic platform takes the approach of abstracting storage behind a programming model and a set of plugins for storage providers.

Chapter 7, *Running Stateful Applications with Kubernetes*, in this chapter, we will learn how to run stateful applications on Kubernetes. Kubernetes takes a lot of work out of our hands by automatically starting and restarting pods across the cluster nodes as needed, based on complex requirements and configurations such as namespaces, limits, and quotas. But when pods run storage-aware software, such as databases and queues, relocating a pod can cause the system to break.

Chapter 8, *Deploying and Updating Applications*, in this chapter, we will explore the automated pod scalability that Kubernetes provides, how it affects rolling updates, and how it interacts with quotas. We will touch on the important topic of provisioning and how to choose and manage the size of the cluster. Finally, we will go over how the Kubernetes team improved the performance of Kubernetes and how they test the limits of Kubernetes with the Kubemark tool.

Chapter 9, *Packaging Applications*, in this chapter, we are going to look into Helm, the Kubernetes package manager. Every successful and non-trivial platform must have a good packaging system. Helm was developed by Deis (acquired by Microsoft on April 4, 2017) and later contributed to the Kubernetes project directly. It became a CNCF project in 2018. We will start by understanding the motivation for Helm, its architecture, and its components.

Chapter 10, *Exploring Kubernetes Networking*, in this chapter, we will examine the important topic of networking. Kubernetes, as an orchestration platform, manages containers/pods running on different machines (physical or virtual) and requires an explicit networking model.

Chapter 11, *Running Kubernetes on Multiple Clusters*, in this chapter, we'll take it to the next level, by running Kubernetes on multiple clouds and clusters. A Kubernetes cluster is a closely knit unit where all the components run in relative proximity and are connected by a fast network (typically a physical data center or cloud provider availability zone). This is great for many use cases, but there are several important use cases where systems need to scale beyond a single cluster.

Chapter 12, *Serverless Computing on Kubernetes*, in this chapter, we will explore the fascinating world of serverless computing in the cloud. The term "serverless" is getting a lot of attention, but it is a misnomer used to describe two different paradigms. A true serverless application runs as a web application in the user's browser or a mobile app and only interacts with external services. The types of serverless systems we build on Kubernetes are different.

Chapter 13, *Monitoring Kubernetes Clusters*, in this chapter, we're going to talk about how to make sure your systems are up and running and performing correctly and how to respond when they aren't. In *Chapter 3, High Availability and Reliability*, we discussed related topics. The focus here is about knowing what's going on in your system and what practices and tools you can use.

Chapter 14, *Utilizing Service Meshes*, in this chapter, we will learn how service meshes allow you to externalize cross-cutting concerns like monitoring and observability from the application code. The service mesh is a true paradigm shift in the way you can design, evolve, and operate distributed systems on Kubernetes. I like to think of it as aspect-oriented programming for cloud-native distributed systems.

Chapter 15, *Extending Kubernetes*, in this chapter, we will dig deep into the guts of Kubernetes. We will start with the Kubernetes API and learn how to work with Kubernetes programmatically via direct access to the API, the Go client, and automating kubectl. Then we'll look into extending the Kubernetes API with custom resources. The last part is all about the various plugins Kubernetes supports. Many aspects of Kubernetes operation are modular and designed for extension. We will examine the API aggregation layer and several types of plugins, such as custom schedulers, authorization, admission control, custom metrics, and volumes. Finally, we'll look into extending kubectl and adding your own commands.

Chapter 16, *Governing Kubernetes*, in this chapter, we will learn about the growing role of Kubernetes in large enterprise organizations, as well as what governance is and how it is applied in Kubernetes. We will look at policy engines, review some popular policy engines, and then dive deep into Kyverno.

Chapter 17, *Running Kubernetes in Production*, in this chapter, we will turn our attention to the overall management of Kubernetes in production. The focus will be on running multiple managed Kubernetes clusters in the cloud. There are so many different aspects to running Kubernetes in production. We will focus mostly on the compute side, which is the most fundamental and dominant.

Chapter 18, *The Future of Kubernetes*, in this chapter, we'll look at the future of Kubernetes from multiple angles. We'll start with the momentum of Kubernetes since its inception, across dimensions such as community, ecosystem, and mindshare. Spoiler alert: Kubernetes won the container orchestration wars by a landslide. As Kubernetes grows and matures, the battle lines shift from beating competitors to fighting against its own complexity. Usability, tooling, and education will play a major role as container orchestration is still new, fast-moving, and not a well-understood domain. Then we will take a look at some very interesting patterns and trends.

To get the most out of this book

To follow the examples in each chapter, you need a recent version of Docker and Kubernetes installed onto your machine, ideally Kubernetes 1.18. If your operating system is Windows 10 Professional, you can enable hypervisor mode; otherwise, you will need to install VirtualBox and use a Linux guest OS. If you use macOS then you're good to go.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Kubernetes-4th-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gXMql`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "If you chose HyperKit instead of VirtualBox, you need to add the flag `--vm-driver=hyperkit` when starting the cluster."

A block of code is set as follows:

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
metadata:
  name: "example-etcd-cluster"
spec:
  size: 3
  version: "3.2.13"
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are highlighted:

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
metadata:
  name: "example-etcd-cluster"
spec:
  size: 3
  version: "3.2.13"
```

Any command-line input or output is written as follows:

```
$ k get pods
NAME                    READY   STATUS    RESTARTS   AGE
echo-855975f9c-r6kj8    1/1     Running   0          2m11s
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share your thoughts

Once you've read *Mastering Kubernetes, Fourth Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781804611395`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Understanding Kubernetes Architecture

In one sentence, Kubernetes is a platform to orchestrate the deployment, scaling, and management of container-based applications. You have probably read about Kubernetes, and maybe even dipped your toes in and used it in a side project or maybe even at work. But to understand what Kubernetes is all about, how to use it effectively, and what the best practices are requires much more.

Kubernetes is a big open source project and ecosystem with a lot of code and a lot of functionality. Kubernetes came out of Google, but joined the **Cloud Native Computing Foundation** (**CNCF**) and is now the de facto standard in the space of container-based applications. According to the 2021 CNCF survey, 96% of organizations use or evaluate Kubernetes.

In this chapter, we will build the foundation necessary to utilize Kubernetes to its full potential. We will start by understanding what Kubernetes is, what Kubernetes isn't, and what container orchestration means exactly. Then we will cover important Kubernetes concepts that will form the vocabulary we will use throughout the book. After that, we will dive into the architecture of Kubernetes properly and look at how it enables all the capabilities it provides for its users. Then, we will discuss how Kubernetes supports multiple container runtimes in a generic way.

The topics we will discuss are:

- What is Kubernetes?
- What Kubernetes is not
- Understanding container orchestration
- Kubernetes concepts
- Diving into Kubernetes architecture in depth
- Kubernetes container runtimes

At the end of this chapter, you will have a solid understanding of container orchestration, what problems Kubernetes addresses, the rationale of Kubernetes design and architecture, and the different runtime engines it supports.

What is Kubernetes?

Kubernetes is a platform that encompasses a huge number of services and capabilities that keeps growing. The core functionality is scheduling workloads in containers across your infrastructure, but it doesn't stop there. Here are some of the other capabilities Kubernetes brings to the table:

- Providing authentication and authorization
- Debugging applications
- Accessing and ingesting logs
- Rolling updates
- Using Cluster Autoscaling
- Using the Horizontal Pod Autoscaler
- Replicating application instances
- Checking application health and readiness
- Monitoring resources
- Balancing loads
- Naming and service discovery
- Distributing secrets
- Mounting storage systems

We will cover all these capabilities in great detail throughout the book. At this point, just absorb and appreciate how much value Kubernetes can add to your system.

Kubernetes has impressive scope, but it is also important to understand what Kubernetes explicitly doesn't provide.

What Kubernetes is not

Kubernetes is not a **Platform as a Service** (**PaaS**). It doesn't dictate many important aspects that are left to you or to other systems built on top of Kubernetes, such as OpenShift and Tanzu. For example:

- Kubernetes doesn't require a specific application type or framework
- Kubernetes doesn't require a specific programming language
- Kubernetes doesn't provide databases or message queues
- Kubernetes doesn't distinguish apps from services
- Kubernetes doesn't have a click-to-deploy service marketplace
- Kubernetes doesn't provide a built-in function as a service solution
- Kubernetes doesn't mandate logging, monitoring, and alerting systems
- Kubernetes doesn't provide a CI/CD pipeline

Understanding container orchestration

The primary responsibility of Kubernetes is container orchestration. That means making sure that all the containers that execute various workloads are scheduled to run on physical or virtual machines. The containers must be packed efficiently following the constraints of the deployment environment and the cluster configuration. In addition, Kubernetes must keep an eye on all running containers and replace dead, unresponsive, or otherwise unhealthy containers. Kubernetes provides many more capabilities that you will learn about in the following chapters. In this section, the focus is on containers and their orchestration.

Physical machines, virtual machines, and containers

It all starts and ends with hardware. In order to run your workloads, you need some real hardware provisioned. That includes actual physical machines, with certain compute capabilities (CPUs or cores), memory, and some local persistent storage (spinning disks or SSDs). In addition, you will need some shared persistent storage and to hook up all these machines using networking, so they can find and talk to each other. At this point, you run multiple virtual machines on the physical machines or stay at the bare-metal level (no virtual machines). Kubernetes can be deployed on a bare-metal cluster (real hardware) or on a cluster of virtual machines. Kubernetes in turn can orchestrate the containers it manages directly on bare-metal or on virtual machines. In theory, a Kubernetes cluster can be composed of a mix of bare-metal and virtual machines, but this is not very common. There are many more esoteric configurations with different levels of encapsulation, such as virtual Kubernetes clusters running inside the namespaces of another Kubernetes cluster.

The benefits of containers

Containers represent a true paradigm shift in the development and operation of large, complicated software systems. Here are some of the benefits compared to more traditional models:

- Agile application creation and deployment
- Continuous development, integration, and deployment
- Development and operations separation of concerns
- Environmental consistency across development, testing, staging, and production
- Cloud and OS distribution portability
- Application-centric management
- Resource isolation
- Resource utilization

Containers in the cloud

Microservices are the dominant architecture for modern large-scale systems. The primary idea is to break down the system into small services with well-defined responsibilities that manage their own data and communicate with other microservices through well-defined APIs.

Containers are ideal to package microservices because, while providing isolation to the microservice, they are very lightweight and you don't incur a lot of overhead when deploying many microservices, as you do with virtual machines. That makes containers ideal for cloud deployment, where allocating a whole virtual machine for each microservice would be cost-prohibitive.

All major cloud providers, such as Amazon AWS, Google's GCE, and Microsoft's Azure, provide container hosting services these days. Many other companies jumped on the Kubernetes bandwagon and offer managed Kubernetes services, including IBM IKS, Alibaba Cloud, DigitalOcean DKS, Oracle OKS, OVH Managed Kubernetes, and Rackspace KaaS.

Google's GKE was always based on Kubernetes. AWS **Elastic Kubernetes Service** (**EKS**) was added in addition to the proprietary AWS ECS orchestration solution. Microsoft Azure's container service used to be based on Apache Mesos, but later switched to Kubernetes with **Azure Kubernetes Service** (**AKS**). You could always deploy Kubernetes on all the cloud platforms, but it wasn't deeply integrated with other services. But, at the end of 2017 all cloud providers announced direct support for Kubernetes. Microsoft launched AKS, and AWS released EKS. Also, various other companies offer managed Kubernetes services such as IBM, Oracle, Digital Ocean, Alibaba, Tencent, and Huawei.

Cattle versus pets

In the olden days, when systems were small, each server had a name. Developers and users knew exactly what software was running on each machine. I remember that, in many of the companies I worked for, we had multi-day discussions to decide on a naming theme for our servers. For example, composers and Greek mythology characters were popular choices. Everything was very cozy. You treated your servers like beloved pets. When a server died it was a major crisis. Everybody scrambled to try to figure out where to get another server, what was even running on the dead server, and how to get it working on the new server. If the server stored some important data, then hopefully you had an up-to-date backup, and maybe you'd even be able to recover it.

Obviously, that approach doesn't scale. When you have tens or hundreds of servers, you must start treating them like cattle. You think about the collective and not individuals. You may still have some pets like your CI/CD machines (although managed CI/CD solutions are becoming more common), but your web servers and backend services are just cattle.

Kubernetes takes the cattle approach to the extreme and takes full responsibility for allocating containers to specific machines. You don't need to interact with individual machines (nodes) most of the time. This works best for stateless workloads. For stateful applications, the situation is a little different, but Kubernetes provides a solution called StatefulSet, which we'll discuss soon.

In this section, we covered the idea of container orchestration and discussed the relationships between hosts (physical or virtual) and containers, as well as the benefits of running containers in the cloud. We then finished with a discussion about cattle versus pets. In the following section, we will get to know the world of Kubernetes and learn its concepts and terminology.

Kubernetes concepts

In this section, we'll briefly introduce many important Kubernetes concepts and give you some context as to why they are needed and how they interact with each other. The goal is to get familiar with these terms and concepts. Later, we will see how these concepts are woven together and organized into API groups and resource categories to achieve awesomeness. You can consider many of these concepts as building blocks. Some concepts, such as nodes and the control plane, are implemented as a set of Kubernetes components. These components are at a different abstraction level, and we will discuss them in detail in a dedicated section, *Kubernetes components*.

Here is the Kubernetes architecture diagram:

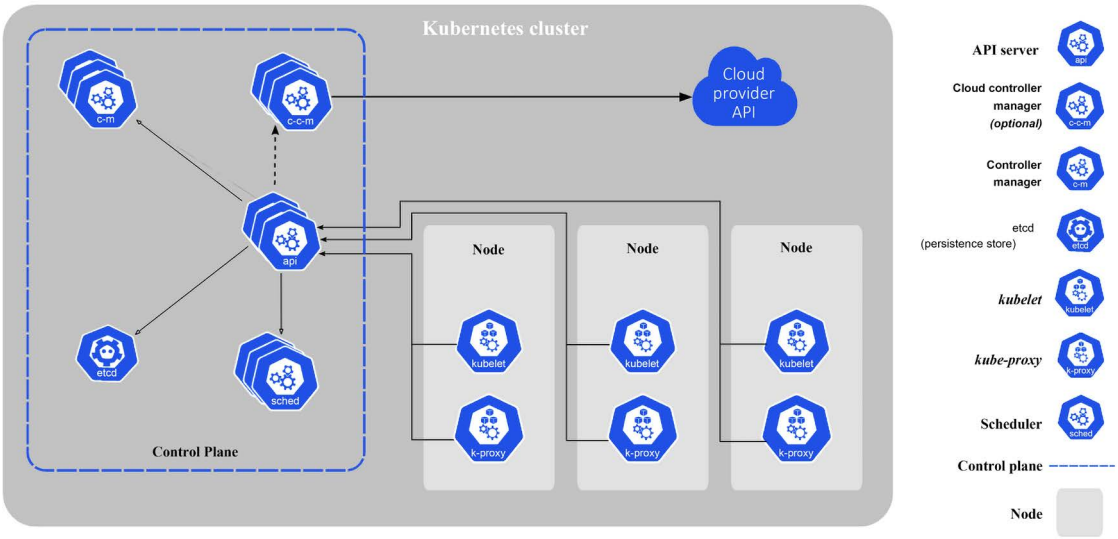


Figure 1.1: Kubernetes architecture

Node

A node is a single host. It may be a physical or virtual machine. Its job is to run pods. Each Kubernetes node runs several Kubernetes components, such as the kubelet, the container runtime, and the kube-proxy. Nodes are managed by the Kubernetes control plane. The nodes are the worker bees of Kubernetes and shoulder all the heavy lifting. In the past they were called **minions**. If you read some old documentation or articles, don't get confused. Minions are just nodes.

Cluster

A cluster is a collection of hosts (nodes) that provide compute, memory, storage, and networking resources. Kubernetes uses these resources to run the various workloads that comprise your system. Note that your entire system may consist of multiple clusters. We will discuss this advanced use case of multi-cluster systems in detail later.

Control plane

The control plane of Kubernetes consists of several components, such as an API server, a scheduler, a controller manager and, optionally, a cloud controller manager. The control plane is responsible for the global state of the cluster, cluster-level scheduling of pods, and handling of events. Usually, all the control plane components are set up on the same host although it's not required. When considering high-availability scenarios or very large clusters, you will want to have control plane redundancy. We will discuss highly available clusters in detail in *Chapter 3*, *High Availability and Reliability*.

Pod

A pod is the unit of work in Kubernetes. Each pod contains one or more containers (so you can think of it as a container container). A pod is scheduled as an atomic unit (all its containers run on the same machine). All the containers in a pod have the same IP address and port space; they can communicate with each other using localhost or standard inter-process communication. In addition, all the containers in a pod can have access to shared local storage on the node hosting the pod. Containers don't get access to local storage or any other storage by default. Volumes of storage must be mounted into each container inside the pod explicitly.

Pods are an important feature of Kubernetes. It is possible to run multiple applications inside a single container by having something like supervisord as the main process that runs multiple processes, but this practice is often frowned upon for the following reasons:

- **Transparency**: Making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring. This facilitates a number of conveniences for users.
- **Decoupling software dependencies**: The individual containers may be versioned, rebuilt, and redeployed independently. Kubernetes may even support live updates of individual containers someday.
- **Ease of use**: Users don't need to run their own process managers, worry about signal and exit-code propagation, and so on.
- **Efficiency**: Because the infrastructure takes on more responsibility, containers can be more lightweight.

Pods provide a great solution for managing groups of closely related containers that depend on each other and need to co-operate on the same host to accomplish their purpose. It's important to remember that pods are considered ephemeral, throwaway entities that can be discarded and replaced at will. Each pod gets a **unique ID** (**UID**), so you can still distinguish between them if necessary.

Label

Labels are key-value pairs that are used to group together sets of objects, very often pods via selectors. This is important for several other concepts, such as replica sets, deployments, and services that operate on dynamic groups of objects and need to identify the members of the group. There is an NxN relationship between objects and labels. Each object may have multiple labels, and each label may be applied to different objects.

There are certain restrictions by design on labels. Each label on an object must have a unique key. The label key must adhere to a strict syntax. Note that labels are dedicated to identifying objects and are not for attaching arbitrary metadata to objects. This is what annotations are for (see the *Annotation* section).

Label selector

Label selectors are used to select objects based on their labels. Equality-based selectors specify a key name and a value. There are two operators, `=` (or `==`) and `!=`, for equality or inequality based on the value. For example:

```
role = webserver
```

This will select all objects that have that label key and value.

Label selectors can have multiple requirements separated by a comma. For example:

```
role = webserver, application != foo
```

Set-based selectors extend the capabilities and allow selection based on multiple values:

```
role in (webserver, backend)
```

Annotation

Annotations let you associate arbitrary metadata with Kubernetes objects. Kubernetes just stores the annotations and makes their metadata available. Annotation key syntax has similar requirements as label keys.

In my experience, you always need such metadata for complicated systems, and it is nice that Kubernetes recognizes this need and provides it out of the box, so you don't have to come up with your own separate metadata store and mapping object to their metadata.

Service

Services are used to expose some functionality to users or other services. They usually encompass a group of pods, usually identified by – you guessed it – a label. You can have services that provide access to external resources, or pods you control directly at the virtual IP level. Native Kubernetes services are exposed through convenient endpoints. Note that services operate at layer 3 (TCP/UDP). Kubernetes 1.2 added the Ingress object, which provides access to HTTP objects – more on that later. Services are published or discovered via one of two mechanisms: DNS, or environment variables. Services can be load-balanced inside the cluster by Kubernetes. But, developers can choose to manage load balancing themselves in case of services that use external resources or require special treatment.

There are many gory details associated with IP addresses, virtual IP addresses, and port spaces. We will discuss them in depth in *Chapter 10*, *Exploring Kubernetes Networking*.

Volume

Local storage used by the pod is ephemeral and goes away with the pod in most cases. Sometimes that's all you need, if the goal is just to exchange data between containers of the node, but sometimes it's important for the data to outlive the pod, or it's necessary to share data between pods. The volume concept supports that need. The essence of a volume is a directory with some data that is mounted into a container.

There are many volume types. Originally, Kubernetes directly supported many volume types, but the modern approach for extending Kubernetes with volume types is through the **Container Storage Interface** (**CSI**), which we'll discuss in detail in *Chapter 6*, *Managing Storage*. Most of the originally built-in volume types will have been (or are in the process of being) phased out in favor of out-of-tree plugins available through the CSI.

Replication controller and replica set

Replication controllers and replica sets both manage a group of pods identified by a label selector and ensure that a certain number is always up and running. The main difference between them is that replication controllers test for membership by name equality and replica sets can use set-based selection. Replica sets are the way to go as they are a superset of replication controllers. I expect replication controllers to be deprecated at some point. Kubernetes guarantees that you will always have the same number of pods running as you specified in a replication controller or a replica set. Whenever the number drops due to a problem with the hosting node or the pod itself, Kubernetes will fire up new instances. Note that, if you manually start pods and exceed the specified number, the replica set controller will kill some extra pods.

Replication controllers used to be central to many workflows, such as rolling updates and running one-off jobs. As Kubernetes evolved, it introduced direct support for many of these workflows, with dedicated objects such as **Deployment**, **Job**, **CronJob**, and **DaemonSet**. We will meet them all later.

StatefulSet

Pods come and go, and if you care about their data then you can use persistent storage. That's all good. But sometimes you want Kubernetes to manage a distributed data store such as **Cassandra** or **CockroachDB**. These clustered stores keep the data distributed across uniquely identified nodes. You can't model that with regular pods and services. Enter **StatefulSet**. If you remember earlier, we discussed pets versus cattle and how cattle is the way to go. Well, StatefulSet sits somewhere in the middle. StatefulSet ensures (similar to a ReplicaSet) that a given number of instances with unique identities are running at any given time. StatefulSet members have the following properties:

- A stable hostname, available in DNS
- An ordinal index
- Stable storage linked to the ordinal and hostname
- Members are created and terminated gracefully in order

StatefulSet can help with peer discovery as well as adding or removing members safely.

Secret

Secrets are small objects that contain sensitive info such as credentials and tokens. They are stored by default as plaintext in etcd, accessible by the Kubernetes API server, and can be mounted as files into pods (using dedicated secret volumes that piggyback on regular data volumes) that need access to them. The same secret can be mounted into multiple pods. Kubernetes itself creates secrets for its components, and you can create your own secrets. Another approach is to use secrets as environment variables. Note that secrets in a pod are always stored in memory (tmpfs in the case of mounted secrets) for better security. The best practice is to enable encryption at rest as well as access control with RBAC. We will discuss it in detail later.

Name

Each object in Kubernetes is identified by a UID and a name. The name is used to refer to the object in API calls. Names should be up to 253 characters long and use lowercase alphanumeric characters, dashes (-), and dots (.). If you delete an object, you can create another object with the same name as the deleted object, but the UIDs must be unique across the lifetime of the cluster. The UIDs are generated by Kubernetes, so you don't have to worry about it.

Namespace

A namespace is a form of isolation that lets you group resources and apply policies. It is also a scope for names. Objects of the same kind must have unique names within a namespace. By default, pods in one namespace can access pods and services in other namespaces.

Note that there are cluster-scope objects like node objects and persistent volumes that don't live in a namespace. Kubernetes may schedule pods from different namespaces to run on the same node. Likewise, pods from different namespaces can use the same persistent storage.

In multi-tenancy scenarios, where it's important to totally isolate namespaces, you can do a passable job with proper network policies and resource quotas to ensure proper access and distribution of the physical cluster resources. But, in general namespaces are considered a weak form of isolation and there are other solutions more appropriated for hard multi-tenancy like virtual clusters, which we will discuss in *Chapter 4*, *Securing Kubernetes*.

We've covered most of Kubernetes' primary concepts; there are a few more I mentioned briefly. In the next section, we will continue our journey into Kubernetes architecture by looking into its design motivations, the internals, and implementation, and even pick at the source code.

Diving into Kubernetes architecture in depth

Kubernetes has very ambitious goals. It aims to manage and simplify the orchestration, deployment, and management of distributed systems across a wide range of environments and cloud providers. It provides many capabilities and services that should work across all these diverse environments and use cases, while evolving and remaining simple enough for mere mortals to use. This is a tall order. Kubernetes achieves this by following a crystal-clear, high-level design and well-thought-out architecture that promotes extensibility and pluggability.

Kubernetes originally had many hard-coded or environment-aware components, but the trend is to refactor them into plugins and keep the core small, generic, and abstract.

In this section, we will peel Kubernetes like an onion, starting with various distributed systems design patterns and how Kubernetes supports them, then go over the surface of Kubernetes, which is its set of APIs, and then take a look at the actual components that comprise Kubernetes. Finally, we will take a quick tour of the source-code tree to gain even better insight into the structure of Kubernetes itself.

At the end of this section, you will have a solid understanding of Kubernetes architecture and implementation, and why certain design decisions were made.

Distributed systems design patterns

All happy (working) distributed systems are alike, to paraphrase Tolstoy in *Anna Karenina*. That means that, to function properly, all well-designed distributed systems must follow some best practices and principles. Kubernetes doesn't want to be just a management system. It wants to support and enable these best practices and provide high-level services to developers and administrators. Let's look at some of those described as design patterns. We will start with single-node patterns such as sidecar, ambassador, and adapter. Then, we will discuss multi-node patterns.

Sidecar pattern

The sidecar pattern is about co-locating another container in a pod in addition to the main application container. The application container is unaware of the sidecar container and just goes about its business. A great example is a central logging agent. Your main container can just log to stdout, but the sidecar container will send all logs to a central logging service where they will be aggregated with the logs from the entire system. The benefits of using a sidecar container versus adding central logging to the main application container are enormous. First, applications are not burdened anymore with central logging, which could be a nuisance. If you want to upgrade or change your central logging policy or switch to a totally new provider, you just need to update the sidecar container and deploy it. None of your application containers change, so you can't break them by accident. The Istio service mesh uses the sidecar pattern to inject its proxies into each pod.

Ambassador pattern

The ambassador pattern is about representing a remote service as if it were local and possibly enforcing some policy. A good example of the ambassador pattern is if you have a Redis cluster with one master for writes and many replicas for reads. A local ambassador container can serve as a proxy and expose Redis to the main application container on the localhost. The main application container simply connects to Redis on `localhost:6379` (Redis's default port), but it connects to the ambassador running in the same pod, which filters the requests, and sends write requests to the real Redis master and read requests randomly to one of the read replicas. Just like with the sidecar pattern, the main application has no idea what's going on. That can help a lot when testing against a real local Redis cluster. Also, if the Redis cluster configuration changes, only the ambassador needs to be modified; the main application remains blissfully unaware.

Adapter pattern

The adapter pattern is about standardizing output from the main application container. Consider the case of a service that is being rolled out incrementally: it may generate reports in a format that doesn't conform to the previous version. Other services and applications that consume that output haven't been upgraded yet. An adapter container can be deployed in the same pod with the new application container and massage their output to match the old version until all consumers have been upgraded. The adapter container shares the filesystem with the main application container, so it can watch the local filesystem, and whenever the new application writes something, it immediately adapts it.

Multi-node patterns

The single-node patterns described earlier are all supported directly by Kubernetes via pods scheduled on a single node. Multi-node patterns involve pods scheduled on multiple nodes. Multi-node patterns such as leader election, work queues, and scatter-gather are not supported directly, but composing pods with standard interfaces to accomplish them is a viable approach with Kubernetes.

Level-triggered infrastructure and reconciliation

Kubernetes is all about control loops. It keeps watching itself and correcting issues. Level-triggered infrastructure means that Kubernetes has a desired state, and it constantly strives toward it. For example, if a replica set has a desired state of 3 replicas and it drops to 2 replicas, Kubernetes (the **ReplicaSet** controller part of Kubernetes) will notice and work to get back to 3 replicas. The alternative approach of edge-triggering is event-based. If the number of replicas dropped from 2 to 3, create a new replica. This approach is very brittle and has many edge cases, especially in distributed systems where events like replicas coming and going can happen simultaneously.

After covering the Kubernetes architecture in depth let's study the Kubernetes APIs.

The Kubernetes APIs

If you want to understand the capabilities of a system and what it provides, you must pay a lot of attention to its API. The API provides a comprehensive view of what you can do with the system as a user. Kubernetes exposes several sets of REST APIs for different purposes and audiences via API groups. Some APIs are used primarily by tools and some can be used directly by developers. An important aspect of the APIs is that they are under constant development. The Kubernetes developers keep it manageable by trying to extend (adding new objects, and new fields to existing objects) and avoid renaming or dropping existing objects and fields. In addition, all API endpoints are versioned, and often have an alpha or beta notation too. For example:

```
/api/v1
/api/v2alpha1
```

You can access the API through the kubectl CLI, via client libraries, or directly through REST API calls. There are elaborate authentication and authorization mechanisms we will explore in *Chapter 4*, *Securing Kubernetes*. If you have the right permissions you can list, view, create, update, and delete various Kubernetes objects. At this point, let's get a glimpse into the surface area of the APIs.

The best way to explore the API is via API groups. Some API groups are enabled by default. Other groups can be enabled/disabled via flags. For example, to disable the `autoscaling/v1` group and enable the `autoscaling/v2beta2` group you can set the `--runtime-config` flag when running the API server as follows:

```
--runtime-config=autoscaling/v1=false,autoscaling/v2beta2=true
```

Note that managed Kubernetes clusters in the cloud don't let you specify flags for the API server (as they manage it).

Resource categories

In addition to API groups, another useful classification of available APIs is by functionality. The Kubernetes API is huge and breaking it down into categories helps a lot when you're trying to find your way around. Kubernetes defines the following resource categories:

- **Workloads**: Objects you use to manage and run containers on the cluster.
- **Discovery and load balancing**: Objects you use to expose your workloads to the world as externally accessible, load-balanced services.
- **Config and storage**: Objects you use to initialize and configure your applications, and to persist data that is outside the container.
- **Cluster**: Objects that define how the cluster itself is configured; these are typically used only by cluster operators.
- **Metadata**: Objects you use to configure the behavior of other resources within the cluster, such as **HorizontalPodAutoscaler** for scaling workloads.

In the following subsections, we'll list the resources that belong to each group with the API group they belong to. We will not specify the version here because APIs move rapidly from alpha to beta to **GA** (**general availability**) and from V1 to V2, and so on.

Workloads resource category

The workloads category contains the following resources with their corresponding API groups:

- **Container**: core
- **CronJob**: batch
- **ControllerRevision**: apps
- **DaemonSet**: apps
- **Deployment**: apps
- **HorizontalPodAutoscaler**: autoscaling
- **Job**: batch
- **Pod**: core
- **PodTemplate**: core
- **PriorityClass**: scheduling.k8s.io
- **ReplicaSet**: apps

- **ReplicationController**: core
- **StatefulSet**: apps

Controllers create containers within pods. Pods execute containers and offer necessary dependencies, such as shared or persistent storage volumes, as well as configuration or secret data injected into the containers.

Here is a detailed description of one of the most common operations, which gets a list of all the pods across all namespaces as a REST API:

```
GET /api/v1/pods
```

It accepts various query parameters (all optional):

- `fieldSelector`: Specifies a selector to narrow down the returned objects based on their fields. The default behavior includes all objects.
- `labelSelector`: Defines a selector to filter the returned objects based on their labels. By default, all objects are included.
- `limit`/`continue`: The `limit` parameter specifies the maximum number of responses to be returned in a list call. If there are more items available, the server sets the `continue` field in the list metadata. This value can be used with the initial query to fetch the next set of results.
- `pretty`: When set to `'true'`, the output is formatted in a human-readable manner.
- `resourceVersion`: Sets a constraint on the acceptable resource versions that can be served by the request. If not specified, it defaults to unset.
- `resourceVersionMatch`: Determines how the `resourceVersion` constraint is applied in list calls. If not specified, it defaults to unset.
- `timeoutSeconds`: Specifies a timeout duration for the list/watch call. This limits the duration of the call, regardless of any activity or inactivity.
- `watch`: Enables the monitoring of changes to the described resources and returns a continuous stream of notifications for additions, updates, and removals. The `resourceVersion` parameter must be specified.

Discovery and load balancing

Workloads in a cluster are only accessible within the cluster by default. To make them accessible externally, either a `LoadBalancer` or a `NodePort` Service needs to be used. However, for development purposes, internally accessible workloads can be accessed through the API server using the "kubectl proxy" command:

- `Endpoints`: `core`
- `EndpointSlice`: `discovery.k8s.io/v1`
- `Ingress`: `networking.k8s.io`
- `IngressClass`: `networking.k8s.io`
- `Service`: `core`

Config and storage

Dynamic configuration without redeployment and secret management are cornerstones of Kubernetes and running complex distributed applications on your Kubernetes cluster. The secret and configuration are not baked into container images and are stored in the Kubernetes state store (usually etcd). Kubernetes also provides a lot of abstractions for managing arbitrary storage. Here are some of the primary resources:

- `ConfigMap`: `core`
- `CSIDriver`: `storage.k8s.io`
- `CSINode`: `storage.k8s.io`
- `CSIStorageCapacity`: `storage.k8s.io`
- `Secret`: `core`
- `PersistentVolumeClaim`: `core`
- `StorageClass`: `storage.k8s.io`
- `Volume`: `core`
- `VolumeAttachment`: `storage.k8s.io`

Metadata

The metadata resources typically show up as sub-resources of the resources they configure. For example, a limit range is defined at the namespace level and can specify:

- The range of compute resource usage (minimum and maximum) for pods or containers within a namespace.
- The range of storage requests (minimum and maximum) per `PersistentVolumeClaim` within a namespace.
- The ratio between the resource request and limit for a specific resource within a namespace.
- The default request/limit for compute resources within a namespace, which are automatically injected into containers at runtime.

You will not interact with these objects directly most of the time. There are many metadata resources. You can find the complete list here: `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#-strong-metadata-apis-strong-`.

Cluster

The resources in the cluster category are designed for use by cluster operators as opposed to developers. There are many resources in this category as well. Here are some of the most important resources:

- `Namespace`: `core`
- `Node`: `core`
- `PersistentVolume`: `core`
- `ResourceQuota`: `core`
- `Role`: `rbac.authorization.k8s.io`

- `RoleBinding`: `rbac.authorization.k8s.io`
- `ClusterRole`: `rbac.authorization.k8s.io`
- `ClusterRoleBinding`: `rbac.authorization.k8s.io`
- `NetworkPolicy`: `networking.k8s.io`

Now that we understand how Kubernetes organizes and exposes its capabilities via API groups and resource categories, let's see how it manages the physical infrastructure and keeps it up with the state of the cluster.

Kubernetes components

A Kubernetes cluster has several control plane components used to control the cluster, as well as node components that run on each worker node. Let's get to know all these components and how they work together.

Control plane components

The control plane components can all run on one node, but in a highly available setup or a very large cluster, they may be spread across multiple nodes.

API server

The Kubernetes API server exposes the Kubernetes REST API. It can easily scale horizontally as it is stateless and stores all the data in the etcd cluster (or another data store in Kubernetes distributions like k3s). The API server is the embodiment of the Kubernetes control plane.

etcd

etcd is a highly reliable distributed data store. Kubernetes uses it to store the entire cluster state. In small, transient clusters a single instance of etcd can run on the same node with all the other control plane components. But, for more substantial clusters, it is typical to have a 3-node or even 5-node etcd cluster for redundancy and high availability.

Kube controller manager

The Kube controller manager is a collection of various managers rolled up into one binary. It contains the replica set controller, the pod controller, the service controller, the endpoints controller, and others. All these managers watch over the state of the cluster via the API, and their job is to steer the cluster into the desired state.

Cloud controller manager

When running in the cloud, Kubernetes allows cloud providers to integrate their platform for the purpose of managing nodes, routes, services, and volumes. The cloud provider code interacts with Kubernetes code. It replaces some of the functionality of the Kube controller manager. When running Kubernetes with a cloud controller manager you must set the Kube controller manager flag `--cloud-provider` to `external`. This will disable the control loops that the cloud controller manager is taking over.

The cloud controller manager was introduced in Kubernetes 1.6, and it's being used by multiple cloud providers already such as:

- GCP
- AWS
- Azure
- BaiduCloud
- Digital Ocean
- Oracle
- Linode

OK. Let's look at some code. The specific code is not that important. The goal is just to give you a taste of what Kubernetes code looks like. Kubernetes is implemented in Go. A quick note about Go to help you parse the code: the method name comes first, followed by the method's parameters in parentheses. Each parameter is a pair, consisting of a name followed by its type. Finally, the return values are specified. Go allows multiple return types. It is very common to return an error object in addition to the actual result. If everything is OK, the error object will be nil.

Here is the main interface of the `cloudprovider` package:

```
package cloudprovider

import (
    "context"
    "errors"
    "fmt"
    "strings"

    v1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/types"
    "k8s.io/client-go/informers"
    clientset "k8s.io/client-go/kubernetes"
    restclient "k8s.io/client-go/rest"
)

// Interface is an abstract, pluggable interface for cloud providers.
type Interface interface {
    Initialize(clientBuilder ControllerClientBuilder, stop <-chan struct{})
    LoadBalancer() (LoadBalancer, bool)
    Instances() (Instances, bool)
    InstancesV2() (InstancesV2, bool)
    Zones() (Zones, bool)
    Clusters() (Clusters, bool)
```

```
    Routes() (Routes, bool)
    ProviderName() string
    HasClusterID() bool
}
```

Most of the methods return other interfaces with their own method. For example, here is the `LoadBalancer` interface:

```
type LoadBalancer interface {
    GetLoadBalancer(ctx context.Context, clusterName string, service *v1.
Service) (status *v1.LoadBalancerStatus, exists bool, err error)
    GetLoadBalancerName(ctx context.Context, clusterName string, service *v1.
Service) string
    EnsureLoadBalancer(ctx context.Context, clusterName string, service *v1.
Service, nodes []*v1.Node) (*v1.LoadBalancerStatus, error)
    UpdateLoadBalancer(ctx context.Context, clusterName string, service *v1.
Service, nodes []*v1.Node) error
    EnsureLoadBalancerDeleted(ctx context.Context, clusterName string, service
*v1.Service) error
}
```

Kube scheduler

The `kube-scheduler` is responsible for scheduling pods into nodes. This is a very complicated task as it needs to consider multiple interacting factors, such as:

- Resource requirements
- Service requirements
- Hardware/software policy constraints
- Node affinity and anti-affinity specifications
- Pod affinity and anti-affinity specifications
- Taints and tolerations
- Local storage requirements
- Data locality
- Deadlines

If you need some special scheduling logic not covered by the default Kube scheduler, you can replace it with your own custom scheduler. You can also run your custom scheduler side by side with the default scheduler and have your custom scheduler schedule only a subset of the pods.

DNS

Starting with Kubernetes 1.3, a DNS service is part of the standard Kubernetes cluster. It is scheduled as a regular pod. Every service (except headless services) receives a DNS name. Pods can receive a DNS name too. This is very useful for automatic discovery.

We covered all the control plane components. Let's look at the Kubernetes components running on each node.

Node components

Nodes in the cluster need a couple of components to interact with the API server, receive workloads to execute, and update the API server regarding their status.

Proxy

The kube-proxy does low-level network housekeeping on each node. It reflects the Kubernetes services locally and can do TCP and UDP forwarding. It finds cluster IPs via environment variables or DNS.

kubelet

The kubelet is the Kubernetes representative on the node. It oversees communicating with the API server and manages the running pods. That includes the following:

- Receive pod specs
- Download pod secrets from the API server
- Mount volumes
- Run the pod's containers (via the configured container runtime)
- Report the status of the node and each pod
- Run container liveness, readiness, and startup probes

In this section, we dug into the guts of Kubernetes and explored its architecture from a very high level of vision and supported design patterns, through its APIs and the components used to control and manage the cluster. In the next section, we will take a quick look at the various runtimes that Kubernetes supports.

Kubernetes container runtimes

Kubernetes originally only supported Docker as a container runtime engine. But that is no longer the case. Kubernetes now supports any runtime that implements the CRI interface.

In this section, you'll get a closer look at the CRI and get to know some runtime engines that implement it. At the end of this section, you'll be able to make a well-informed decision about which container runtime is appropriate for your use case and under what circumstances you may switch or even combine multiple runtimes in the same system.

The Container Runtime Interface (CRI)

The CRI is a gRPC API, containing specifications/requirements and libraries for container runtimes to integrate with the kubelet on a node. In Kubernetes 1.7 the internal Docker integration in Kubernetes was replaced with a CRI-based integration. This was a big deal. It opened the door to multiple implementations that can take advantage of advances in the container world. The kubelet doesn't need to interface directly with multiple runtimes. Instead, it can talk to any CRI-compliant container runtime. The following diagram illustrates the flow:

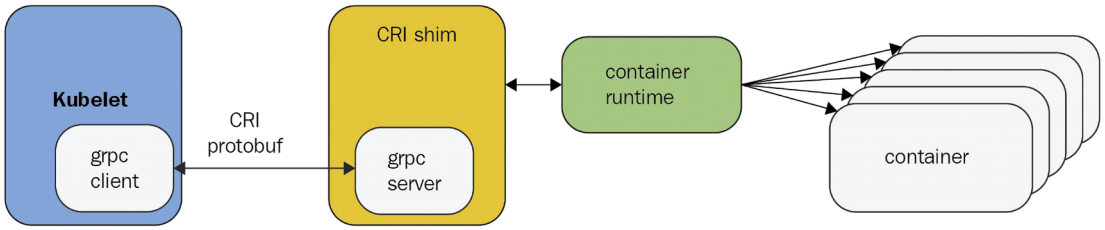


Figure 1.2: kubelet and cri

There are two gRPC service interfaces `ImageService` and `RuntimeService` that CRI container runtimes (or shims) must implement. The `ImageService` is responsible for managing images. Here is the gRPC/protobuf interface (this is not Go):

```
service ImageService {
    rpc ListImages(ListImagesRequest) returns (ListImagesResponse) {}
    rpc ImageStatus(ImageStatusRequest) returns (ImageStatusResponse) {}
    rpc PullImage(PullImageRequest) returns (PullImageResponse) {}
    rpc RemoveImage(RemoveImageRequest) returns (RemoveImageResponse) {}
    rpc ImageFsInfo(ImageFsInfoRequest) returns (ImageFsInfoResponse) {}
}
```

The `RuntimeService` is responsible for managing pods and containers. Here is the gRPC/protobuf interface:

```
service RuntimeService {
    rpc Version(VersionRequest) returns (VersionResponse) {}
    rpc RunPodSandbox(RunPodSandboxRequest) returns (RunPodSandboxResponse) {}
    rpc StopPodSandbox(StopPodSandboxRequest) returns (StopPodSandboxResponse)
{}
    rpc RemovePodSandbox(RemovePodSandboxRequest) returns
(RemovePodSandboxResponse) {}
    rpc PodSandboxStatus(PodSandboxStatusRequest) returns
(PodSandboxStatusResponse) {}
    rpc ListPodSandbox(ListPodSandboxRequest) returns (ListPodSandboxResponse)
{}
    rpc CreateContainer(CreateContainerRequest) returns
(CreateContainerResponse) {}
    rpc StartContainer(StartContainerRequest) returns (StartContainerResponse)
{}
    rpc StopContainer(StopContainerRequest) returns (StopContainerResponse) {}
    rpc RemoveContainer(RemoveContainerRequest) returns
(RemoveContainerResponse) {}
    rpc ListContainers(ListContainersRequest) returns (ListContainersResponse)
{}
```

```
    rpc ContainerStatus(ContainerStatusRequest) returns
(ContainerStatusResponse) {}
    rpc UpdateContainerResources(UpdateContainerResourcesRequest) returns
(UpdateContainerResourcesResponse) {}
    rpc ExecSync(ExecSyncRequest) returns (ExecSyncResponse) {}
    rpc Exec(ExecRequest) returns (ExecResponse) {}
    rpc Attach(AttachRequest) returns (AttachResponse) {}
    rpc PortForward(PortForwardRequest) returns (PortForwardResponse) {}
    rpc ContainerStats(ContainerStatsRequest) returns (ContainerStatsResponse)
{}
    rpc ListContainerStats(ListContainerStatsRequest) returns
(ListContainerStatsResponse) {}
    rpc UpdateRuntimeConfig(UpdateRuntimeConfigRequest) returns
(UpdateRuntimeConfigResponse) {}
    rpc Status(StatusRequest) returns (StatusResponse) {}
}
```

The data types used as arguments and return types are called messages and are also defined as part of the API. Here is one of them:

```
message CreateContainerRequest {
    string pod_sandbox_id = 1;
    ContainerConfig config = 2;
    PodSandboxConfig sandbox_config = 3;
}
```

As you can see messages can be embedded inside each other. The `CreateContainerRequest` message has one string field and two other fields, which are themselves messages: `ContainerConfig` and `PodSandboxConfig`.



To learn more about gRPC and CRI check out the following resources:

`https://grpc.io`

`https://kubernetes.io/docs/concepts/architecture/cri/`

Now that you are familiar at the code level with what Kubernetes considers a runtime engine, let's look at the individual runtime engines briefly.

Docker

Docker used to be the 800-pound gorilla of containers. Kubernetes was originally designed to only manage Docker containers. The multi-runtime capability was first introduced in Kubernetes 1.3 and the CRI in Kubernetes 1.5. Until then, Kubernetes could only manage Docker containers. Even after the CRI was introduced a Dockershim remained in the Kubernetes source code, and it was only removed in Kubernetes 1.24. Since then, Docker doesn't get any special treatment anymore.

I assume you're very familiar with Docker and what it brings to the table if you are reading this book. Docker enjoys tremendous popularity and growth, but there is also a lot of criticism towards it. Critics often mention the following concerns:

- Security
- Difficulty setting up multi-container applications (in particular, networking)
- Development, monitoring, and logging
- Limitations of Docker containers running one command
- Releasing half-baked features too fast

Docker is aware of the criticisms and has addressed some of these concerns. In particular, Docker invested in its Docker Swarm product. Docker Swarm is a Docker-native orchestration solution that competes with Kubernetes. It is simpler to use than Kubernetes, but it's not as powerful or mature.

Starting with Docker 1.11, released in April 2016, Docker has changed the way it runs containers. The runtime now uses containerd and runC to run **Open Container Initiative** (**OCI**) images in containers:

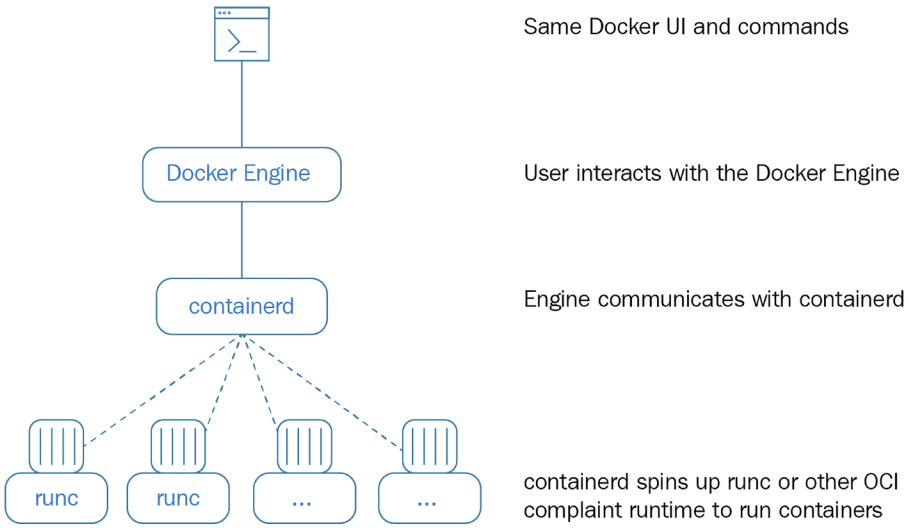


Figure 1.3: Docker and OCI

Starting with Docker 1.12, swarm mode is included in the Docker daemon natively, which upset some people due to bloat and scope creep. As a result, more people turned to other container runtimes.

In September 2021, Docker required a paid subscription to Docker Desktop for large organizations. This wasn't popular, as you might expect, and lots of organizations scrambled to find an alternative. Docker Desktop is a client-side distribution and UI for Docker and doesn't impact the container runtime. But it eroded Docker's reputation and goodwill with the community even further.

containerd

containerd has been a graduated CNCF project since 2019. It is now a mainstream option for Kubernetes containers. All major cloud providers support it and as of Kubernetes 1.24, it is the default container runtime.

In addition, the Docker container runtime is built on top of containerd as well.

CRI-O

CRI-O is a CNCF incubator project. It is designed to provide an integration path between Kubernetes and OCI-compliant container runtimes like Docker. CRI-O provides the following capabilities:

- Support for multiple image formats including the existing Docker image format
- Support for multiple means to download images including trust and image verification
- Container image management (managing image layers, overlay filesystems, etc.)
- Container process life cycle management
- Monitoring and logging required to satisfy the CRI
- Resource isolation as required by the CRI

It supports runC and Kata containers right now, but any OCI-compliant container runtime can be plugged in and integrated with Kubernetes.

Lightweight VMs

Kubernetes runs containers from different applications on the same node, sharing the same OS. This allows for running a lot of containers in a very efficient manner. However, container isolation is a serious security concern and multiple cases of privilege escalation occurred, which drove a lot of interest in a different approach. Lightweight VMs provide strong VM-level isolation, but are not as heavyweight as standard VMs, which allow them to operate as container runtimes on Kubernetes. Some prominent projects are:

- AWS Firecracker
- Google gVisor
- Kata Containers
- Singularity
- SmartOS

In this section, we covered the various runtime engines that Kubernetes supports as well as the trend toward standardization, convergence, and externalizing the runtime support from core Kubernetes.

Summary

In this chapter, we covered a lot of ground. You learned about the organization, design, and architecture of Kubernetes. Kubernetes is an orchestration platform for microservice-based applications running as containers. Kubernetes clusters have a control plane and worker nodes. Containers run within pods. Each pod runs on a single physical or virtual machine. Kubernetes directly supports many concepts, such as services, labels, and persistent storage. You can implement various distributed systems design patterns on Kubernetes. Container runtimes just need to implement the CRI. Docker, containerd, CRI-O, and more are supported.

In *Chapter 2*, *Creating Kubernetes Clusters*, we will explore the various ways to create Kubernetes clusters, discuss when to use different options, and build a local multi-node cluster.

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`



2

Creating Kubernetes Clusters

In the previous chapter, we learned what Kubernetes is all about, how it is designed, what concepts it supports, its architecture, and the various container runtimes it supports.

Creating a Kubernetes cluster from scratch is a non-trivial task. There are many options and tools to select from. There are many factors to consider. In this chapter, we will roll our sleeves up and build some Kubernetes clusters using Minikube, KinD, and k3d. We will discuss and evaluate other tools such as Kubeadm and Kubespray. We will also look into deployment environments such as local, cloud, and bare metal. The topics we will cover are as follows:

- Getting ready for your first cluster
- Creating a single-node cluster with Minikube
- Creating a multi-node cluster with KinD
- Creating a multi-node cluster using k3d
- Creating clusters in the cloud
- Creating bare-metal clusters from scratch
- Reviewing other options for creating Kubernetes clusters

At the end of this chapter, you will have a solid understanding of the various options to create Kubernetes clusters and knowledge of the best-of-breed tools to support the creation of Kubernetes clusters, and you will also build several clusters, both single-node and multi-node.

Getting ready for your first cluster

Before we start creating clusters, we should install a couple of tools such as the Docker client and kubectl. These days, the most convenient way to install Docker and kubectl on Mac and Windows is via Rancher Desktop. If you already have them installed, feel free to skip this section.

Installing Rancher Desktop

Rancher Desktop is a cross-platform desktop application that lets you run Docker on your local machine. It will install additional tools such as:

- Helm
- Kubectl
- Nerdctl
- Moby (open source Docker)
- Docker Compose

Installation on macOS

The most streamlined way to install Rancher Desktop on macOS is via Homebrew:

```
brew install --cask rancher
```

Installation on Windows

The most streamlined way to install Rancher Desktop on Windows is via Chocolatey:

```
choco install rancher-desktop
```

Additional installation methods

For alternative methods to install Docker Desktop, follow the instructions here:

`https://docs.rancherdesktop.io/getting-started/installation/`

Let's verify `docker` was installed correctly. Type the following commands and make sure you don't see any errors (the output doesn't have to be identical if you installed a different version than mine):

```
$ docker version
Client:
 Version:           20.10.9
 API version:       1.41
 Go version:        go1.16.8
 Git commit:        c2ea9bc
 Built:             Thu Nov 18 21:17:06 2021
 OS/Arch:           darwin/arm64
 Context:           rancher-desktop
 Experimental:      true

Server:
 Engine:
  Version:          20.10.14
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.17.9
```

```
 Git commit:       87a90dc786bda134c9eb02adbae2c6a7342fb7f6
 Built:            Fri Apr 15 00:05:05 2022
 OS/Arch:          linux/arm64
 Experimental:     false
containerd:
 Version:          v1.5.11
 GitCommit:        3df54a852345ae127d1fa3092b95168e4a88e2f8
runc:
 Version:          1.0.2
 GitCommit:        52b36a2dd837e8462de8e01458bf02cf9eea47dd
docker-init:
 Version:          0.19.0
 GitCommit:
```

And, while we're at it, let's verify kubectl has been installed correctly too:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"23", GitVersion:"v1.23.4",
GitCommit:"e6c093d87ea4cbb530a7b2ae91e54c0842d8308a", GitTreeState:"clean",
BuildDate:"2022-02-16T12:38:05Z", GoVersion:"go1.17.7", Compiler:"gc",
Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"23", GitVersion:"v1.23.6+k3s1",
GitCommit:"418c3fa858b69b12b9cefbcff0526f666a6236b9", GitTreeState:"clean",
BuildDate:"2022-04-28T22:16:58Z", GoVersion:"go1.17.5", Compiler:"gc",
Platform:"linux/arm64"}
```

The `Server` section may be empty if no active Kubernetes server is up and running. When you see this output, you can rest assured that kubectl is ready to go.

Meet kubectl

Before we start creating clusters, let's talk about kubectl. It is the official Kubernetes CLI, and it interacts with your Kubernetes cluster's API server via its API. It is configured by default using the `~/.kube/config` file, which is a YAML file that contains metadata, connection info, and authentication tokens or certificates for one or more clusters. Kubectl provides commands to view your configuration and switch between clusters if it contains more than one. You can also point kubectl at a different config file by setting the `KUBECONFIG` environment variable or passing the `--kubeconfig` command-line flag.

The code below uses a `kubectl` command to check the pods in the `kube-system` namespace of the current active cluster:

```
$ kubectl get pods -n kube-system
NAME                                     READY   STATUS    RESTARTS         AGE
svclb-traefik-fv84n                      2/2     Running   6 (7d20h ago)    8d
local-path-provisioner-84bb864455-s2xmp  1/1     Running   20 (7d20h ago)   27d
metrics-server-ff9dbcb6c-lsffr           0/1     Running   88 (10h ago)     27d
```

```
coredns-d76bd69b-mc6cn          1/1     Running   11 (22h ago)     8d
traefik-df4ff85d6-2fskv         1/1     Running   7 (3d ago)       8d
```

Kubectl is great, but it is not the only game in town. Let's look at some alternative tools.

Kubectl alternatives — K9S, KUI, and Lens

Kubectl is a no-nonsense command-line tool. It is very powerful, but it may be difficult or less convenient for some people to visually parse its output or remember all the flags and options. There are many tools the community developed that can replace (or more like complement) kubectl. The best ones, in my opinion, are K9S, KUI, and Lens.

K9S

K9S is a terminal-based UI for managing Kubernetes clusters. It has a lot of shortcuts and aggregated views that will require multiple kubectl commands to accomplish.

Here is what the K9S window looks like:



Figure 2.1: K9S window

Check it out here: `https://k9scli.io`

KUI

KUI is a framework for adding graphics to **CLIs** (**command-line interfaces**). This is a very interesting concept. KUI is focused on Kubernetes of course. It lets you run Kubectl commands and returns the results as graphics. KUI also collects a lot of relevant information and presents it in a concise way with tabs and detail panes to explore even deeper.

KUI is based on Electron, but it is fast.

Here is what the KUI window looks like:

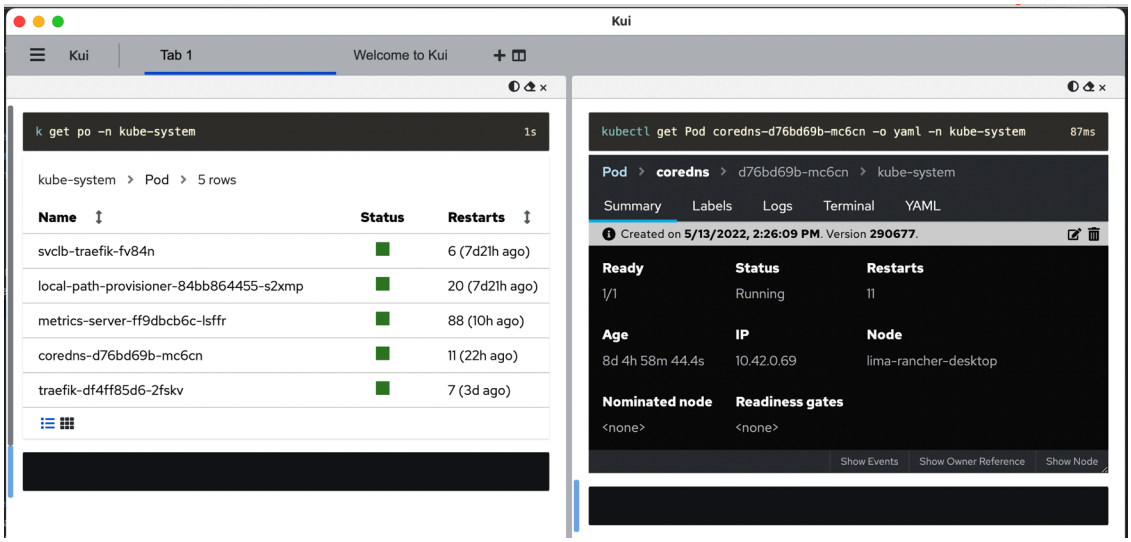


Figure 2.2: KUI window

Check it out here: `https://kui.tools`

Lens

Lens is a very polished application. It also presents a graphical view of clusters and allows you to perform a lot of operations from the UI and drop to a terminal interface when necessary. I especially appreciate the ability to work easily with multiple clusters that Lens provides.

Here is what the Lens window looks like:

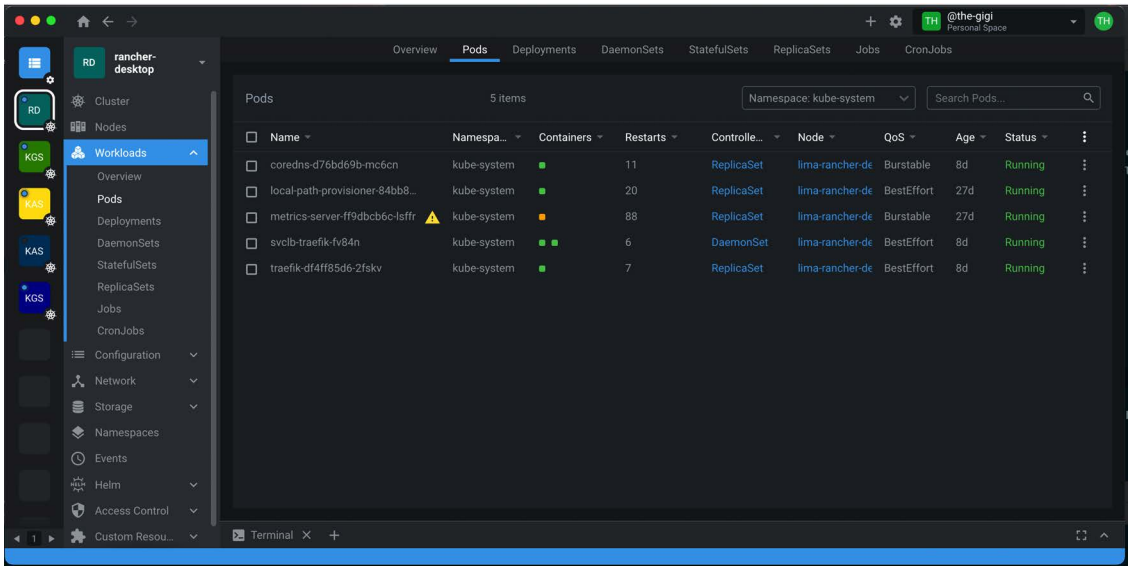


Figure 2.3: Lens window

Check it out here: `https://k8slens.dev`

All these tools are running locally. I highly recommend that you start playing with kubectl and then give these tools a test drive. One of them may just be your speed.

In this section, we covered the installation of Rancher Desktop, introduced kubectl, and looked at some alternatives. We are now ready to create our first Kubernetes cluster.

Creating a single-node cluster with Minikube

In this section, we will create a local single-node cluster using Minikube. Local clusters are most useful for developers that want quick edit-test-deploy-debug cycles on their machine before committing their changes. Local clusters are also very useful for DevOps and operators that want to play with Kubernetes locally without concerns about breaking a shared environment or creating expensive resources in the cloud and forgetting to clean them up. While Kubernetes is typically deployed on Linux in production, many developers work on Windows PCs or Macs. That said, there aren't too many differences if you do want to install Minikube on Linux.



Figure 2.4: minikube

Quick introduction to Minikube

Minikube is the most mature local Kubernetes cluster. It runs the latest stable Kubernetes release. It supports Windows, macOS, and Linux. Minikube provides a lot of advanced options and capabilities:

- LoadBalancer service type - via minikube tunnel
- NodePort service type - via minikube service
- Multiple clusters
- Filesystem mounts
- GPU support - for machine learning
- RBAC
- Persistent Volumes
- Ingress
- Dashboard - via minikube dashboard
- Custom container runtimes - via the `start --container-runtime` flag
- Configuring API server and kubelet options via command-line flags
- Addons

Installing Minikube

The ultimate guide is here: `https://minikube.sigs.k8s.io/docs/start/`

But, to save you a trip, here are the latest instructions at the time of writing.

Installing Minikube on Windows

On Windows, I prefer to install software via the Chocolatey package manager. If you don't have it yet, you can get it here: `https://chocolatey.org/`

If you don't want to use Chocolatey, check the ultimate guide above for alternative methods.

With Chocolatey installed, the installation is pretty simple:

```
PS C:\Windows\system32> choco install minikube -y
Chocolatey v0.12.1
Installing the following packages:
minikube
By installing, you accept licenses for the packages.
Progress: Downloading Minikube 1.25.2... 100%

kubernetes-cli v1.24.0 [Approved]
kubernetes-cli package files install completed. Performing other installation steps.
Extracting 64-bit C:\ProgramData\chocolatey\lib\kubernetes-cli\tools\kubernetes-
client-windows-amd64.tar.gz to C:\ProgramData\chocolatey\lib\kubernetes-cli\tools...
C:\ProgramData\chocolatey\lib\kubernetes-cli\tools
```

```
Extracting 64-bit C:\ProgramData\chocolatey\lib\kubernetes-cli\tools\kubernetes-
client-windows-amd64.tar to C:\ProgramData\chocolatey\lib\kubernetes-cli\tools...
C:\ProgramData\chocolatey\lib\kubernetes-cli\tools
 ShimGen has successfully created a shim for kubectl-convert.exe
 ShimGen has successfully created a shim for kubectl.exe
 The install of kubernetes-cli was successful.
  Software installed to 'C:\ProgramData\chocolatey\lib\kubernetes-cli\tools'

Minikube v1.25.2 [Approved]
minikube package files install completed. Performing other installation steps.
 ShimGen has successfully created a shim for minikube.exe
 The install of minikube was successful.
  Software installed to 'C:\ProgramData\chocolatey\lib\Minikube'

Chocolatey installed 2/2 packages.
 See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).
```

On Windows, you can work in different command-line environments. The most common ones are PowerShell and **WSL** (**Windows System for Linux**). Either one works. You may need to run them in Administrator mode for certain operations.

As far as console windows go, I recommend the official Windows Terminal these days. You can install it with one command:

```
choco install microsoft-windows-terminal --pre
```

If you prefer other console windows such as ConEMU or Cmdr, this is totally fine.

I'll use shortcuts to make life easy. If you want to follow along and copy the aliases into your profile, you can use the following for PowerShell and WSL.

For PowerShell, add the following to your `$profile`:

```
function k { kubectl.exe $args } function mk { minikube.exe $args }
```

For WSL, add the following to `.bashrc`:

```
alias k='kubectl.exe'
alias mk=minikube.exe'
```

Let's verify that minikube was installed correctly:

```
$ mk version
minikube version: v1.25.2
commit: 362d5fdc0a3dbee389b3d3f1034e8023e72bd3a7
```

Let's create a cluster with `mk start`:

```
$ mk start
```

```
😄  minikube v1.25.2 on Microsoft Windows 10 Pro 10.0.19044 Build 19044
✨  Automatically selected the docker driver. Other choices: hyperv, ssh
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
💾  Downloading Kubernetes v1.23.3 preload ...
    > preloaded-images-k8s-v17-v1...: 505.68 MiB / 505.68 MiB  100.00% 3.58 MiB
    > gcr.io/k8s-minikube/kicbase: 379.06 MiB / 379.06 MiB  100.00% 2.61 MiB p/
🔥  Creating docker container (CPUs=2, Memory=8100MB) ...
🤦  docker "minikube" container is missing, will recreate.
🔥  Creating docker container (CPUs=2, Memory=8100MB) ...
💿  Downloading VM boot image ...
    > minikube-v1.25.2.iso.sha256: 65 B / 65 B [-------------] 100.00% ? p/s 0s
    > minikube-v1.25.2.iso: 237.06 MiB / 237.06 MiB [ 100.00% 12.51 MiB p/s 19s
👍  Starting control plane node minikube in cluster minikube
🔥  Creating hyperv VM (CPUs=2, Memory=6000MB, Disk=20000MB) ...
❗  This VM is having trouble accessing https://k8s.gcr.io
💡  To pull new external images, you may need to configure a proxy: https://minikube.
sigs.k8s.io/docs/reference/networki
ng/proxy/
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default"
namespace by default
```

As you can see, the process is pretty complicated even for the default setup, and required multiple retries (automatically). You can customize the cluster creation process with a multitude of command-line flags. Type `mk start -h` to see what's available.

Let's check the status of our cluster:

```
$ mk status
minikube
type: Control Plane
host: Running
```

```
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

All is well!

Now let's stop the cluster and later restart it:

```
$ mk stop
✋ Stopping node "minikube" ...
🛑 Powering off "minikube" via SSH ...
🛑 1 node stopped.
```

Restarting with the time command to measure how long it takes:

```
$ time mk start
😄  minikube v1.25.2 on Microsoft Windows 10 Pro 10.0.19044 Build 19044
✨  Using the hyperv driver based on existing profile
👍  Starting control plane node minikube in cluster minikube
🔄  Restarting existing hyperv VM for "minikube" ...
❗  This VM is having trouble accessing https://k8s.gcr.io
💡  To pull new external images, you may need to configure a proxy: https://minikube.
sigs.k8s.io/docs/reference/networki
ng/proxy/
🐳  Preparing Kubernetes v1.23.3 on Docker 20.10.12 ...
    ▪ kubelet.housekeeping-interval=5m
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default"
namespace by default

real    1m8.666s
user    0m0.004s
sys     0m0.000s
```

It took a little over a minute.

Let's review what Minikube did behind the curtains for you. You'll need to do a lot of it when creating a cluster from scratch:

1. Started a Hyper-V VM
2. Created certificates for the local machine and the VM
3. Downloaded images

4. Set up networking between the local machine and the VM
5. Ran the local Kubernetes cluster on the VM
6. Configured the cluster
7. Started all the Kubernetes control plane components
8. Configured the kubelet
9. Enabled addons (for storage)
10. Configured kubectl to talk to the cluster

Installing Minikube on macOS

On Mac, I recommend installing minikube using Homebrew:

```
$ brew install minikube
Running `brew update --preinstall`...
==> Auto-updated Homebrew!
Updated 2 taps (homebrew/core and homebrew/cask).
==> Updated Formulae
Updated 39 formulae.
==> New Casks
contour                                                    hdfview
rancher-desktop | kube-system
==> Updated Casks
Updated 17 casks.

==> Downloading https://ghcr.io/v2/homebrew/core/kubernetes-cli/manifests/1.24.0
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/kubernetes-cli/blobs/
sha256:e57f8f7ea19d22748d1bcae5cd02b91e71816147712e6dcd
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/
sha256:e57f8f7ea19d22748d1bcae5cd02b91e71816147
######################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/minikube/manifests/1.25.2
Already downloaded: /Users/gigi.sayfan/Library/Caches/Homebrew/downloads/fa0034af
e1330adad087a8b3dc9ac4917982d248b08a4df4cbc52ce01d5eabff--minikube-1.25.2.bottle_
manifest.json
==> Downloading https://ghcr.io/v2/homebrew/core/minikube/blobs/
sha256:6dee5f22e08636346258f4a6daa646e9102e384ceb63f33981745d
Already downloaded: /Users/gigi.sayfan/Library/Caches/Homebrew/downloads/ceeab562
206fd08fd3b6523a85b246d48d804b2cd678d76cbae4968d97b5df1f--minikube--1.25.2.arm64_
monterey.bottle.tar.gz
==> Installing dependencies for minikube: kubernetes-cli
==> Installing minikube dependency: kubernetes-cli
==> Pouring kubernetes-cli--1.24.0.arm64_monterey.bottle.tar.gz
```

```
🍺  /opt/homebrew/Cellar/kubernetes-cli/1.24.0: 228 files, 55.3MB
==> Installing minikube
==> Pouring minikube--1.25.2.arm64_monterey.bottle.tar.gz
==> Caveats
zsh completions have been installed to:
  /opt/homebrew/share/zsh/site-functions
==> Summary
🍺  /opt/homebrew/Cellar/minikube/1.25.2: 9 files, 70.3MB
==> Running `brew cleanup minikube`...
Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).
==> Caveats
==> minikube
zsh completions have been installed to:
  /opt/homebrew/share/zsh/site-functions
```

You can add aliases to your `.bashrc` file (similar to the WSL aliases on Windows):

```
alias k='kubectl'
alias mk='$(brew --prefix)/bin/minikube'
```

Now you can use `k` and `mk` and type less.

Type `mk version` to verify Minikube is correctly installed and functioning:

```
$ mk version
minikube version: v1.25.2
commit: 362d5fdc0a3dbee389b3d3f1034e8023e72bd3a7
```

Type `k version` to verify kubectl is correctly installed and functioning:

```
$ k version
I0522 15:41:13.663004   68055 versioner.go:58] invalid configuration: no
configuration has been provided
Client Version: version.Info{Major:"1", Minor:"23", GitVersion:"v1.23.4",
GitCommit:"e6c093d87ea4cbb530a7b2ae91e54c0842d8308a", GitTreeState:"clean",
BuildDate:"2022-02-16T12:38:05Z", GoVersion:"go1.17.7", Compiler:"gc",
Platform:"darwin/amd64"}
The connection to the server localhost:8080 was refused - did you specify the right
host or port?
```

Note that the client version is 1.23. Don't worry about the error message. There is no cluster running, so kubectl can't connect to anything. That's expected. The error message will disappear when we create the cluster.

You can explore the available commands and flags for both Minikube and kubectl by just typing the commands with no arguments.

To create the cluster on macOS, just run `mk start`.

Troubleshooting the Minikube installation

If something goes wrong during the process, try to follow the error messages. You can add the `--alsologtostderr` flag to get detailed error info to the console. Everything minikube does is organized neatly under `~/.minikube`. Here is the directory structure:

```
$ tree ~/.minikube\ -L 2
C:\Users\the_g\.minikube\
|-- addons
|-- ca.crt
|-- ca.key
|-- ca.pem
|-- cache
|   |-- iso
|   |-- kic
|   `-- preloaded-tarball
|-- cert.pem
|-- certs
|   |-- ca-key.pem
|   |-- ca.pem
|   |-- cert.pem
|   `-- key.pem
|-- config
|-- files
|-- key.pem
|-- logs
|   |-- audit.json
|   `-- lastStart.txt
|-- machine_client.lock
|-- machines
|   |-- minikube
|   |-- server-key.pem
|   `-- server.pem
|-- profiles
|   `-- minikube
|-- proxy-client-ca.crt
`-- proxy-client-ca.key

13 directories, 16 files
```

If you don't have the tree utility, you can install it.

On Windows: `$ choco install -y tree`

On Mac: `brew install tree`

Checking out the cluster

Now that we have a cluster up and running, let's peek inside.

First, let's `ssh` into the VM:

```
$ mk ssh
```

```
                       _             _
            _         _ ( )           ( )
  ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)
```

```
$ uname -a
Linux minikube 4.19.202 #1 SMP Tue Feb 8 19:13:02 UTC 2022 x86_64 GNU/Linux
$
```

Great! That works. The weird symbols are ASCII art for "minikube." Now, let's start using kubectl because it is the Swiss Army knife of Kubernetes and will be useful for all clusters.

Disconnect from the VM via *ctrl+D* or by typing:

```
$ logout
```

We will cover many of the kubectl commands in our journey. First, let's check the cluster status using `cluster-info`:

```
$ k cluster-info
Kubernetes control plane is running at https://172.26.246.89:8443
CoreDNS is running at https://172.26.246.89:8443/api/v1/namespaces/kube-system/
services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

You can see that the control plane is running properly. To see a much more detailed view of all the objects in the cluster as JSON, type: `k cluster-info dump`. The output can be a little daunting let's use more specific commands to explore the cluster.

Let's check out the nodes in the cluster using `get nodes`:

```
$ k get nodes
NAME       STATUS   ROLES                  AGE   VERSION
minikube   Ready    control-plane,master   62m   v1.23.3
```

So, we have one node called minikube. To get a lot more information about it, type:

```
k describe node minikube
```

The output is verbose; I'll let you try it yourself.

Before we start putting our cluster to work, let's check the addons minikube installed by default:

```
 mk addons list
|-----------------------------|----------|-------------|--------------------------
-----|
|         ADDON NAME          | PROFILE  |    STATUS   |           MAINTAINER
|
|-----------------------------|----------|-------------|--------------------------
-----|
| ambassador                  | minikube | disabled    | third-party (ambassador)
|
| auto-pause                  | minikube | disabled    | google
|
| csi-hostpath-driver         | minikube | disabled    | kubernetes
|
| dashboard                   | minikube | disabled    | kubernetes
|
| default-storageclass        | minikube | enabled ✅  | kubernetes
|
| efk                         | minikube | disabled    | third-party (elastic)
|
| freshpod                    | minikube | disabled    | google
|
| gcp-auth                    | minikube | disabled    | google
|
| gvisor                      | minikube | disabled    | google
|
| helm-tiller                 | minikube | disabled    | third-party (helm)
|
| ingress                     | minikube | disabled    | unknown (third-party)
|
| ingress-dns                 | minikube | disabled    | google
|
| istio                       | minikube | disabled    | third-party (istio)
|
| istio-provisioner           | minikube | disabled    | third-party (istio)
|
| kong                        | minikube | disabled    | third-party (Kong HQ)
|
```

```
| kubevirt                    | minikube | disabled     | third-party (kubevirt)
|
| logviewer                   | minikube | disabled     | unknown (third-party)
|
| metallb                     | minikube | disabled     | third-party (metallb)
|
| metrics-server              | minikube | disabled     | kubernetes
|
| nvidia-driver-installer     | minikube | disabled     | google
|
| nvidia-gpu-device-plugin    | minikube | disabled     | third-party (nvidia)
|
| olm                         | minikube | disabled     | third-party (operator
|
|                             |          |              | framework)
|
| pod-security-policy         | minikube | disabled     | unknown (third-party)
|
| portainer                   | minikube | disabled     | portainer.io
|
| registry                    | minikube | disabled     | google
|
| registry-aliases            | minikube | disabled     | unknown (third-party)
|
| registry-creds              | minikube | disabled     | third-party (upmc
enterprises) |
| storage-provisioner         | minikube | enabled ✅   | google
|
| storage-provisioner-gluster | minikube | disabled     | unknown (third-party)
|
| volumesnapshots             | minikube | disabled     | kubernetes
|
|-----------------------------|----------|--------------|-------------------------
-----|
```

As you can see, minikube comes loaded with a lot of addons, but only enables a couple of storage addons out of the box.

Doing work

Before we start, if you have a VPN running, you may need to shut it down when pulling images.

We have a nice empty cluster up and running (well, not completely empty, as the DNS service and dashboard run as pods in the kube-system namespace). It's time to deploy some pods:

```
$ k create deployment echo --image=k8s.gcr.io/e2e-test-images/echoserver:2.5
deployment.apps/echo created
```

Let's check out the pod that was created. The `-w` flag means watch. Whenever the status changes, a new line will be displayed:

```
$ k get po -w
NAME                    READY   STATUS              RESTARTS   AGE
echo-7fd7648898-6hh48   0/1     ContainerCreating   0          5s
echo-7fd7648898-6hh48   1/1     Running             0          6s
```

To expose our pod as a service, type the following:

```
$ k expose deployment echo --type=NodePort --port=8080
service/echo exposed
```

Exposing the service as type `NodePort` means that it is exposed to the host on some port. But it is not the `8080` port we ran the pod on. Ports get mapped in the cluster. To access the service, we need the cluster IP and exposed port:

```
$ mk ip
172.26.246.89

$  k get service echo -o jsonpath='{.spec.ports[0].nodePort}'
32649
```

Now we can access the echo service, which returns a lot of information:

```
n$ curl http://172.26.246.89:32649/hi


Hostname: echo-7fd7648898-6hh48

Pod Information:
        -no pod information available-

Server values:
        server_version=nginx: 1.14.2 - lua: 10015

Request Information:
        client_address=172.17.0.1
        method=GET
        real path=/hi
        query=
        request_version=1.1
        request_scheme=http
```

```
        request_uri=http://172.26.246.89:8080/hi

Request Headers:
        accept=*/*
        host=172.26.246.89:32649
        user-agent=curl/7.79.1

Request Body:
        -no body in request-
```

Congratulations! You just created a local Kubernetes cluster, deployed a service, and exposed it to the world.

Examining the cluster with the dashboard

Kubernetes has a very nice web interface, which is deployed, of course, as a service in a pod. The dashboard is well designed and provides a high-level overview of your cluster as well as drilling down into individual resources, viewing logs, editing resource files, and more. It is the perfect weapon when you want to check out your cluster manually and don't have local tools like KUI or Lens. Minikube provides it as an addon.

Let's enable it:

```
$ mk addons enable dashboard
    ▪ Using image kubernetesui/dashboard:v2.3.1
    ▪ Using image kubernetesui/metrics-scraper:v1.0.7
💡  Some dashboard features require the metrics-server addon. To enable all features please run:

        minikube addons enable metrics-server


🌟  The 'dashboard' addon is enabled
```

To launch it, type:

```
$ mk dashboard
🤔  Verifying dashboard health ...
🚀  Launching proxy ...
🤔  Verifying proxy health ...
🎉  Opening http://127.0.0.1:63200/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/ in your default browser...
```

Minikube will open a browser window with the dashboard UI.

Here is the **Workloads** view, which displays **Deployments**, **Replica Sets**, and **Pods**.

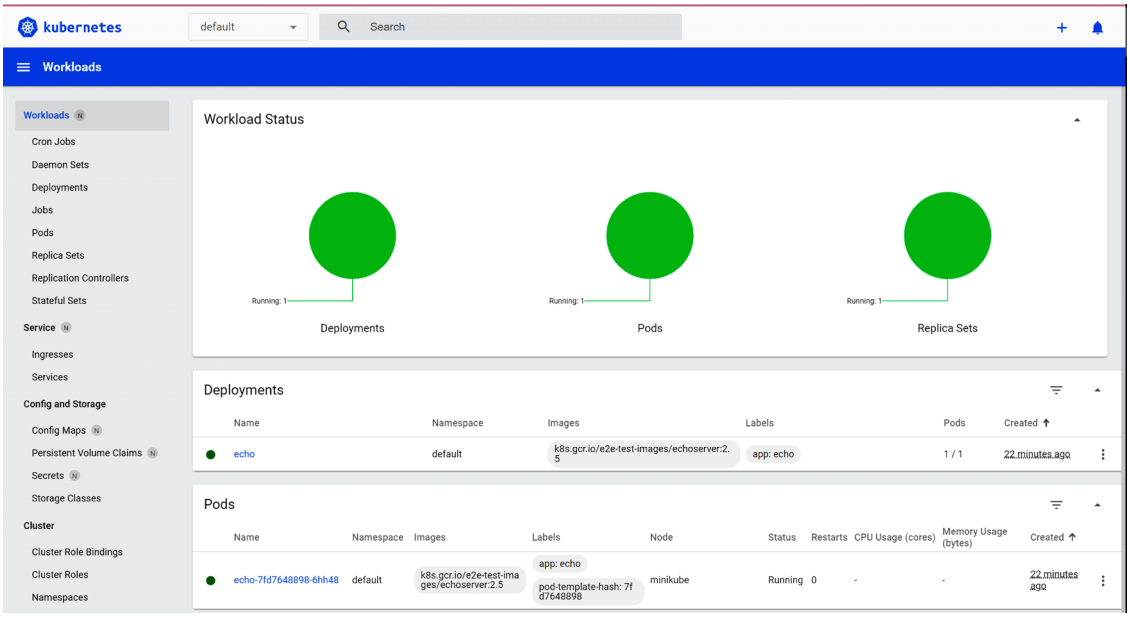


Figure 2.5: Workloads dashboard

It can also display daemon sets, stateful sets, and jobs, but we don't have any in this cluster.

To delete the cluster we created, type:

```
$ mk delete
🔥  Deleting "minikube" in docker ...
🔥  Deleting container "minikube" ...
🔥  Removing /Users/gigi.sayfan/.minikube/machines/minikube ...
💀  Removed all traces of the "minikube" cluster.
```

In this section, we created a local single-node Kubernetes cluster on Windows, explored it a little bit using kubectl, deployed a service, and played with the web UI. In the next section, we'll move to a multi-node cluster.

Creating a multi-node cluster with KinD

In this section, we'll create a multi-node cluster using KinD. We will also repeat the deployment of the echo server we deployed on Minikube and observe the differences. Spoiler alert - everything will be faster and easier!

Quick introduction to KinD

KinD stands for **Kubernetes in Docker**. It is a tool for creating ephemeral clusters (no persistent storage). It was built primarily for running the Kubernetes conformance tests. It supports Kubernetes 1.11+. Under the covers, it uses `kubeadm` to bootstrap Docker containers as nodes in the cluster. KinD is a combination of a library and a CLI. You can use the library in your code for testing or other purposes. KinD can create highly-available clusters with multiple control plane nodes. Finally, KinD is a CNCF conformant Kubernetes installer. It had better be if it's used for the conformance tests of Kubernetes itself.

KinD is super fast to start, but it has some limitations too:

- No persistent storage
- No support for alternative runtimes yet, only Docker

Let's install KinD and get going.

Installing KinD

You must have Docker installed as KinD is literally running as a Docker container. If you have Go installed, you can install the KinD CLI via:

```
go install sigs.k8s.io/kind@v0.14.0
```

Otherwise, on macOS type:

```
brew install kind
```

On Windows type:

```
choco install kind
```

Dealing with Docker contexts

You may have multiple Docker engines on your system and the Docker context determines which one is used. You may get an error like:

```
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker
daemon running?
```

In this case, check your Docker contexts:

```
$ docker context ls
NAME              DESCRIPTION                               DOCKER ENDPOINT
KUBERNETES ENDPOINT                ORCHESTRATOR
colima            colima                                    unix:///Users/gigi.
sayfan/.colima/docker.sock
default *         Current DOCKER_HOST based configuration   unix:///var/run/docker.
sock                       https://127.0.0.1:6443 (default)   swarm
rancher-desktop   Rancher Desktop moby context              unix:///Users/gigi.
sayfan/.rd/docker.sock       https://127.0.0.1:6443 (default)
```

The context marked with * is the current context. If you use Rancher Desktop, then you should set the context to `rancher-desktop`:

```
$ docker context use rancher-desktop
```

Creating a cluster with KinD

Creating a cluster is super easy.

```
$ kind create cluster
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.23.4) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Thanks for using kind! 😊
```

It takes less than 30 seconds to create a single-node cluster.

Now, we can access the cluster using kubectl:

```
$ k config current-context
kind-kind

$ k cluster-info
Kubernetes control plane is running at https://127.0.0.1:51561
CoreDNS is running at https://127.0.0.1:51561/api/v1/namespaces/kube-system/services/
kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

KinD adds its kube context to the default `~/.kube/config` file by default. When creating a lot of temporary clusters, it is sometimes better to store the KinD contexts in separate files and avoid cluttering `~/.kube/config`. This is easily done by passing the `--kubeconfig` flag with a file path.

So, KinD creates a single-node cluster by default:

```
$ k get no
NAME                 STATUS   ROLES                  AGE   VERSION
kind-control-plane   Ready    control-plane,master   4m    v1.23.4
```

Let's delete it and create a multi-node cluster:

```
$ kind delete cluster
Deleting cluster "kind" ...
```

To create a multi-node cluster, we need to provide a configuration file with the specification of our nodes. Here is a configuration file that will create a cluster called `multi-node-cluster` with one control plane node and two worker nodes:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: multi-node-cluster
nodes:
- role: control-plane
- role: worker
- role: worker
```

Let's save the configuration file as `kind-multi-node-config.yaml` and create the cluster storing the kubeconfig with its own file `$TMPDIR/kind-multi-node-config`:

```
$ kind create cluster --config kind-multi-node-config.yaml --kubeconfig $TMPDIR/kind-
multi-node-config
Creating cluster "multi-node-cluster" ...
 ✓ Ensuring node image (kindest/node:v1.23.4)
 ✓ Preparing nodes
 ✓ Writing configuration
 ✓ Starting control-plane
 ✓ Installing CNI
 ✓ Installing StorageClass
 ✓ Joining worker nodes
Set kubectl context to "kind-multi-node-cluster"
You can now use your cluster with:

kubectl cluster-info --context kind-multi-node-cluster --kubeconfig /var/folders/
qv/7l781jhs6j19gw3b89f4fcz40000gq/T//kind-multi-node-config

Have a nice day!
```

Yeah, it works! And we got a local 3-node cluster in less than a minute:

```
$ k get nodes --kubeconfig $TMPDIR/kind-multi-node-config
NAME                                STATUS   ROLES                  AGE     VERSION
multi-node-cluster-control-plane    Ready    control-plane,master   2m17s   v1.23.4
multi-node-cluster-worker           Ready    <none>                 100s    v1.23.4
multi-node-cluster-worker2          Ready    <none>                 100s    v1.23.4
```

KinD is also kind enough (see what I did there) to let us create **HA** (**highly available**) clusters with multiple control plane nodes for redundancy. If you want a highly available cluster with three control plane nodes and two worker nodes, your cluster config file will be very similar:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: ha-multi-node-cluster
nodes:
- role: control-plane
- role: control-plane
- role: control-plane
- role: worker
- role: worker
```

Let's save the configuration file as `kind-ha-multi-node-config.yaml` and create a new HA cluster:

```
$ kind create cluster --config kind-ha-multi-node-config.yaml --kubeconfig $TMPDIR/
kind-ha-multi-node-config
Creating cluster "ha-multi-node-cluster" ...
 ✓ Ensuring node image (kindest/node:v1.23.4)
 ✓ Preparing nodes
 ✓ Configuring the external load balancer
 ✓ Writing configuration
 ✓ Starting control-plane
 ✓ Installing CNI
 ✓ Installing StorageClass
 ✓ Joining more control-plane nodes
 ✓ Joining worker nodes
Set kubectl context to "kind-ha-multi-node-cluster"
You can now use your cluster with:

kubectl cluster-info --context kind-ha-multi-node-cluster --kubeconfig /var/folders/
qv/7l781jhs6j19gw3b89f4fcz40000gq/T//kind-ha-multi-node-config
```

```
Not sure what to do next? 😅 Check out https://kind.sigs.k8s.io/docs/user/quick-start/
```

Hmmm... there is something new here. Now KinD creates an external load balancer as well as joining more control plane nodes before joining the worker nodes. The load balancer is necessary to distribute requests across all the control plane nodes.

Note that the external load balancer doesn't show as a node using kubectl:

```
$ k get nodes --kubeconfig $TMPDIR/kind-ha-multi-node-config
NAME                                      STATUS   ROLES                  AGE
VERSION
ha-multi-node-cluster-control-plane       Ready    control-plane,master   3m31s
v1.23.4
ha-multi-node-cluster-control-plane2      Ready    control-plane,master   3m19s
v1.23.4
ha-multi-node-cluster-control-plane3      Ready    control-plane,master   2m22s
v1.23.4
ha-multi-node-cluster-worker              Ready    <none>                 2m4s
v1.23.4
ha-multi-node-cluster-worker2             Ready    <none>                 2m5s
v1.23.4
```

But, KinD has its own `get nodes` command, where you can see the load balancer:

```
$ kind get nodes --name ha-multi-node-cluster
ha-multi-node-cluster-control-plane2
ha-multi-node-cluster-external-load-balancer
ha-multi-node-cluster-control-plane
ha-multi-node-cluster-control-plane3
ha-multi-node-cluster-worker
ha-multi-node-cluster-worker2
```

Our KinD cluster is up and running; let's put it to work.

Doing work with KinD

Let's deploy our echo service on the KinD cluster. It starts the same:

```
$ k create deployment echo --image=g1g1/echo-server:0.1 --kubeconfig $TMPDIR/kind-ha-multi-node-config
deployment.apps/echo created

$ k expose deployment echo --type=NodePort --port=7070 --kubeconfig $TMPDIR/kind-ha-multi-node-config
service/echo exposed
```

Checking our services, we can see the echo service front and center:

```
$ k get svc echo --kubeconfig $TMPDIR/kind-ha-multi-node-config
NAME   TYPE       CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
echo   NodePort   10.96.52.33   <none>        7070:31953/TCP   10s
```

But, there is no external IP to the service. With minikube, we got the IP of the minikube node itself via `$(minikube ip)` and we could use it in combination with the node port to access the service. That is not an option with KinD clusters. Let's see how to use a proxy to access the echo service.

Accessing Kubernetes services locally through a proxy

We will go into a lot of detail about networking, services, and how to expose them outside the cluster later in the book.

Here, we will just show you how to get it done and keep you in suspense for now. First, we need to run the `kubectl proxy` command that exposes the API server, pods, and services on localhost:

```
$ k proxy --kubeconfig $TMPDIR/kind-ha-multi-node-config &
[1] 32479
Starting to serve on 127.0.0.1:8001
```

Then, we can access the echo service through a specially crafted proxy URL that includes the exposed port (`8080`) and NOT the node port:

```
$ http http://localhost:8001/api/v1/namespaces/default/services/echo:7070/proxy/yeah-it-works
HTTP/1.1 200 OK
Audit-Id: 294cf10b-0d60-467d-8a51-4414834fc173
Cache-Control: no-cache, private
Content-Length: 13
Content-Type: text/plain; charset=utf-8
Date: Mon, 23 May 2022 21:54:01 GMT

yeah-it-works
```

I used httpie in the command above. You can use curl too. To install httpie, follow the instructions here: `https://httpie.org/doc#installation`.

We will deep dive into exactly what's going on in *Chapter 10*, *Exploring Kubernetes Networking*. For now, it is enough to demonstrate how kubectl proxy allows us to access our KinD services.

Let's check out my favorite local cluster solution – k3d.

Creating a multi-node cluster with k3d

In this section, we'll create a multi-node cluster using k3d from Rancher. We will not repeat the deployment of the echo server because it's identical to the KinD cluster including accessing it through a proxy. Spoiler alert – creating clusters with k3d is even faster and more user-friendly than KinD!

Quick introduction to k3s and k3d

Rancher created k3s, which is a lightweight Kubernetes distribution. Rancher says that k3s is 5 less than k8s if that makes any sense. The basic idea is to remove features and capabilities that most people don't need such as:

- Non-default features
- Legacy features
- Alpha features
- In-tree storage drivers
- In-tree cloud providers

K3s removed Docker completely and uses containerd instead. You can still bring Docker back if you depend on it. Another major change is that k3s stores its state in an SQLite DB instead of etcd. For networking and DNS, k3s uses Flannel and CoreDNS.

K3s also added a simplified installer that takes care of SSL and certificate provisioning.

The end result is astonishing – a single binary (less than 40MB) that needs only 512MB of memory.

Unlike Minikube and KinD, k3s is actually designed for production. The primary use case is for edge computing, IoT, and CI systems. It is optimized for ARM devices.

OK. That's k3s, but what's k3d? K3d takes all the goodness that is k3s and packages it in Docker (similar to KinD) and adds a friendly CLI to manage it.

Let's install k3d and see for ourselves.

Installing k3d

Installing k3d on macOS is as simple as:

```
brew install k3d
```

And on Windows, it is just:

```
choco install -y k3d
```

On Windows, optionally add this alias to your WSL `.bashrc` file:

```
alias k3d='k3d.exe'
```

Let's see what we have:

```
$ k3d version
k3d version v5.4.1
k3s version v1.22.7-k3s1 (default)
```

As you see, k3d reports its version, which shows all is well. Now, we can create a cluster with k3d.

Creating the cluster with k3d

Are you ready to be amazed? Creating a single-node cluster with k3d takes less than 20 seconds!

```
$ time k3d cluster create
INFO[0000] Prep: Network
INFO[0000] Created network 'k3d-k3s-default'
INFO[0000] Created image volume k3d-k3s-default-images
INFO[0000] Starting new tools node...
INFO[0000] Starting Node 'k3d-k3s-default-tools'
INFO[0001] Creating node 'k3d-k3s-default-server-0'
INFO[0001] Creating LoadBalancer 'k3d-k3s-default-serverlb'
INFO[0002] Using the k3d-tools node to gather environment information
INFO[0002] HostIP: using network gateway 172.19.0.1 address
INFO[0002] Starting cluster 'k3s-default'
INFO[0002] Starting servers...
INFO[0002] Starting Node 'k3d-k3s-default-server-0'
INFO[0008] All agents already running.
INFO[0008] Starting helpers...
INFO[0008] Starting Node 'k3d-k3s-default-serverlb'
INFO[0015] Injecting records for hostAliases (incl. host.k3d.internal) and for 2
network members into CoreDNS configmap...
INFO[0017] Cluster 'k3s-default' created successfully!
INFO[0018] You can now use it like this:
kubectl cluster-info


real    0m18.154s
user    0m0.005s
sys     0m0.000s
```

Without a load balancer, it takes less than 8 seconds!

What about multi-node clusters? We saw that KinD was much slower, especially when creating a HA cluster with multiple control plane nodes and an external load balancer.

Let's delete the single-node cluster first:

```
$ k3d cluster delete
INFO[0000] Deleting cluster 'k3s-default'
INFO[0000] Deleting cluster network 'k3d-k3s-default'
INFO[0000] Deleting 2 attached volumes...
WARN[0000] Failed to delete volume 'k3d-k3s-default-images' of cluster 'k3s-default':
failed to find volume 'k3d-k3s-default-images': Error: No such volume: k3d-k3s-
default-images -> Try to delete it manually
INFO[0000] Removing cluster details from default kubeconfig...
```

```
INFO[0000] Removing standalone kubeconfig file (if there is one)...
INFO[0000] Successfully deleted cluster k3s-default!
```

Now, let's create a cluster with 3 worker nodes. That takes a little over 30 seconds:

```
$ time k3d cluster create --agents 3
INFO[0000] Prep: Network
INFO[0000] Created network 'k3d-k3s-default'
INFO[0000] Created image volume k3d-k3s-default-images
INFO[0000] Starting new tools node...
INFO[0000] Starting Node 'k3d-k3s-default-tools'
INFO[0001] Creating node 'k3d-k3s-default-server-0'
INFO[0001] Creating node 'k3d-k3s-default-agent-0'
INFO[0002] Creating node 'k3d-k3s-default-agent-1'
INFO[0002] Creating node 'k3d-k3s-default-agent-2'
INFO[0002] Creating LoadBalancer 'k3d-k3s-default-serverlb'
INFO[0002] Using the k3d-tools node to gather environment information
INFO[0002] HostIP: using network gateway 172.22.0.1 address
INFO[0002] Starting cluster 'k3s-default'
INFO[0002] Starting servers...
INFO[0002] Starting Node 'k3d-k3s-default-server-0'
INFO[0008] Starting agents...
INFO[0008] Starting Node 'k3d-k3s-default-agent-0'
INFO[0008] Starting Node 'k3d-k3s-default-agent-2'
INFO[0008] Starting Node 'k3d-k3s-default-agent-1'
INFO[0018] Starting helpers...
INFO[0019] Starting Node 'k3d-k3s-default-serverlb'
INFO[0029] Injecting records for hostAliases (incl. host.k3d.internal) and for 5
network members into CoreDNS configmap...
INFO[0032] Cluster 'k3s-default' created successfully!
INFO[0032] You can now use it like this:
kubectl cluster-info

real    0m32.512s
user    0m0.005s
sys     0m0.000s
```

Let's verify the cluster works as expected:

```
$ k cluster-info
Kubernetes control plane is running at https://0.0.0.0:60490
CoreDNS is running at https://0.0.0.0:60490/api/v1/namespaces/kube-system/services/
kube-dns:dns/proxy
```

```
Metrics-server is running at https://0.0.0.0:60490/api/v1/namespaces/kube-system/
services/https:metrics-server:https/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Here are the nodes. Note that there is just one control plane node called `k3d-k3s-default-server-0`:

```
$ k get nodes
NAME                        STATUS   ROLES                  AGE     VERSION
k3d-k3s-default-server-0    Ready    control-plane,master   5m33s   v1.22.7+k3s1
k3d-k3s-default-agent-0     Ready    <none>                 5m30s   v1.22.7+k3s1
k3d-k3s-default-agent-2     Ready    <none>                 5m30s   v1.22.7+k3s1
k3d-k3s-default-agent-1     Ready    <none>                 5m29s   v1.22.7+k3s1
```

You can stop and start clusters, create multiple clusters, and list existing clusters using the k3d CLI. Here are all the commands. Feel free to explore further:

```
$ k3d
Usage:
  k3d [flags]
  k3d [command]

Available Commands:
  cluster      Manage cluster(s)
  completion   Generate completion scripts for [bash, zsh, fish, powershell | psh]
  config       Work with config file(s)
  help         Help about any command
  image        Handle container images.
  kubeconfig   Manage kubeconfig(s)
  node         Manage node(s)
  registry     Manage registry/registries
  version      Show k3d and default k3s version

Flags:
  -h, --help         help for k3d
      --timestamps   Enable Log timestamps
      --trace        Enable super verbose output (trace logging)
      --verbose      Enable verbose output (debug logging)
      --version      Show k3d and default k3s version

Use "k3d [command] --help" for more information about a command.
```

You can repeat the steps for deploying, exposing, and accessing the echo service on your own. It works just like KinD.

OK. We created clusters using minikube, KinD and k3d. Let's compare them, so you can decide which one works for you.

Comparing Minikube, KinD, and k3d

Minikube is an official local Kubernetes release. It's very mature and very full-featured. That said, it requires a VM and is both slow to install and to start. It also can get into trouble with networking at arbitrary times, and sometimes the only remedy is deleting the cluster and rebooting. Also, minikube supports a single node only. I suggest using Minikube only if it supports some features that you need that are not available in either KinD or k3d. See `https://minikube.sigs.k8s.io/` for more info.

KinD is much faster than Minikube and is used for Kubernetes conformance tests, so by definition, it is a conformant Kubernetes distribution. It is the only local cluster solution that provides an HA cluster with multiple control plane nodes. It is also designed to be used as a library, which I don't find a big attraction because it is very easy to automate CLIs from code. The main downside of KinD for local development is that it is ephemeral. I recommend using KinD if you contribute to Kubernetes itself and want to test against it. See `https://kind.sigs.k8s.io/`.

K3d is the clear winner for me. Lightning fast, and supports multiple clusters and multiple worker nodes per cluster. Easy to stop and start clusters without losing state. See `https://k3d.io/`.

Honorable mention – Rancher Desktop Kubernetes cluster

I use Rancher Desktop as my Docker Engine provider, but it also comes with a built-in Kubernetes cluster. You don't get to customize it and you can't have multiple clusters or even multiple nodes in the same cluster. But, if all you need is a local single-node Kubernetes cluster to play with, then the `rancher-desktop` cluster is there for you.

To use this cluster, type:

```
$ kubectl config use-context rancher-desktop
Switched to context "rancher-desktop".
```

You can decide how many resources you allocate to its node, which is important if you try to deploy a lot of workloads on it because you get just the one node.

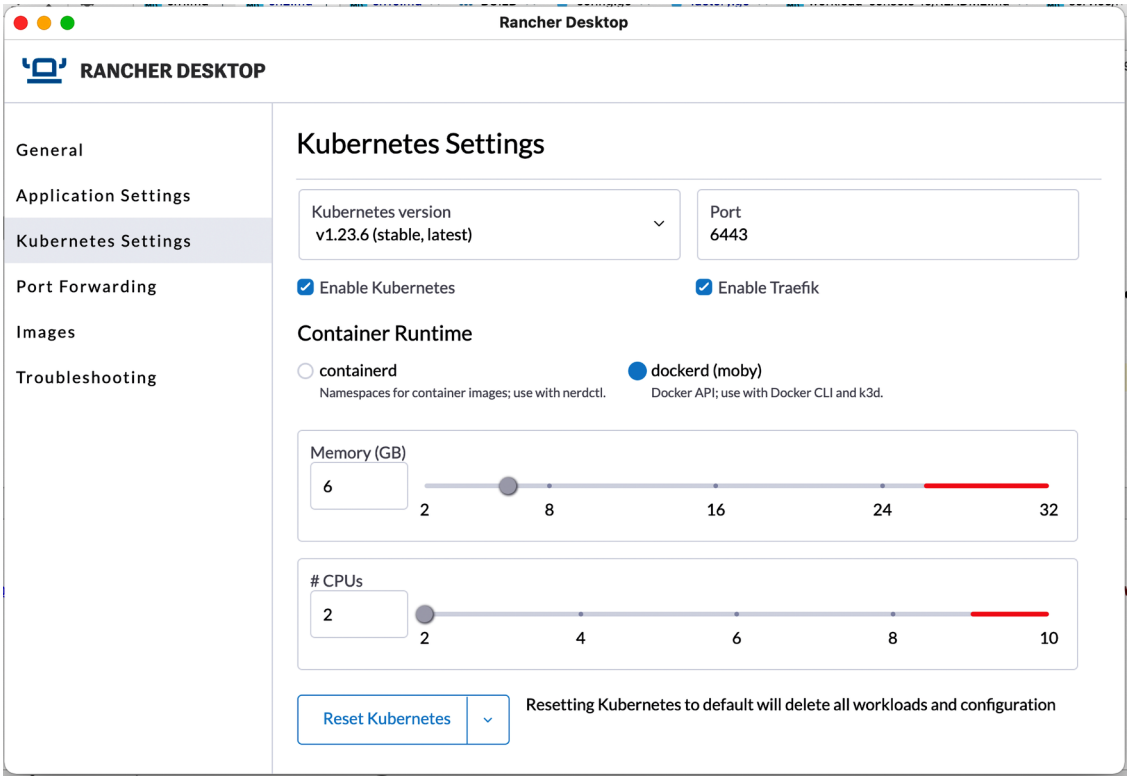


Figure 2.6: Rancher Desktop – Kubernetes settings

In this section, we covered creating Kubernetes clusters locally using Minikube, KinD, and K3d. In the next section, we will look at creating clusters in the cloud.

Creating clusters in the cloud (GCP, AWS, Azure, and Digital Ocean)

Creating clusters locally is fun. It's also important during development and when trying to troubleshoot problems locally. But, in the end, Kubernetes is designed for cloud-native applications (applications that run in the cloud). Kubernetes doesn't want to be aware of individual cloud environments because that doesn't scale. Instead, Kubernetes has the concept of a cloud-provider interface. Every cloud provider can implement this interface and then host Kubernetes.

The cloud-provider interface

The cloud-provider interface is a collection of Go data types and interfaces. It is defined in a file called `cloud.go`, available at: `https://github.com/kubernetes/cloud-provider/blob/master/cloud.go`.

Here is the main interface:

```
type Interface interface {
     Initialize(clientBuilder ControllerClientBuilder, stop <-chan struct{})
    LoadBalancer() (LoadBalancer, bool)
    Instances() (Instances, bool)
    InstancesV2() (InstancesV2, bool)
    Zones() (Zones, bool)
    Clusters() (Clusters, bool)
    Routes() (Routes, bool)
    ProviderName() string
    HasClusterID() bool
}
```

This is very clear. Kubernetes operates in terms of instances, zones, clusters, and routes, and also requires access to a load balancer and provider name. The main interface is primarily a gateway. Most methods of the `Interface` interface above return yet other interfaces.

For example, the `Clusters()` method returns the `Cluster` interface, which is very simple:

```
type Clusters interface {
    ListClusters(ctx context.Context) ([]string, error)
    Master(ctx context.Context, clusterName string) (string, error)
}
```

The `ListClusters()` method returns cluster names. The `Master()` method returns the IP address or DNS name of the control plane of the cluster.

The other interfaces are not much more complicated. The entire file is 313 lines long (at the time of writing) including lots of comments. The take-home point is that it is not too complicated to implement a Kubernetes provider if your cloud utilizes those basic concepts.

Creating Kubernetes clusters in the cloud

Before we look at the cloud providers and their support for managed and non-managed Kubernetes, let's consider how you should create and maintain clusters. If you commit to a single cloud provider, and you are happy with using their tooling, then you are set. All cloud providers let you create and configure Kubernetes clusters using either a Web UI, a CLI, or an API. However, if you prefer a more general approach and want to utilize GitOps to manage your clusters, you should look into Infrastructure as Code solutions such as Terraform and Pulumi.

If you prefer to roll out non-managed Kubernetes clusters in the cloud, then kOps is a strong candidate. See: `https://kops.sigs.k8s.io`.

Later, in *Chapter 17*, *Running Kubernetes in Production*, we will discuss in detail the topic of multi-cluster provisioning and management. There are many technologies, open source projects, and commercial products in this space.

For now, let's look at the various cloud providers.

GCP

The **Google Cloud Platform** (**GCP**) supports Kubernetes out of the box. The so-called **Google Kubernetes Engine** (**GKE**) is a container management solution built on Kubernetes. You don't need to install Kubernetes on GCP, and you can use the Google Cloud API to create Kubernetes clusters and provision them. The fact that Kubernetes is a built-in part of the GCP means it will always be well integrated and well tested, and you don't have to worry about changes in the underlying platform breaking the cloud-provider interface.

If you prefer to manage Kubernetes yourself, then you can just deploy it directly on GCP instances (or use kOps alpha support for GCP), but I would generally advise against it as GKE does a lot of work for you and it's integrated deeply with GCP compute, networking, and core services.

All in all, if you plan to base your system on Kubernetes and you don't have any existing code on other cloud platforms, then GCP is a solid choice. It leads the pack in terms of maturity, polish, and depth of integration to GCP services, and is usually the first to update to newer versions of Kubernetes.

I spent a lot of time with Kubernetes on GKE, managing tens of clusters, upgrading them, and deploying workloads. GKE is production-grade Kubernetes for sure.

GKE Autopilot

GKE also has the Autopilot project, which takes care of managing worker nodes and node pools for you, so you focus on deploying and configuring workloads.

See: `https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview`.

AWS

AWS has its own container management service called ECS, which is not based on Kubernetes. It also has a managed Kubernetes service called EKS. You can run Kubernetes yourself on AWS EC2 instances. Let's talk about how to roll your own Kubernetes first and then we'll discuss EKS.

Kubernetes on EC2

AWS was a supported cloud provider from the get-go. There is a lot of documentation on how to set it up. While you could provision some EC2 instances yourself and use kubeadm to create a cluster, I recommend using the kOps (Kubernetes Operations) project mentioned earlier. kOps initially supported only AWS and is generally considered the most battle-tested and feature-rich tool for self-provisioning Kubernetes clusters on AWS (without using EKS).

It supports the following features:

- Automated Kubernetes cluster CRUD for the cloud (AWS)
- HA Kubernetes clusters
- Uses a state-sync model for dry-run and automatic idempotency
- Custom support for kubectl addons
- kOps can generate Terraform configuration
- Based on a simple meta-model defined in a directory tree
- Easy command-line syntax
- Community support

To create a cluster, you need to do some IAM and DNS configuration, set up an S3 bucket to store the cluster configuration, and then run a single command:

```
kops create cluster \
    --name=${NAME} \
    --cloud=aws \
    --zones=us-west-2a \
    --discovery-store=s3://prefix-example-com-oidc-store/${NAME}/discovery
```

The complete instructions are here: `https://kops.sigs.k8s.io/getting_started/aws/`.

At the end of 2017, AWS joined the CNCF and made two big announcements regarding Kubernetes: its own Kubernetes-based container orchestration solution (EKS) and a container-on-demand solution (Fargate).

Amazon EKS

Amazon Elastic Kubernetes Service (**EKS**) is a fully managed and highly available Kubernetes solution. It has three control plane nodes running in three AZs. EKS also takes care of upgrades and patching. The great thing about EKS is that it runs a stock Kubernetes. This means you can use all the standard plugins and tools developed by the community. It also opens the door to convenient cluster federation with other cloud providers and/or your own on-premise Kubernetes clusters.

EKS provides deep integration with AWS infrastructure like IAM authentication, which is integrated with Kubernetes **Role-Based Access Control** (**RBAC**).

You can also use PrivateLink if you want to access your Kubernetes masters directly from your own Amazon VPC. With PrivateLink, your Kubernetes control plane and the Amazon EKS service endpoint appear as an elastic network interface with private IP addresses in your Amazon VPC.

Another important piece of the puzzle is a special CNI plugin that lets your Kubernetes components talk to each other using AWS networking.

EKS keeps getting better and Amazon demonstrated that it is committed to keeping it up to date and improving it. If you are an AWS shop and getting into Kubernetes, I recommend starting with EKS as opposed to building your own cluster.

The eksctl tool is a great CLI for creating and managing EKS clusters and node groups for testing and development. I successfully created, deleted, and added nodes to several Kubernetes clusters on AWS using eksctl. Check out `https://eksctl.io/`.

Fargate

Fargate lets you run containers directly without worrying about provisioning hardware. It eliminates a huge part of the operational complexity at the cost of losing some control. When using Fargate, you package your application into a container, specify CPU and memory requirements, define networking and IAM policies, and you're off to the races. Fargate can run on top of ECS and EKS. It is a very interesting member of the serverless camp although it's not specific to Kubernetes like GKE's Autopilot.

Azure

Azure used to have its own container management service based on Mesos-based DC/OS or Docker Swarm to manage your containers. But you can also use Kubernetes, of course. You could also provision the cluster yourself (for example, using Azure's desired state configuration) and then create the Kubernetes cluster using kubeadm. kOps has alpha support for Azure, and the Kubespray project is a good option too.

However, in the second half of 2017 Azure jumped on the Kubernetes bandwagon too and introduced **AKS** (**Azure Kubernetes Service**). It is similar to Amazon EKS, although it's a little further ahead in its implementation.

AKS provides a Web UI, CLI, and REST API to manage your Kubernetes clusters. Once, an AKS cluster is configured, you can use kubectl and any other Kubernetes tooling directly.

Here are some of the benefits of using AKS:

- Automated Kubernetes version upgrades and patching
- Easy cluster scaling
- Self-healing hosted control plane (masters)
- Cost savings – pay only for running agent pool nodes

AKS also offers integration with **Azure Container Instances** (**ACI**), which is similar to AWS Fargate and GKE AutoPilot. This means that not only the control plane of your Kubernetes cluster is managed, but also the worker nodes.

Digital Ocean

Digital Ocean is not a cloud provider on the order of the big three (GCP, AWS, Azure), but it does provide a managed Kubernetes solution and it has data centers across the world (US, Canada, Europe, Asia). It is also much cheaper compared to the alternatives, and cost is a major deciding factor when choosing a cloud provider. With Digital Ocean, the control plane doesn't cost anything. Besides lower prices Digital Ocean's claim to fame is simplicity.

DOKS (**Digital Ocean Kubernetes Service**) gives you a managed Kubernetes control plane (which can be highly available) and integration with Digital Ocean's droplets (for nodes and node pools), load balancers, and block storage volumes. This covers all the basic needs. Your clusters are of course CNCF conformant.

Digital Ocean will take care of system upgrades, security patches, and the installed packages on the control plane as well as the worker nodes.

Other cloud providers

GCP, AWS, and Azure are leading the pack, but there are quite a few other companies that offer managed Kubernetes services. In general, I recommend using these providers if you already have significant business connections or integrations.

Once upon a time in China

If you operate in China with its special constraints and limitations, you should probably use a Chinese cloud platform. There are three big ones: Alibaba, Tencent, and Huawei.

The Chinese **Alibaba** cloud is an up-and-comer on the cloud platform scene. It mimics AWS pretty closely, although its English documentation leaves a lot to be desired. The Alibaba cloud supports Kubernetes in several ways via its **ACK** (**Alibaba Container service for Kubernetes**) and allows you to:

- Run your own dedicated Kubernetes cluster (you must create 3 master nodes and upgrade and maintain them)
- Use the managed Kubernetes cluster (you're just responsible for the worker nodes)
- Use the serverless Kubernetes cluster via **ECI** (**Elastic Container Instances**), which is similar to Fargate and ACI

ACK is a CNCF-certified Kubernetes distribution. If you need to deploy cloud-native applications in China, then ACK looks like a solid option.

See `https://www.alibabacloud.com/product/kubernetes`.

Tencent is another large Chinese company with its own cloud platform and Kubernetes support. **TKE** (**Tencent Kubernetes Engine**) seems less mature than ACK. See `https://intl.cloud.tencent.com/products/tke`.

Finally, the Huawei cloud platform offers **CCE** (**Cloud Container Engine**), which is built on Kubernetes. It supports VMs, bare metal, and GPU accelerated instances. See `https://www.huaweicloud.com/intl/en-us/product/cce.html`.

IBM Kubernetes service

IBM is investing heavily in Kubernetes. It acquired Red Hat at the end of 2018. Red Hat was of course a major player in the Kubernetes world, building its OpenShift Kubernetes-based platform and contributing RBAC to Kubernetes. IBM has its own cloud platform and it offers a managed Kubernetes cluster. You can try it out for free with $200 credit and there is also a free tier.

IBM is also involved in the development of Istio and Knative, so you can expect IKS to have deep integration with those technologies.

IKS offers integration with a lot of IBM services.

See `https://www.ibm.com/cloud/kubernetes-service`.

Oracle Container Service

Oracle also has a cloud platform and of course, it offers a managed Kubernetes service too, with high availability, bare-metal instances, and multi-AZ support.

OKE supports ARM and GPU instances and also offers a few control plane options.

See `https://www.oracle.com/cloud/cloud-native/container-engine-kubernetes/`.

In this section, we covered the cloud-provider interface and looked at the recommended ways to create Kubernetes clusters on various cloud providers. The scene is still young and the tools evolve quickly. I believe convergence will happen soon. Kubeadm has matured and is the underlying foundation of many other tools to bootstrap and create Kubernetes clusters on and off the cloud. Let's consider now what it takes to create bare-metal clusters where you have to provision the hardware and low-level networking and storage too.

Creating a bare-metal cluster from scratch

In the previous section, we looked at running Kubernetes on cloud providers. This is the dominant deployment story for Kubernetes. But there are strong use cases for running Kubernetes on bare metal, such as Kubernetes on the edge. We don't focus here on hosted versus on-premise. This is yet another dimension. If you already manage a lot of servers on-premise, you are in the best position to decide.

Use cases for bare metal

Bare-metal clusters are a bear to deal with, especially if you manage them yourself. There are companies that provide commercial support for bare-metal Kubernetes clusters, such as Platform 9, but the offerings are not mature yet. A solid open-source option is Kubespray, which can deploy industrial-strength Kubernetes clusters on bare metal, AWS, GCE, Azure, and OpenStack.

Here are some use cases where it makes sense:

- **Price**: If you already manage large-scale bare-metal clusters, it may be much cheaper to run Kubernetes clusters on your physical infrastructure
- **Low network latency**: If you must have low latency between your nodes, then the VM overhead might be too much
- **Regulatory requirements**: If you must comply with regulations, you may not be allowed to use cloud providers
- **You want total control over hardware**: Cloud providers give you many options, but you may have special needs

When should you consider creating a bare-metal cluster?

The complexities of creating a cluster from scratch are significant. A Kubernetes cluster is not a trivial beast. There is a lot of documentation on the web on how to set up bare-metal clusters, but as the whole ecosystem moves forward, many of these guides get out of date quickly.

You should consider going down this route if you have the operational capability to troubleshoot problems at every level of the stack. Most of the problems will probably be networking-related, but filesystems and storage drivers can bite you too, as well as general incompatibilities and version mismatches between components such as Kubernetes itself, Docker (or other runtimes, if you use them), images, your OS, your OS kernel, and the various addons and tools you use. If you opt for using VMs on top of bare metal, then you add another layer of complexity.

Understanding the process

There is a lot to do. Here is a list of some of the concerns you'll have to address:

- Implementing your own cloud-provider interface or sidestepping it
- Choosing a networking model and how to implement it (CNI plugin, direct compile)
- Whether or not to use network policies
- Selecting images for system components
- The security model and SSL certificates
- Admin credentials
- Templates for components such as API Server, replication controller, and scheduler
- Cluster services: DNS, logging, monitoring, and GUI

I recommend the following guide from the Kubernetes site to get a deeper understanding of what it takes to create a HA cluster from scratch using kubeadm:

`https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/`

Using the Cluster API for managing bare-metal clusters

The Cluster API (AKA CAPI) is a Kubernetes sub-project for managing Kubernetes clusters at scale. It uses kubeadm for provisioning. It can provision and manage Kubernetes clusters in any environment using providers. At work, we use it to manage multiple clusters in the cloud. But, it has multiple providers for bare-metal clusters:

- MAAS
- Equinix metal
- Metal3
- Cidero

See `https://cluster-api.sigs.k8s.io`.

Using virtual private cloud infrastructure

If your use case falls under the bare-metal use cases but you don't have the necessary skilled manpower or the inclination to deal with the infrastructure challenges of bare metal, you have the option to use a private cloud such as OpenStack. If you want to aim a little higher in the abstraction ladder, then Mirantis offers a cloud platform built on top of OpenStack and Kubernetes.

Let's review a few more tools for building Kubernetes clusters on bare metal. Some of these tools support OpenStack as well.

Building your own cluster with Kubespray

Kubespray is a project for deploying production-ready highly available Kubernetes clusters. It uses Ansible and can deploy Kubernetes on a large number of targets such as:

- AWS
- GCE
- Azure
- OpenStack
- vSphere
- Equinix metal
- Oracle Cloud Infrastructure (Experimental)

It is also used to deploy Kubernetes clusters on plain bare-metal machines.

It is highly customizable and supports multiple operating systems for the nodes, multiple CNI plugins for networking, and multiple container runtimes.

If you want to test it locally, it can deploy to a multi-node vagrant setup too. If you're an Ansible fan, Kubespray may be a great choice for you.

See `https://kubespray.io`.

Building your cluster with Rancher RKE

Rancher Kubernetes Engine (**RKE**) is a friendly Kubernetes installer that can install Kubernetes on bare metal as well as virtualized servers. RKE aims to address the complexity of installing Kubernetes. It is open source and has great documentation. Check it out here: `http://rancher.com/docs/rke/v0.1.x/en/`.

Running managed Kubernetes on bare metal or VMs

The cloud providers didn't want to confine themselves to their own cloud only. They all offer multi-cloud and hybrid solutions where you can control Kubernetes clusters on multiple clouds as well as use their managed control plane on VMs anywhere.

GKE Anthos

Anthos is a comprehensive managed platform that facilitates the deployment of applications, encompassing both traditional and cloud-native environments. It empowers you to construct and oversee global fleets of applications while ensuring operational consistency across them.

EKS Anywhere

Amazon EKS Anywhere presents a fresh deployment alternative for Amazon EKS that enables you to establish and manage Kubernetes clusters on your infrastructure with AWS support. It grants you the flexibility to run Amazon EKS Anywhere on your own on-premises infrastructure, utilizing VMware vSphere, as well as bare metal environments.

AKS Arc

Azure Arc encompasses a collection of technologies that extend Azure's security and cloud-native services to hybrid and multi-cloud environments. It empowers you to safeguard and manage your infrastructure and applications across various locations while providing familiar tools and services to accelerate the development of cloud-native apps. These applications can then be deployed on any Kubernetes platform.

In this section, we covered creating bare-metal Kubernetes clusters, which gives you total control, but is highly complicated, and requires a tremendous amount of effort and knowledge. Luckily, there are multiple tools, projects, and frameworks to assist you.

Summary

In this chapter, we got into some hands-on cluster creation. We created single-node and multi-node clusters using tools like Minikube, KinD, and k3d. Then we looked at the various options to create Kubernetes clusters on cloud providers. Finally, we touched on the complexities of creating Kubernetes clusters on bare metal. The current state of affairs is very dynamic. The basic components are changing rapidly, the tooling is getting better, and there are different options for each environment. Kubeadm is now the cornerstone of most installation options, which is great for consistency and consolidation of effort. It's still not completely trivial to stand up a Kubernetes cluster on your own, but with some effort and attention to detail, you can get it done quickly.

I highly recommend considering the Cluster API as the go-to solution for provisioning and managing clusters in any environment – managed, private cloud, VMs, and bare metal. We will discuss the Cluster API in depth in *Chapter 17*, *Running Kubernetes in Production*.

In the next chapter, we will explore the important topics of scalability and high availability. Once your cluster is up and running, you need to make sure it stays that way even as the volume of requests increases. This requires ongoing attention and building the ability to recover from failures as well adjusting to changes in traffic.

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`



3

High Availability and Reliability

In *Chapter 2*, *Creating Kubernetes Clusters*, we learned how to create Kubernetes clusters in different environments, experimented with different tools, and created a couple of clusters. Creating a Kubernetes cluster is just the beginning of the story. Once the cluster is up and running, you need to make sure it stays operational.

In this chapter, we will dive into the topic of highly available clusters. This is a complicated topic. The Kubernetes project and the community haven't settled on one true way to achieve high availability nirvana. There are many aspects to highly available Kubernetes clusters, such as ensuring that the control plane can keep functioning in the face of failures, protecting the cluster state in etcd, protecting the system's data, and recovering capacity and/or performance quickly. Different systems will have different reliability and availability requirements. How to design and implement a highly available Kubernetes cluster will depend on those requirements.

This chapter will explore the following main topics:

- High availability concepts
- High availability best practices
- High availability, scalability, and capacity planning
- Large cluster performance, cost, and design trade-offs
- Choosing and managing the cluster capacity
- Pushing the envelope with Kubernetes
- Testing Kubernetes at scale

At the end of this chapter, you will understand the various concepts associated with high availability and be familiar with Kubernetes' high availability best practices and when to employ them. You will be able to upgrade live clusters using different strategies and techniques, and you will be able to choose between multiple possible solutions based on trade-offs between performance, cost, and availability.

High availability concepts

In this section, we will start our journey into high availability by exploring the concepts and building blocks of reliable and highly available systems. The million (trillion?) dollar question is, how do we build reliable and highly available systems from unreliable components? Components will fail; you can take that to the bank. Hardware will fail, networks will fail, configuration will be wrong, software will have bugs, and people will make mistakes. Accepting that, we need to design a system that can be reliable and highly available even when components fail. The idea is to start with redundancy, detect component failure, and replace bad components quickly.

Redundancy

Redundancy is the foundation of reliable and highly available systems at the hardware and software levels. If a critical component fails and you want the system to keep running, you must have another identical component ready to go. Kubernetes itself takes care of your stateless pods via replication controllers and replica sets. But, your cluster state in etcd and the control plane components themselves need redundancy to function when some components fail. In practice, this means running etcd and the API server on 3 or more nodes.

In addition, if your system's stateful components are not already backed up by redundant persistent storage (for example, on a cloud platform), then you need to add redundancy to prevent data loss.

Hot swapping

Hot swapping is the concept of replacing a failed component on the fly without taking the system down, with minimal (ideally, zero) interruption for users. If the component is stateless (or its state is stored in separate redundant storage), then hot swapping a new component to replace it is easy and just involves redirecting all clients to the new component. But, if it stores local state, including in memory, then hot swapping is not trivial. There are two main options:

- Give up on in-flight transactions (clients will retry)
- Keep a hot replica in sync (active-active)

The first solution is much simpler. Most systems are resilient enough to cope with failures. Clients can retry failed requests and the hot-swapped component will service them.

The second solution is more complicated and fragile and will incur a performance overhead because every interaction must be replicated to both copies (and acknowledged). It may be necessary for some critical parts of the system.

Leader election

Leader election is a common pattern in distributed systems. You often have multiple identical components that collaborate and share the load, but one component is elected as the leader and certain operations are serialized through the leader. You can think of distributed systems with leader election as a combination of redundancy and hot swapping. The components are all redundant and when the current leader fails or becomes unavailable, a new leader is elected and hot-swapped in.

Smart load balancing

Load balancing is about distributing the workload across multiple replicas that service incoming requests. This is useful for scaling up and down under a heavy load by adjusting the number of replicas. When some replicas fail, the load balancer will stop sending requests to failed or unreachable components. Kubernetes will provision new replicas, restore capacity, and update the load balancer. Kubernetes provides great facilities to support this via services, endpoints, replica sets, labels, and ingress controllers.

Idempotency

Many types of failure can be temporary. This is most common with networking issues or with too-stringent timeouts. A component that doesn't respond to a health check will be considered unreachable and another component will take its place. Work that was scheduled for the presumably failed component may be sent to another component. But the original component may still be working and complete the same work. The end result is that the same work may be performed twice. It is very difficult to avoid this situation. To support exactly-once semantics, you need to pay a heavy price in overhead, performance, latency, and complexity. Thus, most systems opt to support at-least-once semantics, which means it is OK for the same work to be performed multiple times without violating the system's data integrity. This property is called idempotency. Idempotent systems maintain their state even if an operation is performed multiple times.

Self-healing

When component failures occur in dynamic systems, you usually want the system to be able to heal itself. Kubernetes replica sets are great examples of self-healing systems. But failure can extend well beyond pods. Self-healing starts with the automated detection of problems followed by an automated resolution. Quotas and limits help create checks and balances to ensure automated self-healing doesn't run amok due to bugs or circumstances such as DDoS attacks. Self-healing systems deal very well with transient failures by retrying failed operations and escalating failures only when it's clear there is no other option. Some self-healing systems have fallback paths including serving cached content if up-to-date content is unavailable. Self-healing systems attempt to degrade gracefully and keep working until the core issue can be fixed.

In this section, we considered various concepts involved in creating reliable and highly available systems. In the next section, we will apply them and demonstrate best practices for systems deployed on Kubernetes clusters.

High availability best practices

Building reliable and highly available distributed systems is a non-trivial endeavor. In this section, we will check some of the best practices that enable a Kubernetes-based system to function reliably and be available in the face of various failure categories. We will also dive deep and see how to go about constructing your own highly available clusters. However, due to the complexity and the large number of factors that impact HA clusters, we will just provide guidance. We will not provide here step by step instructions for building a HA cluster.

Note that you should roll your own highly available Kubernetes cluster only in very special cases. There are multiple robust tools (usually built on top of kubeadm) that provide battle-tested ways to create highly available Kubernetes clusters at the control plane level. You should take advantage of all the work and effort that went into these tools. In particular, the cloud providers offer managed Kubernetes clusters that are highly available.

Creating highly available clusters

To create a highly available Kubernetes cluster, the control plane components must be redundant. That means etcd must be deployed as a cluster (typically across three or five nodes) and the Kubernetes API server must be redundant. Auxiliary cluster-management services such as the observability stack storage should be deployed redundantly too. The following diagram depicts a typical reliable and highly available Kubernetes cluster in a stacked etcd topology. There are several load-balanced control plane nodes, each one containing all the control plane components as well as an etcd component:

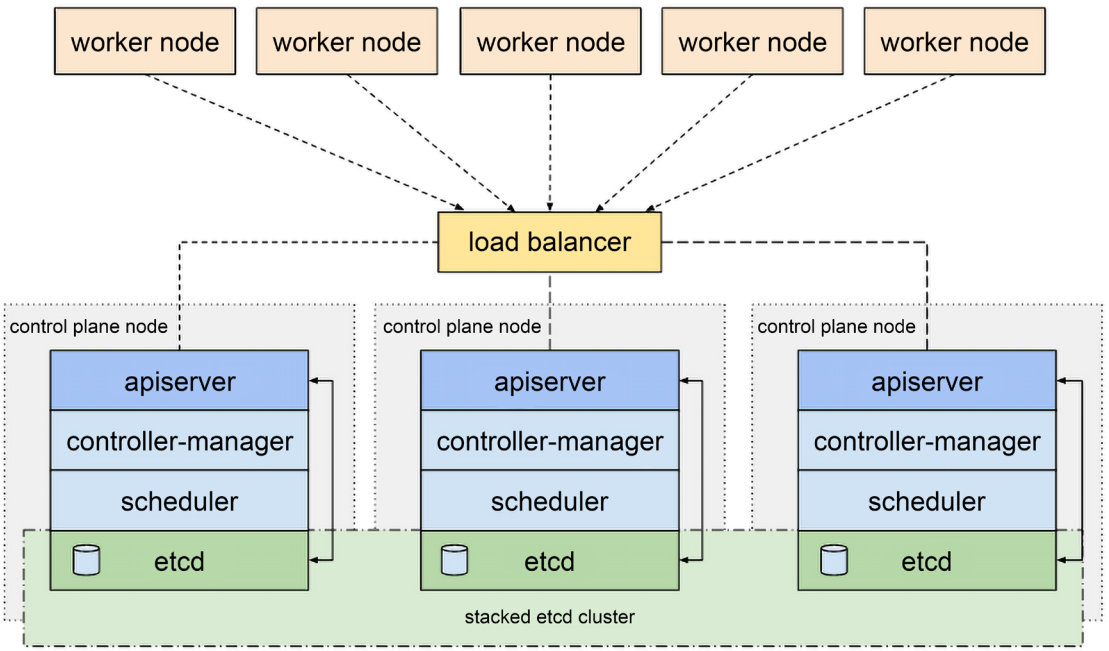


Figure 3.1: A highly available cluster configuration

This is not the only way to configure highly available clusters. You may prefer, for example, to deploy a standalone etcd cluster to optimize the machines to their workload or if you require more redundancy for your etcd cluster than the rest of the control plane nodes.

The following diagram shows a Kubernetes cluster where etcd is deployed as an external cluster:

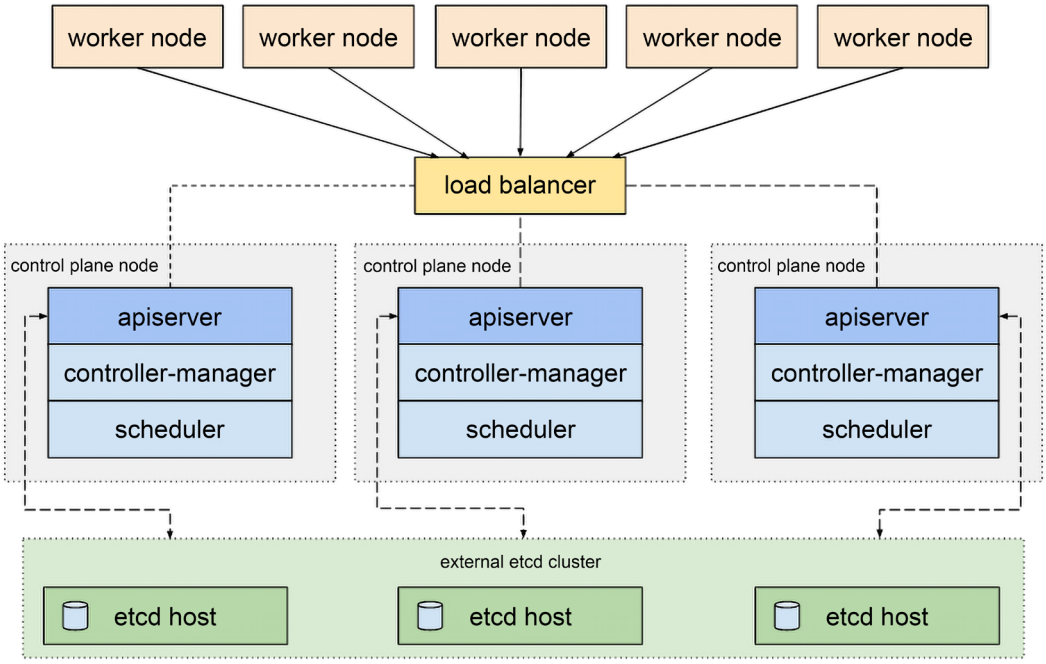


Figure 3.2: etcd used as an external cluster

Self-hosted Kubernetes clusters, where control plane components are deployed as pods and stateful sets in the cluster, are a great approach to simplify the robustness, disaster recovery, and self-healing of the control plane components by applying Kubernetes to Kubernetes. This means that some of the components that manage Kubernetes are themselves managed by Kubernetes. For example, if one of the Kubernetes API server nodes goes down, the other API server pods will notice and provision a new API server.

Making your nodes reliable

Nodes will fail, or some components will fail, but many failures are transient. The basic guarantee is to make sure that the runtime engine (Docker daemon, Containerd, or whatever the CRI implementation is) and the kubelet restart automatically in the event of a failure.

If you run CoreOS, a modern Debian-based OS (including Ubuntu >= 16.04), or any other OS that uses systemd as its init mechanism, then it's easy to deploy Docker and the kubelet as self-starting daemons:

```
systemctl enable docker
systemctl enable kubelet
```

For other operating systems, the Kubernetes project selected the **monit** process monitor for their high availability example, but you can use any process monitor you prefer. The main requirement is to make sure that those two critical components will restart in the event of failure, without external intervention.

See `https://monit-docs.web.cern.ch/base/kubernetes/`.

Protecting your cluster state

The Kubernetes cluster state is typically stored in etcd (some Kubernetes implementations like k3s use alternative storage engines like SQLite). The etcd cluster was designed to be super reliable and distributed across multiple nodes. It's important to take advantage of these capabilities for a reliable and highly available Kubernetes cluster by making sure to have multiple copies of the cluster state in case one of the copies is lost and unreachable.

Clustering etcd

You should have at least three nodes in your etcd cluster. If you need more reliability and redundancy, you can have five, seven, or any other odd number of nodes. The number of nodes must be odd to have a clear majority in case of a network split.

In order to create a cluster, the etcd nodes should be able to discover each other. There are several methods to accomplish that such as:

- static
- etcd discovery
- DNS discovery

The etcd-operator project from CoreOS used to be the go-to solution for deploying etcd clusters. Unfortunately, the project has been archived and is not developed actively anymore. Kubeadm uses the static method for provisioning etcd clusters for Kubernetes, so if you use any tool based on kubeadm, you're all set. If you want to deploy a HA etcd cluster, I recommend following the official documentation: `https://github.com/etcd-io/etcd/blob/release-3.4/Documentation/op-guide/clustering.md`.

Protecting your data

Protecting the cluster state and configuration is great, but even more important is protecting your own data. If somehow the cluster state gets corrupted, you can always rebuild the cluster from scratch (although the cluster will not be available during the rebuild). But if your own data is corrupted or lost, you're in deep trouble. The same rules apply; redundancy is king. But while the Kubernetes cluster state is very dynamic, much of your data may be less dynamic. For example, a lot of historic data is often important and can be backed up and restored. Live data might be lost, but the overall system may be restored to an earlier snapshot and suffer only temporary damage.

You should consider Velero as a solution for backing up your entire cluster including your own data. Heptio (now part of VMWare) developed Velero, which is open source and may be a lifesaver for critical systems.

Check it out here: `https://velero.io/`.

Running redundant API servers

The API servers are stateless, fetching all the necessary data on the fly from the etcd cluster. This means that you can easily run multiple API servers without needing to coordinate between them. Once you have multiple API servers running, you can put a load balancer in front of them to make it transparent to clients.

Running leader election with Kubernetes

Some control plane components, such as the scheduler and the controller manager, can't have multiple instances active at the same time. This will be chaos, as multiple schedulers try to schedule the same pod into multiple nodes or multiple times into the same node. It is possible to run multiple schedulers that are configured to manage different pods. The correct way to have a highly scalable Kubernetes cluster is to have these components run in leader election mode. This means that multiple instances are running but only one is active at a time, and if it fails, another one is elected as leader and takes its place.

Kubernetes supports this mode via the `–leader-elect` flag (the default is `True`). The scheduler and the controller manager can be deployed as pods by copying their respective manifests to `/etc/kubernetes/manifests`.

Here is a snippet from a scheduler manifest that shows the use of the flag:

```
    command:
    - /bin/sh
    - -c
    - /usr/local/bin/kube-scheduler --master=127.0.0.1:8080 --v=2 --leader-elect=true
1>>/var/log/kube-scheduler.log
      2>&1
```

Here is a snippet from a controller manager manifest that shows the use of the flag:

```
  - command:
    - /bin/sh
    - -c
    - /usr/local/bin/kube-controller-manager --master=127.0.0.1:8080 --cluster-
name=e2e-test-bburns
      --cluster-cidr=10.245.0.0/16 --allocate-node-cidrs=true --cloud-provider=gce
--service-account-private-key-file=/srv/kubernetes/server.key
      --v=2 --leader-elect=true 1>>/var/log/kube-controller-manager.log 2>&1
    image: gcr.io/google\_containers/kube-controller-manager:fda24638d51a48baa13c353
37fcd4793
```

There are several other flags to control leader election. All of them have reasonable defaults:

```
 --leader-elect-lease-duration duration     Default: 15s
--leader-elect-renew-deadline duration     Default: 10s
--leader-elect-resource-lock endpoints     Default: "endpoints" ("configmaps" is the
other option)
--leader-elect-retry-period duration       Default: 2s
```

Note that it is not possible to have these components restarted automatically by Kubernetes like other pods because these are exactly the Kubernetes components responsible for restarting failed pods, so they can't restart themselves if they fail. There must be a ready-to-go replacement already running.

Making your staging environment highly available

High availability is not trivial to set up. If you go to the trouble of setting up high availability, it means there is a business case for a highly available system. It follows that you want to test your reliable and highly available cluster before you deploy it to production (unless you're Netflix, where you test in production). Also, any change to the cluster may, in theory, break your high availability without disrupting other cluster functions. The essential point is that, just like anything else, if you don't test it, assume it doesn't work.

We've established that you need to test reliability and high availability. The best way to do it is to create a staging environment that replicates your production environment as closely as possible. This can get expensive. There are several ways to manage the cost:

- **Ad hoc HA staging environment**: Create a large HA cluster only for the duration of HA testing.
- **Compress time**: Create interesting event streams and scenarios ahead of time, feed the input, and simulate the situations in rapid succession.
- **Combine HA testing with performance and stress testing**: At the end of your performance and stress tests, overload the system and see how the reliability and high availability configuration handles the load.

It's also important to practice chaos engineering and intentionally instigate failure at different levels to verify the system can handle those failure modes.

Testing high availability

Testing high availability takes planning and a deep understanding of your system. The goal of every test is to reveal flaws in the system's design and/or implementation and to provide good enough coverage that, if the tests pass, you'll be confident that the system behaves as expected.

In the realm of reliability, self-healing, and high availability, it means you need to figure out ways to break the system and watch it put itself back together.

That requires several pieces, as follows:

- Comprehensive list of possible failures (including reasonable combinations)
- For each possible failure, it should be clear how the system should respond

- A way to induce the failure
- A way to observe how the system reacts

None of the pieces are trivial. The best approach in my experience is to do it incrementally and try to come up with a relatively small number of generic failure categories and generic responses, rather than an exhaustive, ever-changing list of low-level failures.

For example, a generic failure category is node-unresponsive. The generic response could be rebooting the node, the way to induce the failure can be stopping the VM of the node (if it's a VM), and the observation should be that, while the node is down, the system still functions properly based on standard acceptance tests; the node is eventually up, and the system gets back to normal. There may be many other things you want to test, such as whether the problem was logged, whether relevant alerts went out to the right people, and whether various stats and reports were updated.

But, beware of over-generalizing. In the case of the generic unresponsive node failure mode, a key component is detecting that the node is unresponsive. If your method of detection is faulty, then your system will not react properly. Use best practices like health checks and readiness checks.

Note that sometimes, a failure can't be resolved in a single response. For example, in our unresponsive node case, if it's a hardware failure, then a reboot will not help. In this case, the second line of response gets into play and maybe a new node is provisioned to replace the failed node. In this case, you can't be too generic and you may need to create tests for specific types of pods/roles that were on the node (etcd, master, worker, database, and monitoring).

If you have high-quality requirements, be prepared to spend much more time setting up the proper testing environments and tests than even the production environment.

One last important point is to try to be as non-intrusive as possible. That means that, ideally, your production system will not have testing features that allow shutting down parts of it or cause it to be configured to run at reduced capacity for testing. The reason is that it increases the attack surface of your system and it can be triggered by accident by mistakes in configuration. Ideally, you can control your testing environment without resorting to modifying the code or configuration that will be deployed in production. With Kubernetes, it is usually easy to inject pods and containers with custom test functionality that can interact with system components in the staging environment but will never be deployed in production.

The Chaos Mesh CNCF incubating project is a good starting point: `https://chaos-mesh.org`.

In this section, we looked at what it takes to actually have a reliable and highly available cluster, including etcd, the API server, the scheduler, and the controller manager. We considered best practices for protecting the cluster itself, as well as your data, and paid special attention to the issue of starting environments and testing.

High availability, scalability, and capacity planning

Highly available systems must also be scalable. The load on most complicated distributed systems can vary dramatically based on time of day, weekday vs weekend, seasonal effects, marketing campaigns, and many other factors. Successful systems will have more users over time and accumulate more and more data. That means that the physical resources of the clusters - mostly nodes and storage - will have to grow over time too. If your cluster is under-provisioned, it will not be able to satisfy all demands and it will not be available because requests will time out or be queued up and not processed fast enough.

This is the realm of capacity planning. One simple approach is to over-provision your cluster. Anticipate the demand and make sure you have enough of a buffer for spikes of activity. But, this approach suffers from several deficiencies:

- For highly dynamic and complicated distributed systems, it's difficult to predict the demand even approximately.
- Over-provisioning is expensive. You spend a lot of money on resources that are rarely or never used.
- You have to periodically redo the whole process because the average and peak load on the system changes over time.
- You have to do the entire process for multiple groups of workloads that use specific resources (e.g. workloads that use high-memory nodes and workloads that require GPUs).

A much better approach is to use intent-based capacity planning where high-level abstraction is used and the system adjusts itself accordingly. In the context of Kubernetes, there is the **Horizontal Pod Autoscaler** (**HPA**), which can grow and shrink the number of pods needed to handle requests for a particular workload. But, that works only to change the ratio of resources allocated to different workloads. When the entire cluster (or node pool) approaches saturation, you simply need more resources. This is where the cluster autoscaler comes into play. It is a Kubernetes project that became available with Kubernetes 1.8. It works particularly well in cloud environments where additional resources can be provisioned via programmatic APIs.

When the **cluster autoscaler** (**CAS**) determines that pods can't be scheduled (are in a pending state) it provisions a new node for the cluster. It can also remove nodes from the cluster (downscaling) if it determines that the cluster has more nodes than necessary to handle the load. The CAS will check for pending pods every 30 seconds by default. It will remove nodes only after 10 minutes of low usage to avoid thrashing.

The CAS makes its scale-down decision based on CPU and memory usage. If the sum of CPU and memory requests of all pods running on a node is smaller than 50% (by default, it is configurable) of the node's allocatable resources, then the node will be considered for removal. All pods (except DaemonSet pods) must be movable (some pods can't be moved due to factors like scheduling constraints or local storage) and the node must not have scale-down disabled.

Here are some issues to consider:

- A cluster may require more nodes even if the total CPU or memory utilization is low due to control mechanisms like affinity, anti-affinity, taints, tolerations, pod priorities, max pods per node, max persistent volumes per node, and pod disruption budgets.
- In addition to the built-in delays in triggering scale up or scale down of nodes, there is an additional delay of several minutes when provisioning a new node from the cloud provider.
- Some nodes (e.g with local storage) can't be removed by default (require special annotation).
- The interactions between HPA and the CAS can be subtle.

Installing the cluster autoscaler

Note that you can't test the CAS locally. You must have a Kubernetes cluster running on one of the supported cloud providers:

- AWS
- BaiduCloud
- Brightbox
- CherryServers
- CloudStack
- HuaweiCloud
- External gRPC
- Hetzner
- Equinix Metal
- IonosCloud
- OVHcloud
- Linode
- OracleCloud
- ClusterAPI
- BizflyCloud
- Vultr
- TencentCloud

I have installed it successfully on GKE, EKS, and AKS. There are two reasons to use a CAS in a cloud environment:

1. You installed non-managed Kubernetes yourself and you want to benefit from the CAS.
2. You use a managed Kubernetes, but you want to modify some of its settings (e.g higher CPU utilization threshold). In this case, you will need to disable the cloud provider's CAS to avoid conflicts.

Let's look at the manifests for installing CAS on AWS. There are several ways to do it. I chose the multi-ASG (auto-scaling groups option), which is the most production-ready. It supports multiple node groups with different configurations. The file contains all the Kubernetes resources needed to install the cluster autoscaler. It involves creating a service account, and giving it various RBAC permissions because it needs to monitor node usage across the cluster and be able to act on it. Finally, there is a Deployment that actually deploys the cluster autoscaler image itself with a command line that includes the range of nodes (minimum and maximum number) it should maintain, and in the case of EKS, node groups are needed too. The maximum number is important to prevent a situation where an attack or error causes the cluster autoscaler to just add more and more nodes uncontrollably and rack up a huge bill. The full file is here: `https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/examples/cluster-autoscaler-multi-asg.yaml`.

Here is a snippet from the pod template of the Deployment:

```
    spec:
      priorityClassName: system-cluster-critical
      securityContext:
        runAsNonRoot: true
        runAsUser: 65534
        fsGroup: 65534
      serviceAccountName: cluster-autoscaler
      containers:
        - image: k8s.gcr.io/autoscaling/cluster-autoscaler:v1.22.2
          name: cluster-autoscaler
          resources:
            limits:
              cpu: 100m
              memory: 600Mi
            requests:
              cpu: 100m
              memory: 600Mi
          command:
            - ./cluster-autoscaler
            - --v=4
            - --stderrthreshold=info
            - --cloud-provider=aws
            - --skip-nodes-with-local-storage=false
            - --expander=least-waste
            - --nodes=1:10:k8s-worker-asg-1
            - --nodes=1:3:k8s-worker-asg-2
          volumeMounts:
            - name: ssl-certs
```

```
              mountPath: /etc/ssl/certs/ca-certificates.crt #/etc/ssl/certs/ca-
bundle.crt for Amazon Linux Worker Nodes
              readOnly: true
          imagePullPolicy: "Always"
      volumes:
        - name: ssl-certs
          hostPath:
            path: "/etc/ssl/certs/ca-bundle.crt"
```

See `https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/examples/cluster-autoscaler-multi-asg.yaml#L120`.

The combination of the HPA and CAS provides a truly elastic cluster where the HPA ensures that services use the proper amount of pods to handle the load per service and the CAS makes sure that the number of nodes matches the overall load on the cluster.

Considering the vertical pod autoscaler

The vertical pod autoscaler is another autoscaler that operates on pods. Its job is to adjust the CPU and memory requests and limits of pods based on actual usage. It is configured using a CRD for each workload and has three components:

- Recommender - Watches CPU and memory usage and provides recommendations for new values for CPU and memory requests
- Updater - Kills managed pods whose CPU and memory requests don't match the recommendations made by the recommender
- Admission control webhook - Sets the CPU and memory requests for new or recreated pods based on recommendations

The VPA can run in recommendation mode only or actively resizing pods. When the VPA decides to resize a pod, it evicts the pod. When the pod is rescheduled, it modifies the requests and limits based on the latest recommendation.

Here is an example that defines a VPA custom resource for a Deployment called `awesome-deployment` in recommendation mode:

```
apiVersion: autoscaling.k8s.io/v1beta2
kind: VerticalPodAutoscaler
metadata:
  name: awesome-deployment
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment
    name:       awesome-deployment
```

```
  updatePolicy:
    updateMode: "Off"
```

Here are some of the main requirements and limitations of using the VPA:

- Requires the metrics server
- Can't set memory to less than 250Mi
- Unable to update running pod (hence the updater kills pods to get them restarted with the correct requests)
- Can't evict pods that aren't managed by a controller
- It is not recommended to run the VPA alongside the HPA

Autoscaling based on custom metrics

The HPA operates by default on CPU and memory metrics. But, it can be configured to operate on arbitrary custom metrics like a queue depth (e.g. AWS SQS queue) or a number of threads, which may become the bottleneck due to concurrency even if there is still available CPU and memory. The Keda project (`https://keda.sh/`) provides a strong solution for custom metrics instead of starting from scratch. They use the concept of event-based autoscaling as a generalization.

This section covered the interactions between auto-scalability and high availability and looked at different approaches for scaling Kubernetes clusters and the applications running on these clusters.

Large cluster performance, cost, and design trade-offs

In the previous section, we looked at various ways to provision and plan for capacity and autoscale clusters and workloads. In this section, we will consider the various options and configurations of large clusters with different reliability and high availability properties. When you design your cluster, you need to understand your options and choose wisely based on the needs of your organization.

The topics we will cover include various availability requirements, from best effort all the way to the holy grail of zero downtime. Finally, we will settle down on the practical site reliability engineering approach. For each category of availability, we will consider what it means from the perspectives of performance and cost.



Availability requirements

Different systems have very different requirements for reliability and availability. Moreover, different sub-systems have very different requirements. For example, billing systems are always a high priority because if the billing system is down, you can't make money. But, even within the billing system, if the ability to dispute charges is sometimes unavailable, it may be OK from the business point of view.

Best effort

Best effort means, counter-intuitively, no guarantee whatsoever. If it works, great! If it doesn't work – oh well, what are you going to do? This level of reliability and availability may be appropriate for internal components that change often and the effort to make them robust is not worth it. As long the services or clients that invoke the unreliable services are able to handle the occasional errors or outages, then all is well. It may also be appropriate for services released in the wild as beta.

Best effort is great for developers. Developers can move fast and break things. They are not worried about the consequences and they don't have to go through a gauntlet of rigorous tests and approvals. The performance of best effort services may be better than more robust services because the best effort service can often skip expensive steps such as verifying requests, persisting intermediate results, and replicating data. But, on the other hand, more robust services are often heavily optimized and their supporting hardware is fine-tuned to their workload. The cost of best effort services is usually lower because they don't need to employ redundancy unless the operators neglect to do basic capacity planning and just over-provision needlessly.

In the context of Kubernetes, the big question is whether all the services provided by the cluster are best effort. If this is the case, then the cluster itself doesn't have to be highly available. You can probably have a single master node with a single instance of etcd, and a monitoring solution may not even need to be deployed. This is typically appropriate for local development clusters only. Even a shared development cluster that multiple developers use should have a decent level of reliability and robustness or else all the developers will be twiddling their thumbs whenever the cluster goes down unexpectedly.

Maintenance windows

In a system with maintenance windows, special times are dedicated to performing various maintenance activities, such as applying security patches, upgrading software, pruning log files, and database cleanups. With a maintenance window, the system (or a sub-system) becomes unavailable. This is typically planned for off-hours and often, users are notified. The benefit of maintenance windows is that you don't have to worry about how your maintenance actions are going to interact with live requests coming into the system. It can drastically simplify operations. System administrators and operators love maintenance windows just as much as developers love best effort systems.

The downside, of course, is that the system is down during maintenance. It may only be acceptable for systems where user activity is limited to certain times (e.g. US office hours or weekdays only).

With Kubernetes, you can do maintenance windows by redirecting all incoming requests via the load balancer to a web page (or JSON response) that notifies users about the maintenance window.

But in most cases, the flexibility of Kubernetes should allow you to do live maintenance. In extreme cases, such as upgrading the Kubernetes version, or the switch from etcd v2 to etcd v3, you may want to resort to a maintenance window. Blue-green deployment is another alternative. But the larger the cluster, the more expansive the blue-green alternative because you must duplicate your entire production cluster, which is both costly and can cause you to run into problems like an insufficient quota.

Quick recovery

Quick recovery is another important aspect of highly available clusters. Something will go wrong at some point. Your unavailability clock starts running. How quickly can you get back to normal? **Mean time to recovery** (**MTTR**) is an important measure to track and ensure your system can deal adequately with disasters.

Sometimes it's not up to you. For example, if your cloud provider has an outage (and you didn't implement a federated cluster, as we will discuss later in *Chapter 11*, *Running Kubernetes on Multiple Clusters*), then you just have to sit and wait until they sort it out. But the most likely culprit is a problem with a recent deployment. There are, of course, time-related issues, and even calendar-related issues. Do you remember the leap-year bug that took down Microsoft Azure on February 29, 2012?

The poster boy of quick recovery is, of course, the blue-green deployment–if you keep the previous version running when the problem is discovered. But, that's usually good for problems that happen during deployment or shortly after. If a sneaky bug lays dormant and is discovered only hours after the deployment, then you will have torn down your blue deployment already and you will not be able to revert to it.

On the other hand, rolling updates mean that if the problem is discovered early, then most of your pods still run the previous version.

Data-related problems can take a long time to reverse, even if your backups are up to date and your restore procedure actually works (definitely test this regularly).

Tools like Velero can help in some scenarios by creating a snapshot backup of your cluster that you can just restore, in case something goes wrong and you're not sure how to fix it.

Zero downtime

Finally, we arrive at the zero-downtime system. There is no such thing as a system that truly has zero downtime. All systems fail and all software systems definitely fail. The reliability of a system is often measured in the "number of nines." See `https://en.wikipedia.org/wiki/High_availability#%22Nines%22`.

Sometimes the failure is serious enough that the system or some of its services will be down. Think about zero downtime as a best-effort distributed system design. You design for zero downtime in the sense that you provide a lot of redundancy and mechanisms to address expected failures without bringing the system down. As always, remember that, even if there is a business case for zero downtime, it doesn't mean that every component must be zero downtime. Reliable (within reason) systems can be constructed from highly unreliable components.

The plan for zero downtime is as follows:

- **Redundancy at every level:** This is a required condition. You can't have a single point of failure in your design because when it fails, your system is down.

- **Automated hot-swapping of failed components**: Redundancy is only as good as the ability of the redundant components to kick into action as soon as the original component has failed. Some components can share the load (for example, stateless web servers), so there is no need for explicit action. In other cases, such as the Kubernetes scheduler and controller manager, you need a leader election in place to make sure the cluster keeps humming along.
- **Tons of metrics, monitoring, and alerts to detect problems early**: Even with careful design, you may miss something or some implicit assumption might invalidate your design. Often, such subtle issues creep up on you and with enough attention, you may discover them before they become an all-out system failure. For example, suppose there is a mechanism in place to clean up old log files when disk space is over 90% full, but for some reason, it doesn't work. If you set an alert for when disk space is over 95% full, then you'll catch it and be able to prevent the system failure.
- **Tenacious testing before deployment to production**: Comprehensive tests have proven themselves as a reliable way to improve quality. It is hard work to have comprehensive tests for something as complicated as a large Kubernetes cluster running a massive distributed system, but you need it. What should you test? Everything. That's right. For zero downtime, you need to test both the application and the infrastructure together. Your 100% passing unit tests are a good start, but they don't provide much confidence that when you deploy your application on your production Kubernetes cluster, it will still run as expected. The best tests are, of course, on your production cluster after a blue-green deployment or identical cluster. In lieu of a full-fledged identical cluster, consider a staging environment with as much fidelity as possible to your production environment. Here is a list of tests you should run. Each of these tests should be comprehensive because if you leave something untested, it might be broken:
    - Unit tests
    - Acceptance tests
    - Performance tests
    - Stress tests
    - Rollback tests
    - Data restore tests
    - Penetration tests
- **Keep the raw data**: For many systems, the data is the most critical asset. If you keep the raw data, you can recover from any data corruption and processed data loss that happens later. This will not really help you with zero downtime because it can take a while to reprocess the raw data, but it will help with zero-data loss, which is often more important. The downside to this approach is that the raw data is often huge compared to the processed data. A good option may be to store the raw data in cheaper storage compared to the processed data.
- **Perceived uptime as a last resort**: OK. Some part of the system is down. You may still be able to maintain some level of service. In many situations, you may have access to a slightly stale version of the data or can let the user access some other part of the system. It is not a great user experience, but technically the system is still available.

Does that sound crazy? Good. Zero-downtime large-scale systems are hard (actually impossible). There is a reason why Microsoft, Google, Amazon, Facebook, and other big companies have tens of thousands of software engineers (combined) just working on infrastructure, operations, and making sure things are up and running.

Site reliability engineering

SRE is a real-world approach for operating reliable distributed systems. SRE embraces failures and works with **service-level indicators** (**SLIs**), **service-level objectives** (**SLOs**), and **service-level agreements** (**SLAs**). Each service has objectives such as latency below 50 milliseconds for 95% of requests. If a service violates its objectives, then the team focuses on fixing the issue before going back to work on new features and capabilities.

The beauty of SRE is that you get to play with the knobs for cost and performance. If you want to invest more in reliability, then be ready to pay for it in resources and development time.

Performance and data consistency

When you develop or operate distributed systems, the CAP theorem should always be in the back of your mind. CAP stands for consistency, availability, and partition tolerance:

- Consistency means that every read receives the most recent write or an error
- Availability means that every request receives a non-error response (but the response may be stale)
- Partition tolerance means the system continues to operate even when an arbitrary number of messages between nodes are dropped or delayed by the network

The theorem says that you can have at most two out of the three. Since any distributed system can suffer from a network partition, in practice you can choose between CP or AP. CP means that in order to remain consistent, the system will not be available in the event of a network partition. AP means that the system will always be available but might not be consistent. For example, reads from different partitions might return different results because one of the partitions didn't receive a write.

In this section, we have focused on highly available systems, which means AP. To achieve high availability, we must sacrifice consistency. But that doesn't mean that our system will have corrupt or arbitrary data. The key concept is eventual consistency. Our system may be a little bit behind and provide access to somewhat stale data, but eventually, you'll get what you expect.

When you start thinking in terms of eventual consistency, it opens the door to potentially significant performance improvements. For example, if some important value is updated frequently (let's say, every second), but you send its value only every minute, you have reduced your network traffic by a factor of 60 and you're on average only 30 seconds behind real-time updates. This is very significant. This is huge. You have just scaled your system to handle 60 times more users or requests with the same amount of resources.

As we discussed earlier, redundancy is key to highly available systems. However, there is tension between redundancy and cost. In the next section, we will discuss choosing and managing your cluster capacity.

Choosing and managing the cluster capacity

With Kubernetes' horizontal pod autoscaling, DaemonSets, StatefulSets, and quotas, we can scale and control our pods, storage, and other objects. However, in the end, we're limited by the physical (virtual) resources available to our Kubernetes cluster. If all your nodes are running at 100% capacity, you need to add more nodes to your cluster. There is no way around it. Kubernetes will just fail to scale. On the other hand, if you have very dynamic workloads, then Kubernetes can scale down your pods, but if you don't scale down your nodes correspondingly, you will still pay for the excess capacity. In the cloud, you can stop and start instances on demand. Combining it with the cluster autoscaler can solve the compute capacity problem automatically. That's the theory. In practice, there are always nuances.

Choosing your node types

The simplest solution is to choose a single node type with a known quantity of CPU, memory, and local storage. But that is typically not the most efficient and cost-effective solution. It makes capacity planning simple because the only question is how many nodes are needed. Whenever you add a node, you add a known quantity of CPU and memory to your cluster, but most Kubernetes clusters and components within the cluster handle different workloads. We may have a stream processing pipeline where many pods receive some data and process it in one place.

This workload is CPU-heavy and may or may not need a lot of memory. Some components, such as a distributed memory cache, need a lot of memory, but very little CPU. Other components, such as a Cassandra cluster, need multiple SSD disks attached to each node. Machine learning workloads can benefit from GPUs. In addition, cloud providers offer spot instances – nodes that are cheaper but may be snatched away from you if another customer is willing to pay the regular price.

At scale, costs start to add up and you should try to align your workloads with the configuration of the nodes they run on, which means multiple node pools with different node (instance) types.

For each type of node, you should assign proper labels and make sure that Kubernetes schedules the pods that are designed to run on that node type.

Choosing your storage solutions

Storage is a huge factor in scaling a cluster. There are three categories of scalable storage solutions:

- Roll your own
- Use your cloud platform storage solution
- Use an out-of-cluster solution

When you roll your own, you install some type of storage solution in your Kubernetes cluster. The benefits are flexibility and full control, but you have to manage and scale it yourself.

When you use your cloud platform storage solution, you get a lot out of the box, but you lose control, you typically pay more, and, depending on the service, you may be locked into that provider.

When you use an out-of-cluster solution, the performance and cost of data transfer may be much greater. You typically use this option if you need to integrate with an existing system.

Of course, large clusters may have multiple data stores from all categories. This is one of the most critical decisions you have to make, and your storage needs may change and evolve over time.

Trading off cost and response time

If money was not an issue, you could just over-provision your cluster. Every node would have the best hardware configuration available, you would have way more nodes than are needed to process your workloads, and you would have copious amounts of available storage. But guess what? Money is always an issue!

You may get by with over-provisioning when you're just starting and your cluster doesn't handle a lot of traffic. You may just run five nodes, even if two nodes are enough most of the time. However, multiply everything by 1,000, and someone from the finance department will come asking questions if you have thousands of idle machines and petabytes of empty storage.

OK. So, you measure and optimize carefully and you get 99.99999% utilization of every resource. Congratulations, you just created a system that can't handle an iota of extra load or the failure of a single node without dropping requests on the floor or delaying responses.

You need to find the middle ground. Understand the typical fluctuations of your workloads and consider the cost/benefit ratio of having excess capacity versus having reduced response time or processing ability.

Sometimes, if you have strict availability and reliability requirements, you can build redundancy into the system, and then you over-provision by design. For example, you want to be able to hot-swap a failed component with no downtime and no noticeable effects. Maybe you can't lose even a single transaction. In this case, you'll have a live backup for all critical components, and that extra capacity can be used to mitigate temporary fluctuations without any special actions.

Using multiple node configurations effectively

Effective capacity planning requires you to understand the usage patterns of your system and the load each component can handle. That may include a lot of data streams generated inside the system. When you have a solid understanding of the typical workloads via metrics, you can look at workflows and which components handle which parts of the load. Then you can compute the number of pods and their resource requirements. In my experience, there are some relatively fixed workloads, some workloads that vary predictably (such as office hours versus non-office hours), and then you have your completely crazy workloads that behave erratically. You have to plan accordingly for each workload, and you can design several families of node configurations that can be used to schedule pods that match a particular workload.

Benefiting from elastic cloud resources

Most cloud providers let you scale instances automatically, which is a perfect complement to Kubernetes' horizontal pod autoscaling. If you use cloud storage, it also grows magically without you having to do anything. However, there are some gotchas that you need to be aware of.

Autoscaling instances

All the big cloud providers have instance autoscaling in place. There are some differences, but scaling up and down based on CPU utilization is always available, and sometimes, custom metrics are available too. Sometimes, load balancing is offered as well. As you can see, there is some overlap with Kubernetes here.

If your cloud provider doesn't have adequate autoscaling with proper control and is not supported by the cluster autoscaler, it is relatively easy to roll your own, where you monitor your cluster resource usage and invoke cloud APIs to add or remove instances. You can extract the metrics from Kubernetes. Here is a diagram that shows how two new instances are added based on a CPU load monitor:

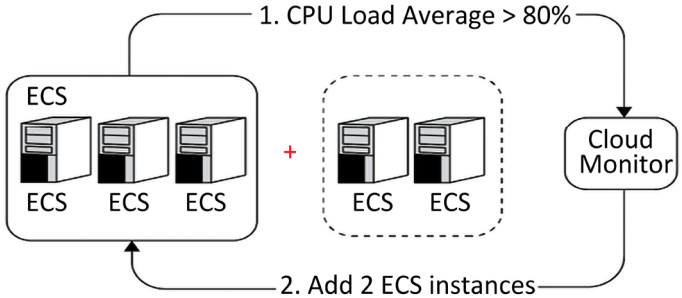


Figure 3.3: CPU-based autoscaling

Mind your cloud quotas

When working with cloud providers, some of the most annoying things are quotas. I've worked with four different cloud providers (AWS, GCP, Azure, and Alibaba Cloud) and I was always bitten by quotas at some point. The quotas exist to let the cloud providers do their own capacity planning (and also to protect you from inadvertently starting 1,000,000 instances that you won't be able to pay for), but from your point of view, it is yet one more thing that can trip you up. Imagine that you set up a beautiful autoscaling system that works like magic, and suddenly the system doesn't scale when you hit 100 nodes. You quickly discover that you are limited to 100 nodes and you open a support request to increase the quota. However, a human must approve quota requests, and that can take a day or two. In the meantime, your system is unable to handle the load.

Manage regions carefully

Cloud platforms are organized in regions and availability zones. The cost difference between regions can be up to 20% on cloud providers like GCP and Azure. On AWS, it may be even more extreme (30%-70%). Some services and machine configurations are available only in some regions.

Cloud quotas are also managed at the regional level. Performance and cost of data transfers within regions are much lower (often free) than across regions. When planning your clusters, you should carefully consider your geo-distribution strategy. If you need to run your workloads across multiple regions, you may have some tough decisions to make regarding redundancy, availability, performance, and cost.

Considering container-native solutions

A container-native solution is when your cloud provider offers a way to deploy containers directly into their infrastructure. You don't need to provision instances and then install a container runtime (like the Docker daemon) and only then deploy your containers. Instead, you just provide your containers and the platform is responsible for finding a machine to run your container. You are totally separated from the actual machines your containers are running on.

All the major cloud providers now provide solutions that abstract instances completely:

- AWS Fargate
- Azure Container Instances
- Google Cloud Run

These solutions are not Kubernetes-specific, but they can work great with Kubernetes. The cloud providers already provide a managed Kubernetes control plane with Google's **Google Kubernetes Engine** (**GKE**), Microsoft's **Azure Kubernetes Service** (**AKS**), and Amazon Web Services' **Elastic Kubernetes Service** (**EKS**). But managing the data plane (the nodes) was left to the cluster administrator. The container-native solution allows the cloud provider to do that on your behalf. Google Run for GKE, AKS with ACI, and AWS EKS with Fargate can manage both the control plane and the data plane.

For example, in AKS, you can provision virtual nodes. A virtual node is not backed up by an actual VM. Instead, it utilizes ACI to deploy containers when necessary. You pay for it only when the cluster needs to scale beyond the capacity of the regular nodes. It is faster to scale than using the cluster autoscaler that needs to provision an actual VM-backed node.

The following diagram illustrates this burst to the ACI approach:

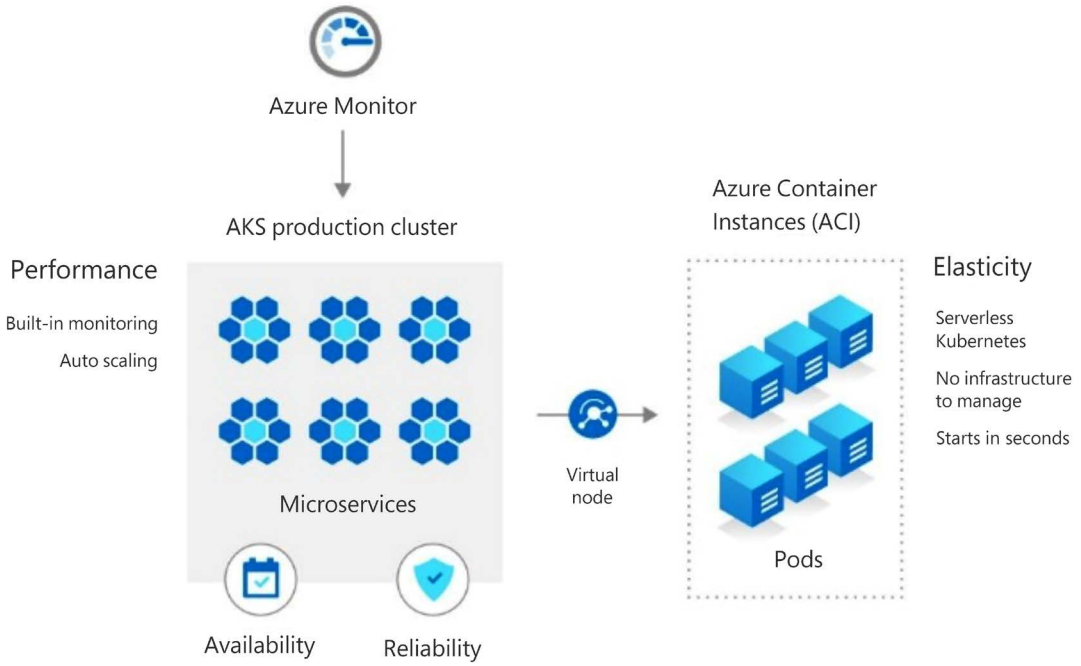


Figure 3.4: AKS and ACI

In this section, we looked at various factors that impact your decision regarding cluster capacity as well as cloud-provider solutions that do the heavy lifting on your behalf. In the next section, we will see how far you can stress a single Kubernetes cluster.

Pushing the envelope with Kubernetes

In this section, we will see how the Kubernetes team pushes Kubernetes to its limit. The numbers are quite telling, but some of the tools and techniques, such as Kubemark, are ingenious, and you may even use them to test your clusters. Kubernetes is designed to support clusters with the following properties:

- Up to 110 pods per node
- Up to 5,000 nodes
- Up to 150,000 pods
- Up to 300,000 total containers

Those numbers are just guidelines and not hard limits. Clusters that host specialized workloads with specialized deployment and runtime patterns regarding new pods coming and going may support very different numbers.

At CERN, the OpenStack team achieved 2 million requests per second: `http://superuser.openstack.org/articles/scaling-magnum-and-kubernetes-2-million-requests-per-second`.

Mirantis conducted a performance and scaling test in their scaling lab where they deployed 5,000 Kubernetes nodes (in VMs) on 500 physical servers.

OpenAI scaled their machine learning Kubernetes cluster to 2,500 nodes on Azure and learned some valuable lessons such as minding the query load of logging agents and storing events in a separate etcd cluster: `https://openai.com/research/scaling-kubernetes-to-2500-nodes`.

There are many more interesting use cases here: `https://www.cncf.io/projects/case-studies`.

By the end of this section, you'll appreciate the effort and creativeness that goes into improving Kubernetes on a large scale, you will know how far you can push a single Kubernetes cluster and what performance to expect, and you'll get an inside look at some tools and techniques that can help you evaluate the performance of your own Kubernetes clusters.

Improving the performance and scalability of Kubernetes

The Kubernetes team focused heavily on performance and scalability in Kubernetes 1.6. When Kubernetes 1.2 was released, it supported clusters of up to 1,000 nodes within the Kubernetes service-level objectives. Kubernetes 1.3 doubled the number to 2,000 nodes, and Kubernetes 1.6 brought it to a staggering 5,000 nodes per cluster. 5,000 nodes can carry you very far, especially if you use large nodes. But, when you run large nodes, you need to pay attention to pods per node guideline too. Note that cloud providers still recommend up to 1,000 nodes per cluster.

We will get into the numbers later, but first, let's look under the hood and see how Kubernetes achieved these impressive improvements.

Caching reads in the API server

Kubernetes keeps the state of the system in etcd, which is very reliable, though not super-fast (although etcd 3 delivered massive improvement specifically to enable larger Kubernetes clusters). The various Kubernetes components operate on snapshots of that state and don't rely on real-time updates. That fact allows the trading of some latency for throughput. All the snapshots used to be updated by etcd watches. Now, the API server has an in-memory read cache that is used for updating state snapshots. The in-memory read cache is updated by etcd watches. These schemes significantly reduce the load on etcd and increase the overall throughput of the API server.

The pod lifecycle event generator

Increasing the number of nodes in a cluster is key for horizontal scalability, but pod density is crucial too. Pod density is the number of pods that the kubelet can manage efficiently on one node. If pod density is low, then you can't run too many pods on one node. That means that you might not benefit from more powerful nodes (more CPU and memory per node) because the kubelet will not be able to manage more pods. The other alternative is to force the developers to compromise their design and create coarse-grained pods that do more work per pod. Ideally, Kubernetes should not force your hand when it comes to pod granularity. The Kubernetes team understands this very well and invested a lot of work in improving pod density.

In Kubernetes 1.1, the official (tested and advertised) number was 30 pods per node. I actually ran 40 pods per node on Kubernetes 1.1, but I paid for it in excessive kubelet overhead that stole CPU from the worker pods. In Kubernetes 1.2, the number jumped to 100 pods per node. The kubelet used to poll the container runtime constantly for each pod in its own goroutine. That put a lot of pressure on the container runtime that, during peaks to performance, has reliability issues, in particular CPU utilization. The solution was the **Pod Lifecycle Event Generator** (**PLEG**). The way the PLEG works is that it lists the state of all the pods and containers and compares it to the previous state. This is done once for all the pods and containers. Then, by comparing the state to the previous state, the PLEG knows which pods need to sync again and invokes only those pods. That change resulted in a significant four-times-lower CPU usage by the kubelet and the container runtime. It also reduced the polling period, which improves responsiveness.

The following diagram shows the CPU utilization for 120 pods on Kubernetes 1.1 versus Kubernetes 1.2. You can see the 4X factor very clearly:

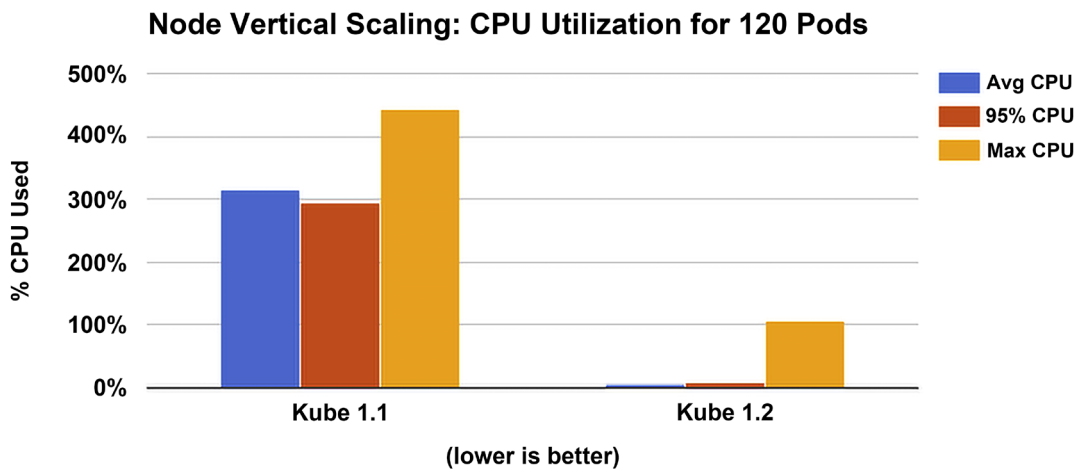


Figure 3.5: CPU utilization for 120 pods with Kube 1.1 and Kube 1.2

Serializing API objects with protocol buffers

The API server has a REST API. REST APIs typically use JSON as their serialization format, and the Kubernetes API server was no different. However, JSON serialization implies marshaling and unmarshaling JSON to native data structures. This is an expensive operation. In a large-scale Kubernetes cluster, a lot of components need to query or update the API server frequently. The cost of all that JSON parsing and composition adds up quickly. In Kubernetes 1.3, the Kubernetes team added an efficient protocol buffers serialization format. The JSON format is still there, but all internal communication between Kubernetes components uses the protocol buffers serialization format.

etcd3

Kubernetes switched from etcd2 to etcd3 in Kubernetes 1.6. This was a big deal. Scaling Kubernetes to 5,000 nodes wasn't possible due to limitations of etcd2, especially related to the watch implementation. The scalability needs of Kubernetes drove many of the improvements of etcd3, as CoreOS used Kubernetes as a measuring stick. Some of the big ticket items are talked about here.

gRPC instead of REST

etcd2 has a REST API, and etcd3 has a gRPC API (and a REST API via gRPC gateway). The HTTP/2 protocol at the base of gRPC can use a single TCP connection for multiple streams of requests and responses.

Leases instead of TTLs

etcd2 uses **Time to Live** (**TTL**) per key as the mechanism to expire keys, while etcd3 uses leases with TTLs where multiple keys can share the same key. This significantly reduces keep-alive traffic.

Watch implementation

The watch implementation of etcd3 takes advantage of gRPC bidirectional streams and maintains a single TCP connection to send multiple events, which reduced the memory footprint by at least an order of magnitude.

State storage

With etcd3, Kubernetes started storing all the state as protocol buffers, which eliminated a lot of wasteful JSON serialization overhead.

Other optimizations

The Kubernetes team made many other optimizations such as:

- Optimizing the scheduler (which resulted in 5-10x higher scheduling throughput)
- Switching all controllers to a new recommended design using shared informers, which reduced resource consumption of controller-manager
- Optimizing individual operations in the API server (conversions, deep copies, and patch)
- Reducing memory allocation in the API server (which significantly impacts the latency of API calls)

Measuring the performance and scalability of Kubernetes

In order to improve performance and scalability, you need a sound idea of what you want to improve and how you're going to measure the improvements. You must also make sure that you don't violate basic properties and guarantees in the quest for improved performance and scalability. What I love about performance improvements is that they often buy you scalability improvements for free. For example, if a pod needs 50% of the CPU of a node to do its job and you improve performance so that the pod can do the same work using 33% CPU, then you can suddenly run three pods instead of two on that node, and you've improved the scalability of your cluster by 50% overall (or reduced your cost by 33%).

The Kubernetes SLOs

Kubernetes has **service level objectives** (**SLOs**). Those guarantees must be respected when trying to improve performance and scalability. Kubernetes has a one-second response time for API calls (99 percentile). That's 1,000 milliseconds. It actually achieves an order of magnitude faster response times most of the time.

Measuring API responsiveness

The API has many different endpoints. There is no simple API responsiveness number. Each call has to be measured separately. In addition, due to the complexity and the distributed nature of the system, not to mention networking issues, there can be a lot of volatility in the results. A solid methodology is to break the API measurements into separate endpoints and then run a lot of tests over time and look at percentiles (which is standard practice).

It's also important to use enough hardware to manage a large number of objects. The Kubernetes team used a 32-core VM with 120 GB for the master in this test.

The following diagram describes the 50th, 90th, and 99th percentile of various important API call latencies for Kubernetes 1.3. You can see that the 90th percentile is very low, below 20 milliseconds. Even the 99th percentile is less than 125 milliseconds for the `DELETE` pods operation and less than 100 milliseconds for all other operations:

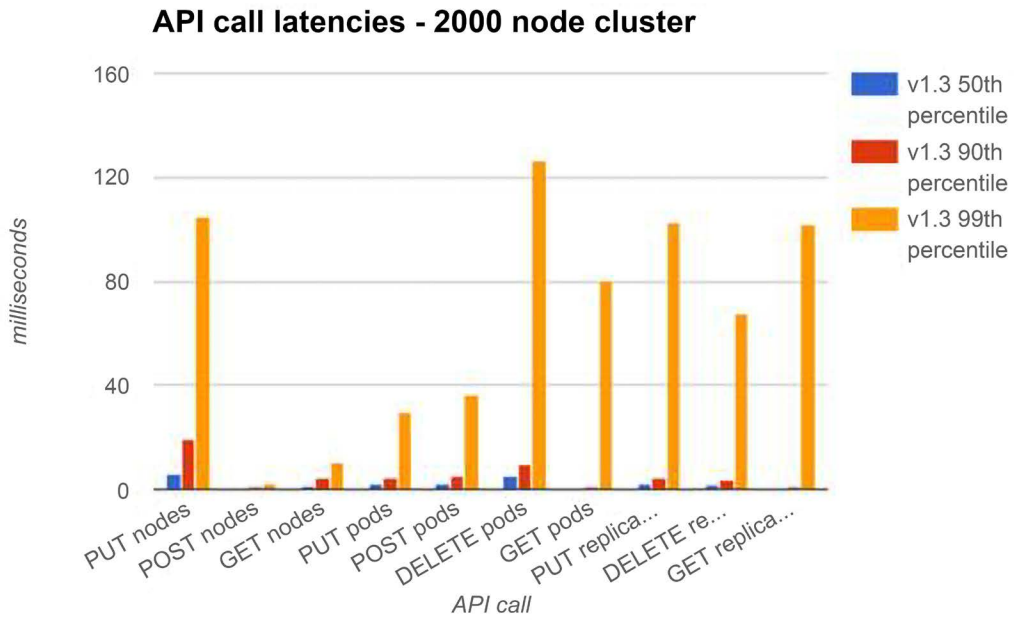


Figure 3.6: API call latencies

Another category of API calls is `LIST` operations. Those calls are more expansive because they need to collect a lot of information in a large cluster, compose the response, and send a potentially large response. This is where performance improvements such as the in-memory read cache and the protocol buffers serialization really shine. The response time is understandably greater than the single API calls, but it is still way below the SLO of one second (1,000 milliseconds):

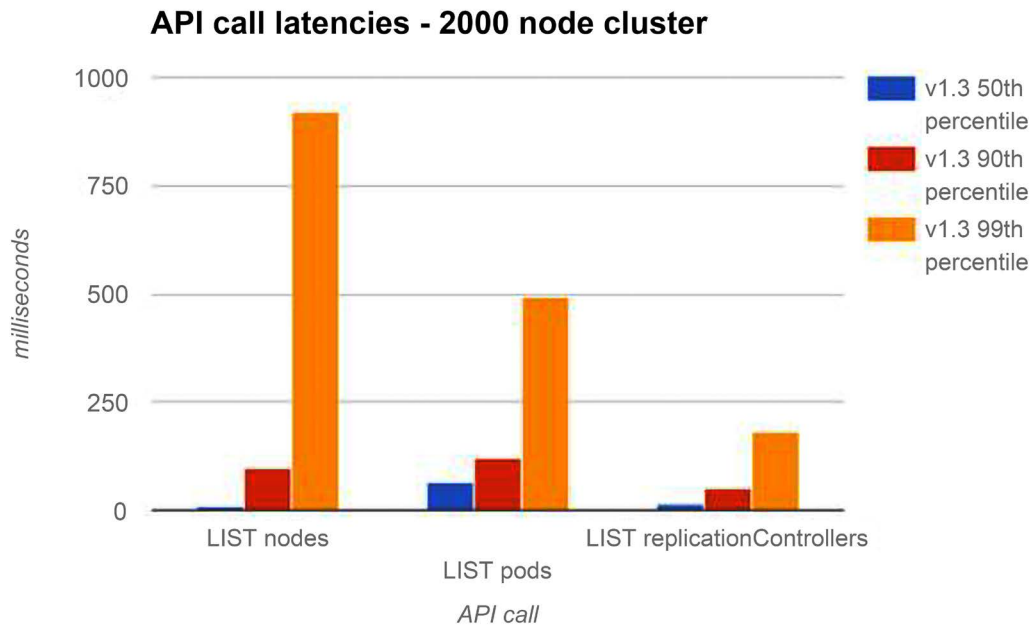


Figure 3.7: LIST API call latencies

Measuring end-to-end pod startup time

One of the most important performance characteristics of a large dynamic cluster is end-to-end pod startup time. Kubernetes creates, destroys, and shuffles pods around all the time. You could say that the primary function of Kubernetes is to schedule pods. In the following diagram, you can see that pod startup time is less volatile than API calls. This makes sense since there is a lot of work that needs to be done, such as launching a new instance of a runtime, that doesn't depend on cluster size. With Kubernetes 1.2 on a 1,000-node cluster, the 99th percentile end-to-end time to launch a pod was less than 3 seconds. With Kubernetes 1.3, the 99th percentile end-to-end time to launch a pod was a little over 2.5 seconds.

It's remarkable that the time is very close, but a little better with Kubernetes 1.3, on a 2,000-node cluster versus a 1,000-node cluster:

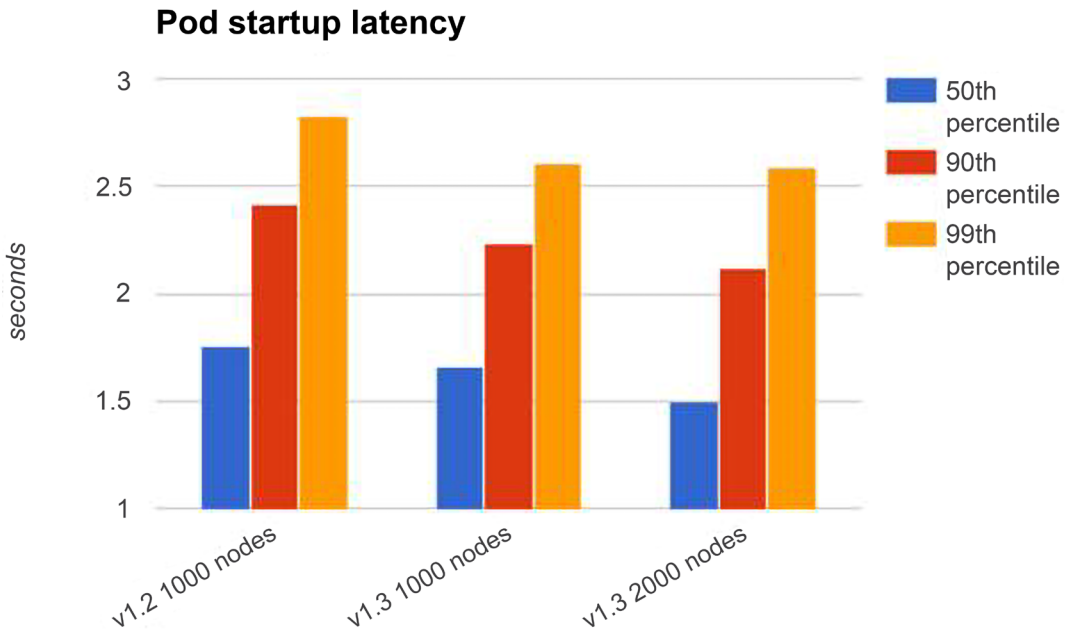


Figure 3.8: Pod startup latencies 1

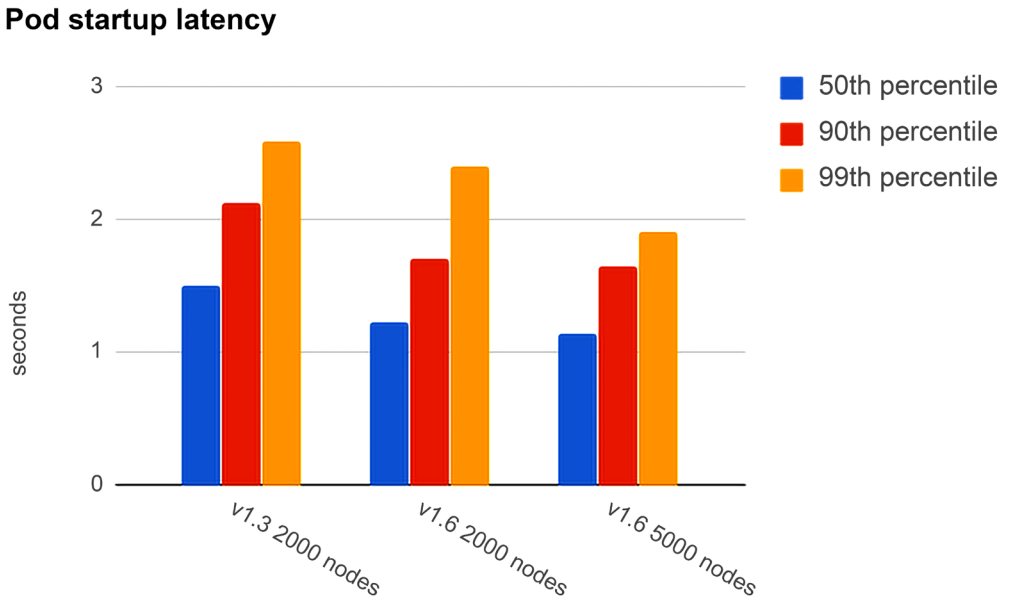


Figure 3.9: Pod startup latencies 2

In this section, we looked at how far we can push Kubernetes and how clusters with a large number of nodes perform. In the next section, we will examine some of the creative ways the Kubernetes developers test Kubernetes at scale.

Testing Kubernetes at scale

Clusters with thousands of nodes are expensive. Even a project such as Kubernetes that enjoys the support of Google and other industry giants still needs to come up with reasonable ways to test without breaking the bank.

The Kubernetes team runs a full-fledged test on a real cluster at least once per release to collect real-world performance and scalability data. However, there is also a need for a lightweight and cheaper way to experiment with potential improvements and detect regressions. Enter Kubemark.

Introducing the Kubemark tool

Kubemark is a Kubernetes cluster that runs mock nodes called hollow nodes used for running lightweight benchmarks against large-scale (hollow) clusters. Some of the Kubernetes components that are available on a real node such as the kubelet are replaced with a hollow kubelet. The hollow kubelet fakes a lot of the functionality of a real kubelet. A hollow kubelet doesn't actually start any containers, and it doesn't mount any volumes. But from the Kubernetes point of view – the state stored in etcd – all those objects exist and you can query the API server. The hollow kubelet is actually the real kubelet with an injected mock Docker client that doesn't do anything.

Another important hollow component is the hollow proxy, which mocks the kube-proxy component. It again uses the real kube-proxy code with a mock proxier interface that does nothing and avoids touching iptables.

Setting up a Kubemark cluster

A Kubemark cluster uses the power of Kubernetes. To set up a Kubemark cluster, perform the following steps:

1. Create a regular Kubernetes cluster where we can run *N* hollow nodes.
2. Create a dedicated VM to start all master components for the Kubemark cluster.
3. Schedule *N* hollow node pods on the base Kubernetes cluster. Those hollow nodes are configured to talk to the Kubemark API server running on the dedicated VM.
4. Create add-on pods by scheduling them on the base cluster and configuring them to talk to the Kubemark API server.

A full-fledged guide is available here:

`https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scalability/kubemark-guide.md`

Comparing a Kubemark cluster to a real-world cluster

The performance of Kubemark clusters is mostly similar to the performance of real clusters. For the pod startup end-to-end latency, the difference is negligible. For the API responsiveness, the differences are greater, though generally less than a factor of two. However, trends are exactly the same: an improvement/regression on a real cluster is visible as a similar percentage drop/increase in metrics on Kubemark.

Summary

In this chapter, we looked at reliable and highly available large-scale Kubernetes clusters. This is arguably the sweet spot for Kubernetes. While it is useful to be able to orchestrate a small cluster running a few containers, it is not necessary, but at scale, you must have an orchestration solution in place that you can trust to scale with your system and provide the tools and the best practices to do that.

You now have a solid understanding of the concepts of reliability and high availability in distributed systems. You delved into the best practices for running reliable and highly available Kubernetes clusters. You explored the complex issues surrounding scaling Kubernetes clusters and measuring their performance. You can now make wise design choices regarding levels of reliability and availability, as well as their performance and cost.

In the next chapter, we will address the important topic of security in Kubernetes. We will also discuss the challenges of securing Kubernetes and the risks involved. We will learn all about namespaces, service accounts, admission control, authentication, authorization, and encryption.

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`



4

Securing Kubernetes

In *Chapter 3*, *High Availability and Reliability*, we looked at reliable and highly available Kubernetes clusters, the basic concepts, the best practices, and the many design trade-offs regarding scalability, performance, and cost.

In this chapter, we will explore the important topic of security. Kubernetes clusters are complicated systems composed of multiple layers of interacting components. The isolation and compartmentalization of different layers is very important when running critical applications. To secure a system and ensure proper access to resources, capabilities, and data, we must first understand the unique challenges facing Kubernetes as a general-purpose orchestration platform that runs unknown workloads. Then we can take advantage of various securities, isolation, and access control mechanisms to make sure the cluster, the applications running on it, and the data are all safe. We will discuss various best practices and when it is appropriate to use each mechanism.

This chapter will explore the following main topics:

- Understanding Kubernetes security challenges
- Hardening Kubernetes
- Running multi-tenant clusters

By the end of this chapter, you will have a good understanding of Kubernetes security challenges. You will gain practical knowledge of how to harden Kubernetes against various potential attacks, establishing defense in depth, and will even be able to safely run a multi-tenant cluster while providing different users full isolation as well as full control over their part of the cluster.

Understanding Kubernetes security challenges

Kubernetes is a very flexible system that manages very low-level resources in a generic way. Kubernetes itself can be deployed on many operating systems and hardware or virtual machine solutions, on-premises, or in the cloud. Kubernetes runs workloads implemented by runtimes it interacts with through a well-defined runtime interface, but without understanding how they are implemented. Kubernetes manipulates critical resources such as networking, DNS, and resource allocation on behalf of or in service of applications it knows nothing about.

This means that Kubernetes is faced with the difficult task of providing good security mechanisms and capabilities in a way that application developers and cluster administrators can utilize, while protecting itself, the developers, and the administrators from common mistakes.

In this section, we will discuss security challenges in several layers or components of a Kubernetes cluster: nodes, network, images, pods, and containers. Defense in depth is an important security concept that requires systems to protect themselves at each level, both to mitigate attacks that penetrate other layers and to limit the scope and damage of a breach. Recognizing the challenges in each layer is the first step toward defense in depth.

This is often described as the 4 Cs of cloud-native security:

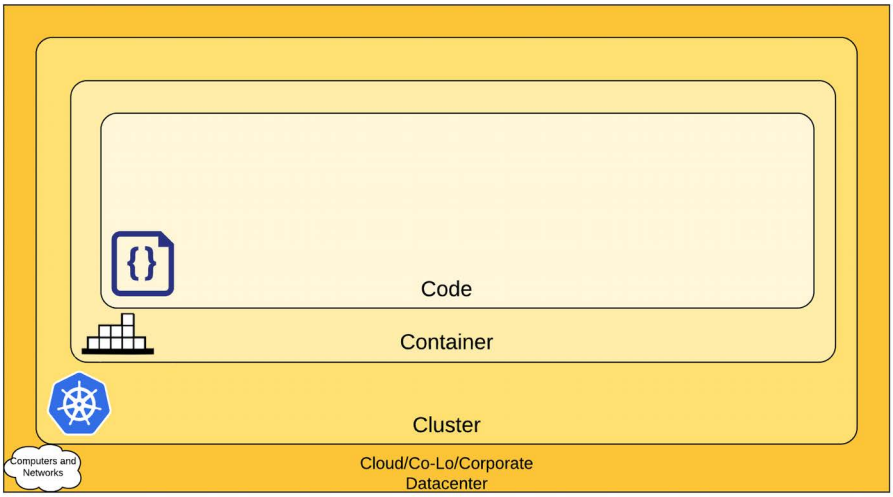


Figure 4.1: The 4 Cs of cloud-native security

However, the 4 Cs model is a coarse-grained approach for security. Another approach is building a threat model based on the security challenges across different dimensions, such as:

- Node challenges
- Network challenges
- Image challenges
- Deployment and configuration challenges
- Pod and container challenges
- Organizational, cultural, and process challenges

Let's examine these challenges.

Node challenges

The nodes are the hosts of the runtime engines. If an attacker gets access to a node, this is a serious threat. They can control at least the host itself and all the workloads running on it. But it gets worse.

The node has a kubelet running that talks to the API server. A sophisticated attacker can replace the kubelet with a modified version and effectively evade detection by communicating normally with the Kubernetes API server while running their own workloads instead of the scheduled workloads, collecting information about the overall cluster, and disrupting the API server and the rest of the cluster by sending malicious messages. The node will have access to shared resources and to secrets that may allow the attacker to infiltrate even deeper. A node breach is very serious, because of both the possible damage and the difficulty of detecting it after the fact.

Nodes can be compromised at the physical level too. This is more relevant on bare-metal machines where you can tell which hardware is assigned to the Kubernetes cluster.

Another attack vector is resource drain. Imagine that your nodes become part of a bot network that, unrelated to your Kubernetes cluster, just runs its own workloads like cryptocurrency mining and drains CPU and memory. The danger here is that your cluster will choke and run out of resources to run your workloads or, alternatively, your infrastructure may scale automatically and allocate more resources.

Another problem is the installation of debugging and troubleshooting tools or modifying the configuration outside of an automated deployment. Those are typically untested and, if left behind and active, can lead to at least degraded performance, but can also cause more sinister problems. At the least, it increases the attack surface.

Where security is concerned, it's a numbers game. You want to understand the attack surface of the system and where you're vulnerable. Let's list some possible node challenges:

- An attacker takes control of the host
- An attacker replaces the kubelet
- An attacker takes control of a node that runs master components (such as the API server, scheduler, or controller manager)
- An attacker gets physical access to a node
- An attacker drains resources unrelated to the Kubernetes cluster
- Self-inflicted damage occurs through the installation of debugging and troubleshooting tools or a configuration change

Mitigating node challenges requires several layers of defense such as controlling physical access, preventing privilege escalation, and reducing the attack surface by controlling the OS and software installed on the node.

Network challenges

Any non-trivial Kubernetes cluster spans at least one network. There are many challenges related to networking. You need to understand how your system components are connected at a very fine level. Which components are supposed to talk to each other? What network protocols do they use? What ports? What data do they exchange? How is your cluster connected to the outside world?

There is a complex chain of exposing ports and capabilities or services:

- Container to host
- Host to host within the internal network
- Host to the world

Using overlay networks (which will be discussed more in *Chapter 10*, *Exploring Kubernetes Networking*) can help with defense in depth where, even if an attacker gains access to a container, they are sandboxed and can't escape to the underlay network's infrastructure.

Discovering components is a big challenge too. There are several options here, such as DNS, dedicated discovery services, and load balancers. Each comes with a set of pros and cons that take careful planning and insight to get right for your situation. Making sure two containers can find each other and exchange information is not trivial.

You need to decide which resources and endpoints should be publicly accessible. Then you need to come up with a proper way to authenticate users, services, and authorize them to operate on resources. Often you may want to control access between internal services too.

Sensitive data must be encrypted on the way into and out of the cluster and sometimes at rest, too. That means key management and safe key exchange, which is one of the most difficult problems to solve in security.

If your cluster shares networking infrastructure with other Kubernetes clusters or non-Kubernetes processes, then you have to be diligent about isolation and separation.

The ingredients are network policies, firewall rules, and **software-defined networking** (**SDN**). The recipe is often customized. This is especially challenging with on-premises and bare-metal clusters. Here are some of the network challenges you will face:

- Coming up with a connectivity plan
- Choosing components, protocols, and ports
- Figuring out dynamic discovery
- Public versus private access
- Authentication and authorization (including between internal services)
- Designing firewall rules
- Deciding on a network policy
- Key management and exchange
- Encrypted communication

There is a constant tension between making it easy for containers, users, and services to find and talk to each other at the network level versus locking down access and preventing attacks through the network or attacks on the network itself.

Many of these challenges are not Kubernetes-specific. However, the fact that Kubernetes is a generic platform that manages key infrastructure and deals with low-level networking makes it necessary to think about dynamic and flexible solutions that can integrate system-specific requirements into Kubernetes. These solutions often involve monitoring and automatically injecting firewall rules or applying network policies based on namespaces and pod labels.

Image challenges

Kubernetes runs containers that comply with one of its runtime engines. It has no idea what these containers are doing (except collecting metrics). You can put certain limits on containers via quotas. You can also limit their access to other parts of the network via network policies. But, in the end, containers do need access to host resources, other hosts in the network, distributed storage, and external services. The image determines the behavior of a container. The infamous software supply chain problem is at the core of how these container images are created. There are two categories of problems with images:

- Malicious images
- Vulnerable images

Malicious images are images that contain code or configuration that was designed by an attacker to do some harm, collect information, or just take advantage of your infrastructure for their purposes (for example, crypto mining). Malicious code can be injected into your image preparation pipeline, including any image repositories you use. Alternatively, you may install third-party images that were compromised themselves and now contain malicious code.

Vulnerable images are images you designed (or third-party images you install) that just happen to contain some vulnerability that allows an attacker to take control of the running container or cause some other harm, including injecting their own code later.

It's hard to tell which category is worse. At the extreme, they are equivalent because they allow seizing total control of the container. The other defenses that are in place (remember defense in depth?) and the restrictions you put on the container will determine how much damage it can do. Minimizing the danger of bad images is very challenging. Fast-moving companies utilizing microservices may generate many images daily. Verifying an image is not an easy task either. In addition, some containers require wide permissions to do their legitimate job. If such a container is compromised it can do extreme damage.

The base images that contain the operating system may become vulnerable any time a new vulnerability is discovered. Moreover, if you rely on base images prepared by someone else (very common) then malicious code may find its way into those base images, which you have no control over and you trust implicitly.

When a vulnerability in a third-party dependency is discovered, ideally there is already a fixed version and you should patch it as soon as possible.

We can summarize the image challenges that developers are likely to face as follows:

- Kubernetes doesn't know what containers are doing
- Kubernetes must provide access to sensitive resources for the designated function
- It's difficult to protect the image preparation and delivery pipeline (including image repositories)
- The speed of development and deployment of new images conflict with the careful review of changes
- Base images that contain the OS or other common dependencies can easily get out of date and become vulnerable
- Base images are often not under your control and might be more prone to the injection of malicious code

Integrating a static image analyzer like CoreOS Clair or the Anchore Engine into your CI/CD pipeline can help a lot. In addition, minimizing the blast radius by limiting the resource access of containers only to what they need to perform their job can reduce the impact on your system if a container gets compromised. You must also be diligent about patching known vulnerabilities.

Configuration and deployment challenges

Kubernetes clusters are administered remotely. Various manifests and policies determine the state of the cluster at each point in time. If an attacker gets access to a machine with administrative control over the cluster, they can wreak havoc, such as collecting information, injecting bad images, weakening security, and tampering with logs. As usual, bugs and mistakes can be just as harmful; by neglecting important security measures, you leave the cluster open for attack. It is very common these days for employees with administrative access to the cluster to work remotely from home or from a coffee shop and have their laptops with them, where you are just one kubectl command from opening the floodgates.

Let's reiterate the challenges:

- Kubernetes is administered remotely
- An attacker with remote administrative access can gain complete control over the cluster
- Configuration and deployment is typically more difficult to test than code
- Remote or out-of-office employees risk extended exposure, allowing an attacker to gain access to their laptops or phones with administrative access

There are some best practices to minimize this risk, such as a layer of indirection in the form of a jump box where a developer connects from the outside to a dedicated machine in the cluster with tight controls that manages secure interaction with internal services, requiring a VPN connection (which authenticates and encrypts all communication), and using multi-factor authentication and one-time passwords to protect against trivial password cracking attacks.

Pod and container challenges

In Kubernetes, pods are the unit of work and contain one or more containers. The pod is a grouping and deployment construct. But often, containers that are deployed together in the same pod interact through direct mechanisms. The containers all share the same localhost network and often share mounted volumes from the host. This easy integration between containers in the same pod can result in exposing parts of the host to all the containers. This might allow one bad container (either malicious or just vulnerable) to open the way for an escalated attack on other containers in the pod, later taking over the node itself and the entire cluster. Control plane add-ons are often collocated with control plane components and present that kind of danger, especially because many of them are experimental. The same goes for DaemonSets that run pods on every node. The practice of sidecar containers where additional containers are deployed in a pod along with your application container is very popular, especially with service meshes. This increases that risk because the sidecar containers are often outside your control, and if compromised, can provide access to your infrastructure.

Multi-container pod challenges include the following:

- The same pod containers share the localhost network
- The same pod containers sometimes share a mounted volume on the host filesystem
- Bad containers might poison other containers in the pod
- Bad containers have an easier time attacking the node if collocated with another container that accesses crucial node resources
- Experimental add-ons that are collocated with master components might be experimental and less secure
- Service meshes introduce a sidecar container that might become an attack vector

Consider carefully the interaction between containers running in the same pod. You should realize that a bad container might try to compromise its sibling containers in the same pod as its first attack. This means that you should be able to detect rogue containers injected into a pod (e.g. by a malicious admission control webhook or a compromised CI/CD pipeline). You should also apply the least privilege principle and minimize the damage such a rogue container can do.

Organizational, cultural, and process challenges

Security is often held in contrast to productivity. This is a normal trade-off and nothing to worry about. Traditionally, when developers and operations were separate, this conflict was managed at an organizational level. Developers pushed for more productivity and treated security requirements as the cost of doing business. Operations controlled the production environment and were responsible for access and security procedures. The DevOps movement brought down the wall between developers and operations. Now, speed of development often takes a front-row seat. Concepts such as continuous deployment deploying multiple times a day without human intervention were unheard of in most organizations. Kubernetes was designed for this new world of cloud-native applications. But, it was developed based on Google's experience.

Google had a lot of time and skilled experts to develop the proper processes and tooling to balance rapid deployments with security. For smaller organizations, this balancing act might be very challenging and security could be weakened by focusing too much on productivity.

The challenges facing organizations that adopt Kubernetes are as follows:

- Developers that control the operation of Kubernetes might be less security-oriented
- The speed of development might be considered more important than security
- Continuous deployment might make it difficult to detect certain security problems before they reach production
- Smaller organizations might not have the knowledge and expertise to manage security properly in Kubernetes clusters

There are no easy answers here. You should be deliberate in striking the right balance between security and agility. I recommend having a dedicated security team (or at least one person focused on security) participate in all planning meetings and advocate for security. It's important to bake security into your system from the get-go.

In this section, we reviewed the many challenges you face when you try to build a secure Kubernetes cluster. Most of these challenges are not specific to Kubernetes, but using Kubernetes means there is a large part of your system that is generic and unaware of what the system is doing.

This can pose problems when trying to lock down a system. The challenges are spread across different levels:

- Node challenges
- Network challenges
- Image challenges
- Configuration and deployment challenges
- Pod and container challenges
- Organizational and process challenges

In the next section, we will look at the facilities Kubernetes provides to address some of those challenges. Many of the challenges require solutions at the larger system scope. It is important to realize that just utilizing all of Kubernetes' security features is not enough.

Hardening Kubernetes

The previous section cataloged and listed the variety of security challenges facing developers and administrators deploying and maintaining Kubernetes clusters. In this section, we will hone in on the design aspects, mechanisms, and features offered by Kubernetes to address some of the challenges. You can get to a pretty good state of security by judicious use of capabilities such as service accounts, network policies, authentication, authorization, admission control, AppArmor, and secrets.

Remember that a Kubernetes cluster is one part of a bigger system that includes other software systems, people, and processes. Kubernetes can't solve all problems. You should always keep in mind general security principles, such as defense in depth, a need-to-know basis, and the principle of least privilege.

In addition, log everything you think may be useful in the event of an attack and have alerts for early detection when the system deviates from its state. It may be just a bug or it may be an attack. Either way, you want to know about it and respond.

Understanding service accounts in Kubernetes

Kubernetes has regular users that are managed outside the cluster for humans connecting to the cluster (for example, via the `kubectl` command), and it has service accounts.

Regular user accounts are global and can access multiple namespaces in the cluster. Service accounts are constrained to one namespace. This is important. It ensures namespace isolation, because whenever the API server receives a request from a pod, its credentials will apply only to its own namespace.

Kubernetes manages service accounts on behalf of the pods. Whenever Kubernetes instantiates a pod, it assigns the pod a service account unless the service account or the pod explicitly opted out by setting `automountServiceAccountToken` to `False`. The service account identifies all the pod processes when they interact with the API server. Each service account has a set of credentials mounted in a secret volume. Each namespace has a default service account called default. When you create a pod, it is automatically assigned the default service account unless you specify a different service account.

You can create additional service accounts if you want different pods to have different identities and permissions. Then you can bind different service accounts to different roles.

Create a file called `custom-service-account.yaml` with the following content:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-service-account
```

Now type the following:

```
$ kubectl create -f custom-service-account.yaml
serviceaccount/custom-service-account created
```

Here is the service account listed alongside the default service account:

```
$ kubectl get serviceaccounts
NAME                     SECRETS   AGE
custom-service-account   1         6s
default                  1         2m28s
```

Note that a secret was created automatically for your new service account:

```
$ kubectl get secret
NAME                                                  TYPE
DATA   AGE
custom-service-account-token-vbrbm   kubernetes.io/service-account-token   3
62s
```

```
default-token-m4nfk                              kubernetes.io/service-account-token
3          3m24s
```

To get more detail, type the following:

```
$ kubectl get serviceAccounts/custom-service-account -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2022-06-19T18:38:22Z"
  name: custom-service-account
  namespace: default
  resourceVersion: "784"
  uid: f70f70cf-5b42-4a46-a2ff-b07792bf1220
secrets:
- name: custom-service-account-token-vbrbm
```

You can see the secret itself, which includes a `ca.crt` file and a token, by typing the following:

```
$ kubectl get secret custom-service-account-token-vbrbm -o yaml
```

How does Kubernetes manage service accounts?

The API server has a dedicated component called the service account admission controller. It is responsible for checking, at pod creation time, if the API server has a custom service account and, if it does, that the custom service account exists. If there is no service account specified, then it assigns the default service account.

It also ensures the pod has `ImagePullSecrets`, which are necessary when images need to be pulled from a remote image registry. If the pod spec doesn't have any secrets, it uses the service account's `ImagePullSecrets`.

Finally, it adds a `volume` with an API token for API access and a `volumeSource` mounted at `/var/run/secrets/kubernetes.io/serviceaccount`.

The API token is created and added to the secret by another component called the **token controller** whenever a service account is created. The token controller also monitors secrets and adds or removes tokens wherever secrets are added to or removed from a service account.

The **service account controller** ensures the default service account exists for every namespace.

Accessing the API server

Accessing the API server requires a chain of steps that include authentication, authorization, and admission control. At each stage, the request may be rejected. Each stage consists of multiple plugins that are chained together.

The following diagram illustrates this:

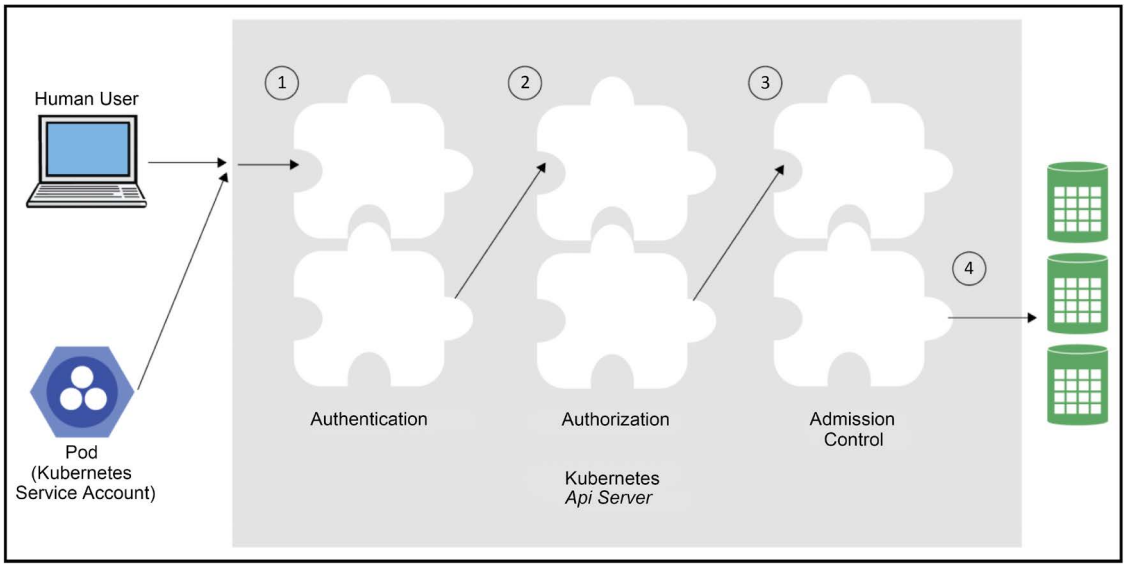


Figure 4.2: Accessing the API server

Authenticating users

When you first create the cluster, some keys and certificates are created for you to authenticate against the cluster. These credentials are typically stored in the file `~/.kube/config`, which may contain credentials for multiple clusters. You can also have multiple configuration files and control which file will be used by setting the `KUBECONFIG` environment variable or passing the `--kubeconfig` flag to kubectl. kubectl uses the credentials to authenticate itself to the API server and vice versa over TLS (an encrypted HTTPS connection). Let's create a new KinD cluster and store its credentials in a dedicated config file by setting the `KUBECONFIG` environment variable:

```
$ export KUBECONFIG=~/.kube/kind-config
$ kind create cluster
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.23.4) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-kind"
You can now use your cluster with:
```

```
kubectl cluster-info --context kind-kind
```

You can view your configuration using this command:

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://127.0.0.1:61022
  name: kind-kind
contexts:
- context:
    cluster: kind-kind
    user: kind-kind
  name: kind-kind
current-context: kind-kind
kind: Config
preferences: {}
users:
- name: kind-kind
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

This is the configuration for a KinD cluster. It may look different for other types of clusters.

Note that if multiple users need to access the cluster, the creator should provide the necessary client certificates and keys to the other users in a secure manner.

This is just establishing basic trust with the Kubernetes API server itself. You're not authenticated yet. Various authentication modules may look at the request and check for various additional client certificates, passwords, bearer tokens, and JWT tokens (for service accounts). Most requests require an authenticated user (either a regular user or a service account), although there are some anonymous requests too. If a request fails to authenticate with all the authenticators it will be rejected with a 401 HTTP status code (unauthorized, which is a bit of a misnomer).

The cluster administrator determines what authentication strategies to use by providing various command-line arguments to the API server:

- `--client-ca-file=` (for x509 client certificates specified in a file)
- `--token-auth-file=` (for bearer tokens specified in a file)
- `--basic-auth-file=` (for user/password pairs specified in a file)
- `--enable-bootstrap-token-auth` (for bootstrap tokens used by kubeadm)

Service accounts use an automatically loaded authentication plugin. The administrator may provide two optional flags:

- `--service-account-key-file=` (If not specified, the API server's TLS private key will be utilized as the PEM-encoded key for signing bearer tokens.)
- `--service-account-lookup` (When enabled, the revocation of tokens will take place if they are deleted from the API.)

There are several other methods, such as OpenID Connect, webhooks, Keystone (the OpenStack identity service), and an authenticating proxy. The main theme is that the authentication stage is extensible and can support any authentication mechanism.

The various authentication plugins will examine the request and, based on the provided credentials, will associate the following attributes:

- **Username** (a user-friendly name)
- **UID** (a unique identifier and more consistent than the username)
- **Groups** (a set of group names the user belongs to)
- **Extra fields** (these map string keys to string values)

In Kubernetes 1.11, kubectl gained the ability to use credential plugins to receive an opaque token from a provider such as an organizational LDAP server. These credentials are sent by kubectl to the API server that typically uses a webhook token authenticator to authenticate the credentials and accept the request.

The authenticators have no knowledge whatsoever of what a particular user is allowed to do. They just map a set of credentials to a set of identities. The authenticators run in an unspecified order; the first authenticator to accept the passed credentials will associate an identity with the incoming request and the authentication is considered successful. If all authenticators reject the credentials then authentication failed.

It's interesting to note that Kubernetes has no idea who its regular users are. There is no list of users in etcd. Authentication is granted to any user who presents a valid certificate that has been signed by the **certificate authority** (**CA**) associated with the cluster.

Impersonation

It is possible for users to impersonate different users (with proper authorization). For example, an admin may want to troubleshoot some issue as a different user with fewer privileges. This requires passing impersonation headers to the API request. The headers are as follows:

- **Impersonate-User**: Specifies the username to act on behalf of.
- **Impersonate-Group**: Specifies a group name to act on behalf of. Multiple groups can be provided by specifying this option multiple times. This option is optional but requires Impersonate-User to be set.
- **Impersonate-Extra-(extra name)**: A dynamic header used to associate additional fields with the user. This option is optional but requires Impersonate-User to be set.

With kubectl, you pass the `--as` and `--as-group` parameters.

To impersonate a service account, type the following:

```
kubectl --as system:serviceaccount:<namespace>:<service account name>
```

Authorizing requests

Once a user is authenticated, authorization commences. Kubernetes has generic authorization semantics. A set of authorization modules receives the request, which includes information such as the authenticated username and the request's verb (list, get, watch, create, and so on). Unlike authentication, all authorization plugins will get a shot at any request. If a single authorization plugin rejects the request or no plugin had an opinion then it will be rejected with a 403 HTTP status code (forbidden). A request will continue only if at least one plugin accepts it and no other plugin rejected it.

The cluster administrator determines what authorization plugins to use by specifying the `--authorization-mode` command-line flag, which is a comma-separated list of plugin names.

The following modes are supported:

- `--authorization-mode=AlwaysDeny` rejects all requests. Use if you don't need authorization.
- `--authorization-mode=AlwaysAllow` allows all requests. Use if you don't need authorization. This is useful during testing.
- `--authorization-mode=ABAC` allows for a simple, local-file-based, user-configured authorization policy. ABAC stands for Attribute-Based Access Control.
- `--authorization-mode=RBAC` is a role-based mechanism where authorization policies are stored and driven by the Kubernetes API. RBAC stands for Role-Based Access Control.
- `--authorization-mode=Node` is a special mode designed to authorize API requests made by kubelets.
- `--authorization-mode=Webhook` allows for authorization to be driven by a remote service using REST.

You can add your own custom authorization plugin by implementing the following straightforward Go interface:

```
type Authorizer interface {
    Authorize(ctx context.Context, a Attributes) (authorized Decision, reason
string, err error)
}
```

The `Attributes` input argument is also an interface that provides all the information you need to make an authorization decision:

```
type Attributes interface {
  GetUser() user.Info
  GetVerb() string
  IsReadOnly() bool
  GetNamespace() string
  GetResource() string
```

```
    GetSubresource() string
    GetName() string
    GetAPIGroup() string
    GetAPIVersion() string
    IsResourceRequest() bool
    GetPath() string
  }
```

You can find the source code at `https://github.com/kubernetes/apiserver/blob/master/pkg/authorization/authorizer/interfaces.go`.

Using the `kubectl can-i` command, you can check what actions you can perform and even impersonate other users:

```
$ kubectl auth can-i create deployments
Yes

$ kubectl auth can-i create deployments --as jack
no
```

kubectl supports plugins. We will discuss plugins in depth later in *Chapter 15*, *Extending Kubernetes*. In the meantime, I'll just mention that one of my favorite plugins is rolesum. This plugin gives you a summary of all the permissions a user or service account has. Here is an example:

```
$ kubectl rolesum job-controller -n kube-system
ServiceAccount: kube-system/job-controller
Secrets:
• */job-controller-token-tp72d

Policies:

• [CRB] */system:controller:job-controller ⟶  [CR] */system:controller:job-
controller
  Resource                Name  Exclude  Verbs  G L W C U P D DC
  events.[,events.k8s.io] [*]     [-]     [-]   ✗✗✗✓✓✓✗✗
  jobs.batch              [*]     [-]     [-]   ✓✓✓✗✓✓✗✗
  jobs.batch/finalizers   [*]     [-]     [-]   ✗✗✗✗✓✗✗✗
  jobs.batch/status       [*]     [-]     [-]   ✗✗✗✗✓✗✗✗
  pods                    [*]     [-]     [-]   ✗✓✓✓✗✓ ✓✗
```

Check it out here: `https://github.com/Ladicle/kubectl-rolesum`.

Using admission control plugins

OK. The request was authenticated and authorized, but there is one more step before it can be executed. The request must go through a gauntlet of admission-control plugins. Similar to the authorizers, if a single admission controller rejects a request, it is denied. Admission controllers are a neat concept. The idea is that there may be global cluster concerns that could be grounds for rejecting a request. Without admission controllers, all authorizers would have to be aware of these concerns and reject the request. But, with admission controllers, this logic can be performed once. In addition, an admission controller may modify the request. Admission controllers run in either validating mode or mutating mode. As usual, the cluster administrator decides which admission control plugins run by providing a command-line argument called `admission-control`. The value is a comma-separated and ordered list of plugins. Here is the list of recommended plugins for Kubernetes >= 1.9 (the order matters):

```
--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,Persisten
tVolumeLabel,DefaultStorageClass,MutatingAdmissionWebhook,ValidatingAdmissi-
onWebhook,ResourceQuota,DefaultTolerationSeconds
```

Let's look at some available plugins (more are added all the time):

- `DefaultStorageClass`: Adds a default storage class to requests for the creation of a `PersistentVolumeClaim` that doesn't specify a storage class.
- `DefaultTolerationSeconds`: Sets the default toleration of pods for taints (if not set already): `notready:NoExecute` and `notreachable:NoExecute`.
- `EventRateLimit`: Limits flooding of the API server with events.
- `ExtendedResourceToleration`: Combines taints on nodes with special resources such as GPUs and **Field Programmable Gate Arrays** (**FPGAs**) with toleration on pods that request those resources. The end result is that the node with the extra resources will be dedicated for pods with the proper toleration.
- `ImagePolicyWebhook`: This complicated plugin connects to an external backend to decide whether a request should be rejected based on the image.
- `LimitPodHardAntiAffinity`: In the `requiredDuringSchedulingRequiredDuringExecution` field, any pod that specifies an AntiAffinity topology key other than `kubernetes.io/hostname` will be denied.
- `LimitRanger`: Rejects requests that violate resource limits.
- `MutatingAdmissionWebhook`: Calls registered mutating webhooks that are able to modify their target object. Note that there is no guarantee that the change will be effective due to potential changes by other mutating webhooks.
- `NamespaceAutoProvision`: Creates the namespace in the request if it doesn't exist already.
- `NamespaceLifecycle`: Rejects object creation requests in namespaces that are in the process of being terminated or don't exist.
- `ResourceQuota`: Rejects requests that violate the namespace's resource quota.
- `ServiceAccount`: Automation for service accounts.
- `ValidatingAdmissionWebhook`: The admission controller invokes validating webhooks that match the request. The matching webhooks are called concurrently, and if any of them reject the request, the overall request fails.

As you can see, the admission control plugins have very diverse functionality. They support namespace-wide policies and enforce the validity of requests mostly from the resource management and security points of view. This frees up the authorization plugins to focus on valid operations. `ImagePolicyWebHook` is the gateway to validating images, which is a big challenge. `MutatingAdmissionWebhook` and `ValidatingAdmissionWebhook` are the gateways to dynamic admission control, where you can deploy your own admission controller without compiling it into Kubernetes. Dynamic admission control is suitable for tasks like semantic validation of resources (do all pods have the standard set of labels?). We will discuss dynamic admission control in depth later, in *Chapter 16*, *Governing Kubernetes*, as this is the foundation of policy management governance in Kubernetes.

The division of responsibility for validating an incoming request through the separate stages of authentication, authorization, and admission, each with its own plugins, makes a complicated process much more manageable to understand, use, and extend.

The mutating admission controllers provide a lot of flexibility and the ability to automatically enforce certain policies without burdening the users (for example, creating a namespace automatically if it doesn't exist).

Securing pods

Pod security is a major concern, since Kubernetes schedules the pods and lets them run. There are several independent mechanisms for securing pods and containers. Together these mechanisms support defense in depth, where, even if an attacker (or a mistake) bypasses one mechanism, it will get blocked by another.

Using a private image repository

This approach gives you a lot of confidence that your cluster will only pull images that you have previously vetted, and you can manage upgrades better. In light of the rise in software supply chain attacks it is an important countermeasure. You can configure your `HOME/.docker/config.json` on each node. But, on many cloud providers, you can't do this because nodes are provisioned automatically for you.

ImagePullSecrets

This approach is recommended for clusters on cloud providers. The idea is that the credentials for the registry will be provided by the pod, so it doesn't matter what node it is scheduled to run on. This circumvents the problem with `.dockercfg` at the node level.

First, you need to create a secret object for the credentials:

```
$ kubectl create secret docker-registry the-registry-secret \
    --docker-server=<docker registry server> \
    --docker-username=<username> \
    --docker-password=<password> \
    --docker-email=<email>
secret 'docker-registry-secret' created.
```

You can create secrets for multiple registries (or multiple users for the same registry) if needed. The kubelet will combine all `ImagePullSecrets`.

But, since pods can access secrets only in their own namespace, you must create a secret on each namespace where you want the pod to run. Once the secret is defined, you can add it to the pod spec and run some pods on your cluster. The pod will use the credentials from the secret to pull images from the target image registry:

```
apiVersion: v1
kind: Pod
metadata:
  name: cool-pod
  namespace: the-namespace
spec:
  containers:
 - name: cool-container
   image: cool/app:v1
   imagePullSecrets:
   - name: the-registry-secret
```

Specifying a security context for pods and containers

Kubernetes allows setting a security context at the pod level and additional security contexts at the container level. The pod security context is a set of operating-system-level security settings such as UID, GID, capabilities, and SELinux role. The pod security context can also apply its security settings (in particular, `fsGroup` and `seLinuxOptions`) to volumes.

Here is an example of a pod security context:

```
apiVersion: v1
kind: Pod
metadata:
name: some-pod
spec:
  securityContext:
    fsGroup: 1234
    supplementalGroups: [5678]
    seLinuxOptions:
      level: 's0:c123,c456'
  containers:
    ...
```

For the complete list of pod security context fields, check out `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#podsecuritycontext-v1-core`.

The container security context is applied to each container and it adds container-specific settings. Some fields of the container security context overlap with fields in the pod security context. If the container security context specifies these fields they override the values in the pod security context. Container context settings can't be applied to volumes, which remain at the pod level even if mounted into specific containers only.

Here is a pod with a container security context:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - name: some-container
    ...
    securityContext:
      privileged: true
      seLinuxOptions:
        level: 's0:c123,c456'
```

For the complete list of container security context fields, check out `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.24/#securitycontext-v1-core`.

Pod security standards

Kubernetes defines security profiles that are appropriate for different security needs and aggregates recommended settings. The privileged profile provides all permissions and is, unfortunately, the default. The baseline profile is a minimal security profile that just prevents privilege escalation. The restricted profile follows hardening best practices.

See more info here: `https://kubernetes.io/docs/concepts/security/pod-security-standards/`.

Protecting your cluster with AppArmor

AppArmor is a Linux kernel security module. With AppArmor, you can restrict a process running in a container to a limited set of resources such as network access, Linux capabilities, and file permissions. You configure AppArmor through profiles.

AppArmor requirements

AppArmor support was added as beta in Kubernetes 1.4. It is not available for every OS, so you must choose a supported OS distribution in order to take advantage of it. Ubuntu and SUSE Linux support AppArmor and enable it by default. Other distributions have optional support.

To check if AppArmor is enabled connect to a node (e.g. via `ssh`) and type the following:

```
$ cat /sys/module/apparmor/parameters/enabled
Y
```

If the result is `Y` then it's enabled. If the file doesn't exist or the result is not `Y` it is not enabled.

The profile must be loaded into the kernel. Check the following file:

```
/sys/kernel/security/apparmor/profiles
```

Kubernetes doesn't provide a built-in mechanism to load profiles to nodes. You typically need a DaemonSet with node-level privileges to load the necessary AppArmor profiles into the nodes.

Check out the following link for more details on loading AppArmor profiles into nodes: `https://kubernetes.io/docs/tutorials/security/apparmor/#setting-up-nodes-with-profiles`.

Securing a pod with AppArmor

Since AppArmor is still in beta, you specify the metadata as annotations and not as bonafide fields. When it gets out of beta, this will change.

To apply a profile to a container, add the following annotation:

```
container.apparmor.security.beta.kubernetes.io/<container name>: <profile
reference>
```

The profile reference can be either the `default` profile, `runtime/default`, or a profile file on the host/localhost.

Here is a sample profile that prevents writing to files:

```
> \#include \<tunables/global\>
>
> profile k8s-apparmor-example-deny-write flags=(attach\\\_disconnected)
> {
>
> \#include \<abstractions/base\>
>
> file,
>
> \# Deny all file writes.
>
> deny /\\\*\\\* w,
>
> }
```

AppArmor is not a Kubernetes resource, so the format is not the YAML or JSON you're familiar with.

To verify the profile was attached correctly, check the attributes of process 1:

```
kubectl exec <pod-name> cat /proc/1/attr/current
```

Pods can be scheduled on any node in the cluster by default. This means the profile should be loaded into every node. This is a classic use case for a DaemonSet.

Writing AppArmor profiles

Writing profiles for AppArmor by hand is not trivial. There are some tools that can help: `aa-genprof` and `aa-logprof` can generate a profile for you and assist in fine-tuning it by running your application with AppArmor in complain mode. The tools keep track of your application's activity and AppArmor warnings and create a corresponding profile. This approach works, but it feels clunky.

My favorite tool is bane, which generates AppArmor profiles from a simpler profile language based on the TOML syntax. bane profiles are very readable and easy to grasp. Here is a sample bane profile:

```
# name of the profile, we will auto prefix with `docker-`
# so the final profile name will be `docker-nginx-sample`
Name = "nginx-sample"

[Filesystem]
# read only paths for the container
ReadOnlyPaths = [
    "/bin/**",
    "/boot/**",
    "/dev/**",
    "/etc/**",
    "/home/**",
    "/lib/**",
    "/lib64/**",
    "/media/**",
    "/mnt/**",
    "/opt/**",
    "/proc/**",
    "/root/**",
    "/sbin/**",
    "/srv/**",
    "/tmp/**",
    "/sys/**",
    "/usr/**",
]

# paths where you want to log on write
LogOnWritePaths = [
```

```
    "/**"
]

# paths where you can write
WritablePaths = [
    "/var/run/nginx.pid"
]

# allowed executable files for the container
AllowExec = [
    "/usr/sbin/nginx"
]

# denied executable files
DenyExec = [
    "/bin/dash",
    "/bin/sh",
    "/usr/bin/top"
]

# allowed capabilities
[Capabilities]
Allow = [
    "chown",
    "dac_override",
    "setuid",
    "setgid",
    "net_bind_service"
]

[Network]
# if you don't need to ping in a container, you can probably
# set Raw to false and deny network raw
Raw = false
Packet = false
Protocols = [
    "tcp",
    "udp",
    "icmp"
]
```

The generated AppArmor profile is pretty gnarly (verbose and complicated).

You can find more information about bane here: `https://github.com/genuinetools/bane`.

Pod Security Admission

Pod Security Admission is an admission controller that is responsible for managing the Pod Security Standards (`https://kubernetes.io/docs/concepts/security/pod-security-standards/`). The pod security restrictions are applied at the namespace level. All pods in the target namespace will be checked for the same security profile (**privileged**, **baseline**, or **restricted**).

Note that Pod Security Admission doesn't set the relevant security contexts. It only validates that the pod conforms to the target policy.

There are three modes:

- `enforce`: Policy violations will result in the rejection of the pod.
- `audit`: Policy violations will result in the addition of an audit annotation to the event recorded in the audit log, but the pod will still be allowed.
- `warn`: Policy violations will trigger a warning for the user, but the pod will still be allowed.

To activate Pod Security Admission on a namespace you simply add a label to the target namespace:

```
$ MODE=warn # One of enforce, audit, or warn
$ LEVEL=baseline # One of privileged, baseline, or restricted
$ kubectl label namespace/ns-1 pod-security.kubernetes.io/${MODE}: ${LEVEL}

namespace/ns-1 created
```

Managing network policies

Node, pod, and container security are imperative, but it's not enough. Network segmentation is critical to design secure Kubernetes clusters that allow multi-tenancy, as well as to minimize the impact of security breaches. Defense in depth mandates that you compartmentalize parts of the system that don't need to talk to each other, while also carefully managing the direction, protocols, and ports of network traffic.

Network policies allow the fine-grained control and proper network segmentation of your cluster. At the core, a network policy is a set of firewall rules applied to a set of namespaces and pods selected by labels. This is very flexible because labels can define virtual network segments and be managed at the Kubernetes resource level.

This is a huge improvement over trying to segment your network using traditional approaches like IP address ranges and subnet masks, where you often run out of IP addresses or allocate too many just in case.

However, if you use a service mesh you may not need to use network policies since the service mesh can fulfill the same role. More on service meshes later, in *Chapter 14*, *Utilizing Service Meshes*.

Choosing a supported networking solution

Some networking backends (network plugins) don't support network policies. For example, the popular Flannel can't be used to apply policies. This is critical. You will be able to define network policies even if your network plugin doesn't support them. Your policies will simply have no effect, giving you a false sense of security. Here is a list of network plugins that support network policies (both ingress and egress):

- Calico
- WeaveNet
- Canal
- Cillium
- Kube-Router
- Romana
- Contiv

If you run your cluster on a managed Kubernetes service then the choice has already been made for you, although you can also install custom CNI plugins on some managed Kubernetes offerings.

We will explore the ins and outs of network plugins in *Chapter 10*, *Exploring Kubernetes Networking*. Here we focus on network policies.

Defining a network policy

You define a network policy using a standard YAML manifest. The supported protocols are TCP, UDP, and SCTP (since Kubernetes 1.20).

Here is a sample policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: the-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          project: cool-project
    - podSelector:
        matchLabels:
```

```
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
```

The `spec` part has two important parts, the `podSelector` and the `ingress`. The `podSelector` governs which pods this network policy applies to. The `ingress` governs which namespaces and pods can access these pods and which protocols and ports they can use.

In the preceding sample network policy, the pod selector specified the target for the network policy to be all the pods that are labeled `role: db`. The `ingress` section has a `from` sub-section with a namespace selector and a pod selector. All the namespaces in the cluster that are labeled `project: cool-project`, and within these namespaces, all the pods that are labeled `role: frontend`, can access the target pods labeled `role: db`. The `ports` section defines a list of pairs (protocol and port) that further restrict what protocols and ports are allowed. In this case, the protocol is `tcp` and the port is `6379` (the standard Redis port). If you want to target a range of ports, you can use `endPort`, as in:

```
  ports:
  - protocol: TCP
    port: 6379
    endPort: 7000
```

Note that the network policy is cluster-wide, so pods from multiple namespaces in the cluster can access the target namespace. The current namespace is always included, so even if it doesn't have the `project:cool` label, pods with `role:frontend` can still have access.

It's important to realize that the network policy operates in a whitelist fashion. By default, all access is forbidden, and the network policy can open certain protocols and ports to certain pods that match the labels. However, the whitelist nature of the network policy applies only to pods that are selected for at least one network policy. If a pod is not selected it will allow all access. Always make sure all your pods are covered by a network policy.

Another implication of the whitelist nature is that, if multiple network policies exist, then the unified effect of all the rules applies. If one policy gives access to port `1234` and another gives access to port `5678` for the same set of pods, then a pod may be accessed through either `1234` or `5678`.

To use network policies responsibly, consider starting with a `deny-all` network policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Then, start adding network policies to allow ingress to specific pods explicitly. Note that you must apply the `deny-all` policy for each namespace:

```
$ k create -n ${NAMESPACE} -f deny-all-network-policy.yaml
```

Limiting egress to external networks

Kubernetes 1.8 added egress network policy support, so you can control outbound traffic too. Here is an example that prevents access to the external IP `1.2.3.4`. The `order: 999` ensures the policy is applied before other policies:

```
apiVersion: networking.k8s.io/ v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
spec:
  order: 999
  egress:
  - action: deny
    destination:
      net: 1.2.3.4
    source: {}
```

Cross-namespace policies

If you divide your cluster into multiple namespaces, it can come in handy sometimes if pods can communicate across namespaces. You can specify the `ingress.namespaceSelector` field in your network policy to enable access from multiple namespaces. This is useful, for example, if you have production and staging namespaces and you periodically populate your staging environments with snapshots of your production data.

The costs of network policies

Network policies are not free. Your CNI plugin may install additional components in your cluster and on every node. These components use precious resources and in addition may cause your pods to get evicted due to insufficient capacity. For example, the Calico CNI plugin installs several deployments in the `kube-system` namespace:

```
$ k get deploy -n kube-system -o name | grep calico
deployment.apps/calico-node-vertical-autoscaler
deployment.apps/calico-typha
deployment.apps/calico-typha-horizontal-autoscaler
deployment.apps/calico-typha-vertical-autoscaler
```

It also provisions a DaemonSet that runs a pod on every node:

```
$ k get ds -n kube-system -o name | grep calico-node
daemonset.apps/calico-node
```

Using secrets

Secrets are paramount in secure systems. They can be credentials such as usernames and passwords, access tokens, API keys, certificates, or crypto keys. Secrets are typically small. If you have large amounts of data you want to protect, you should encrypt it and keep the encryption/decryption keys as secrets.

Storing secrets in Kubernetes

Kubernetes used to store secrets in etcd as plaintext by default. This means that direct access to etcd should be limited and carefully guarded. Starting with Kubernetes 1.7, you can now encrypt your secrets at rest (when they're stored by etcd).

Secrets are managed at the namespace level. Pods can mount secrets either as files via secret volumes or as environment variables. From a security standpoint, this means that any user or service that can create a pod in a namespace can have access to any secret managed for that namespace. If you want to limit access to a secret, put it in a namespace accessible to a limited set of users or services.

When a secret is mounted into a container, it is never written to disk. It is stored in `tmpfs`. When the kubelet communicates with the API server, it normally uses TLS, so the secret is protected in transit.

Kubernetes secrets are limited to 1 MB.

Configuring encryption at rest

You need to pass this argument when you start the API server: `--encryption-provider-config`.

Here is a sample encryption config:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - identity: {}
      - aesgcm:
          keys:
            - name: key1
              secret: c2VjcmV0IGlzIHNlY3VyZQ==
            - name: key2
              secret: dGhpcyBpcyBwYXNzd29yZA==
      - aescbc:
          keys:
            - name: key1
              secret: c2VjcmV0IGlzIHNlY3VyZQ==
            - name: key2
```

```
            secret: dGhpcyBpcyBwYXNzd29yZA==
      - secretbox:
          keys:
            - name: key1
              secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=
```

Creating secrets

Secrets must be created before you try to create a pod that requires them. The secret must exist; otherwise, the pod creation will fail.

You can create secrets with the following command: `kubectl create secret`.

Here I create a generic secret called `hush-hush`, which contains two keys, a username and password:

```
$ k create secret generic hush-hush \
    --from-literal=username=tobias  \
    --from-literal=password=cutoffs

secret/hush-hush created
```

The resulting secret is opaque:

```
$ k describe secrets/hush-hush
Name:         hush-hush
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
password:  7 bytes
username:  6 bytes
```

You can create secrets from files using `--from-file` instead `--from-literal`, and you can also create secrets manually if you encode the secret value as base64.

Key names inside a secret must follow the rules for DNS sub-domains (without the leading dot).

Decoding secrets

To get the content of a secret you can use `kubectl get secret`:

```
$ k get secrets/hush-hush -o yaml
apiVersion: v1
data:
```

```
  password: Y3V0b2Zmcw==
  username: dG9iaWFz
kind: Secret
metadata:
  creationTimestamp: "2022-06-20T19:49:56Z"
  name: hush-hush
  namespace: default
  resourceVersion: "51831"
  uid: 93e8d6d1-4c7f-4868-b146-32d1eb02b0a6
type: Opaque
```

The values are base64-encoded. You need to decode them yourself:

```
$ k get secrets/hush-hush -o jsonpath='{.data.password}' | base64 --decode

cutoffs
```

Using secrets in a container

Containers can access secrets as files by mounting volumes from the pod. Another approach is to access the secrets as environment variables. Finally, a container (given that its service account has the permission) can access the Kubernetes API directly or use `kubectl get secret`.

To use a secret mounted as a volume, the pod manifest should declare the volume and it should be mounted in the container's spec:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-secret
spec:
  containers:
  - name: container-with-secret
    image: g1g1/py-kube:0.3
    command: ["/bin/bash", "-c", "while true ; do sleep 10 ; done"]
    volumeMounts:
    - name: secret-volume
      mountPath: "/mnt/hush-hush"
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: hush-hush
```

The volume name (secret-volume) binds the pod volume to the mount in the container. Multiple containers can mount the same volume. When this pod is running, the username and password are available as files under `/etc/hush-hush`:

```
$ k create -f pod-with-secret.yaml
pod/pod-with-secret created

$ k exec pod-with-secret -- cat /mnt/hush-hush/username
tobias

$ k exec pod-with-secret -- cat /mnt/hush-hush/password
cutoffs
```

Managing secrets with Vault

Kubernetes secrets are a good foundation for storing and managing sensitive data on Kubernetes. However, storing encrypted data in etcd is just the tip of the iceberg of an industrial-strength secret management solution. This is where Vault comes in.

Vault is an identity-based source secret management system developed by HashiCorp since 2015. Vault is considered best in class and doesn't really have any significant non-proprietary competitors. It exposes an HTTP API, CLI, and UI to manage your secrets. Vault is a mature and battle-tested solution that is being used by a large number of enterprise organizations as well as smaller companies. Vault has well-defined security and threat models that cover a lot of ground. Practically, as long as you can ensure the physical security of your Vault deployment, Vault will keep your secrets safe and make it easy to manage and audit them.

When running Vault on Kubernetes there are several other important measures to ensure the Vault security model remains intact such as:

- Considerations for multi-tenant clusters (single Vault will be shared by all tenants)
- End-to-end TLS (Kubernetes may skip TLS under some conditions)
- Turn off process core dumps to avoid revealing Vault encryption keys
- Ensure mlock is enabled to avoid swapping memory to disk and revealing Vault encryption keys
- Container supervisor and pods should run as non-root

You can find a lot of information about Vault here: `https://www.vaultproject.io/`.

Deploying and configuring Vault is pretty straightforward. If you want to try it out, follow this tutorial: `https://learn.hashicorp.com/tutorials/vault/kubernetes-minikube-raft`.

Running a multi-tenant cluster

In this section, we will look briefly at the option to use a single cluster to host systems for multiple users or multiple user communities (which is also known as multi-tenancy). The idea is that those users are totally isolated and may not even be aware that they share the cluster with other users.

Each user community will have its own resources, and there will be no communication between them (except maybe through public endpoints). The Kubernetes namespace concept is the ultimate expression of this idea. But, they don't provide absolute isolation. Another solution is to use virtual clusters where each namespace appears as a completely independent cluster to the users.

Check out `https://www.vcluster.com/` for more details about virtual clusters.

The case for multi-tenant clusters

Why should you run a single cluster for multiple isolated users or deployments? Isn't it simpler to just have a dedicated cluster for each user? There are two main reasons: cost and operational complexity. If you have many relatively small deployments and you want to create a dedicated cluster for each one, then you'll have a separate control plane node and possibly a three-node etcd cluster for each one. The cost can add up. Operational complexity is very important too. Managing tens, hundreds, or thousands of independent clusters is no picnic. Every upgrade and every patch needs to be applied to each cluster. Operations might fail and you'll have to manage a fleet of clusters where some of them are in a slightly different state than the others. Meta-operations across all clusters may be more difficult. You'll have to aggregate and write your tools to perform operations and collect data from all clusters.

Let's look at some use cases and requirements for multiple isolated communities or deployments:

- A platform or service provider for software-as-a-service
- Managing separate testing, staging, and production environments
- Delegating responsibility to community/deployment admins
- Enforcing resource quotas and limits on each community
- Users see only resources in their community

Using namespaces for safe multi-tenancy

Kubernetes namespaces are a good start for safe multi-tenant clusters. This is not a surprise, as this was one of the design goals of namespaces.

You can easily create namespaces in addition to the built-in `kube-system` and default. Here is a YAML file that will create a new namespace called `custom-namespace`. All it has is a metadata item called name. It doesn't get any simpler:

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```

Let's create the namespace:

```
$ k create -f custom-namespace.yaml
namespace/custom-namespace created


$ k get ns
```

```
NAME                STATUS   AGE
custom-namespace    Active   5s
default             Active   24h
kube-node-lease     Active   24h
kube-public         Active   24h
kube-system         Active   24h
```

We can see the default namespace, our new `custom-namespace`, and a few other system namespaces prefixed with `kube-`.

The status field can be `Active` or `Terminating`. When you delete a namespace, it will move into the `Terminating` state. When the namespace is in this state, you will not be able to create new resources in this namespace. This simplifies the clean-up of namespace resources and ensures the namespace is really deleted. Without it, the replication controllers might create new pods when existing pods are deleted.

Sometimes, namespaces may hang during termination. I wrote a little Go tool called k8s-namespace-deleter to delete stubborn namespaces.

Check it out here: `https://github.com/the-gigi/k8s-namespace-deleter`

It can also be used as kubectl plugin.

To work with a namespace, you add the `--namespace` (or `-n` for short) argument to kubectl commands. Here is how to run a pod in interactive mode in the `custom-namespace` namespace:

```
$ k run trouble -it -n custom-namespace --image=g1g1/py-kube:0.3 bash
If you don't see a command prompt, try pressing enter.
root@trouble:/#
```

Listing pods in the `custom-namespace` returns only the pod we just launched:

```
$ k get po -n custom-namespace
NAME      READY   STATUS    RESTARTS      AGE
trouble   1/1     Running   1 (15s ago)   57s
```

Avoiding namespace pitfalls

Namespaces are great, but they can add some friction. When you use just the default namespace, you can simply omit the namespace. When using multiple namespaces, you must qualify everything with the namespace. This can add some burden but doesn't present any danger.

However, if some users (for example, cluster administrators) can access multiple namespaces, then you're open to accidentally modifying or querying the wrong namespace. The best way to avoid this situation is to hermetically seal the namespace and require different users and credentials for each namespace, just like you should use a user account for most operations on your machine or remote machines and use root via sudo only when you have to.

In addition, you should use tools that help make it clear what namespace you're operating on (for example, shell prompt if working from the command line or listing the namespace prominently in a web interface). One of the most popular tools is kubens (available along with kubectx), available at `https://github.com/ahmetb/kubectx`.

Make sure that users that can operate on a dedicated namespace don't have access to the default namespace. Otherwise, every time they forget to specify a namespace, they'll operate quietly on the default namespace.

Using virtual clusters for strong multi-tenancy

Namespaces are fine, but they don't really cut it for strong multi-tenancy. Namespace isolation obviously works for namespaced resources only. But, Kubernetes has many cluster-level resources (in particular CRDs). Tenants will share those. In addition, the control plane version, security, and audit will be shared.

One trivial solution is not to use multi-tenancy. Just have a separate cluster for each tenant. But, that is not efficient especially if you have a lot of small tenants.

The vcluster project (`https://www.vcluster.com`) from Loft.sh utilizes an innovative approach where a physical Kubernetes cluster can host multiple virtual clusters that appear as regular Kubernetes clusters to their users, totally isolated from the other virtual clusters and the host cluster. This reaps all benefits of multi-tenancy without the downsides of namespace-level isolation.

Here is the vcluster architecture:

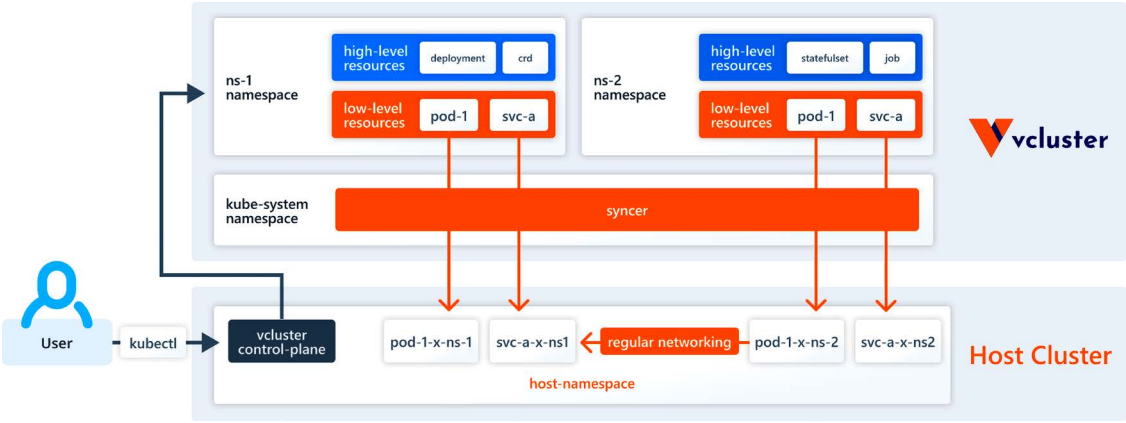


Figure 4.3: vcluster architecture

Let's create a couple of virtual clusters. First install the vcluster CLI: `https://www.vcluster.com/docs/getting-started/setup`.

Make sure it's installed correctly:

```
$ vcluster version
vcluster version 0.10.1
```

Now, we can create some virtual clusters. You can create virtual clusters using the vcluster CLI, Helm, or kubectl. Let's go with the vcluster CLI.

```
$ vcluster create tenant-1
info   Creating namespace vcluster-tenant-1
info   Detected local kubernetes cluster kind. Will deploy vcluster with a NodePort
info   Create vcluster tenant-1...
done √ Successfully created virtual cluster tenant-1 in namespace vcluster-tenant-1
info   Waiting for vcluster to come up...
warn   vcluster is waiting, because vcluster pod tenant-1-0 has status:
ContainerCreating
info   Starting proxy container...
done √ Switched active kube context to vcluster_tenant-1_vcluster-tenant-1_kind-kind
- Use `vcluster disconnect` to return to your previous kube context
- Use `kubectl get namespaces` to access the vcluster
```

Let's create another virtual cluster:

```
$ vcluster create tenant-2

? You are creating a vcluster inside another vcluster, is this desired?
  [Use arrows to move, enter to select, type to filter]
> No, switch back to context kind-kind
  Yes
```

Oops. After creating the `tenant-1` virtual cluster the Kubernetes context changed to this cluster. When I tried to create `tenant-2`, the vcluster CLI was smart enough to warn me. Let's try again:

```
$ k config use-context kind-kind
Switched to context "kind-kind".

$ vcluster create tenant-2
info   Creating namespace vcluster-tenant-2
info   Detected local kubernetes cluster kind. Will deploy vcluster with a NodePort
info   Create vcluster tenant-2...
done √ Successfully created virtual cluster tenant-2 in namespace vcluster-tenant-2
info   Waiting for vcluster to come up...
info   Stopping docker proxy...
info   Starting proxy container...
done √ Switched active kube context to vcluster_tenant-2_vcluster-tenant-2_kind-kind
- Use `vcluster disconnect` to return to your previous kube context
- Use `kubectl get namespaces` to access the vcluster
```

Let's check our clusters:

```
$ k config get-contexts -o name
kind-kind
vcluster_tenant-1_vcluster-tenant-1_kind-kind
vcluster_tenant-2_vcluster-tenant-2_kind-kind
```

Yes, our two virtual clusters are available. Let's see the namespaces in our host `kind` cluster:

```
$ k get ns --context kind-kind
NAME                 STATUS   AGE
custom-namespace     Active   3h6m
default              Active   27h
kube-node-lease      Active   27h
kube-public          Active   27h
kube-system          Active   27h
local-path-storage   Active   27h
vcluster-tenant-1    Active   15m
vcluster-tenant-2    Active   3m48s
```

We can see the two new namespaces for the virtual clusters. Let's see what's running in the `vcluster-tenant-1` namespace:

```
$ k get all -n vcluster-tenant-1 --context kind-kind
NAME                                                     READY   STATUS    RESTARTS
AGE
pod/coredns-5df468b6b7-rj4nr-x-kube-system-x-tenant-1   1/1     Running   0
16m
pod/tenant-1-0                                           2/2     Running   0
16m

NAME                                          TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)                  AGE
service/kube-dns-x-kube-system-x-tenant-1     ClusterIP   10.96.200.106   <none>
53/UDP,53/TCP,9153/TCP   16m
service/tenant-1                              NodePort    10.96.107.216   <none>
443:32746/TCP            16m
service/tenant-1-headless                     ClusterIP   None            <none>
443/TCP                  16m
service/tenant-1-node-kind-control-plane      ClusterIP   10.96.235.53    <none>
10250/TCP                16m

NAME                          READY   AGE
statefulset.apps/tenant-1     1/1     16m
```

Now, let's see what namespaces are in the virtual cluster:

```
$ k get ns --context vcluster_tenant-1_vcluster-tenant-1_kind-kind
NAME              STATUS   AGE
kube-system       Active   17m
default           Active   17m
kube-public       Active   17m
kube-node-lease   Active   17m
```

Just the default namespaces of a k3s cluster (vcluster is based on k3s). Let's create a new namespace and verify it shows up only in the virtual cluster:

```
$ k create ns new-ns --context vcluster_tenant-1_vcluster-tenant-1_kind-kind
namespace/new-ns created

$ k get ns new-ns --context vcluster_tenant-1_vcluster-tenant-1_kind-kind
NAME     STATUS   AGE
new-ns   Active   19s

$ k get ns new-ns --context vcluster_tenant-2_vcluster-tenant-2_kind-kind
 Error from server (NotFound): namespaces "new-ns" not found


$ k get ns new-ns --context kind-kind
Error from server (NotFound): namespaces "new-ns" not found
```

The new namespace is only visible in the virtual cluster it was created in as expected.

In this section, we covered multi-tenant clusters, why they are useful, and different approaches for isolating tenants such as namespaces and virtual clusters.

Summary

In this chapter, we covered the many security challenges facing developers and administrators building systems and deploying applications on Kubernetes clusters. But we also explored the many security features and the flexible plugin-based security model that provides many ways to limit, control, and manage containers, pods, and nodes. Kubernetes already provides versatile solutions to most security challenges, and it will only get better as capabilities such as AppArmor and various plugins move from alpha/beta status to general availability. Finally, we considered how to use namespaces and virtual clusters to support multi-tenant communities or deployments in the same Kubernetes cluster.

In the next chapter, we will look in detail into many Kubernetes resources and concepts and how to use them and combine them effectively. The Kubernetes object model is built on top of a solid foundation of a small number of generic concepts such as resources, manifests, and metadata. This empowers an extensible, yet surprisingly consistent, object model to expose a very diverse set of capabilities for developers and administrators.

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`



5

Using Kubernetes Resources in Practice

In this chapter, we will design a fictional massive-scale platform that will challenge Kubernetes' capabilities and scalability. The Hue platform is all about creating an omniscient and omnipotent digital assistant. Hue is a digital extension of you. Hue will help you do anything, find anything, and, in many cases, will do a lot on your behalf. It will obviously need to store a lot of information, integrate with many external services, respond to notifications and events, and be smart about interacting with you.

We will take the opportunity in this chapter to get to know kubectl and related tools a little better and explore in detail familiar resources we've seen before, such as pods, as well as new resources, such as jobs. We will explore advanced scheduling and resource management.

This chapter will cover the following topics:

- Designing the Hue platform
- Using Kubernetes to build the Hue platform
- Separating internal and external services
- Advanced scheduling
- Using namespaces to limit access
- Using Kustomization for hierarchical cluster structures
- Launching jobs
- Mixing non-cluster components
- Managing the Hue platform with Kubernetes
- Evolving the Hue platform with Kubernetes

Just to be clear, this is a design exercise! We are not actually going to build the Hue platform. The motivation behind this exercise is to showcase the vast range of capabilities available with Kubernetes in the context of a large system with multiple moving parts.

At the end of this chapter, you will have a clear picture of how impressive Kubernetes is and how it can be used as the foundation for hugely complex systems.

Designing the Hue platform

In this section, we will set the stage and define the scope of the amazing Hue platform. Hue is not Big Brother; Hue is Little Brother! Hue will do whatever you allow it to do. Hue will be able to do a lot, which might concern some people, but you get to pick how much or how little Hue can help you with. Get ready for a wild ride!

Defining the scope of Hue

Hue will manage your digital persona. It will know you better than you know yourself. Here is a list of some of the services Hue can manage and help you with:

- Search and content aggregation
- Medical – electronic heath records, DNA sequencing
- Smart homes
- Finance – banking, savings, retirement, investing
- Office
- Social
- Travel
- Wellbeing
- Family

Let's look at some of the capabilities of the Hue platform, such as smart reminders and notifications, security, identity, and privacy.

Smart reminders and notifications

Let's think of the possibilities. Hue will know you, but also know your friends and the aggregate of other users across all domains. Hue will update its models in real time. It will not be confused by stale data. It will act on your behalf, present relevant information, and learn your preferences continuously. It can recommend new shows or books that you may like, make restaurant reservations based on your schedule and that of your family or friends, and control your house's automation.

Security, identity, and privacy

Hue is your proxy online. The ramifications of someone stealing your Hue identity, or even just eavesdropping on your Hue interactions, are devastating. Potential users may even be reluctant to trust the Hue organization with their identity. Let's devise a non-trust system where users have the power to pull the plug on Hue at any time. Here are a few ideas:

- Strong identity via a dedicated device with multi-factor authorization, including multiple biometric factors
- Frequently rotating credentials

- Quick service pause and identity verification of all external services (will require original proof of identity for each provider)
- The Hue backend will interact with all external services via short-lived tokens
- Architecting Hue as a collection of loosely coupled microservices with strong compartmentalization
- GDPR compliance
- End-to-end encryption
- Avoid owning critical data (let external providers manage it)

Hue's architecture will need to support enormous variation and flexibility. It will also need to be very extensible where existing capabilities and external services are constantly upgraded, and new capabilities and external services are integrated into the platform. That level of scale calls for microservices, where each capability or service is totally independent of other services except for well-defined interfaces via standard and/or discoverable APIs.

Hue components

Before embarking on our microservice journey, let's review the types of components we need to construct for Hue.

User profile

The user profile is a major component, with lots of sub-components. It is the essence of the user, their preferences, their history across every area, and everything that Hue knows about them. The benefit you can get from Hue is affected strongly by the richness of the profile. But the more information is managed by the profile, the more damage you can suffer if the data (or part of it) is compromised.

A big piece of managing the user profile is the reports and insights that Hue will provide to the user. Hue will employ sophisticated machine learning to better understand the user and their interactions with other users and external service providers.

User graph

The user graph component models networks of interactions between users across multiple domains. Each user participates in multiple networks: social networks such as Facebook, Instagram, and Twitter; professional networks; hobby networks; and volunteer communities. Some of these networks are ad hoc and Hue will be able to structure them to benefit users. Hue can take advantage of the rich profiles it has of user connections to improve interactions even without exposing private information.

Identity

Identity management is critical, as mentioned previously, so it merits a separate component. A user may prefer to manage multiple mutually exclusive profiles with separate identities. For example, maybe users are not comfortable with mixing their health profile with their social profile at the risk of inadvertently exposing personal health information to their friends. While Hue can find useful connections for you, you may prefer to trade off capabilities for more privacy.

Authorizer

The authorizer is a critical component where the user explicitly authorizes Hue to perform certain actions or collect various data on their behalf. This involves access to physical devices, accounts of external services, and levels of initiative.

External services

Hue is an aggregator of external services. It is not designed to replace your bank, your health provider, or your social network. It will keep a lot of metadata about your activities, but the content will remain with your external services. Each external service will require a dedicated component to interact with the external service API and policies. When no API is available, Hue emulates the user by automating the browser or native apps.

Generic sensor

A big part of Hue's value proposition is to act on the user's behalf. In order to do that effectively, Hue needs to be aware of various events. For example, if Hue reserved a vacation for you but it senses that a cheaper flight is available, it can either automatically change your flight or ask you for confirmation. There is an infinite number of things to sense. To reign in sensing, a generic sensor is needed. The generic sensor will be extensible but exposes a generic interface that the other parts of Hue can utilize uniformly even as more and more sensors are added.

Generic actuator

This is the counterpart of the generic sensor. Hue needs to perform actions on your behalf; for example, reserving a flight or a doctor's appointment. To do that, Hue needs a generic actuator that can be extended to support particular functions but can interact with other components, such as the identity manager and the authorizer, in a uniform fashion.

User learner

This is the brain of Hue. It will constantly monitor all your interactions (that you authorize) and update its model of you and other users in your networks. This will allow Hue to become more and more useful over time, predict what you need and what will interest you, provide better choices, surface more relevant information at the right time, and avoid being annoying and overbearing.

Hue microservices

The complexity of each of the components is enormous. Some of the components, such as the external service, the generic sensor, and the generic actuator, will need to operate across hundreds, thousands, or even more external services that constantly change outside the control of Hue. Even the user learner needs to learn the user's preferences across many areas and domains. Microservices address this need by allowing Hue to evolve gradually and grow more isolated capabilities without collapsing under its own complexity. Each microservice interacts with generic Hue infrastructure services through standard interfaces and, optionally, with a few other services through well-defined and versioned interfaces. The surface area of each microservice is manageable and the orchestration between microservices is based on standard best practices.

Plugins

Plugins are the key to extending Hue without a proliferation of interfaces. The thing about plugins is that often, you need plugin chains that cross multiple abstraction layers. For example, if you want to add a new integration for Hue with YouTube, then you can collect a lot of YouTube-specific information – your channels, favorite videos, recommendations, and videos you have watched. To display this information to users and allow them to act on it, you need plugins across multiple components and, eventually, in the user interface as well. Smart design will help by aggregating categories of actions such as recommendations, selections, and delayed notifications to many services.

The great thing about plugins is that they can be developed by anyone. Initially, the Hue development team will have to develop the plugins, but as Hue becomes more popular, external services will want to integrate with Hue and build Hue plugins to enable their service. That will lead, of course, to a whole ecosystem of plugin registration, approval, and curation.

Data stores

Hue will need several types of data stores, and multiple instances of each type, to manage its data and metadata:

- Relational database
- Graph database
- Time-series database
- In-memory caching
- Blob storage

Due to the scope of Hue, each one of these databases will have to be clustered, scalable, and distributed. In addition, Hue will use local storage on edge devices.

Stateless microservices

The microservices should be mostly stateless. This will allow specific instances to be started and killed quickly and migrated across the infrastructure as necessary. The state will be managed by the stores and accessed by the microservices with short-lived access tokens. Hue will store frequently accessed data in easily hydrated fast caches when appropriate.

Serverless functions

A big part of Hue's functionality per user will involve relatively short interactions with external services or other Hue services. For those activities, it may not be necessary to run a full-fledged persistent microservice that needs to be scaled and managed. A more appropriate solution may be to use a serverless function that is more lightweight.

Event-driven interactions

All these microservices need to talk to each other. Users will ask Hue to perform tasks on their behalf. External services will notify Hue of various events. Queues coupled with stateless microservices provide the perfect solution.

Multiple instances of each microservice will listen to various queues and respond when relevant events or requests are popped from the queue. Serverless functions may be triggered as a result of particular events too. This arrangement is very robust and easy to scale. Every component can be redundant and highly available. While each component is fallible, the system is very fault-tolerant.

A queue can be used for asynchronous RPC or request-response style interactions too, where the calling instance provides a private queue name and the response is posted to the private queue.

That said, sometimes direct service-to-service interaction (or serverless function-to-service interaction) through a well-defined interface makes more sense and simplifies the architecture.

Planning workflows

Hue often needs to support workflows. A typical workflow will take a high-level task, such as making a dentist appointment. It will extract the user's dentist's details and schedule, match it with the user's schedule, choose between multiple options, potentially confirm with the user, make the appointment, and set up a reminder. We can classify workflows into fully automatic workflows and human workflows where humans are involved. Then there are workflows that involve spending money and might require an additional level of approval.

Automatic workflows

Automatic workflows don't require human intervention. Hue has full authority to execute all the steps from start to finish. The more autonomy the user allocates to Hue, the more effective it will be. The user will be able to view and audit all workflows, past and present.

Human workflows

Human workflows require interaction with a human. Most often it will be the user that needs to make a choice from multiple options or approve an action. But it may involve a person on another service. For example, to make an appointment with a dentist, Hue may have to get a list of available times from the secretary. In the future, Hue will be able to handle conversations with humans and possibly automate some of these workflows too.

Budget-aware workflows

Some workflows, such as paying bills or purchasing a gift, require spending money. While, in theory, Hue can be granted unlimited access to the user's bank account, most users will probably be more comfortable setting budgets for different workflows or just making spending a human-approved activity. Potentially, the user could grant Hue access to a dedicated account or set of accounts and, based on reminders and reports, allocate more or fewer funds to Hue as needed.

At this point, we have covered a lot of ground and looked at the different components that comprise the Hue platform and its design. Now is a good time to see how Kubernetes can help with building a platform like Hue.

Using Kubernetes to build the Hue platform

In this section, we will look at various Kubernetes resources and how they can help us build Hue. First, we'll get to know the versatile kubectl a little better, then we will look at how to run long-running processes in Kubernetes, exposing services internally and externally, using namespaces to limit access, launching ad hoc jobs, and mixing in non-cluster components. Obviously, Hue is a huge project, so we will demonstrate the ideas on a local cluster and not actually build a real Hue Kubernetes cluster. Consider it primarily a thought experiment. If you wish to explore building a real microservice-based distributed system on Kubernetes, check out *Hands-On Microservices with Kubernetes*: `https://www.packtpub.com/product/hands-on-microservices-with-kubernetes/9781789805468`.

Using kubectl effectively

kubectl is your Swiss Army knife. It can do pretty much anything around a cluster. Under the hood, kubectl connects to your cluster via the API. It reads your `~/.kube/config` file (by default, this can be overridden with the `KUBECONFIG` environment variable or the `--kubeconfig` command-line argument), which contains the information necessary to connect to your cluster or clusters. The commands are divided into multiple categories:

- **Generic commands**: Deal with resources in a generic way: `create`, `get`, `delete`, `run`, `apply`, `patch`, `replace`, and so on
- **Cluster management commands**: Deal with nodes and the cluster at large: `cluster-info`, `certificate`, `drain`, and so on
- **Troubleshooting commands**: `describe`, `logs`, `attach`, `exec`, and so on
- **Deployment commands**: Deal with deployment and scaling: `rollout`, `scale`, `auto-scale`, and so on
- **Settings commands**: Deal with labels and annotations: `label`, `annotate`, and so on
- **Misc commands**: `help`, `config`, and `version`
- **Customization commands**: Integrate the kustomize.io capabilities into kubectl
- **Configuration commands:** Deal with contexts, switch between clusters and namespaces, set current context and namespace, and so on

You can view the configuration with Kubernetes' `config view` command.

Here is the configuration for my local KinD cluster:

```
$ k config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://127.0.0.1:50615
  name: kind-kind
contexts:
```

```
- context:
    cluster: kind-kind
    user: kind-kind
  name: kind-kind
current-context: kind-kind
kind: Config
preferences: {}
users:
- name: kind-kind
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

Your `kubeconfig` file may or may not be similar to the code sample above, but as long as it points to a running Kubernetes cluster, you will be able to follow along. Let's take an in-depth look into the kubectl manifest files.

Understanding kubectl manifest files

Many kubectl operations, such as `create`, require a complicated hierarchical structure (since the API requires this structure). kubectl uses YAML or JSON manifest files. YAML is more concise and human-readable so we will use YAML mostly. Here is a YAML manifest file for creating a pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: ""
  labels:
    name: ""
  namespace: ""
  annotations: []
  generateName: ""
spec:
    ...
```

Let's examine the various fields of the manifest.

apiVersion

The very important Kubernetes API keeps evolving and can support different versions of the same resource via different versions of the API.

kind

`kind` tells Kubernetes what type of resource it is dealing with; in this case, `Pod`. This is always required.

metadata

`metadata` contains a lot of information that describes the pod and where it operates:

- `name`: Identifies the pod uniquely within its namespace
- `labels`: Multiple labels can be applied
- `namespace`: The namespace the pod belongs to
- `annotations`: A list of annotations available for query

spec

`spec` is a pod template that contains all the information necessary to launch a pod. It can be quite elaborate, so we'll explore it in multiple parts:

```
spec:
  containers: [
      ...
  ],
  "restartPolicy": "",
  "volumes": []
```

Container spec

The pod spec's `containers` section is a list of container specs. Each container spec has the following structure:

```
name: "",
image: "",
command: [""],
args: [""],
env:
    - name: "",
      value: ""

imagePullPolicy: "",
ports:
    - containerPort": 0,
      name: "",
      protocol: ""
resources:
  requests:
    cpu: ""
    memory: ""
  limits:
    cpu: ""
    memory: ""
```

Each container has an `image`, a command that, if specified, replaces the Docker image command. It also has arguments and environment variables. Then, there are of course the image pull policy, ports, and resource limits. We covered those in earlier chapters.

If you want to explore the pod resource, or other Kubernetes resources, further, then the following command can be very useful: `kubectl explain`.

It can explore resources as well as specific sub-resources and fields.

Try the following commands:

```
kubectl explain pod
kubectl explain pod.spec
```

Deploying long-running microservices in pods

Long-running microservices should run in pods and be stateless. Let's look at how to create pods for one of Hue's microservices – the Hue learner – which is responsible for learning the user's preferences across different domains. Later, we will raise the level of abstraction and use a deployment.

Creating pods

Let's start with a regular pod configuration file for creating a Hue learner internal service. This service doesn't need to be exposed as a public service and it will listen to a queue for notifications and store its insights in some persistent storage.

We need a simple container that will run in the pod. Here is possibly the simplest Docker file ever, which will simulate the Hue learner:

```
FROM busybox

CMD ash -c "echo 'Started...'; while true ; do sleep 10 ; done"
```

It uses the `busybox` base image, prints to standard output `Started...`, and then goes into an infinite loop, which is, by all accounts, long-running.

I have built two Docker images tagged as `g1g1/hue-learn:0.3` and `g1g1/hue-learn:0.4` and pushed them to the Docker Hub registry (`g1g1` is my username):

```
$ docker build . -t g1g1/hue-learn:0.3
$ docker build . -t g1g1/hue-learn:0.4
$ docker push g1g1/hue-learn:0.3
$ docker push g1g1/hue-learn:0.4
```

Now these images are available to be pulled into containers inside of Hue's pods.

We'll use YAML here because it's more concise and human-readable. Here are the boilerplate and metadata labels:

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: hue-learner
  labels:
    app: hue
    service: learner
    runtime-environment: production
    tier: internal-service
```

Next comes the important `containers` spec, which defines for each container the mandatory name and image:

```
spec:
  containers:
  - name: hue-learner
    image: g1g1/hue-learn:0.3
```

The `resources` section tells Kubernetes the resource requirements of the container, which allows for more efficient and compact scheduling and allocations. Here, the container requests 200 milli-cpu units (0.2 core) and 256 MiB (2 to the power of 28 bytes):

```
    resources:
      requests:
        cpu: 200m
        memory: 256Mi
```

The environment section allows the cluster administrator to provide environment variables that will be available to the container. Here it tells it to discover the queue and the store via DNS. In a testing environment, it may use a different discovery method:

```
    env:
    - name: DISCOVER_QUEUE
      value: dns
    - name: DISCOVER_STORE
      value: dns
```

Decorating pods with labels

Labeling pods wisely is key for flexible operations. It lets you evolve your cluster live, organize your microservices into groups you can operate on uniformly, and drill down on the fly to observe different subsets.

For example, our Hue learner pod has the following labels (and a few others):

- `runtime-environment : production`
- `tier : internal-service`

The `runtime-environment` label allows performing global operations on all pods that belong to a certain environment. The `tier` label can be used to query all pods that belong to a particular tier. These are just examples; your imagination is the limit here.

Here is how to list the labels with the `get pods` command:

```
$ k get po -n kube-system --show-labels
NAME                                          READY   STATUS    RESTARTS   AGE
LABELS
coredns-64897985d-gzrm4                       1/1     Running   0          2d2h   k8s-
app=kube-dns,pod-template-hash=64897985d
coredns-64897985d-m8nm9                       1/1     Running   0          2d2h   k8s-
app=kube-dns,pod-template-hash=64897985d
etcd-kind-control-plane                       1/1     Running   0          2d2h
component=etcd,tier=control-plane
kindnet-wx7kl                                 1/1     Running   0          2d2h
app=kindnet,controller-revision-hash=9d779cb4d,k8s-app=kindnet,pod-template-
generation=1,tier=node
kube-apiserver-kind-control-plane             1/1     Running   0          2d2h
component=kube-apiserver,tier=control-plane
kube-controller-manager-kind-control-plane    1/1     Running   0          2d2h
component=kube-controller-manager,tier=control-plane
kube-proxy-bgcrq                              1/1     Running   0          2d2h
controller-revision-hash=664d4bb79f,k8s-app=kube-proxy,pod-template-generation=1
kube-scheduler-kind-control-plane             1/1     Running   0          2d2h
component=kube-scheduler,tier=control-plane
```

Now, if you want to filter and list only the **kube-dns** pods, type the following:

```
$ k get po -n kube-system -l k8s-app=kube-dns
NAME                       READY   STATUS    RESTARTS   AGE
coredns-64897985d-gzrm4    1/1     Running   0          2d2h
coredns-64897985d-m8nm9    1/1     Running   0          2d2h
```

Deploying long-running processes with deployments

In a large-scale system, pods should never be just created and let loose. If a pod dies unexpectedly for whatever reason, you want another one to replace it to maintain overall capacity. You can create replication controllers or replica sets yourself, but that leaves the door open to mistakes, as well as the possibility of partial failure. It makes much more sense to specify how many replicas you want when you launch your pods in a declarative manner. This is what Kubernetes deployments are for.

Let's deploy three instances of our Hue learner microservice with a Kubernetes deployment resource:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
    name: hue-learn
    labels:
      app: hue
  spec:
    replicas: 3
    selector:
      matchLabels:
        app: hue
    template:
      metadata:
        labels:
          app: hue
      spec:
        containers:
        - name: hue-learner
          image: g1g1/hue-learn:0.3
          resources:
            requests:
              cpu: 200m
              memory: 256Mi
          env:
          - name: DISCOVER_QUEUE
            value: dns
          - name: DISCOVER_STORE
            value: dns
```

The pod spec is identical to the spec section from the pod configuration file previously.

Let's create the deployment and check its status:

```
$ k create -f hue-learn-deployment.yaml
deployment.apps/hue-learn created

$ k get deployment hue-learn
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
hue-learn   3/3     3            3           25s

$ k get pods -l app=hue
NAME                          READY   STATUS    RESTARTS   AGE
hue-learn-67d4649b58-qhc88    1/1     Running   0          45s
hue-learn-67d4649b58-qpm2q    1/1     Running   0          45s
hue-learn-67d4649b58-tzzq7    1/1     Running   0          45s
```

You can get a lot more information about the deployment using the `kubectl describe` command:

```
$ k describe deployment hue-learn
Name:                   hue-learn
Namespace:              default
CreationTimestamp:      Tue, 21 Jun 2022 21:11:50 -0700
Labels:                 app=hue
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=hue
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=hue
  Containers:
   hue-learner:
    Image:      g1g1/hue-learn:0.3
    Port:       <none>
    Host Port:  <none>
    Requests:
      cpu:     200m
      memory:  256Mi
    Environment:
      DISCOVER_QUEUE:  dns
      DISCOVER_STORE:  dns
    Mounts:            <none>
  Volumes:             <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   hue-learn-67d4649b58 (3/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  106s  deployment-controller  Scaled up replica set hue-
learn-67d4649b58 to 3
```

Updating a deployment

The Hue platform is a large and ever-evolving system. You need to upgrade constantly. Deployments can be updated to roll out updates in a painless manner. You change the pod template to trigger a rolling update fully managed by Kubernetes. Currently, all the pods are running with version 0.3:

```
$ k get pods -o jsonpath='{.items[*].spec.containers[0].image}' -l app=hue | xargs
printf "%s\n"
g1g1/hue-learn:0.3
g1g1/hue-learn:0.3
g1g1/hue-learn:0.3
```

Let's update the deployment to upgrade to version 0.4. Modify the image version in the deployment file. Don't modify labels; it will cause an error. Save it to `hue-learn-deployment-0.4.yaml`. Then we can use the `kubectl apply` command to upgrade the version and verify that the pods now run 0.4:

```
$ k apply -f hue-learn-deployment-0.4.yaml
Warning: resource deployments/hue-learn is missing the kubectl.kubernetes.io/last-
applied-configuration annotation which is required by kubectl apply. kubectl apply
should only be used on resources created declaratively by either kubectl create
--save-config or kubectl apply. The missing annotation will be patched automatically.
deployment.apps/hue-learn configured


$ k get pods -o jsonpath='{.items[*].spec.containers[0].image}' -l app=hue | xargs
printf "%s\n"
g1g1/hue-learn:0.4
g1g1/hue-learn:0.4
g1g1/hue-learn:0.4
```

Note that new pods are created and the original 0.3 pods are terminated in a rolling update manner.

```
$ kubectl get pods
NAME                          READY   STATUS        RESTARTS   AGE
hue-learn-67d4649b58-fgt7m    1/1     Terminating   0          99s
hue-learn-67d4649b58-klhz5    1/1     Terminating   0          100s
hue-learn-67d4649b58-lgpl9    1/1     Terminating   0          101s
hue-learn-68d74fd4b7-bxxnm    1/1     Running       0          4s
hue-learn-68d74fd4b7-fh55c    1/1     Running       0          3s
hue-learn-68d74fd4b7-rnsj4    1/1     Running       0          2s
```

We've covered how kubectl manifest files are structured and how they can be applied to deploy and update workloads on our cluster. Let's see how these workloads can discover and call each other via internal services as well as be called from outside the cluster via externally exposed services.

Separating internal and external services

Internal services are services that are accessed directly only by other services or jobs in the cluster (or administrators that log in and run ad hoc tools). There are also workloads that are not accessed at all. These workloads may watch for some events and perform their function without exposing any API.

But some services need to be exposed to users or external programs. Let's look at a fake Hue service that manages a list of reminders for a user. It doesn't really do much – just returns a fixed list of reminders – but we'll use it to illustrate how to expose services. I already pushed a `hue-reminders` image to Docker Hub:

```
docker push g1g1/hue-reminders:3.0
```

Deploying an internal service

Here is the deployment, which is very similar to the `hue-learner` deployment, except that I dropped the annotations, `env`, and `resources` sections, kept just one or two labels to save space, and added a `ports` section to the container. That's crucial because a service must expose a port through which other services can access it:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hue-reminders
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hue
      service: reminders
  template:
    metadata:
      name: hue-reminders
      labels:
        app: hue
        service: reminders
    spec:
      containers:
      - name: hue-reminders
        image: g1g1/hue-reminders:3.0
        ports:
        - containerPort: 8080
```

When we run the deployment, two `hue-reminders` pods are added to the cluster:

```
$ k create -f hue-reminders-deployment.yaml
deployment.apps/hue-reminders created

$ k get pods
NAME                              READY   STATUS    RESTARTS   AGE
hue-learn-68d74fd4b7-bxxnm        1/1     Running   0          12h
hue-learn-68d74fd4b7-fh55c        1/1     Running   0          12h
hue-learn-68d74fd4b7-rnsj4        1/1     Running   0          12h
hue-reminders-9bdcd7489-4jqhc     1/1     Running   0          11s
hue-reminders-9bdcd7489-bxh59     1/1     Running   0          11s
```

OK. The pods are running. In theory, other services can look up or be configured with their internal IP address and just access them directly because they are all in the same network address space. But this doesn't scale. Every time a reminder's pod dies and is replaced by a new one, or when we just scale up the number of pods, all the services that access these pods must know about it. Kubernetes services solve this issue by providing a single stable access point to all the pods that share a set of selector labels. Here is the service definition:

```
apiVersion: v1
kind: Service
metadata:
  name: hue-reminders
  labels:
    app: hue
    service: reminders
spec:
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
  selector:
    app: hue
    service: reminders
```

The service has a `selector` that determines the backing pods by their matching labels. It also exposes a port, which other services will use to access it. It doesn't have to be the same port as the container's port. You can define a `targetPort`.

The `protocol` field can be one of the following: k'TCP, UDP, or (since Kubernetes 1.12) SCTP.

Creating the hue-reminders service

Let's create the service and explore it:

```
$ k create -f hue-reminders-service.yaml
service/hue-reminders created

$ k describe svc hue-reminders
Name:              hue-reminders
Namespace:         default
Labels:            app=hue
                   service=reminders
Annotations:       <none>
Selector:          app=hue,service=reminders
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.96.152.254
IPs:               10.96.152.254
Port:              <unset>  8080/TCP
TargetPort:        8080/TCP
Endpoints:         10.244.0.32:8080,10.244.0.33:8080
Session Affinity:  None
Events:            <none>
```

The service is up and running. Other pods can find it through environment variables or DNS. The environment variables for all services are set at pod creation time. That means that if a pod is already running when you create your service, you'll have to kill it and let Kubernetes recreate it with the environment variables for the new service.

For example, the pod `hue-learn-68d74fd4b7-bxxnm` was created before the `hue-reminders` service was created, so it doesn't have the environment variable for `HUE_REMINDERS_SERVICE`. Printing the environment for the pod shows the environment variable doesn't exist:

```
$ k exec hue-learn-68d74fd4b7-bxxnm -- printenv | grep HUE_REMINDERS_SERVICE
```

Let's kill the pod and, when a new pod replaces it, let's try again:

```
$ k delete po hue-learn-68d74fd4b7-bxxnm
pod "hue-learn-68d74fd4b7-bxxnm" deleted
```

Let's check the `hue-learn` pods again:

```
$ k get pods | grep hue-learn
hue-learn-68d74fd4b7-fh55c      1/1     Running   0          13h
hue-learn-68d74fd4b7-rnsj4      1/1     Running   0          13h
hue-learn-68d74fd4b7-rw4qr      1/1     Running   0          2m
```

Great. We have a new fresh pod – `hue-learn-68d74fd4b7-rw4qr`. Let's see if it has the environment variable for the `HUE_REMINDERS_SERVICE` service:

```
$ k exec hue-learn-68d74fd4b7-rw4qr -- printenv | grep HUE_REMINDERS_SERVICE
HUE_REMINDERS_SERVICE_PORT=8080
HUE_REMINDERS_SERVICE_HOST=10.96.152.254
```

Yes, it does! But using DNS is much simpler. Kubernetes assigns an internal DNS name to every service. The service DNS name is:

```
<service name>.<namespace>.svc.cluster.local
$ kubectl exec hue-learn-68d74fd4b7-rw4qr -- nslookup hue-reminders.default.svc.
cluster.local
Server:     10.96.0.10
Address:    10.96.0.10:53


Name:   hue-reminders.default.svc.cluster.local
Address: 10.96.152.254
```

Now, every pod in the cluster can access the `hue-reminders` service through its service endpoint and port `8080`:

```
$ kubectl exec hue-learn-68d74fd4b7-fh55c -- wget -q -O - hue-reminders.default.svc.
cluster.local:8080
Dentist appointment at 3pm
Dinner at 7pm
```

Yes, at the moment `hue-reminders` always returns the same two reminders:

```
Dentist appointment at 3pm
Dinner at 7pm
```

This is for demonstration purposes only. If `hue-reminders` was a real system it would return live and dynamic reminders.

Now that we've covered internal services and how to access them, let's look at external services.

Exposing a service externally

The service is accessible inside the cluster. If you want to expose it to the world, Kubernetes provides several ways to do it:

- Configure `NodePort` for direct access
- Configure a cloud load balancer if you run it in a cloud environment
- Configure your own load balancer if you run on bare metal

Before you configure a service for external access, you should make sure it is secure. We've already covered the principles of this in *Chapter 4*, *Securing Kubernetes*. The Kubernetes documentation has a good example that covers all the gory details here: `https://github.com/kubernetes/examples/blob/master/staging/https-nginx/README.md`.

Here is the `spec` section of the `hue-reminders` service when exposed to the world through `NodePort`:

```
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
    protocol: TCP
    name: http

  - port: 443
    protocol: TCP
    name: https
  selector:
    app: hue-reminders
```

The main downside of exposing services though `NodePort` is that the port numbers are shared across all services. You must coordinate them globally across your entire cluster to avoid conflicts. This is not trivial at scale for large clusters with lots of developers deploying services.

But there are other reasons that you may want to avoid exposing a Kubernetes service directly, such as security and lack of abstraction, and you may prefer to use an Ingress resource in front of the service.

Ingress

Ingress is a Kubernetes configuration object that lets you expose a service to the outside world and takes care of a lot of details. It can do the following:

- Provide an externally visible URL to your service
- Load balance traffic
- Terminate SSL
- Provide name-based virtual hosting

To use Ingress, you must have an Ingress controller running in your cluster. Ingress was introduced in Kubernetes 1.1, and became stable in Kubernetes 1.19. One of the current limitations of the Ingress controller is that it isn't built for scale. As such, it is not a good option for the Hue platform yet. We'll cover the Ingress controller in greater detail in *Chapter 10*, *Exploring Kubernetes Networking*.

Here is what an Ingress resource looks like:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

Note the annotation, which hints that it is an Ingress object that works with the Nginx Ingress controller. There are many other Ingress controllers and they typically use annotations to encode information they need that is not captured by the Ingress object itself and its rules.

Other Ingress controllers include:

- Traefik
- Gloo
- Contour
- AWS ALB Ingress controller
- HAProxy Ingress
- Voyager

An `IngressClass` resource can be created and specified in the `Ingress` resource. If it is not specified, then the default `IngressClass` is used.

In this section, we looked at how the different components of the Hue platform can discover and talk to each other via services as well as exposing public-facing services to the outside world. In the next section, we will look at how Hue workloads can be scheduled efficiently and cost-effectively on Kubernetes.

Advanced scheduling

One of the strongest suits of Kubernetes is its powerful yet flexible scheduler. The job of the scheduler, put simply, is to choose nodes to run newly created pods. In theory, the scheduler could even move existing pods around between nodes, but in practice, it doesn't do that at the moment and instead leaves this functionality for other components.

By default, the scheduler follows several guiding principles, including:

- Split pods from the same replica set or stateful set across nodes
- Schedule pods to nodes that have enough resources to satisfy the pod requests
- Balance out the overall resource utilization of nodes

This is pretty good default behavior, but sometimes you may want better control over specific pod placement. Kubernetes 1.6 introduced several advanced scheduling options that give you fine-grained control over which pods are scheduled or not scheduled on which nodes as well as which pods are to be scheduled together or separately.

Let's review these mechanisms in the context of Hue.

First, let's create a k3d cluster with two worker nodes:

```
$ k3d cluster create --agents 2
...
INFO[0026] Cluster 'k3s-default' created successfully!

$ k get no
NAME                       STATUS   ROLES                  AGE   VERSION
k3d-k3s-default-agent-0    Ready    <none>                 22s   v1.23.6+k3s1
k3d-k3s-default-agent-1    Ready    <none>                 22s   v1.23.6+k3s1
k3d-k3s-default-server-0   Ready    control-plane,master   31s   v1.23.6+k3s1
```

Let's look at the various ways that pods can be scheduled to nodes and when each method is appropriate.

Node selector

The node selector is pretty simple. A pod can specify which nodes it wants to be scheduled on in its spec. For example, the `trouble-shooter` pod has a `nodeSelector` that specifies the `kubernetes.io/hostname` label of the `worker-2` node:

```
apiVersion: v1
kind: Pod
metadata:
  name: trouble-shooter
  labels:
    role: trouble-shooter
spec:
  nodeSelector:
    kubernetes.io/hostname: k3d-k3s-default-agent-1
  containers:
  - name: trouble-shooter
    image: g1g1/py-kube:0.3
    command: ["bash"]
    args: ["-c", "echo started...; while true ; do sleep 1 ; done"]
```

When creating this pod, it is indeed scheduled to the `k3d-k3s-default-agent-1` node:

```
$ k apply -f trouble-shooter.yaml
pod/trouble-shooter created

$ k get po trouble-shooter -o jsonpath='{.spec.nodeName}'
k3d-k3s-default-agent-1
```

Taints and tolerations

You can taint a node in order to prevent pods from being scheduled on the node. This can be useful, for example, if you don't want pods to be scheduled on your control plane nodes. Tolerations allow pods to declare that they can "tolerate" a specific node taint and then these pods can be scheduled on the tainted node. A node can have multiple taints and a pod can have multiple tolerations. A taint is a triplet: key, value, effect. The key and value are used to identify the taint. The effect is one of:

- `NoSchedule` (no pods will be scheduled to the node unless they tolerate the taint)
- `PreferNoSchedule` (soft version of `NoSchedule`; the scheduler will attempt to not schedule pods that don't tolerate the taint)
- `NoExecute` (no new pods will be scheduled, but also existing pods that don't tolerate the taint will be evicted)

Let's deploy `hue-learn` and `hue-reminders` on our k3d cluster:

```
$ k apply -f hue-learn-deployment.yaml
deployment.apps/hue-learn created

$ k apply -f hue-reminders-deployment.yaml
deployment.apps/hue-reminders created
```

Currently, there is a `hue-learn` pod that runs on the control plane node (`k3d-k3s-default-server-0`):

```
$ k get po -o wide
NAME                            READY   STATUS    RESTARTS   AGE     IP          NODE
NOMINATED NODE   READINESS GATES
trouble-shooter                 1/1     Running   0          2m20s   10.42.2.4   k3d-
k3s-default-agent-1    <none>           <none>
hue-learn-67d4649b58-tklxf      1/1     Running   0          18s     10.42.1.8   k3d-
k3s-default-server-0   <none>           <none>
hue-learn-67d4649b58-wk55w      1/1     Running   0          18s     10.42.0.3   k3d-
k3s-default-agent-0    <none>           <none>
hue-learn-67d4649b58-jkwwg      1/1     Running   0          18s     10.42.2.5   k3d-
k3s-default-agent-1    <none>           <none>
hue-reminders-9bdcd7489-2j65p   1/1     Running   0          6s      10.42.2.6   k3d-
k3s-default-agent-1    <none>           <none>
```

```
hue-reminders-9bdcd7489-wntpx   1/1     Running   0          6s      10.42.0.4   k3d-
k3s-default-agent-0    <none>           <none>
```

Let's taint our control plane node:

```
$ k taint nodes k3d-k3s-default-server-0 control-plane=true:NoExecute
node/k3d-k3s-default-server-0 tainted
```

We can now review the taint:

```
$ k get nodes k3d-k3s-default-server-0 -o jsonpath='{.spec.taints[0]}'
map[effect:NoExecute key:control-plane value:true]
```

Yeah, it worked! There are now no pods scheduled on the master node. The `hue-learn` pod on `k3d-k3s-default-server-0` was evicted and a new pod (`hue-learn-67d4649b58-bl8cn`) is now running on `k3d-k3s-default-agent-0`:

```
$ k get po -o wide
NAME                            READY   STATUS    RESTARTS   AGE     IP          NODE
NOMINATED NODE   READINESS GATES
trouble-shooter                 1/1     Running   0          33m     10.42.2.4   k3d-
k3s-default-agent-1   <none>           <none>
hue-learn-67d4649b58-wk55w      1/1     Running   0          31m     10.42.0.3   k3d-
k3s-default-agent-0   <none>           <none>
hue-learn-67d4649b58-jkwwg      1/1     Running   0          31m     10.42.2.5   k3d-
k3s-default-agent-1   <none>           <none>
hue-reminders-9bdcd7489-2j65p   1/1     Running   0          30m     10.42.2.6   k3d-
k3s-default-agent-1   <none>           <none>
hue-reminders-9bdcd7489-wntpx   1/1     Running   0          30m     10.42.0.4   k3d-
k3s-default-agent-0   <none>           <none>
hue-learn-67d4649b58-bl8cn      1/1     Running   0          2m53s   10.42.0.5   k3d-
k3s-default-agent-0   <none>           <none>
```

To allow pods to tolerate the taint, add a toleration to their spec such as:

```
tolerations:
- key: "control-plane"
  operator: "Equal"
  value: "true"
  effect: "NoSchedule"
```

Node affinity and anti-affinity

Node affinity is a more sophisticated form of the `nodeSelector`. It has three main advantages:

- Rich selection criteria (`nodeSelector` is just `AND` of exact matches on the labels)
- Rules can be soft
- You can achieve anti-affinity using operators like `NotIn` and `DoesNotExist`

Note that if you specify both `nodeSelector` and `nodeAffinity`, then the pod will be scheduled only to a node that satisfies both requirements.

For example, if we add the following section to our `trouble-shooter` pod, it will not be able to run on any node because it conflicts with the `nodeSelector`:

```
affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/hostname
            operator: NotIn
            values:
            - k3d-k3s-default-agent-1
```

Pod affinity and anti-affinity

Pod affinity and anti-affinity provide yet another avenue for managing where your workloads run. All the methods we discussed so far – node selectors, taints/tolerations, node affinity/anti-affinity – were about assigning pods to nodes. But pod affinity is about the relationships between different pods. Pod affinity has several other concepts associated with it: namespacing (since pods are namespaced), topology zone (node, rack, cloud provider zone, cloud provider region), and weight (for preferred scheduling). A simple example is if you want `hue-reminders` to always be scheduled with a `trouble-shooter` pod. Let's see how to define it in the pod template spec of the `hue-reminders` deployment:

```
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: role
                operator: In
                values:
                - trouble-shooter
            topologyKey: topology.kubernetes.io/zone # for clusters on cloud
providers
```

The topology key is a node label that Kubernetes will treat as identical for scheduling purposes. On cloud providers, it is recommended to use `topology.kubernetes.io/zone` when workloads should run in proximity to each other. In the cloud, a zone is the equivalent of a data center.

Then, after re-deploying `hue-reminders`, all the `hue-reminders` pods are scheduled to run on `k3d-k3s-default-agent-1` next to the `trouble-shooter` pod:

```
$ k apply -f hue-reminders-deployment-with-pod-affinity.yaml
deployment.apps/hue-reminders configured
$ k get po -o wide
NAME                              READY   STATUS    RESTARTS   AGE    IP          NODE
NOMINATED NODE   READINESS GATES
trouble-shooter                   1/1     Running   0          117m   10.42.2.4   k3d-
k3s-default-agent-1   <none>           <none>
hue-learn-67d4649b58-wk55w        1/1     Running   0          115m   10.42.0.3   k3d-
k3s-default-agent-0   <none>           <none>
hue-learn-67d4649b58-jkwwg        1/1     Running   0          115m   10.42.2.5   k3d-
k3s-default-agent-1   <none>           <none>
hue-learn-67d4649b58-bl8cn        1/1     Running   0          87m    10.42.0.5   k3d-
k3s-default-agent-0   <none>           <none>
hue-reminders-544d96785b-pd62t    0/1     Pending   0          50s    10.42.2.4   k3d-
k3s-default-agent-1   <none>           <none>
hue-reminders-544d96785b-wpmjj    0/1     Pending   0          50s    10.42.2.4   k3d-
k3s-default-agent-1   <none>           <none>
```

Pod topology spread constraints

Node affinity/anti-affinity and pod affinity/anti-affinity are sometimes too strict. You may want to spread your pods – it's okay if some pods of the same deployment end up on the same node. Pod topology spread constraints give you this flexibility. You can specify the max skew, which is how far you can be from the optimal spread, as well as the behavior when the constraint can't be satisfied (`DoNotSchedule` or `ScheduleAnyway`).

Here is our `hue-reminders` deployment with pod topology spread constraints:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hue-reminders
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hue
      service: reminders
  template:
    metadata:
      name: hue-reminders
      labels:
```

```
        app: hue
        service: reminders
    spec:
      topologySpreadConstraints:
        - maxSkew: 1
          topologyKey: node.kubernetes.io/instance-type
          whenUnsatisfiable: DoNotSchedule
          labelSelector:
            matchLabels:
              app: hue
              service: hue-reminders
      containers:
      - name: hue-reminders
        image: g1g1/hue-reminders:3.0
        ports:
        - containerPort: 80
```

We can see that after applying the manifest, the three pods were spread across the two agent nodes (the server node has a taint as you recall):

```
$  k apply -f hue-reminders-deployment-with-spread-contraitns.yaml
deployment.apps/hue-reminders created


$ k get po -o wide -l app=hue,service=reminders
NAME                             READY   STATUS    RESTARTS   AGE     IP
NODE                       NOMINATED NODE   READINESS GATES
hue-reminders-6664fccb8f-8bvf6   1/1     Running   0          4m40s   10.42.0.11
k3d-k3s-default-agent-0   <none>           <none>
hue-reminders-6664fccb8f-8qrbl   1/1     Running   0          3m59s   10.42.0.12
k3d-k3s-default-agent-0   <none>           <none>
hue-reminders-6664fccb8f-b5pbp   1/1     Running   0          56s     10.42.2.14
k3d-k3s-default-agent-1   <none>           <none>
```

The descheduler

Kubernetes is great at scheduling pods to nodes according to sophisticated placement rules. But, once a pod is scheduled, Kubernetes will not move it to another node if the original conditions changed. Here are some use cases that would benefit from moving workloads around:

- Certain nodes are experiencing under-utilization or over-utilization.
- The initial scheduling decision is no longer valid when taints or labels are modified on nodes, causing pod/node affinity requirements to no longer be met.
- Certain nodes have encountered failures, resulting in the migration of their pods to other nodes.
- Additional nodes are introduced to the clusters.

This is where the descheduler comes into play. The descheduler is not part of vanilla Kubernetes. You need to install it and define policies that determine which running pods may be evicted. It can run as a `Job`, `CronJob`, or `Deployment`. The descheduler will periodically check the current placement of pods and will evict pods that violate some policy. The pods will get rescheduled and then the standard Kubernetes scheduler will take care of scheduling them according to the current conditions.

Check it out here: `https://github.com/kubernetes-sigs/descheduler`.

In this section, we saw how the advanced scheduling mechanisms Kubernetes provides, as well as projects like the descheduler, can help Hue schedule its workload in an optimal way across the available infrastructure. In the next section, we will look at how to divide Hue workloads to a namespace to manage access to different resources.

Using namespaces to limit access

The Hue project is moving along nicely, and we have a few hundred microservices and about 100 developers and DevOps engineers working on it. Groups of related microservices emerge, and you notice that many of these groups are pretty autonomous. They are completely oblivious to the other groups. Also, there are some sensitive areas such as health and finance that you want to control access to more effectively. Enter namespaces.

Let's create a new service, `hue-finance`, and put it in a new namespace called `restricted`.

Here is the YAML file for the new `restricted` namespace:

```
kind: Namespace
apiVersion: v1
metadata:
  name: restricted
  labels:
    name: restricted
```

We can create it as usual:

```
$ kubectl create -f restricted-namespace.yaml
namespace "restricted" created
```

Once the namespace has been created, we can configure a context for the namespace:

```
$ k config set-context k3d-k3s-restricted --cluster k3d-k3s-default
--namespace=restricted --user restricted@k3d-k3s-default
Context "restricted" created.

$ k config use-context restricted
Switched to context "restricted".
```

Let's check our cluster configuration:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://0.0.0.0:53829
  name: k3d-k3s-default
contexts:
- context:
    cluster: k3d-k3s-default
    user: admin@k3d-k3s-default
  name: k3d-k3s-default
- context:
    cluster: ""
    namespace: restricted
    user: restricted@k3d-k3s-default
  name: restricted
current-context: restricted
kind: Config
preferences: {}
users:
- name: admin@k3d-k3s-default
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

As you can see, there are two contexts now and the current context is `restricted`. If we wanted to, we could even create dedicated users with their own credentials that are allowed to operate in the restricted namespace. Depending on the environment this can be easy or difficult and may involve creating certificates via Kubernetes certificate authorities. Cloud providers offer integration with their IAM systems.

To move along, I'll use the `admin@k3d-k3s-default` user's credentials and create a user named `restricted@k3d-k3s-default` directly in the `kubeconfig` file of the cluster:

```
users:
- name: restricted@k3d-k3s-default
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
```

Now, in this empty namespace, we can create our `hue-finance` service, and it will be separate from the other services in the default namespace:

```
$ k create -f hue-finance-deployment.yaml
deployment.apps/hue-finance created

$ k get pods
NAME                            READY   STATUS    RESTARTS   AGE
hue-finance-84c445f684-vh8qv    1/1     Running   0          7s
hue-finance-84c445f684-fjkxs    1/1     Running   0          7s
hue-finance-84c445f684-sppkq    1/1     Running   0          7s
```

You don't have to switch contexts. You can also use the `--namespace=<namespace>` and `--all-namespaces` command-line switches, but when operating for a while in the same non-default namespace, it's nice to set the context to that namespace.

Using Kustomization for hierarchical cluster structures

This is not a typo. Kubectl recently incorporated the functionality of Kustomize (`https://kustomize.io/`). It is a way to configure Kubernetes without templates. There was a lot of drama about the way the Kustomize functionality was integrated into kubectl itself, since there are other options and it was an open question if kubectl should be that opinionated. But, that's all in the past. The bottom line is that `kubectl apply -k` unlocks a treasure trove of configuration options. Let's understand what problem it helps us to solve and take advantage of it to help us manage Hue.

Understanding the basics of Kustomize

Kustomize was created as a response to template-heavy approaches like Helm to configure and customize Kubernetes clusters. It is designed around the principle of declarative application management. It takes a valid Kubernetes YAML manifest (base) and specializes it or extends it by overlaying additional YAML patches (overlays). Overlays depend on their bases. All files are valid YAML files. There are no placeholders.

A `kustomization.yaml` file controls the process. Any directory that contains a `kustomization.yaml` file is called a root. For example:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: staging
commonLabels:
    environment: staging
bases:
  - ../base

patchesStrategicMerge:
```

```
  - hue-learn-patch.yaml

resources:
  - namespace.yaml
```

Kustomize can work well in a GitOps environment where different Kustomizations live in a Git repo and changes to the bases, overlays, or `kustomization.yaml` files trigger a deployment.

One of the best use cases for Kustomize is organizing your system into multiple namespaces such as staging and production. Let's restructure the Hue platform deployment manifests.

Configuring the directory structure

First, we need a base directory that will contain the commonalities of all the manifests. Then we will have an `overlays` directory that contains `staging` and `production` sub-directories:

```
$ tree
.
├── base
│   ├── hue-learn.yaml
│   └── kustomization.yaml
├── overlays
│   ├── production
│   │   ├── kustomization.yaml
│   │   └── namespace.yaml
│   └── staging
│       ├── hue-learn-patch.yaml
│       ├── kustomization.yaml
│       └── namespace.yaml
```

The `hue-learn.yaml` file in the base directory is just an example. There may be many files there. Let's review it quickly:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-learner
  labels:
    tier: internal-service
spec:
  containers:
  - name: hue-learner
    image: g1g1/hue-learn:0.3
    resources:
      requests:
```

```
        cpu: 200m
        memory: 256Mi
    env:
    - name: DISCOVER_QUEUE
      value: dns
    - name: DISCOVER_STORE
      value: dns
```

It is very similar to the manifest we created earlier, but it doesn't have the `app: hue` label. It is not necessary because the label is provided by the `kustomization.yaml` file as a common label for all the listed resources:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
commonLabels:
  app: hue

resources:
  - hue-learn.yaml
```

Applying Kustomizations

We can observe the results by running the `kubectl kustomize` command on the base directory. You can see that the common label `app: hue` was added:

```
$ k kustomize base

apiVersion: v1
kind: Pod
metadata:
  labels:
    app: hue
    tier: internal-service
  name: hue-learner
spec:
  containers:
  - env:
    - name: DISCOVER_QUEUE
      value: dns
    - name: DISCOVER_STORE
      value: dns
    image: g1g1/hue-learn:0.3
    name: hue-learner
    resources:
```

```
      requests:
        cpu: 200m
        memory: 256Mi
```

In order to actually deploy the Kustomization, we can run `kubectl -k apply`. But, the base is not supposed to be deployed on its own. Let's dive into the `overlays/staging` directory and examine it.

The `namespace.yaml` file just creates the `staging` namespace. It will also benefit from all the Kustomizations as we'll soon see:

```
apiVersion: v1
kind: Namespace
metadata:
  name: staging
```

The `kustomization.yaml` file adds the common label `environment: staging`. It depends on the base directory and adds the `namespace.yaml` file to the resources list (which already includes `hue-learn.yaml` from the base):

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: staging
commonLabels:
    environment: staging
bases:
  - ../../base

patchesStrategicMerge:
  - hue-learn-patch.yaml

resources:
  - namespace.yaml
```

But, that's not all. The most interesting part of Kustomizations is patching.

Patching

Patches add or replace parts of manifests. They never remove existing resources or parts of resources. The `hue-learn-patch.yaml` updates the image from `g1g1/hue-learn:0.3` to `g1g1/hue-learn:0.4`:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-learner
spec:
  containers:
```

```
    - name: hue-learner
      image: g1g1/hue-learn:0.4
```

This is a strategic merge. Kustomize supports another type of patch called `JsonPatches6902`. It is based on RFC 6902 (`https://tools.ietf.org/html/rfc6902`). It is often more concise than a strategic merge. We can use YAML syntax for JSON 6902 patches. Here is the same patch of changing the image version to version 0.4 using `JsonPatches6902` syntax:

```
  - op: replace
    path: /spec/containers/0/image
    value: g1g1/hue-learn:0.4
```

Kustomizing the entire staging namespace

Here is what Kustomize generates when running it on the `overlays/staging` directory:

```
$ k kustomize overlays/staging

apiVersion: v1
kind: Namespace
metadata:
  labels:
    environment: staging
  name: staging
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: hue
    environment: staging
    tier: internal-service
  name: hue-learner
  namespace: staging
spec:
  containers:
  - env:
    - name: DISCOVER_QUEUE
      value: dns
    - name: DISCOVER_STORE
      value: dns
    image: g1g1/hue-learn:0.4
    name: hue-learner
    resources:
```

```
      requests:
        cpu: 200m
        memory: 256Mi
```

Note that the namespace didn't inherit the `app: hue` label from the base, but only the `environment: staging` label from its local Kustomization file. The `hue-learn` pod, on the other hand, got all labels as well the namespace designation.

It's time to deploy it to the cluster:

```
$ k apply -k overlays/staging
namespace/staging created
pod/hue-learner created
```

Now, we can review the pod in the newly created `staging` namespace:

```
$ k get po -n staging
NAME          READY   STATUS    RESTARTS   AGE
hue-learner   1/1     Running   0          21s
```

Let's check that the overlay worked and the image version is indeed 0.4:

```
$ k get po hue-learner -n staging -o jsonpath='{.spec.containers[0].image}'
g1g1/hue-learn:0.4
```

In this section, we covered the powerful structuring and reusability afforded by the Kustomize option. This is very important for a large-scale system like the Hue platform where a lot of workloads can benefit from a uniform structure and consistent foundation. In the next section, we will look at launching short-term jobs.

Launching jobs

Hue has evolved and has a lot of long-running processes deployed as microservices, but it also has a lot of tasks that run, accomplish some goal, and exit. Kubernetes supports this functionality via the `Job` resource. A Kubernetes job manages one or more pods and ensures that they run until success or failure. If one of the pods managed by the job fails or is deleted, then the job will run a new pod until it succeeds.

There are also many serverless or function-as-a-service solutions for Kubernetes, but they are built-on top of native Kubernetes. We will cover serverless computing in depth in *Chapter 12*, *Serverless Computing on Kubernetes*.

Here is a job that runs a Python process to compute the factorial of 5 (hint: it's 120):

```
apiVersion: batch/v1
kind: Job
metadata:
  name: factorial5
spec:
```

```
  template:
    metadata:
      name: factorial5
    spec:
      containers:
      - name: factorial5
        image: g1g1/py-kube:0.3
        command: ["python",
                  "-c",
                  "import math; print(math.factorial(5))"]
      restartPolicy: Never
```

Note that the `restartPolicy` must be either `Never` or `OnFailure`. The default value – `Always` – is invalid because a job doesn't restart after successful completion.

Let's start the job and check its status:

```
$ k create -f factorial-job.yaml
job.batch/factorial5 created

$ k get jobs
NAME         COMPLETIONS   DURATION   AGE
factorial5   1/1           4s         27s
```

The pods of completed tasks are displayed with a status of `Completed`. Note that job pods have a label called `job-name` with the name of the job, so it's easy to filter just the job pods:

```
$ k get po -l job-name=factorial5
NAME               READY   STATUS      RESTARTS   AGE
factorial5-dddzz   0/1     Completed   0          114s
```

Let's check out its output in the logs:

```
$ k logs factorial5-dddzz
120
```

Launching jobs one after another is fine for some use cases, but it is often useful to run jobs in parallel. In addition, it's important to clean up jobs after they are done as well as to run jobs periodically. Let's see how it's done.

Running jobs in parallel

You can also run a job with parallelism. There are two fields in the spec called `completions` and `parallelism`. The completions are set to 1 by default. If you want more than one successful completion, then increase this value. Parallelism determines how many pods to launch. A job will not launch more pods than needed for successful completions, even if the parallelism number is greater.

Let's run another job that just sleeps for 20 seconds until it has three successful completions. We'll use a parallelism factor of six, but only three pods will be launched:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: sleep20
spec:
  completions: 3
  parallelism: 6
  template:
    metadata:
      name: sleep20
    spec:
      containers:
      - name: sleep20
        image: g1g1/py-kube:0.3
        command: ["python",
                  "-c",
                  "import time; print('started...');
                   time.sleep(20); print('done.')"]
      restartPolicy: Never
```

Let's run the job and wait for all the pods to complete:

```
$ k create -f parallel-job.yaml
job.batch/sleep20 created
```

We can now see that all three pods completed and the pods are not ready because they already did their work:

```
$ k get pods -l job-name=sleep20
NAME            READY   STATUS      RESTARTS   AGE
sleep20-fqgst   0/1     Completed   0          4m5s
sleep20-2dv8h   0/1     Completed   0          4m5s
sleep20-kvn28   0/1     Completed   0          4m5s
```

Completed pods don't take up resources on the node, so other pods can get scheduled there.

Cleaning up completed jobs

When a job completes, it sticks around – and its pods, too. This is by design, so you can look at logs or connect to pods and explore. But normally, when a job has completed successfully, it is not needed anymore. It's your responsibility to clean up completed jobs and their pods.

The easiest way is to simply delete the `job` object, which will delete all the pods too:

```
$ kubectl get jobs
NAME         COMPLETIONS   DURATION   AGE
factorial5   1/1           2s         6h59m
sleep20      3/3           3m7s       5h54m

$ kubectl delete job factorial5
job.batch "factorial5" deleted

$ kubectl delete job sleep20
job.batch "sleep20" deleted
```

Scheduling cron jobs

Kubernetes cron jobs are jobs that run at a specified time, once or repeatedly. They behave as regular Unix cron jobs specified in the `/etc/crontab` file.

The `CronJob` resource became stable with Kubernetes 1.21. Here is the configuration to launch a cron job every minute to remind you to stretch. In the schedule, you may replace the '*' with '?':

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: cron-demo
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            cronjob-name: cron-demo
        spec:
          containers:
            - name: cron-demo
              image: g1g1/py-kube:0.3
              args:
                - python
                - -c
                - from datetime import datetime; print(f'[{datetime.now()}]
CronJob demo here...remember to stretch')
          restartPolicy: OnFailure
```

In the pod spec, under the job template, I added the label `cronjob-name: cron-demo`. The reason is that cron jobs and their pods are assigned names with a random prefix by Kubernetes. The label allows you to easily discover all the pods of a particular cron job. The pods will also have the `job-name` label because a cron job creates a `job` object for each invocation. However, the job name itself has a random prefix, so it doesn't help us discover the pods.

Let's run the cron job and observe the results after a minute:

```
$ k get cj
NAME        SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cron-demo   */1 * * * *   False     0        <none>          16s

$ k get job
NAME                 COMPLETIONS   DURATION   AGE
cron-demo-27600079   1/1           3s         2m45s
cron-demo-27600080   1/1           3s         105s
cron-demo-27600081   1/1           3s         45s

$ k get pods
NAME                          READY   STATUS      RESTARTS   AGE
cron-demo-27600080-dmcmq      0/1     Completed   0          2m6s
cron-demo-27600081-gjsvd      0/1     Completed   0          66s
cron-demo-27600082-sgjlh      0/1     Completed   0          6s
```

As you can see, every minute the cron job creates a new job with a different name. The pod of each job is labeled with its job name, but also with the cron job name – `cronjob-demo` – for easily aggregating all pods originating from this cron job.

As usual, you can check the output of a completed job's pod using the `logs` command:

```
$ k logs cron-demo-27600082-sgjlh
[2022-06-23 17:22:00.971343] CronJob demo here...remember to stretch
```

When you delete a cron job it stops scheduling new jobs and deletes all the existing `job` objects and all the pods it created.

You can use the designated label (`name=cron-demo` in this case) to locate all the `job` objects launched by the cron job:

```
$ k delete job -l name=cron-demo
job.batch "cron-demo-27600083" deleted
job.batch "cron-demo-27600084" deleted
job.batch "cron-demo-27600085" deleted
```

You can also suspend a cron job so it doesn't create more jobs without deleting completed jobs and pods. You can also manage how many jobs stick around by setting it in the spec history limits: `.spec.successfulJobsHistoryLimit` and `.spec.failedJobsHistoryLimit`.

In this section, we covered the important topics of launching jobs and controlling them. This is a critical aspect of the Hue platform, which needs to react to real-time events and handle them by launching jobs as well as periodically performing short tasks.

Mixing non-cluster components

Most real-time system components in the Kubernetes cluster will communicate with out-of-cluster components. Those could be completely external third-party services accessible through some API, but can also be internal services running in the same local network that, for various reasons, are not part of the Kubernetes cluster.

There are two categories here: inside the cluster network and outside the cluster network.

Outside-the-cluster-network components

These components have no direct access to the cluster. They can only access it through APIs, externally visible URLs, and exposed services. These components are treated just like any external user. Often, cluster components will just use external services, which pose no security issue. For example, in a previous company, we had a Kubernetes cluster that reported exceptions to a third-party service called Sentry (`https://sentry.io/welcome/`). It was one-way communication from the Kubernetes cluster to the third-party service. The Kubernetes cluster had the credentials to access Sentry and that was the extent of this one-way communication.

Inside-the-cluster-network components

These are components that run inside the network but are not managed by Kubernetes. There are many reasons to run such components. They could be legacy applications that have not been "kubernetized" yet, or some distributed data store that is not easy to run inside Kubernetes. The reason to run these components inside the network is for performance, and to have isolation from the outside world so traffic between these components and pods can be more secure. Being part of the same network ensures low latency, and the reduced need for opening up the network for communication is both convenient and can be more secure.

Managing the Hue platform with Kubernetes

In this section, we will look at how Kubernetes can help operate a huge platform such as Hue. Kubernetes itself provides a lot of capabilities to orchestrate pods and manage quotas and limits, detecting and recovering from certain types of generic failures (hardware malfunctions, process crashes, and unreachable services). But, in a complicated system such as Hue, pods and services may be up and running but in an invalid state or waiting for other dependencies in order to perform their duties. This is tricky because if a service or pod is not ready yet but is already receiving requests, then you need to manage it somehow: fail (puts responsibility on the caller), retry (how many times? for how long? how often?), and queue for later (who will manage this queue?).

It is often better if the system at large can be aware of the readiness state of different components, or if components are visible only when they are truly ready. Kubernetes doesn't know Hue, but it provides several mechanisms such as liveness probes, readiness probes, startup probes, and init containers to support application-specific management of your cluster.

Using liveness probes to ensure your containers are alive

The kubelet watches over your containers. If a container process crashes, the kubelet will take care of it based on the restart policy. But this is not enough in many cases. Your process may not crash but instead run into an infinite loop or a deadlock. The restart policy might not be nuanced enough. With a liveness probe, you get to decide when a container is considered alive. If a liveness probe fails, Kubernetes will restart your container. Here is a pod template for the Hue music service. It has a `livenessProbe` section, which uses the `httpGet` probe. An HTTP probe requires a scheme (`http` or `https`, default to `http`), a host (default to `PodIp`), a path, and a port. The probe is considered successful if the HTTP status is between `200` and `399`. Your container may need some time to initialize, so you can specify an `initialDelayInSeconds`. The kubelet will not hit the liveness check during this period:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: music
    service: music
  name: hue-music
spec:
  containers:
    - image: g1g1/hue-music
      livenessProbe:
        httpGet:
          path: /pulse
          port: 8888
          httpHeaders:
            - name: X-Custom-Header
              value: ItsAlive
        initialDelaySeconds: 30
        timeoutSeconds: 1
      name: hue-music
```

If a liveness probe fails for any container, then the pod's restart policy goes into effect. Make sure your restart policy is not `Never`, because that will make the probe useless.

There are three other types of liveness probes:

- `TcpSocket` – Just checks that a port is open
- `Exec` – Runs a command that returns 0 for success
- `gRPC` – Follows the gRPC health-checking protocol (`https://github.com/grpc/grpc/blob/master/doc/health-checking.md`)

Using readiness probes to manage dependencies

Readiness probes are used for a different purpose. Your container may be up and running and pass its liveness probe, but it may depend on other services that are unavailable at the moment. For example, `hue-music` may depend on access to a data service that contains your listening history. Without access, it is unable to perform its duties. In this case, other services or external clients should not send requests to the `hue-music` service, but there is no need to restart it. Readiness probes address this use case. When a readiness probe fails for a container, the container's pod will be removed from any service endpoint it is registered with. This ensures that requests don't flood services that can't process them. Note that you can also use readiness probes to temporarily remove pods that are overbooked until they drain some internal queue.

Here is a sample readiness probe. I use the `exec` probe here to execute a custom command. If the command exits a non-zero exit code, the container will be torn down:

```
readinessProbe:
  exec:
    command:
        - /usr/local/bin/checker
        - --full-check
        - --data-service=hue-multimedia-service
  initialDelaySeconds: 60
  timeoutSeconds: 5
```

It is fine to have both a readiness probe and a liveness probe on the same container as they serve different purposes.

Using startup probes

Some applications (mostly legacy) may have long initialization periods. In this case, liveness probes may fail and cause the container to restart before it finishes initialization. This is where startup probes come in. If a startup probe is configured, liveness and readiness checks are skipped until the startup is completed. At this point, the startup probe is not invoked anymore and normal liveness and readiness probes take over.

For example, in the following configuration snippet, the startup probe will check for 5 minutes every 10 seconds if the container has started (using the same liveness check as the liveness probe). If the startup probe fails 30 times (300 seconds = 5 minutes) then the container will be restarted and get 5 more minutes to try and initialize itself. But, if it passes the startup probe check within 5 minutes, then the liveness probe is in effect and any failure of the liveness check will result in a restart:

```
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080
```

```
livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 1
  periodSeconds: 10

startupProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10
```

Employing init containers for orderly pod bring-up

Liveness, readiness, and startup probes are great. They recognize that, at startup, there may be a period where the container is not ready yet, but shouldn't be considered failed. To accommodate that, there is the `initialDelayInSeconds` setting where containers will not be considered failed. But, what if this initial delay is potentially very long? Maybe, in most cases, a container is ready after a couple of seconds and ready to process requests, but because the initial delay is set to 5 minutes just in case, we waste a lot of time when the container is idle. If the container is part of a high-traffic service, then many instances can all sit idle for five minutes after each upgrade and pretty much make the service unavailable.

Init containers address this problem. A pod may have a set of init containers that run to completion before other containers are started. An init container can take care of all the non-deterministic initialization and let application containers with their readiness probe have a minimal delay.

Init containers are especially useful for pod-level initialization purposes like waiting for volumes to be ready. There is some overlap between init containers and startup probes and the choice depends on the specific use case.

Init containers came out of beta in Kubernetes 1.6. You specify them in the pod spec as the `initContainers` field, which is very similar to the `containers` field. Here is an example:

```
kind: Pod
metadata:
  name: hue-fitness
spec:
  containers:
  - name: hue-fitness
    image: busybox
  initContainers:
  - name: install
    image: busybox
```

Pod readiness and readiness gates

Pod readiness was introduced in Kubernetes 1.11 and became stable in Kubernetes 1.14. While readiness probes allow you to determine at the container level if it's ready to serve requests, the overall infrastructure that supports delivering traffic to the pod might not be ready yet. For example, the service, network policy, and load balancer might take some extra time. This can be a problem, especially during rolling deployments where Kubernetes might terminate the old pods before the new pods are really ready, which will cause degradation in service capacity and even cause a service outage in extreme cases (all old pods were terminated and no new pod is fully ready).

This is the problem that the pod readiness gates address. The idea is to extend the concept of pod readiness to check additional conditions in addition to making sure all the containers are ready. This is done by adding a new field to the `PodSpec` called `readinessGates`. You can specify a set of conditions that must be satisfied for the pod to be considered ready. In the following example, the pod is not ready because the `www.example.com/feature-1` condition has the `status: False`:

```
Kind: Pod
...
spec:
  readinessGates:
    - conditionType: "www.example.com/feature-1"
status:
  conditions:
    - type: Ready  # this is a builtin PodCondition
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2023-01-01T00:00:00Z
    - type: "www.example.com/feature-1"   # an extra PodCondition
      status: "False"
      lastProbeTime: null
      lastTransitionTime: 2023-01-01T00:00:00Z
  containerStatuses:
    - containerID: docker://abcd...
      ready: true
...
```

Sharing with DaemonSet pods

DaemonSet pods are pods that are deployed automatically, one per node (or a designated subset of the nodes). They are typically used for keeping an eye on nodes and ensuring they are operational. This is a very important function, which we will cover in *Chapter 13*, *Monitoring Kubernetes Clusters*. But they can be used for much more. The nature of the default Kubernetes scheduler is that it schedules pods based on resource availability and requests. If you have lots of pods that don't require a lot of resources, similarly many pods will be scheduled on the same node.

Let's consider a pod that performs a small task and then, every second, sends a summary of all its activities to a remote service. Now, imagine that, on average, 50 of these pods are scheduled on the same node. This means that, every second, 50 pods make 50 network requests with very little data. How about we cut it down by 50× to just a single network request? With a `DaemonSet` pod, all the other 50 pods can communicate with it instead of talking directly to the remote service. The `DaemonSet` pod will collect all the data from the 50 pods and, once a second, will report it in aggregate to the remote service. Of course, that requires the remote service API to support aggregate reporting. The nice thing is that the pods themselves don't have to be modified; they will just be configured to talk to the `DaemonSet` pod on `localhost` instead of the remote service. The `DaemonSet` pod serves as an aggregating proxy. It can also implement retry and other similar functions.

The interesting part about this configuration file is that the `hostNetwork`, `hostPID`, and `hostIPC` options are set to `true`. This enables the pods to communicate efficiently with the proxy, utilizing the fact they are running on the same physical host:

```
apiVersion: apps/v1
kind-fission | fission
kind: DaemonSet
metadata:
  name: hue-collect-proxy kind-fission | fission
  labels:
    tier: stats
    app: hue-collect-proxy
spec:
  selector:
    matchLabels:
      tier: stats
      app: hue-collect-proxy
  template:
    metadata:
      labels:
        tier: stats
        app: hue-collect-proxy
    spec:
      hostPID: true
      hostIPC: true
      hostNetwork: true
      containers:
      - name: hue-collect-proxy
        image: busybox
```

In this section, we looked at how to manage the Hue platform on Kubernetes and ensure that Hue components are deployed reliably and accessed when they are ready using capabilities such as init containers, readiness gates, and DaemonSets. In the next section, we'll see where the Hue platform could potentially go in the future.

Evolving the Hue platform with Kubernetes

In this section, we'll discuss other ways to extend the Hue platform and service additional markets and communities. The question is always, what Kubernetes features and capabilities can we use to address new challenges or requirements?

This is a hypothetical section for thinking big and using Hue as an example of a massively complicated system.

Utilizing Hue in an enterprise

An enterprise often can't run in the cloud, either due to security and compliance reasons or for performance reasons because the system has to work with data and legacy systems that are not cost-effective to move to the cloud. Either way, Hue for enterprise must support on-premises clusters and/or bare-metal clusters.

While Kubernetes is most often deployed in the cloud and even has a special cloud-provider interface, it doesn't depend on the cloud and can be deployed anywhere. It does require more expertise, but enterprise organizations that already run systems on their own data centers may have that expertise or develop it.

Advancing science with Hue

Hue is so great at integrating information from multiple sources that it would be a boon for the scientific community. Consider how Hue can help with multi-disciplinary collaboration between scientists from different disciplines.

A network of scientific communities might require deployment across multiple geographically distributed clusters. Enter multi-cluster Kubernetes. Kubernetes has this use case in mind and evolves its support. We will discuss it at length in *Chapter 11*, *Running Kubernetes on Multiple Clusters*.

Educating the kids of the future with Hue

Hue can be utilized for education and provide many services to online education systems. But, privacy concerns may prevent deploying Hue for kids as a single, centralized system. One possibility is to have a single cluster, with namespaces for different schools. Another deployment option is that each school or county has its own Hue Kubernetes cluster. In the second case, Hue for education must be extremely easy to operate to cater to schools without a lot of technical expertise. Kubernetes can help a lot by providing self-healing and auto-scaling features and capabilities for Hue, to be as close to zero-administration as possible.

Summary

In this chapter, we designed and planned the development, deployment, and management of the Hue platform – an imaginary omniscient and omnipotent system – built on microservice architecture. We used Kubernetes as the underlying orchestration platform, of course, and delved into many of its concepts and resources. In particular, we focused on deploying pods for long-running services as opposed to jobs for launching short-term or cron jobs, explored internal services versus external services, and also used namespaces to segment a Kubernetes cluster. We looked into the various workload scheduling mechanisms of Kubernetes. Then, we looked at the management of a large system such as Hue with liveness probes, readiness probes, startup probes, init containers, and daemon sets.

You should now feel comfortable architecting web-scale systems composed of microservices and understand how to deploy and manage them in a Kubernetes cluster.

In the next chapter, we will look into the super-important area of storage. Data is king, but it is often the least flexible element of the system. Kubernetes provides a storage model and many options for integrating with various storage solutions.

6

Managing Storage

In this chapter, we'll look at how Kubernetes manages storage. Storage is very different from compute, but at a high level they are both resources. Kubernetes as a generic platform takes the approach of abstracting storage behind a programming model and a set of plugins for storage providers. First, we'll go into detail about the conceptual storage model and how storage is made available to containers in the cluster. Then, we'll cover the common cloud platform storage providers, such as **Amazon Web Services** (**AWS**), **Google Compute Engine** (**GCE**), and Azure. Then we'll look at a prominent open source storage provider, GlusterFS from Red Hat, which provides a distributed filesystem. We'll also look into another solution – Ceph – that manages your data in containers as part of the Kubernetes cluster using the Rook operator. We'll see how Kubernetes supports the integration of existing enterprise storage solutions. Finally, we will explore the **Constrainer Storage Interface** (**CSI**) and all the advanced capabilities it brings to the table.

This chapter will cover the following main topics:

- Persistent volumes walk-through
- Demonstrating persistent volume storage end to end
- Public cloud storage volume types – GCE, AWS, and Azure
- GlusterFS and Ceph volumes in Kubernetes
- Integrating enterprise storage into Kubernetes
- The Container Storage Interface

At the end of this chapter, you'll have a solid understanding of how storage is represented in Kubernetes, the various storage options in each deployment environment (local testing, public cloud, and enterprise), and how to choose the best option for your use case.

You should try the code samples in this chapter on minikube, or another cluster that supports storage adequately. The KinD cluster has some problems related to labeling nodes, which is necessary for some storage solutions.

Persistent volumes walk-through

In this section, we will understand the Kubernetes storage conceptual model and see how to map persistent storage into containers, so they can read and write. Let's start by understanding the problem of storage.

Containers and pods are ephemeral. Anything a container writes to its own filesystem gets wiped out when the container dies. Containers can also mount directories from their host node and read or write to them. These will survive container restarts, but the nodes themselves are not immortal. Also, if the pod itself is evicted and scheduled to a different node, the pod's containers will not have access to the old node host's filesystem.

There are other problems, such as ownership of mounted hosted directories when the container dies. Just imagine a bunch of containers writing important data to various data directories on their host and then going away, leaving all that data all over the nodes with no direct way to tell what container wrote what data. You can try to record this information, but where would you record it? It's pretty clear that for a large-scale system, you need persistent storage accessible from any node to reliably manage the data.

Understanding volumes

The basic Kubernetes storage abstraction is the volume. Containers mount volumes that are bound to their pod, and they access the storage wherever it may be as if it's in their local filesystem. This is nothing new, and it is great, because as a developer who writes applications that need access to data, you don't have to worry about where and how the data is stored. Kubernetes supports many types of volumes with their own distinctive features. Let's review some of the main volume types.

Using emptyDir for intra-pod communication

It is very simple to share data between containers in the same pod using a shared volume. Container 1 and container 2 simply mount the same volume and can communicate by reading and writing to this shared space. The most basic volume is the `emptyDir`. An `emptyDir` volume is an empty directory on the host. Note that it is not persistent because when the pod is evicted or deleted, the contents are erased. If a container just crashes, the pod will stick around, and the restarted container can access the data in the volume. Another very interesting option is to use a RAM disk, by specifying the medium as `Memory`. Now, your containers communicate through shared memory, which is much faster, but more volatile of course. If the node is restarted, the `emptyDir`'s volume contents are lost.

Here is a pod configuration file that has two containers that mount the same volume, called `shared-volume`. The containers mount it in different paths, but when the `hue-global-listener` container is writing a file to `/notifications`, the `hue-job-scheduler` will see that file under `/incoming`:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-scheduler
spec:
```

```
    containers:
    - image: g1g1/hue-global-listener:1.0
      name: hue-global-listener
      volumeMounts:
      - mountPath: /notifications
        name: shared-volume
    - image: g1g1/hue-job-scheduler:1.0
      name: hue-job-scheduler
      volumeMounts:
      - mountPath: /incoming
        name: shared-volume
    volumes:
    - name: shared-volume
      emptyDir: {}
```

To use the shared memory option, we just need to add `medium: Memory` to the `emptyDir` section:

```
    volumes:
    - name: shared-volume
      emptyDir:
       medium: Memory
```

Note that memory-based `emptyDir` counts toward the container's memory limit.

To verify it worked, let's create the pod and then write a file using one container and read it using the other container:

```
$ k create -f hue-scheduler.yaml
pod/hue-scheduler created
```

Note that the pod has two containers:

```
$ k get pod hue-scheduler -o json | jq .spec.containers
[
  {
    "image": "g1g1/hue-global-listener:1.0",
    "name": "hue-global-listener",
    "volumeMounts": [
      {
        "mountPath": "/notifications",
        "name": "shared-volume"
      },
      ...
    ]
    ...
```

```
  },
  {
    "image": "g1g1/hue-job-scheduler:1.0",
    "name": "hue-job-scheduler",
    "volumeMounts": [
      {
        "mountPath": "/incoming",
        "name": "shared-volume"
      },
      ...
    ]
    ...
  }
]
```

Now, we can create a file in the `/notifications` directory of the `hue-global-listener` container and list it in the `/incoming` directory of the `hue-job-scheduler` container:

```
$ kubectl exec -it hue-scheduler -c hue-global-listener -- touch /notifications/1.txt
$ kubectl exec -it hue-scheduler -c hue-job-scheduler -- ls /incoming
1.txt
```

As you can see, we are able to see a file that was created in one container in the file system of another container; thereby, the containers can communicate via the shared file system.

Using HostPath for intra-node communication

Sometimes, you want your pods to get access to some host information (for example, the Docker daemon) or you want pods on the same node to communicate with each other. This is useful if the pods know they are on the same host. Since Kubernetes schedules pods based on available resources, pods usually don't know what other pods they share the node with. There are several cases where a pod can rely on other pods being scheduled with it on the same node:

- In a single-node cluster, all pods obviously share the same node
- `DaemonSet` pods always share a node with any other pod that matches their selector
- Pods with required pod affinity are always scheduled together

For example, in *Chapter 5*, *Using Kubernetes Resources in Practice*, we discussed a `DaemonSet` pod that serves as an aggregating proxy to other pods. Another way to implement this behavior is for the pods to simply write their data to a mounted volume that is bound to a host directory, and the `DaemonSet` pod can directly read it and act on it.

A `HostPath` volume is a host file or directory that is mounted into a pod. Before you decide to use the `HostPath` volume, make sure you understand the consequences:

- It is a security risk since access to the host filesystem can expose sensitive data (e.g. kubelet keys)
- The behavior of pods with the same configuration might be different if they are data-driven and the files on their host are different
- It can violate resource-based scheduling because Kubernetes can't monitor `HostPath` resources
- The containers that access host directories must have a security context with `privileged` set to `true` or, on the host side, you need to change the permissions to allow writing
- It's difficult to coordinate disk usage across multiple pods on the same node.
- You can easily run out of disk space

Here is a configuration file that mounts the `/coupons` directory into the `hue-coupon-hunter` container, which is mapped to the host's `/etc/hue/data/coupons` directory:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-coupon-hunter
spec:
  containers:
  - image: the_g1g1/hue-coupon-hunter
    name: hue-coupon-hunter
    volumeMounts:
    - mountPath: /coupons
      name: coupons-volume
  volumes:
  - name: coupons-volume
    host-path:
        path: /etc/hue/data/coupons
```

Since the pod doesn't have a privileged security context, it will not be able to write to the host directory. Let's change the container spec to enable it by adding a security context:

```
  - image: the_g1g1/hue-coupon-hunter
    name: hue-coupon-hunter
    volumeMounts:
    - mountPath: /coupons
      name: coupons-volume
    securityContext:
      privileged: true
```

In the following diagram, you can see that each container has its own local storage area inaccessible to other containers or pods, and the host's `/data` directory is mounted as a volume into both container 1 and container 2:

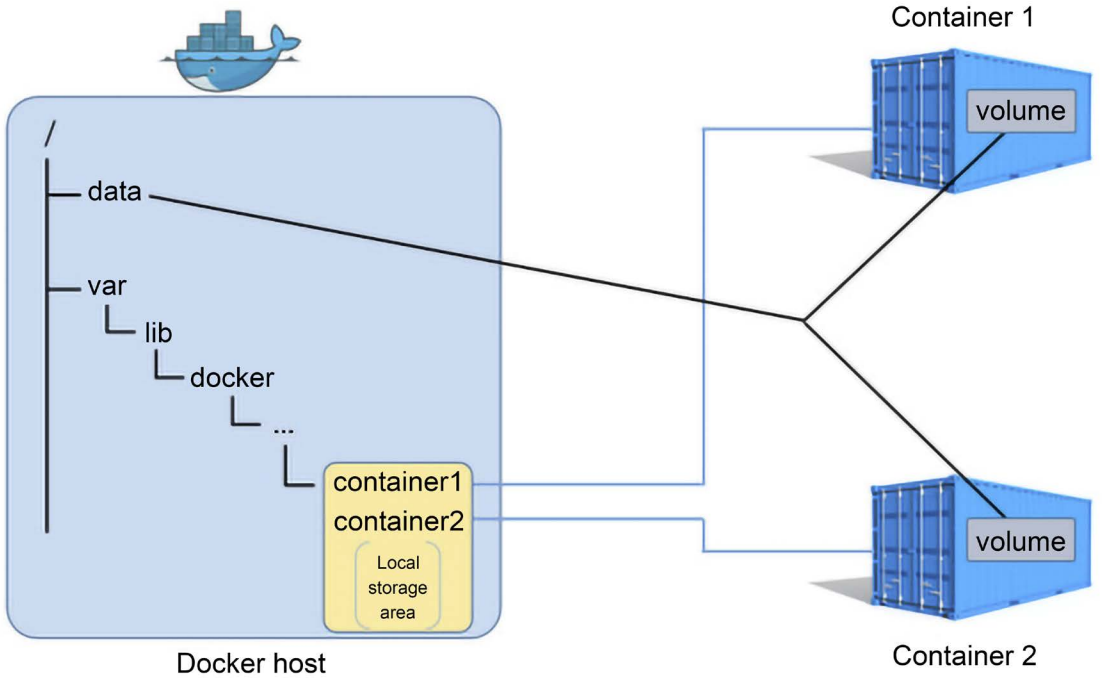


Figure 6.1: Container local storage

Using local volumes for durable node storage

Local volumes are similar to `HostPath`, but they persist across pod restarts and node restarts. In that sense they are considered persistent volumes. They were added in Kubernetes 1.7. As of Kubernetes 1.14 they are considered stable. The purpose of local volumes is to support Stateful Sets where specific pods need to be scheduled on nodes that contain specific storage volumes. Local volumes have node affinity annotations that simplify the binding of pods to the storage they need to access.

We need to define a storage class for using local volumes. We will cover storage classes in depth later in this chapter. In one sentence, storage classes use a provisioner to allocate storage to pods. Let's define the storage class in a file called `local-storage-class.yaml` and create it:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
$ k create -f local-storage-class.yaml
storageclass.storage.k8s.io/local-storage created
```

Now, we can create a persistent volume using the storage class that will persist even after the pod that's using it is terminated:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
  labels:
    release: stable
    capacity: 10Gi
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/disk-1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - k3d-k3s-default-agent-1
```

Provisioning persistent volumes

While `emptyDir` volumes can be mounted and used by containers, they are not persistent and don't require any special provisioning because they use existing storage on the node. `HostPath` volumes persist on the original node, but if a pod is restarted on a different node, it can't access the `HostPath` volume from its previous node. Local volumes are real persistent volumes that use storage provisioned ahead of time by administrators or dynamic provisioning via storage classes. They persist on the node and can survive pod restarts and rescheduling and even node restarts. Some persistent volumes use external storage (not a disk physically attached to the node) provisioned ahead of time by administrators. In cloud environments, the provisioning may be very streamlined, but it is still required, and as a Kubernetes cluster administrator you have to at least make sure your storage quota is adequate and monitor usage versus quota diligently.

Remember that persistent volumes are resources that the Kubernetes cluster is using, similar to nodes. As such they are not managed by the Kubernetes API server.

You can provision resources statically or dynamically.

Provisioning persistent volumes statically

Static provisioning is straightforward. The cluster administrator creates persistent volumes backed up by some storage media ahead of time, and these persistent volumes can be claimed by containers.

Provisioning persistent volumes dynamically

Dynamic provisioning may happen when a persistent volume claim doesn't match any of the statically provisioned persistent volumes. If the claim specified a storage class and the administrator configured that class for dynamic provisioning, then a persistent volume may be provisioned on the fly. We will see examples later when we discuss persistent volume claims and storage classes.

Provisioning persistent volumes externally

Kubernetes originally contained a lot of code for storage provisioning "in-tree" as part of the main Kubernetes code base. With the introduction of CSI, storage provisioners started to migrate out of Kubernetes core into volume plugins (AKA out-of-tree). External provisioners work just like in-tree dynamic provisioners but can be deployed and updated independently. Most in-tree storage provisioners have been migrated out-of-tree. Check out this project for a library and guidelines for writing external storage provisioners: `https://github.com/kubernetes-sigs/sig-storage-lib-external-provisioner`.

Creating persistent volumes

Here is the configuration file for an NFS persistent volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-777
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
    - ReadOnlyMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.2
  nfs:
    path: /tmp
    server: nfs-server.default.svc.cluster.local
```

A persistent volume has a spec and metadata that possibly includes a storage class name. Let's focus on the spec here. There are six sections: capacity, volume mode, access modes, reclaim policy, storage class, and the volume type (`nfs` in the example).

Capacity

Each volume has a designated amount of storage. Storage claims may be satisfied by persistent volumes that have at least that amount of storage. In the example, the persistent volume has a capacity of 10 gibibytes (a single gibibyte is 2 to the power of 30 bytes).

```
capacity:
    storage: 10Gi
```

It is important when allocating static persistent volumes to understand the storage request patterns. For example, if you provision 20 persistent volumes with 100 GiB capacity and a container claims a persistent volume with 150 GiB, then this claim will not be satisfied even though there is enough capacity overall in the cluster.

Volume mode

The optional volume mode was added in Kubernetes 1.9 as an Alpha feature (moved to Beta in Kubernetes 1.13) for static provisioning. It lets you specify if you want a file system (`Filesystem`) or raw storage (`Block`). If you don't specify volume mode, then the default is `Filesystem`, just like it was pre-1.9.

Access modes

There are three access modes:

- `ReadOnlyMany`: Can be mounted read-only by many nodes
- `ReadWriteOnce`: Can be mounted as read-write by a single node
- `ReadWriteMany`: Can be mounted as read-write by many nodes

The storage is mounted to nodes, so even with `ReadWriteOnce`, multiple containers on the same node can mount the volume and write to it. If that causes a problem, you need to handle it through some other mechanism (for example, claim the volume only in DaemonSet pods that you know will have just one per node).

Different storage providers support some subset of these modes. When you provision a persistent volume, you can specify which modes it will support. For example, NFS supports all modes, but in the example, only these modes were enabled:

```
accessModes:
    - ReadWriteMany
    - ReadOnlyMany
```

Reclaim policy

The reclaim policy determines what happens when a persistent volume claim is deleted. There are three different policies:

- `Retain` – the volume will need to be reclaimed manually
- `Delete` – the content, the volume, and the backing storage are removed
- `Recycle` – delete content only (`rm -rf /volume/*`)

The `Retain` and `Delete` policies mean the persistent volume is not available anymore for future claims. The `Recycle` policy allows the volume to be claimed again.

At the moment, NFS and `HostPath` support the recycle policy, while AWS EBS, GCE PD, Azure disk, and Cinder volumes support the delete policy. Note that dynamically provisioned volumes are always deleted.

Storage class

You can specify a storage class using the optional `storageClassName` field of the spec. If you do then only persistent volume claims that specify the same storage class can be bound to the persistent volume. If you don't specify a storage class, then only PV claims that don't specify a storage class can be bound to it.

```
    storageClassName: slow
```

Volume type

The volume type is specified by name in the spec. There is no `volumeType` stanza in the spec. In the preceding example, `nfs` is the volume type:

```
nfs:
    path: /tmp
    server: 172.17.0.8
```

Each volume type may have its own set of parameters. In this case, it's a path and server.

We will go over various volume types later.

Mount options

Some persistent volume types have additional mount options you can specify. The mount options are not validated. If you provide an invalid mount option, the volume provisioning will fail. For example, NFS supports additional mount options:

```
    mountOptions:
      - hard
      - nfsvers=4.1
```

Now that we have looked at provisioning a single persistent volume, let's look at projected volumes, which add more flexibility and abstraction of storage.

Projected volumes

Projected volumes allow you to mount multiple persistent volumes into the same directory. You need to be careful of naming conflicts of course.

The following volume types support projected volumes:

- `ConfigMap`
- `Secret`
- `SownwardAPI`
- `ServiceAccountToken`

The snippet below projects a `ConfigMap` and a `Secret` into the same directory:

```
apiVersion: v1
kind: Pod
metadata:
  name: projected-volumes-demo
spec:
  containers:
  - name: projected-volumes-demo
    image: busybox:1.28
    volumeMounts:
    - name: projected-volumes-demo
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: projected-volumes-demo
    projected:
      sources:
      - secret:
          name: the-user
          items:
            - key: username
              path: the-group/the-user
      - configMap:
          name: the-config-map
          items:
            - key: config
              path: the-group/the-config-map
```

The parameters for projected volumes are very similar to regular volumes. The exceptions are:

- To maintain consistency with `ConfigMap` naming, the field `secretName` has been updated to `name` for secrets.
- The `defaultMode` can only be set at the projected level and cannot be specified individually for each volume source (but you can specify the mode explicitly for each projection).

Let's look at a special kind of projected volume – the `serviceAccountToken` exceptions.

serviceAccountToken projected volumes

Kubernetes pods can access the Kubernetes API server using the permissions of the service account associated with the pod. `serviceAccountToken` projected volumes give you more granularity and control from a security standpoint. The token can have an expiration and a specific audience.

More details are available here: `https://kubernetes.io/docs/concepts/storage/projected-volumes/#serviceaccounttoken`.

Creating a local volume

Local volumes are static persistent disks that are allocated on a specific node. They are similar to `HostPath` volumes, but Kubernetes knows which node a local volume belongs to and will schedule pods that bind to that local volume always to that node. This means the pod will not be evicted and scheduled to another node where the data is not available.

Let's create a local volume. First, we need to create a backing directory. For KinD and k3d clusters you can access the node through Docker:

```
$ docker exec -it k3d-k3s-default-agent-1 mkdir -p /mnt/disks/disk-1
$ docker exec -it k3d-k3s-default-agent-1 ls -la /mnt/disks
total 12
drwxr-xr-x 3 0 0 4096 Jun 29 21:40 .
drwxr-xr-x 3 0 0 4096 Jun 29 21:40 ..
drwxr-xr-x 2 0 0 4096 Jun 29 21:40 disk-1
```

For minikube you need to use `minikube ssh`.

Now, we can create a local volume backed by the `/mnt/disks/disk1` directory:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
  labels:
    release: stable
    capacity: 10Gi
spec:
  capacity:
```

```
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/disk-1
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - k3d-k3s-default-agent-1
```

Here is the `create` command:

```
$ k create -f local-volume.yaml
persistentvolume/local-pv created

$ k get pv
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS    REASON   AGE
local-pv   10Gi       RWO            Delete           Bound    default/local-storage-
claim   local-storage            6m44s
```

Making persistent volume claims

When containers want access to some persistent storage they make a claim (or rather, the developer and cluster administrator coordinate on necessary storage resources to claim). Here is a sample claim that matches the persistent volume from the previous section - *Creating a local volume*:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-storage-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
```

```
  storageClassName: local-storage
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: capacity, operator: In, values: [8Gi, 10Gi]}
```

Let's create the claim and then explain what the different pieces do:

```
$ k create -f local-persistent-volume-claim.yaml
persistentvolumeclaim/local-storage-claim created

$ k get pvc
NAME                  STATUS    VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS
AGE
local-storage-claim   WaitForFirstConsumer   local-pv   10Gi       RWO
local-storage   21m
```

The name `local-storage-claim` will be important later when mounting the claim into a container.

The access mode in the spec is `ReadWriteOnce`, which means if the claim is satisfied no other claim with the `ReadWriteOnce` access mode can be satisfied, but claims for `ReadOnlyMany` can still be satisfied.

The resources section requests 8 GiB. This can be satisfied by our persistent volume, which has a capacity of 10 Gi. But, this is a little wasteful because 2 Gi will not be used by definition.

The storage class name is `local-storage`. As mentioned earlier it must match the class name of the persistent volume. However, with PVC there is a difference between an empty class name (`""`) and no class name at all. The former (an empty class name) matches persistent volumes with no storage class name. The latter (no class name) will be able to bind to persistent volumes only if the `DefaultStorageClass` admission plugin is turned on and the default storage class is used.

The selector section allows you to filter available volumes further. For example, here the volume must match the label `release:stable` and also have a label with either `capacity:8Gi` or `capacity:10Gi`. Imagine that we have several other volumes provisioned with capacities of 20 Gi and 50 Gi. We don't want to claim a 50 Gi volume when we only need 8 Gi.



Kubernetes always tries to match the smallest volume that can satisfy a claim, but if there are no 8 Gi or 10 Gi volumes then the labels will prevent assigning a 20 Gi or 50 Gi volume and use dynamic provisioning instead.

It's important to realize that claims don't mention volumes by name. You can't claim a specific volume. The matching is done by Kubernetes based on storage class, capacity, and labels.

Finally, persistent volume claims belong to a namespace. Binding a persistent volume to a claim is exclusive. That means that a persistent volume will be bound to a namespace. Even if the access mode is `ReadOnlyMany` or `ReadWriteMany`, all the pods that mount the persistent volume claim must be from that claim's namespace.

Mounting claims as volumes

OK. We have provisioned a volume and claimed it. It's time to use the claimed storage in a container. This turns out to be pretty simple. First, the persistent volume claim must be used as a volume in the pod and then the containers in the pod can mount it, just like any other volume. Here is a pod manifest that specifies the persistent volume claim we created earlier (bound to the local persistent volume we provisioned):

```
kind: Pod
apiVersion: v1
metadata:
  name: the-pod
spec:
  containers:
    - name: the-container
      image: g1g1/py-kube:0.3
      volumeMounts:
      - mountPath: "/mnt/data"
        name: persistent-volume
  volumes:
    - name: persistent-volume
      persistentVolumeClaim:
        claimName: local-storage-claim
```

The key is in the `persistentVolumeClaim` section under volumes. The claim name (`local-storage-claim` here) uniquely identifies within the current namespace the specific claim and makes it available as a volume (named `persistent-volume` here). Then, the container can refer to it by its name and mount it to `"/mnt/data"`.

Before we create the pod it's important to note that the persistent volume claim didn't actually claim any storage yet and wasn't bound to our local volume. The claim is pending until some container actually attempts to mount a volume using the claim:

```
$ k get pvc
NAME                  STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS
AGE
local-storage-claim   Pending                                      local-storage
6m14s
```

Now, the claim will be bound when creating the pod:

```
$ k create -f pod-with-local-claim.yaml
pod/the-pod created

$ k get pvc
NAME                  STATUS   VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS
AGE
local-storage-claim   Bound    local-pv   100Gi      RWO            local-storage
20m
```

Raw block volumes

Kubernetes 1.9 added this capability as an Alpha feature. Kubernetes 1.13 moved it to Beta. Since Kubernetes 1.18 it is GA.

Raw block volumes provide direct access to the underlying storage, which is not mediated via a file system abstraction. This is very useful for applications that require high-performance storage like databases or when consistent I/O performance and low latency are needed. The following storage providers support raw block volumes:

- AWSElasticBlockStore
- AzureDisk
- **FC** (**Fibre Channel**)
- GCE Persistent Disk
- iSCSI
- Local volume
- OpenStack Cinder
- RBD (Ceph Block Device)
- VsphereVolume

In addition many CSI storage providers also offer raw block volume. For the full list check out: `https://kubernetes-csi.github.io/docs/drivers.html`.

Here is how to define a raw block volume using the FireChannel provider:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
```

```
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
```

A matching **Persistent Volume Claim** (**PVC**) MUST specify `volumeMode: Block` as well. Here is what it looks like:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 10Gi
```

Pods consume raw block volumes as devices under `/dev` and NOT as mounted filesystems. Containers can then access these devices and read/write to them. In practice this means that I/O requests to block storage go directly to the underlying block storage and don't pass through the file system drivers. This is in theory faster, but in practice it can actually decrease performance if your application benefits from file system buffering.

Here is a pod with a container that binds the `block-pvc` with the raw block storage as a device named `/dev/xdva`:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: ["tail -f /dev/null"]
      volumeDevices:
        - name: data
          devicePath: /dev/xvda
```

```
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc
```

CSI ephemeral volumes

We will cover the **Container Storage Interface** (**CSI**) in detail later in the chapter in the section *The Container Storage Interface*. CSI ephemeral volumes are backed by local storage on the node. These volumes' lifecycles are tied to the pod's lifecycle. In addition, they can only be mounted by containers of that pod, which is useful for populating secrets and certificates directly into a pod, without going through a Kubernetes secret object.

Here is an example of a pod with a CSI ephemeral volume:

```
kind: Pod
apiVersion: v1
metadata:
  name: the-pod
spec:
  containers:
    - name: the-container
      image: g1g1/py-kube:0.3
      volumeMounts:
        - mountPath: "/data"
          name: the-volume
      command: [ "sleep", "1000000" ]
  volumes:
    - name: the-volume
      csi:
        driver: inline.storage.kubernetes.io
        volumeAttributes:
          key: value
```

CSI ephemeral volumes have been GA since Kubernetes 1.25. However, they may not be supported by all CSI drivers. As usual check the list: `https://kubernetes-csi.github.io/docs/drivers.html`.

Generic ephemeral volumes

Generic ephemeral volumes are yet another volume type that is tied to the pod lifecycle. When the pod is gone the generic ephemeral volume is gone.

This volume type actually creates a full-fledged persistent volume claim. This provides several capabilities:

- The storage for the volume can be either local or network-attached.
- The volume has the option to be provisioned with a fixed size.
- Depending on the driver and specified parameters, the volume may contain initial data.
- If supported by the driver, typical operations such as snapshotting, cloning, resizing, and storage capacity tracking can be performed on the volumes.

Here is an example of a pod with a generic ephemeral volume:

```
kind: Pod
apiVersion: v1
metadata:
  name: the-pod
spec:
  containers:
    - name: the-container
      image: g1g1/py-kube:0.3
      volumeMounts:
        - mountPath: "/data"
          name: the-volume
      command: [ "sleep", "1000000" ]
  volumes:
    - name: the-volume
      ephemeral:
        volumeClaimTemplate:
          metadata:
            labels:
              type: generic-ephemeral-volume
          spec:
            accessModes: [ "ReadWriteOnce" ]
            storageClassName: generic-storage
            resources:
              requests:
                storage: 1Gi
```

Note that from a security point of view users that have permission to create pods, but not PVCs, can now create PVCs via generic ephemeral volumes. To prevent that it is possible to use admission control.

Storage classes

We've run into storage classes already. What are they exactly? Storage classes let an administrator configure a cluster with custom persistent storage (as long as there is a proper plugin to support it). A storage class has a name in the metadata (it must be specified in the `storageClassName` file of the claim), a provisioner, a reclaim policy, and parameters.

We declared a storage class for local storage earlier. Here is a sample storage class that uses AWS EBS as a provisioner (so, it works only on AWS):

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
reclaimPolicy: Retain
allowVolumeExpansion: true
mountOptions:
  - debug
volumeBindingMode: Immediate
```

You may create multiple storage classes for the same provisioner with different parameters. Each provisioner has its own parameters.

The currently supported provisioners are:

- AWSElasticBlockStore
- AzureFile
- AzureDisk
- CephFS
- Cinder
- FC
- FlexVolume
- Flocker
- GCE Persistent Disk
- GlusterFS
- iSCSI
- Quobyte
- NFS
- RBD
- VsphereVolume

- PortworxVolume
- ScaleIO
- StorageOS
- Local

This list doesn't contain provisioners for other volume types, such as `configMap` or `secret`, that are not backed by your typical network storage. Those volume types don't require a storage class. Utilizing volume types intelligently is a major part of architecting and managing your cluster.

Default storage class

The cluster administrator can also assign a default storage class. When a default storage class is assigned and the `DefaultStorageClass` admission plugin is turned on, then claims with no storage class will be dynamically provisioned using the default storage class. If the default storage class is not defined or the admission plugin is not turned on, then claims with no storage class can only match volumes with no storage class.

We covered a lot of ground and a lot of options for provisioning storage and using it in different ways. Let's put everything together and show the whole process from start to finish.

Demonstrating persistent volume storage end to end

To illustrate all the concepts, let's do a mini demonstration where we create a `HostPath` volume, claim it, mount it, and have containers write to it. We will use k3d for this part.

Let's start by creating a `hostPath` volume using the `dir` storage class. Save the following in `dir-persistent-volume.yaml`:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: dir-pv
spec:
  storageClassName: dir
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/tmp/data"
```

Then, let's create it:

```
$ k create -f dir-persistent-volume.yaml
persistentvolume/dir-pv created
```

To check out the available volumes, you can use the resource type `persistentvolumes` or `pv` for short:

```
$ k get pv
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS    REASON   AGE
dir-pv     1Gi        RWX            Retain           Available           dir
22s
```

The capacity is 1 GiB as requested. The reclaim policy is `Retain` because host path volumes are retained (not destroyed). The status is `Available` because the volume has not been claimed yet. The access mode is specified as `RWX`, which means `ReadWriteMany`. All of the access modes have a shorthand version:

- `RWO` – `ReadWriteOnce`
- `ROX` – `ReadOnlyMany`
- `RWX` – `ReadWriteMany`

We have a persistent volume. Let's create a claim. Save the following to `dir-persistent-volume-claim.yaml`:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: dir-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Then, run the following command:

```
$ k create -f  dir-persistent-volume-claim.yaml
persistentvolumeclaim/dir-pvc created
```

Let's check the claim and the volume:

```
$ k get pvc
NAME                  STATUS   VOLUME     CAPACITY   ACCESS MODES   STORAGECLASS
AGE
dir-pvc               Bound    dir-pv     1Gi        RWX            dir
106s

$ k get pv
NAME       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS    REASON   AGE
dir-pv     1Gi        RWX            Retain           Bound    default/dir-pvc   dir
4m25s
```

As you can see, the claim and the volume are bound to each other and reference each other. The reason the binding works is that the same storage class is used by the volume and the claim. But, what happens if they don't match? Let's remove the storage class from the persistent volume claim and see what happens. Save the following persistent volume claim to `some-persistent-volume-claim.yaml`:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: some-pvc
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Then, create it:

```
$ k create -f some-persistent-volume-claim.yaml
persistentvolumeclaim/some-pvc created
```

Ok. It was created. Let's check it out:

```
$ k get pvc some-pvc
NAME       STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
some-pvc   Pending                                      local-path     3m29s
```

Very interesting. The `some-pvc` claim was associated with the `local-path` storage class that we never specified, but it is still pending. Let's understand why.

Here is the `local-path` storage class:

```
$ k get storageclass local-path -o yaml
kind: StorageClass
metadata:
  annotations:
    objectset.rio.cattle.io/applied: H4sIAAAAAAAAA/4yRT+vUMBCGv4rMua1bu1tKwIO
u7EUEQdDzNJlux6aZkkwry7LfXbIqrIffn2PyZN7hfXIFXPg7xcQSwEBSiXimaupSxfJ2q6GAiYMDA9
/+oKPHlKCAmRQdKoK5AoYgisoSUj5K/5OsJtIqslQWVT3lNM4xUDzJ5VegWJ63CQxMTXogW128+czBvf/
gnIQXIwLOBAa8WPTl30qvGkoL2jw5rT2V6ZKUZij+SbG5eZVRDKR0F8SpdDTg6rW8YzCgcSW4FeCxJ/+
sjxHTCAbqrhmag20Pw9DbZtfu210z7JuhPnQ719m2w3cOe7fPof81W1DHfLlE2Th/IEUwEDHYkWJe8PCs
gJgL8PxVPNsLGPhEnjRr2cSvM33k4Dicv4jLC34g60niiWPSo4S0zhTh9jsAAP//ytgh5S0CAAA
    objectset.rio.cattle.io/id: ""
    objectset.rio.cattle.io/owner-gvk: k3s.cattle.io/v1, Kind=Addon
    objectset.rio.cattle.io/owner-name: local-storage
    objectset.rio.cattle.io/owner-namespace: kube-system
```

```
    storageclass.kubernetes.io/is-default-class: "true"
  creationTimestamp: "2022-06-22T18:16:56Z"
  labels:
    objectset.rio.cattle.io/hash: 183f35c65ffbc3064603f43f1580d8c68a2dabd4
  name: local-path
  resourceVersion: "290"
  uid: b51cf456-f87e-48ac-9062-4652bf8f683e
provisioner: rancher.io/local-path
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

It is a storage class that comes with k3d (k3s).

Note the annotation: `storageclass.kubernetes.io/is-default-class: "true"`. It tells Kubernetes that this is the default storage class. Since our PVC had no storage class name it was associated with the default storage class. But, why is the claim still pending? The reason is that `volumeBindingMode` is `WaitForFirstConsumer`. This means that the volume for the claim will be provisioned dynamically only when a container attempts to mount the volume via the claim.

Back to our `dir-pvc`. The final step is to create a pod with two containers and assign the claim as a volume to both of them. Save the following to `shell-pod.yaml`:

```
kind: Pod
apiVersion: v1
metadata:
  name: just-a-shell
  labels:
    name: just-a-shell
spec:
  containers:
    - name: a-shell
      image: g1g1/py-kube:0.3
      command: ["sleep", "10000"]
      volumeMounts:
        - mountPath: "/data"
          name: pv
    - name: another-shell
      image: g1g1/py-kube:0.3
      command: ["sleep", "10000"]
      volumeMounts:
        - mountPath: "/another-data"
          name: pv
  volumes:
    - name: pv
```

```
      persistentVolumeClaim:
        claimName: dir-pvc
```

This pod has two containers that use the `g1g1/py-kube:0.3` image and both just sleep for a long time. The idea is that the containers will keep running, so we can connect to them later and check their file system. The pod mounts our persistent volume claim with a volume name of `pv`. Note that the volume specification is done at the pod level just once and multiple containers can mount it into different directories.

Let's create the pod and verify that both containers are running:

```
$ k create -f shell-pod.yaml
pod/just-a-shell created

$ k get po just-a-shell -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP            NODE
NOMINATED NODE   READINESS GATES
just-a-shell   2/2     Running   0          74m   10.42.2.104   k3d-k3s-default-
agent-1   <none>           <none>
```

Then, connect to the node (`k3d-k3s-default-agent-1`). This is the host whose `/tmp/data` is the pod's volume that is mounted as `/data` and `/another-data` into each of the running containers:

```
$ docker exec -it k3d-k3s-default-agent-1 sh
/ #
```

Then, let's create a file in the `/tmp/data` directory on the host. It should be visible by both containers via the mounted volume:

```
/ # echo "yeah, it works" > /tmp/data/cool.txt
```

Let's verify from the outside that the file `cool.txt` is indeed available:

```
$ docker exec -it k3d-k3s-default-agent-1 cat /tmp/data/cool.txt
yeah, it works
```

Next, let's verify the file is available in the containers (in their mapped directories):

```
$ k exec -it just-a-shell -c a-shell -- cat  /data/cool.txt
yeah, it works

$ k exec -it just-a-shell -c another-shell -- cat  /another-data/cool.txt
yeah, it works
```

We can even create a new file, `yo.txt`, in one of the containers and see that it's available to the other container or to the node itself:

```
$ k exec -it just-a-shell -c another-shell – bash –c "echo yo  > /another-data/
yo.txt"
yo /another-data/yo.txt
```

```
$ k exec -it just-a-shell -c a-shell cat /data/yo.txt
yo

$ k exec -it just-a-shell -c another-shell cat /another-data/yo.txt
yo
```

Yes. Everything works as expected and both containers share the same storage.

Public cloud storage volume types – GCE, AWS, and Azure

In this section, we'll look at some of the common volume types available in the leading public cloud platforms. Managing storage at scale is a difficult task that eventually involves physical resources, similar to nodes. If you choose to run your Kubernetes cluster on a public cloud platform, you can let your cloud provider deal with all these challenges and focus on your system. But it's important to understand the various options, constraints, and limitations of each volume type.



Many of the volume types we will go over used to be handled by in-tree plugins (part of core Kubernetes), but have now migrated to out-of-tree CSI plugins.

The CSI migration feature allows in-tree plugins that have corresponding out-of-tree CSI plugins to direct operations toward the out-of-tree plugins as a transitioning measure.

We will cover the CSI itself later.

AWS Elastic Block Store (EBS)

AWS provides the **Elastic Block Store** (**EBS**) as persistent storage for EC2 instances. An AWS Kubernetes cluster can use AWS EBS as persistent storage with the following limitations:

- The pods must run on AWS EC2 instances as nodes
- Pods can only access EBS volumes provisioned in their availability zone
- An EBS volume can be mounted on a single EC2 instance

Those are severe limitations. The restriction for a single availability zone, while great for performance, eliminates the ability to share storage at scale or across a geographically distributed system without custom replication and synchronization. The limit of a single EBS volume to a single EC2 instance means even within the same availability zone, pods can't share storage (even for reading) unless you make sure they run on the same node.

This is an example of an in-tree plugin that also has a CSI driver and supports CSIMigration. That means that if the CSI driver for AWS EBS (`ebs.csi.aws.com`) is installed, then the in-tree plugin will redirect all plugin operations to the out-of-tree plugin.

It is also possible to disable loading the in-tree `awsElasticBlockStore` storage plugin from being loaded by setting the `InTreePluginAWSUnregister` feature gate to `true` (the default is `false`).

Check out all the feature gates here: `https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/`.

Let's see how to define an AWS EBS persistent volume (static provisioning):

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-pv
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 5Gi
  csi:
    driver: ebs.csi.aws.com
    volumeHandle: {EBS volume ID}
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: topology.ebs.csi.aws.com/zone
              operator: In
              values:
                - {availability zone}
```

Then you need to define a PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Finally, a pod can mount the PVC:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - image: some-container
    name: some-container
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: ebs-claim
```

AWS Elastic File System (EFS)

AWS has a service called the **Elastic File System** (**EFS**). This is really a managed NFS service. It uses the NFS 4.1 protocol and has many benefits over EBS:

- Multiple EC2 instances can access the same files across multiple availability zones (but within the same region)
- Capacity is automatically scaled up and down based on actual usage
- You pay only for what you use
- You can connect on-premise servers to EFS over VPN
- EFS runs off SSD drives that are automatically replicated across availability zones

That said, EFS is more expansive than EBS even when you consider the automatic replication to multiple AZs (assuming you fully utilize your EBS volumes). The recommended way to use EFS via its dedicated CSI driver: `https://github.com/kubernetes-sigs/aws-efs-csi-driver`.

Here is an example of static provisioning. First, define the persistent volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
```

```
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  csi:
    driver: efs.csi.aws.com
    volumeHandle: <Filesystem Id>
```

You can find the `Filesystem Id` using the AWS CLI:

```
aws efs describe-file-systems --query "FileSystems[*].FileSystemId"
```

Then define a PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ""
  resources:
    requests:
      storage: 1Gi
```

Here is a pod that consumes it:

```
piVersion: v1
kind: Pod
metadata:
  name: efs-app
spec:
  containers:
  - name: app
    image: centos
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo $(date -u) >> /data/out.txt; sleep 5;
done"]
    volumeMounts:
    - name: persistent-storage
      mountPath: /data
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: efs-claim
```

You can also use dynamic provisioning by defining a proper storage class instead of creating a static volume:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: efs-sc
provisioner: efs.csi.aws.com
parameters:
  provisioningMode: efs-ap
  fileSystemId: <Filesystem Id>
  directoryPerms: "700"
  gidRangeStart: "1000" # optional
  gidRangeEnd: "2000" # optional
  basePath: "/dynamic_provisioning" # optional
```

The PVC is similar, but now uses the storage class name:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-claim
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: efs-sc
  resources:
    requests:
      storage: 5Gi
```

The pod consumes the PVC just like before:

```
apiVersion: v1
kind: Pod
metadata:
  name: efs-app
spec:
  containers:
    - name: app
      image: centos
      command: ["/bin/sh"]
      args: ["-c", "while true; do echo $(date -u) >> /data/out; sleep 5;
done"]
      volumeMounts:
```

```
        - name: persistent-storage
          mountPath: /data
  volumes:
    - name: persistent-storage
      persistentVolumeClaim:
        claimName: efs-claim
```

GCE persistent disk

The `gcePersistentDisk` volume type is very similar to `awsElasticBlockStore`. You must provision the disk ahead of time. It can only be used by GCE instances in the same project and zone. But the same volume can be used as read-only on multiple instances. This means it supports `ReadWriteOnce` and `ReadOnlyMany`. You can use a GCE persistent disk to share data as read-only between multiple pods in the same zone.

It also has a CSI driver called `pd.csi.storage.gke.io` and supports CSIMigration.

If the pod that's using a persistent disk in `ReadWriteOnce` mode is controlled by a replication controller, a replica set, or a deployment, the replica count must be 0 or 1. Trying to scale beyond 1 will fail for obvious reasons.

Here is a storage class for GCE persistent disk using the CSI driver:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-gce-pd
provisioner: pd.csi.storage.gke.io
parameters:
  labels: key1=value1,key2=value2
volumeBindingMode: WaitForFirstConsumer
```

Here is the PVC:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: gce-pd-pvc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: csi-gce-pd
  resources:
    requests:
      storage: 200Gi
```

A Pod can consume it for dynamic provisioning:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - image: some-image
    name: some-container
    volumeMounts:
    - mountPath: /pd
      name: some-volume
  volumes:
  - name: some-volume
    persistentVolumeClaim:
       claimName: gce-pd-pvc
       readOnly: false
```

The GCE persistent disk has supported a regional disk option since Kubernetes 1.10 (in Beta). Regional persistent disks automatically sync between two zones. Here is what the storage class looks like for a regional persistent disk:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-gce-pd
provisioner: pd.csi.storage.gke.io
parameters:
  type: pd-standard
  replication-type: regional-pd
volumeBindingMode: WaitForFirstConsumer
```

Google Cloud Filestore

Google Cloud Filestore is the managed NFS file service of GCP. Kubernetes doesn't have an in-tree plugin for it and there is no general-purpose supported CSI driver.

However, there is a CSI driver used on GKE and if you are adventurous, you may want to try it even if you're installing Kubernetes yourself on GCP and want to use Google Cloud Storage as a storage option.

See: `https://github.com/kubernetes-sigs/gcp-filestore-csi-driver`.

Azure data disk

The Azure data disk is a virtual hard disk stored in Azure storage. It's similar in capabilities to AWS EBS or a GCE persistent disk.

It also has a CSI driver called `disk.csi.azure.com` and supports CSIMigration. See: `https://github.com/kubernetes-sigs/azuredisk-csi-driver`.

Here is an example of defining an Azure disk persistent volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-azuredisk
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: managed-csi
  csi:
    driver: disk.csi.azure.com
    readOnly: false
    volumeHandle: /subscriptions/{sub-id}/resourcegroups/{group-name}/
providers/microsoft.compute/disks/{disk-id}
    volumeAttributes:
      fsType: ext4
```

In addition to the mandatory `diskName` and `diskURI` parameters, it also has a few optional parameters:

- `kind`: The available options for disk storage configurations are `Shared` (allowing multiple disks per storage account), `Dedicated` (providing a single blob disk per storage account), or `Managed` (offering an Azure-managed data disk). The default is `Shared`.
- `cachingMode`: The disk caching mode. This must be one of `None`, `ReadOnly`, or `ReadWrite`. The default is `None`.
- `fsType`: The filesystem type set to mount. The default is `ext4`.
- `readOnly`: Whether the filesystem is used as `readOnly`. The default is `false`.

Azure data disks are limited to 32 GiB. Each Azure VM can have up to 32 data disks. Larger VM sizes can have more disks attached. You can attach an Azure data disk to a single Azure VM.

As usual you should create a PVC and consume it in a pod (or a pod controller).

Azure file

In addition to the data disk, Azure has also a shared filesystem similar to AWS EFS. However, Azure file storage uses the SMB/CIFS protocol (it supports SMB 2.1 and SMB 3.0). It is based on the Azure storage platform and has the same availability, durability, scalability, and geo-redundancy capabilities as Azure Blob, Table, or Queue storage.

In order to use Azure file storage, you need to install on each client VM the `cifs-utils` package. You also need to create a secret, which is a required parameter:

```
apiVersion: v1
kind: Secret
metadata:
  name: azure-file-secret
type: Opaque
data:
  azurestorageaccountname: <base64 encoded account name>
  azurestorageaccountkey: <base64 encoded account key>
```

Here is a pod that uses Azure file storage:

```
apiVersion: v1
kind: Pod
metadata:
 name: some-pod
spec:
 containers:
  - image: some-container
    name: some-container
    volumeMounts:
      - name: some-volume
        mountPath: /azure
 volumes:
      - name: some-volume
        azureFile:
          secretName: azure-file-secret
         shareName: azure-share
          readOnly: false
```

Azure file storage supports sharing within the same region as well as connecting on-premise clients.

This covers the public cloud storage volume types. Let's look at some distributed storage volumes you can install on your own in your cluster.

GlusterFS and Ceph volumes in Kubernetes

GlusterFS and Ceph are two distributed persistent storage systems. GlusterFS is, at its core, a network filesystem. Ceph is, at its core, an object store. Both expose block, object, and filesystem interfaces. Both use the `xfs` filesystem under the hood to store the data and metadata as `xattr` attributes. There are several reasons why you may want to use GlusterFS or Ceph as persistent volumes in your Kubernetes cluster:

- You run on-premises and cloud storage is not available
- You may have a lot of data and applications that access the data in GlusterFS or Ceph
- You have operational expertise managing GlusterFS or Ceph
- You run in the cloud, but the limitations of the cloud platform persistent storage are a non-starter

Let's take a closer look at GlusterFS.

Using GlusterFS

GlusterFS is intentionally simple, exposing the underlying directories as they are and leaving it to clients (or middleware) to handle high availability, replication, and distribution. GlusterFS organizes the data into logical volumes, which encompass multiple nodes (machines) that contain bricks, which store files. Files are allocated to bricks according to DHT (distributed hash table). If files are renamed or the GlusterFS cluster is expanded or rebalanced, files may be moved between bricks. The following diagram shows the GlusterFS building blocks:

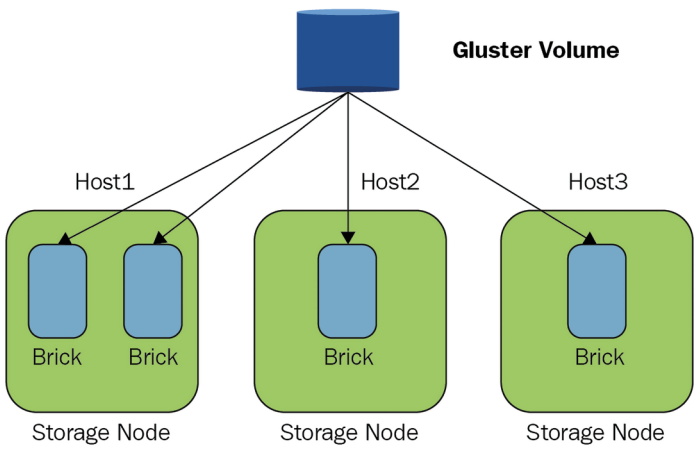


Figure 6.2: GlusterFS building blocks

To use a GlusterFS cluster as persistent storage for Kubernetes (assuming you have an up-and-running GlusterFS cluster), you need to follow several steps. In particular, the GlusterFS nodes are managed by the plugin as a Kubernetes service.

Creating endpoints

Here is an example of an endpoints resource that you can create as a normal Kubernetes resource using `kubectl create`:

```
kind: Endpoints
apiVersion: v1
metadata:
  name: glusterfs-cluster
subsets:
- addresses:
  - ip: 10.240.106.152
  ports:
  - port: 1
- addresses:
  - ip: 10.240.79.157
  ports:
  - port: 1
```

Adding a GlusterFS Kubernetes service

To make the endpoints persistent, you use a Kubernetes service with no selector to indicate the endpoints are managed manually:

```
kind: Service
apiVersion: v1
metadata:
  name: glusterfs-cluster
spec:
  ports:
  - port: 1
```

Creating pods

Finally, in the pod spec's `volumes` section, provide the following information:

```
volumes:
- name: glusterfsvol
  glusterfs:
    endpoints: glusterfs-cluster
    path: kube_vol
    readOnly: true
```

The containers can then mount `glusterfsvol` by name.

The endpoints tell the GlusterFS volume plugin how to find the storage nodes of the GlusterFS cluster.

There was an effort to create a CSI driver for GlusterFS, but it was abandoned: `https://github.com/gluster/gluster-csi-driver`.

After covering GlusterFS let's look at CephFS.

Using Ceph

Ceph's object store can be accessed using multiple interfaces. Unlike GlusterFS, Ceph does a lot of work automatically. It does distribution, replication, and self-healing all on its own. The following diagram shows how RADOS – the underlying object store – can be accessed in multiple ways.

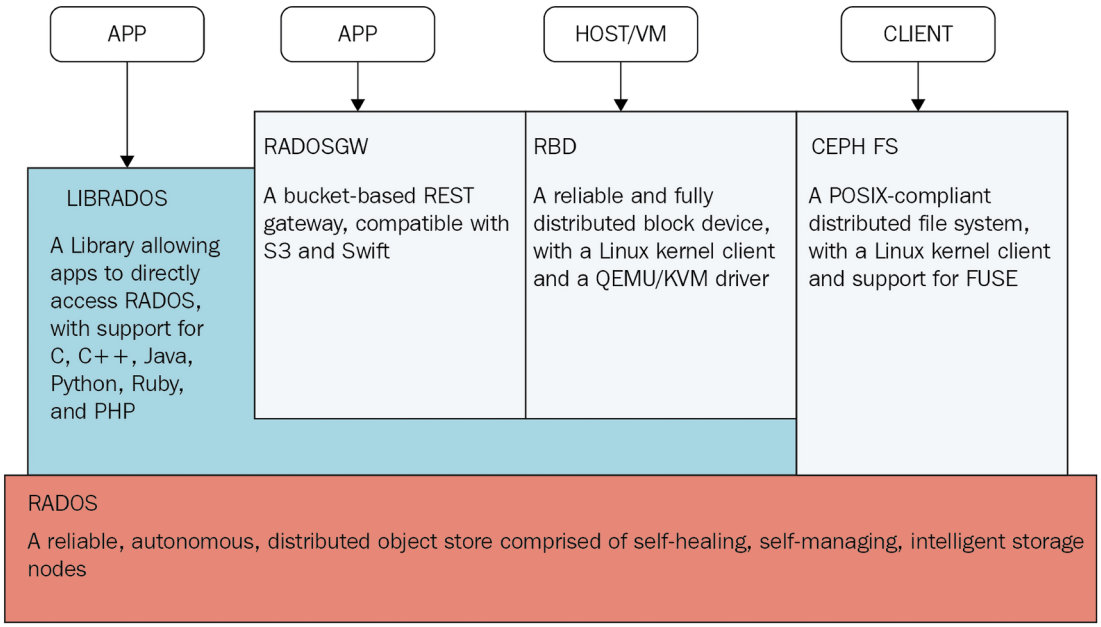


Figure 6.3: Accessing RADOS

Kubernetes supports Ceph via the **Rados Block Device** (**RBD**) interface.

Connecting to Ceph using RBD

You must install `ceph-common` on each node of the Kubernetes cluster. Once you have your Ceph cluster up and running, you need to provide some information required by the Ceph RBD volume plugin in the pod configuration file:

- `monitors`: Ceph monitors.
- `pool`: The name of the RADOS pool. If not provided, the default RBD pool is used.
- `image`: The image name that RBD has created.
- `user`: The RADOS username. If not provided, the default admin is used.

- `keyring`: The path to the keyring file. If not provided, the default `/etc/ceph/keyring` is used.
- `secretName`: The name of the authentication secrets. If provided, `secretName` overrides `keyring`. Note: see the following paragraph about how to create a secret.
- `fsType`: The filesystem type (`ext4`, `xfs`, and so on) that is formatted on the device.
- `readOnly`: Whether the filesystem is used as `readOnly`.

If the Ceph authentication secret is used, you need to create a secret object:

```
apiVersion: v1
kind: Secret
metadata:
  name: ceph-secret
type: "kubernetes.io/rbd"
data:
  key: QVFCMTZWMVZvRjVtRXhBQTVrQ1FzN2JCajhWVUxSdzI2Qzg0SEE9PQ==
```

The secret type is `kubernetes.io/rbd`.

Here is a sample pod that uses Ceph through RBD with a secret using the in-tree provider:

```
apiVersion: v1
kind: Pod
metadata:
  name: rbd2
spec:
  containers:
    - image: kubernetes/pause
      name: rbd-rw
      volumeMounts:
      - name: rbdpd
        mountPath: /mnt/rbd
  volumes:
    - name: rbdpd
      rbd:
```

```
        monitors:
        - '10.16.154.78:6789'
        - '10.16.154.82:6789'
        - '10.16.154.83:6789'
        pool: kube
        image: foo
        fsType: ext4
        readOnly: true
        user: admin
        secretRef:
          name: ceph-secret
```

Ceph RBD supports `ReadWriteOnce` and `ReadOnlyMany` access modes. But, these days it is best to work with Ceph via Rook.

Rook

Rook is an open source cloud native storage orchestrator. It is currently a graduated CNCF project. It used to provide a consistent experience on top of multiple storage solutions like Ceph, edgeFS, Cassandra, Minio, NFS, CockroachDB, and YugabyteDB. But, eventually it laser-focused on supporting only Ceph. Here are the features Rook provides:

- Automating deployment
- Bootstrapping
- Configuration
- Provisioning
- Scaling
- Upgrading
- Migration
- Scheduling
- Lifecycle management
- Resource management
- Monitoring
- Disaster recovery

Rook takes advantage of modern Kubernetes best practices like CRDs and operators.

Here is the Rook architecture:

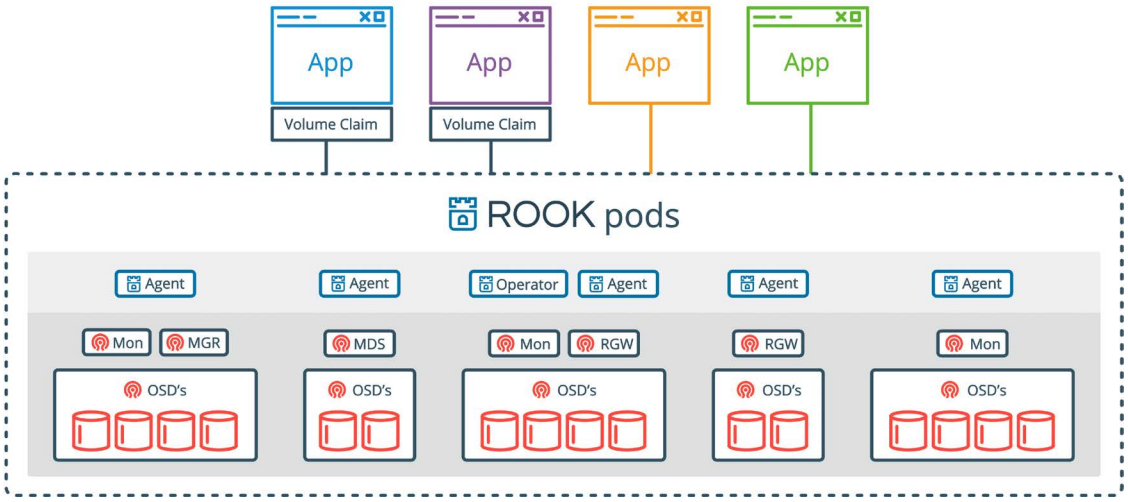


Figure 6.4: Rook architecture

Once you install the Rook operator you can create a Ceph cluster using a Rook CRD such as: `https://github.com/rook/rook/blob/release-1.10/deploy/examples/cluster.yaml`.

Here is a shortened version (without the comments):

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
metadata:
  name: rook-ceph
  namespace: rook-ceph # namespace:cluster
spec:
  cephVersion:
    image: quay.io/ceph/ceph:v17.2.5
    allowUnsupported: false
  dataDirHostPath: /var/lib/rook
  skipUpgradeChecks: false
  continueUpgradeAfterChecksEvenIfNotHealthy: false
  waitTimeoutForHealthyOSDInMinutes: 10
  mon:
    count: 3
    allowMultiplePerNode: false
  mgr:
```

```
    count: 2
    allowMultiplePerNode: false
    modules:
      - name: pg_autoscaler
        enabled: true
  dashboard:
    enabled: true
    ssl: true
  monitoring:
   enabled: false
  network:
    connections:
      encryption:
        enabled: false
      compression:
        enabled: false
  crashCollector:
    disable: false
  logCollector:
    enabled: true
    periodicity: daily # one of: hourly, daily, weekly, monthly
    maxLogSize: 500M # SUFFIX may be 'M' or 'G'. Must be at least 1M.
  cleanupPolicy:
    confirmation: ""
    sanitizeDisks:
      method: quick
      dataSource: zero
      iteration: 1
    allowUninstallWithVolumes: false
  annotations:
  labels:
  resources:
  removeOSDsIfOutAndSafeToRemove: false
  priorityClassNames:
    mon: system-node-critical
    osd: system-node-critical
    mgr: system-cluster-critical
  storage: # cluster level storage configuration and selection
    useAllNodes: true
    useAllDevices: true
    config:
```

```
    onlyApplyOSDPlacement: false
  disruptionManagement:
    managePodBudgets: true
    osdMaintenanceTimeout: 30
    pgHealthCheckTimeout: 0
    manageMachineDisruptionBudgets: false
    machineDisruptionBudgetNamespace: openshift-machine-api

  healthCheck:
    daemonHealth:
      mon:
        disabled: false
        interval: 45s
      osd:
        disabled: false
        interval: 60s
      status:
        disabled: false
        interval: 60s
    livenessProbe:
      mon:
        disabled: false
      mgr:
        disabled: false
      osd:
        disabled: false
    startupProbe:
      mon:
        disabled: false
      mgr:
        disabled: false
      osd:
        disabled: false
```

Here is a storage class for CephFS:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: rook-ceph-retain-bucket
provisioner: rook-ceph.ceph.rook.io/bucket # driver:namespace:cluster
# set the reclaim policy to retain the bucket when its OBC is deleted
```

```
reclaimPolicy: Retain
parameters:
   objectStoreName: my-store # port 80 assumed
   objectStoreNamespace: rook-ceph # namespace:cluster
```

The full code is available here: `https://github.com/rook/rook/blob/release-1.10/deploy/examples/storageclass-bucket-retain.yaml`.

Now that we've covered using distributed storage using GlusterFS, Ceph, and Rook, let's look at enterprise storage options.

Integrating enterprise storage into Kubernetes

If you have an existing **Storage Area Network** (**SAN**) exposed over the iSCSI interface, Kubernetes has a volume plugin for you. It follows the same model as other shared persistent storage plugins we've seen earlier. It supports the following features:

- Connecting to one portal
- Mounting a device directly or via `multipathd`
- Formatting and partitioning any new device
- Authenticating via CHAP

You must configure the iSCSI initiator, but you don't have to provide any initiator information. All you need to provide is the following:

- IP address of the iSCSI target and port (if not the default `3260`)
- Target's **IQN** (**iSCSI Qualified Name**) – typically the reversed domain name
- **LUN** (**Logical Unit Number**)
- Filesystem type
- `Readonly` Boolean flag

The iSCSI plugin supports `ReadWriteOnce` and `ReadonlyMany`. Note that you can't partition your device at this time. Here is an example pod with an iSCSI volume spec:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: iscsipd
spec:
  containers:
  - name: iscsipd-rw
    image: kubernetes/pause
    volumeMounts:
    - mountPath: "/mnt/iscsipd"
```

```
      name: iscsipd-rw
  volumes:
  - name: iscsipd-rw
    iscsi:
      targetPortal: 10.0.2.15:3260
      portals: ['10.0.2.16:3260', '10.0.2.17:3260']
      iqn: iqn.2001-04.com.example:storage.kube.sys1.xyz
      lun: 0
      fsType: ext4
      readOnly: true
```

Other storage providers

The Kubernetes storage scene keeps innovating. A lot of companies adapt their products to Kubernetes and some companies and organizations build Kubernetes-dedicated storage solutions. Here are some of the more popular and mature solutions:

- OpenEBS
- Longhorn
- Portworx

The Container Storage Interface

The **Container Storage Interface** (**CSI**) is a standard interface for the interaction between container orchestrators and storage providers. It was developed by Kubernetes, Docker, Mesos, and Cloud Foundry. The idea is that storage providers implement just one CSI driver and all container orchestrators need to support only the CSI. It is the equivalent of CNI for storage.

A CSI volume plugin was added in Kubernetes 1.9 as an Alpha feature and has been generally available since Kubernetes 1.13. The older FlexVolume approach (which you may have come across) is deprecated now.

Here is a diagram that demonstrates how CSI works within Kubernetes:

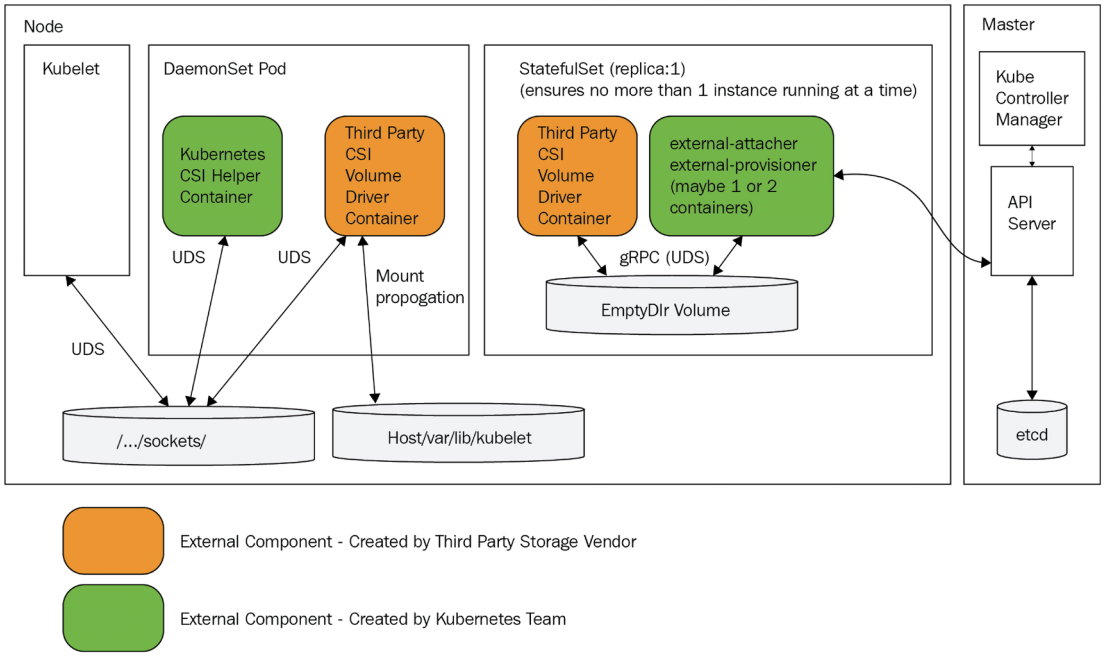


Figure 6.5: CSI architecture

The migration effort to port all in-tree plugins to out-of-tree CSI drivers is well underway. See `https://kubernetes-csi.github.io` for more details.

Advanced storage features

These features are available only to CSI drivers. They represent the benefits of a uniform storage model that allows adding optional advanced functionality across all storage providers with a uniform interface.

Volume snapshots

Volume snapshots are generally available as of Kubernetes 1.20. They are exactly what they sound like – a snapshot of a volume at a certain point in time. You can create and later restore volumes from a snapshot. It's interesting that the API objects associated with snapshots are CRDs and not part of the core Kubernetes API. The objects are:

- `VolumeSnapshotClass`
- `VolumeSnapshotContents`
- `VolumeSnapshot`

Volume snapshots work using an `external-prosnapshotter` sidecar container that the Kubernetes team developed. It watches for snapshot CRDs to be created and interacts with the snapshot controller, which can invoke the `CreateSnapshot` and `DeleteSnapshot` operations of CSI drivers that implement snapshot support.

Here is how to declare a volume snapshot:

```
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: new-snapshot-test
spec:
  volumeSnapshotClassName: csi-hostpath-snapclass
  source:
    persistentVolumeClaimName: pvc-test
```

You can also provision volumes from a snapshot.

Here is a persistent volume claim bound to a snapshot:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restore-pvc
spec:
  storageClassName: csi-hostpath-sc
  dataSource:
    name: new-snapshot-test
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

See `https://github.com/kubernetes-csi/external-snapshotter#design` for more details.

CSI volume cloning

Volume cloning is available in GA as of Kubernetes 1.18. Volume clones are new volumes that are populated with the content of an existing volume. Once the volume cloning is complete there is no relation between the original and the clone. Their content will diverge over time. You can perform a clone manually by creating a snapshot and then create a new volume from the snapshot. But, volume cloning is more streamlined and efficient.

It only works for dynamic provisioning and uses the storage class of the source volume for the clone as well. You initiate a volume clone by specifying an existing persistent volume claim as a data source of a new persistent volume claim. That triggers the dynamic provisioning of a new volume that clones the source claim's volume:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: clone-of-pvc-1
    namespace: myns
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: cloning
  resources:
    requests:
      storage: 5Gi
  dataSource:
    kind: PersistentVolumeClaim
    name: pvc-1
```

See `https://kubernetes.io/docs/concepts/storage/volume-pvc-datasource/` for more details.

Storage capacity tracking

Storage capacity tracking (GA as of Kubernetes 1.24) allows the scheduler to better schedule pods that require storage into nodes that can provide that storage. This requires a CSI driver that supports storage capacity tracking.

The CSI driver will create a `CSIStorageCapacity` object for each storage class and determine which nodes have access to this storage. In addition the `CSIDriverSpec`'s field `StorageCapacity` must be set to true.

When a pod specifies a storage class name in `WaitForFirstConsumer` mode and the CSI driver has `StorageCapacity` set to true the Kubernetes scheduler will consider the CSIStorageCapacity object associated with the storage class and schedule the pod only to nodes that have sufficient storage.

Check out: `https://kubernetes.io/docs/concepts/storage/storage-capacity` for more details.

Volume health monitoring

Volume health monitoring is a recent addition to the storage APIs. It has been in Alpha since Kubernetes 1.21. It involves two components:

- An external health monitor
- The kubelet

CSI drivers that support volume health monitoring will update PVCs with events on abnormal conditions of associated storage volumes. The external health monitor also watches nodes for failures and will report events on PVCs bound to these nodes.

In the case where a CSI driver enables volume health monitoring from the node side, any abnormal condition detected will result in an event being reported for every pod that utilizes a PVC with the corresponding issue.

There is also a new metric associated with volume health: `kubelet_volume_stats_health_status_abnormal`.

It has two labels: `namespace` and `persistentvolumeclaim`. The values are 0 or 1.

More details are available here: `https://kubernetes.io/docs/concepts/storage/volume-health-monitoring/`.

CSI is an exciting initiative that simplified the Kubernetes code base itself by externalizing storage drivers. It simplified the life of storage solutions that can develop out-of-tree drivers and added a lot of advanced capabilities to the Kubernetes storage story.

Summary

In this chapter, we took a deep look into storage in Kubernetes. We've looked at the generic conceptual model based on volumes, claims, and storage classes, as well as the implementation of volume plugins. Kubernetes eventually maps all storage systems into mounted filesystems in containers or devices of raw block storage. This straightforward model allows administrators to configure and hook up any storage system from local host directories, through cloud-based shared storage, all the way to enterprise storage systems. The transition of storage provisioners from in-tree to CSI-based out-of-tree drivers bodes well for the storage ecosystem. You should now have a clear understanding of how storage is modeled and implemented in Kubernetes and be able to make intelligent choices on how to implement storage in your Kubernetes cluster.

In *Chapter 7*, *Running Stateful Applications with Kubernetes*, we'll see how Kubernetes can raise the level of abstraction and, on top of storage, help to develop, deploy, and operate stateful applications using concepts such as stateful sets.

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`



7

Running Stateful Applications with Kubernetes

In this chapter, we will learn how to run stateful applications on Kubernetes. Kubernetes takes a lot of work out of our hands by automatically starting and restarting pods across the cluster nodes as needed, based on complex requirements and configurations such as namespaces, limits, and quotas. But when pods run storage-aware software, such as databases and queues, relocating a pod can cause a system to break.

First, we'll explore the essence of stateful pods and why they are much more complicated to manage in Kubernetes. We will look at a few ways to manage the complexity, such as shared environment variables and DNS records. In some situations, a redundant in-memory state, a `DaemonSet`, or persistent storage claims can do the trick. The main solution that Kubernetes promotes for state-aware pods is the `StatefulSet` (previously called `PetSet`) resource, which allows us to manage an indexed collection of pods with stable properties. Finally, we will dive deep into a full-fledged example of running a Cassandra cluster on top of Kubernetes.

This chapter will cover the following main topics:

- Stateful versus stateless applications in Kubernetes
- Running a Cassandra cluster in Kubernetes

By the end of this chapter, you will understand the challenges of state management in Kubernetes, get a deep look into a specific example of running Cassandra as a data store on Kubernetes, and be able to determine the state management strategy for your workloads.

Stateful versus stateless applications in Kubernetes

A stateless Kubernetes application is an application that doesn't manage its state in the Kubernetes cluster. All the state is stored in memory or outside the cluster, and the cluster containers access it in some manner. A stateful Kubernetes application, on the other hand, has a persistent state that is managed in the cluster. In this section, we'll learn why state management is critical to the design of a distributed system and the benefits of managing the state within the Kubernetes cluster.

Understanding the nature of distributed data-intensive apps

Let's start with the basics here. Distributed applications are a collection of processes that run on multiple machines, process inputs, manipulate data, expose APIs, and possibly have other side effects. Each process is a combination of its program, its runtime environment, and its inputs and outputs.

The programs you write at school get their input as command-line arguments; maybe they read a file or access a database, and then write their results to the screen, a file, or a database. Some programs keep state in memory and can serve requests over a network. Simple programs run on a single machine and can hold all their state in memory or read from a file. Their runtime environment is their operating system. If they crash, the user has to restart them manually. They are tied to their machine.

A distributed application is a different animal. A single machine is not enough to process all the data or serve all the requests quickly enough. A single machine can't hold all the data. The data that needs to be processed is so large that it can't be downloaded cost-effectively into each processing machine. Machines can fail and need to be replaced. Upgrades need to be performed over all the processing machines. Users may be distributed across the globe.

Taking all these issues into account, it becomes clear that the traditional approach doesn't work. The limiting factor becomes the data. Users/clients must receive only summary or processed data. All massive data processing must be done close to the data itself because transferring data is prohibitively slow and expensive. Instead, the bulk of processing code must run in the same data center and network environment of the data.

Why manage the state in Kubernetes?

The main reason to manage the state in Kubernetes itself as opposed to a separate cluster is that a lot of the infrastructure needed to monitor, scale, allocate, secure, and operate a storage cluster is already provided by Kubernetes. Running a parallel storage cluster will lead to a lot of duplicated effort.

Why manage the state outside of Kubernetes?

Let's not rule out the other option. It may be better in some situations to manage the state in a separate non-Kubernetes cluster, as long as it shares the same internal network (data proximity trumps everything).

Some valid reasons are as follows:

- You already have a separate storage cluster and you don't want to rock the boat
- Your storage cluster is used by other non-Kubernetes applications
- Kubernetes support for your storage cluster is not stable or mature enough
- You may want to approach stateful applications in Kubernetes incrementally, starting with a separate storage cluster and integrating more tightly with Kubernetes later

Shared environment variables versus DNS records for discovery

Kubernetes provides several mechanisms for global discovery across the cluster. If your storage cluster is not managed by Kubernetes, you still need to tell Kubernetes pods how to find it and access it.

There are two common methods:

- DNS
- Environment variables

In some cases, you may want to use both, as environment variables can override DNS.

Accessing external data stores via DNS

The DNS approach is simple and straightforward. Assuming your external storage cluster is load-balanced and can provide a stable endpoint, then pods can just hit that endpoint directly and connect to the external cluster.

Accessing external data stores via environment variables

Another simple approach is to use environment variables to pass connection information to an external storage cluster. Kubernetes offers the `ConfigMap` resource as a way to keep configuration separate from the container image. The configuration is a set of `key-value` pairs. The configuration information can be exposed in two ways. One way is as environment variables. The other way is as a configuration file mounted as a volume in the container. You may prefer to use secrets for sensitive connection information like passwords.

Creating a ConfigMap

The following file is a `ConfigMap` that keeps a list of addresses:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  db-ip-addresses: 1.2.3.4,5.6.7.8
```

Save it as `db-config-map.yaml` and run:

```
$ k create -f db-config-map.yaml
configmap/db-config created
```

The `data` section contains all the `key-value` pairs, in this case, just a single pair with a key name of `db-ip-addresses`. It will be important later when consuming the `ConfigMap` in a pod. You can check out the content to make sure it's OK:

```
$ k get configmap db-config -o yaml
apiVersion: v1
data:
  db-ip-addresses: 1.2.3.4,5.6.7.8
kind: ConfigMap
metadata:
  creationTimestamp: "2022-07-17T17:39:05Z"
```

```
  name: db-config
  namespace: default
  resourceVersion: "504571"
  uid: 11e49df0-ed1e-4bee-9fd7-bf38bb2aa38a
```

There are other ways to create a `ConfigMap`. You can directly create one using the `--from-value` or `--from-file` command-line arguments.

Consuming a ConfigMap as an environment variable

When you are creating a pod, you can specify a `ConfigMap` and consume its values in several ways. Here is how to consume our configuration map as an environment variable:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - name: some-container
    image: busybox
    command: ["/bin/sh", "-c", "env"]
    env:
    - name: DB_IP_ADDRESSES
      valueFrom:
        configMapKeyRef:
          name: db-config
          key: db-ip-addresses
  restartPolicy: Never
```

This pod runs the `busybox` minimal container and executes an `env bash` command and it immediately exists. The `db-ip-addresses` key from the `db-configmap` is mapped to the `DB_IP_ADDRESSES` environment variable, and is reflected in the logs:

```
$ k create -f pod-with-db.yaml
pod/some-pod created

$ k logs some-pod | grep DB_IP
DB_IP_ADDRESSES=1.2.3.4,5.6.7.8
```

Using a redundant in-memory state

In some cases, you may want to keep a transient state in memory. Distributed caching is a common case. Time-sensitive information is another one. For these use cases, there is no need for persistent storage, and multiple pods accessed through a service may be just the right solution.

We can use standard Kubernetes techniques, such as labeling, to identify pods that belong to the distributed cache, store redundant copies of the same state, and expose them through a service. If a pod dies, Kubernetes will create a new one and, until it catches up, the other pods will serve the state. We can even use the pod's anti-affinity feature to ensure that pods that maintain redundant copies of the same state are not scheduled to the same node.

Of course, you could also use something like Memcached or Redis.

Using DaemonSet for redundant persistent storage

Some stateful applications, such as distributed databases or queues, manage their state redundantly and sync their nodes automatically (we'll take a very deep look into Cassandra later). In these cases, it is important that pods are scheduled to separate nodes. It is also important that pods are scheduled to nodes with a particular hardware configuration or are even dedicated to the stateful application. The `DaemonSet` feature is perfect for this use case. We can label a set of nodes and make sure that the stateful pods are scheduled on a one-by-one basis to the selected group of nodes.

Applying persistent volume claims

If the stateful application can use effectively shared persistent storage, then using a persistent volume claim in each pod is the way to go, as we demonstrated in *Chapter 6*, *Managing Storage*. The stateful application will be presented with a mounted volume that looks just like a local filesystem.

Utilizing StatefulSet

StatefulSets are specially designed to support distributed stateful applications where the identities of the members are important, and if a pod is restarted, it must retain its identity in the set. It provides ordered deployment and scaling. Unlike regular pods, the pods of a `StatefulSet` are associated with persistent storage.

When to use StatefulSet

`StatefulSets` are great for applications that necessitate any of the following capabilities:

- Consistent and distinct network identifiers
- Persistent and enduring storage
- Methodical and orderly deployment and scaling
- Systematic and organized deletion and termination

The components of StatefulSet

There are several elements that need to be configured correctly in order to have a working `StatefulSet`:

- A headless service responsible for managing the network identity of the `StatefulSet` pods
- The `StatefulSet` itself with a number of replicas
- Local storage on nodes or persistent storage provisioned dynamically or by an administrator

Here is an example of a headless service called `nginx` that will be used for a `StatefulSet`:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  selector:
    app: nginx
  ports:
  - port: 80
    name: web
    clusterIP: None
```

Now, the `StatefulSet` manifest file will reference the service:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
```

The next part is the pod template, which includes a mounted volume named `www`:

```
spec:
  terminationGracePeriodSeconds: 10
  containers:
  - name: nginx
    image: gcr.io/google_containers/nginx-slim:0.8
    ports:
    - containerPort: 80
      name: web
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html
```

Last but not least, `volumeClaimTemplates` use a claim named `www` matching the mounted volume. The claim requests 1 Gib of storage with `ReadWriteOnce` access:

```
volumeClaimTemplates:
    - metadata:
      name: www
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
            requests:
                storage: 1Gib
```

Working with StatefulSets

Let's create the `nginx` headless service and `statefulset`:

```
k apply -f nginx-headless-service.yaml
service/nginx created

$ k apply -f nginx-stateful-set.yaml
statefulset.apps/nginx created
```

We can use the `kubectl get all` command to see all the resources that were created:

```
$ k get all
NAME          READY   STATUS    RESTARTS   AGE
pod/nginx-0   1/1     Running   0          107s
pod/nginx-1   1/1     Running   0          104s
pod/nginx-2   1/1     Running   0          102s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/nginx        ClusterIP   None         <none>        80/TCP    2m5s

NAME                     READY   AGE
statefulset.apps/nginx   3/3     107s
```

As expected, we have the `statefulset` with three replicas and the headless service. What is not pre-set is a `ReplicaSet`, which you find when you create a Deployment. StatefulSets manage their pods directly.

Note that the `kubectl get all` doesn't actually show all resources. The StatefulSet also creates a persistent volume claim backed by a persistent volume for each pod. Here they are:

```
$ k get pvc
NAME          STATUS   VOLUME                                     CAPACITY   ACCESS
MODES   STORAGECLASS   AGE
```

```
www-nginx-0   Bound    pvc-40ac1c62-bba0-4e3c-9177-eda7402755b3   10Mi       RWO
standard       1m37s
www-nginx-1   Bound    pvc-94022a60-e4cb-4495-825d-eb744088266f   10Mi       RWO
standard       1m43s
www-nginx-2   Bound    pvc-8c60523f-a3e8-4ae3-a91f-6aaa53b02848   10Mi       RWO
standard       1m52h

$ k get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS   CLAIM                 STORAGECLASS   REASON   AGE
pvc-40ac1c62-bba0-4e3c-9177-eda7402755b3   10Mi       RWO            Delete
Bound    default/www-nginx-0   standard                1m59s
pvc-8c60523f-a3e8-4ae3-a91f-6aaa53b02848   10Mi       RWO            Delete
Bound    default/www-nginx-2   standard                2m2s
pvc-94022a60-e4cb-4495-825d-eb744088266f   10Mi       RWO            Delete
Bound    default/www-nginx-1   standard                2m1s
```

If we delete a pod, the StatefulSet will create a new pod and bind it to the corresponding persistent volume claim. The pod `nginx-1` is bound to the `www-nginx-1` pvc:

```
$ k get po nginx-1 -o yaml | yq '.spec.volumes[0]'
name: www
persistentVolumeClaim:
  claimName: www-nginx-1
```

Let's delete the `nginx-1` pod and check all remaining pods:

```
$ k delete po nginx-1
pod "nginx-1" deleted

$ k get po
NAME      READY   STATUS    RESTARTS   AGE
nginx-0   1/1     Running   0          12m
nginx-1   1/1     Running   0          14s
nginx-2   1/1     Running   0          12m
```

As you can see, the StatefulSet immediately replaced it with a new `nginx-1` pod (14 seconds old). The new pod is bound to the same persistent volume claim:

```
$ k get po nginx-1 -o yaml | yq '.spec.volumes[0]'
name: www
persistentVolumeClaim:
  claimName: www-nginx-1
```

The persistent volume claim and its backing persistent volume were not deleted when the old `nginx-1` pod was deleted, as you can tell by their age:

```
$ k get pvc www-nginx-1
NAME          STATUS   VOLUME                                     CAPACITY   ACCESS
MODES   STORAGECLASS   AGE
www-nginx-1   Bound    pvc-94022a60-e4cb-4495-825d-eb744088266f   10Mi       RWO
standard       143s

$ k get pv pvc-94022a60-e4cb-4495-825d-eb744088266f
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS   CLAIM                 STORAGECLASS   REASON   AGE
pvc-94022a60-e4cb-4495-825d-eb744088266f   10Mi       RWO            Delete
Bound    default/www-nginx-1   standard                2m1s
```

That means that the state of the StatefulSet is preserved even as pods come and go. Each pod identified by its index is always bound to a specific shard of the state, backed up by the corresponding persistent volume claim.

At this point, we understand what StatefulSets are all about and how to work with them. Let's dive into the implementation of an industrial-strength data store and see how it can be deployed as a StatefulSet in Kubernetes.

Running a Cassandra cluster in Kubernetes

In this section, we will explore in detail a very large example of configuring a Cassandra cluster to run on a Kubernetes cluster. I will dissect and give some context for interesting parts. If you wish to explore this even further, the full example can be accessed here:

`https://kubernetes.io/docs/tutorials/stateful-application/cassandra`

The goal here is to get a sense of what it takes to run a real-world stateful workload on Kubernetes and how StatefulSets help. Don't worry if you don't understand every little detail.

First, we'll learn a little bit about Cassandra and its idiosyncrasies, and then follow a step-by-step procedure to get it running using several of the techniques and strategies we covered in the previous section.

A quick introduction to Cassandra

Cassandra is a distributed columnar data store. It was designed from the get-go for big data. Cassandra is fast, robust (no single point of failure), highly available, and linearly scalable. It also has multi-data center support. It achieves all this by having a laser focus and carefully crafting the features it supports and—just as importantly—the features it doesn't support.

In a previous company, I ran a Kubernetes cluster that used Cassandra as the main data store for sensor data (about 100 TB). Cassandra allocates the data to a set of nodes (node ring) based on a **distributed hash table** (**DHT**) algorithm.

The cluster nodes talk to each other via a gossip protocol and learn quickly about the overall state of the cluster (what nodes joined and what nodes left or are unavailable). Cassandra constantly compacts the data and balances the cluster. The data is typically replicated multiple times for redundancy, robustness, and high availability.

From a developer's point of view, Cassandra is very good for time-series data and provides a flexible model where you can specify the consistency level in each query. It is also idempotent (a very important feature for a distributed database), which means repeated inserts or updates are allowed.

Here is a diagram that shows how a Cassandra cluster is organized, how a client can access any node, and how a request will be forwarded automatically to the nodes that have the requested data:

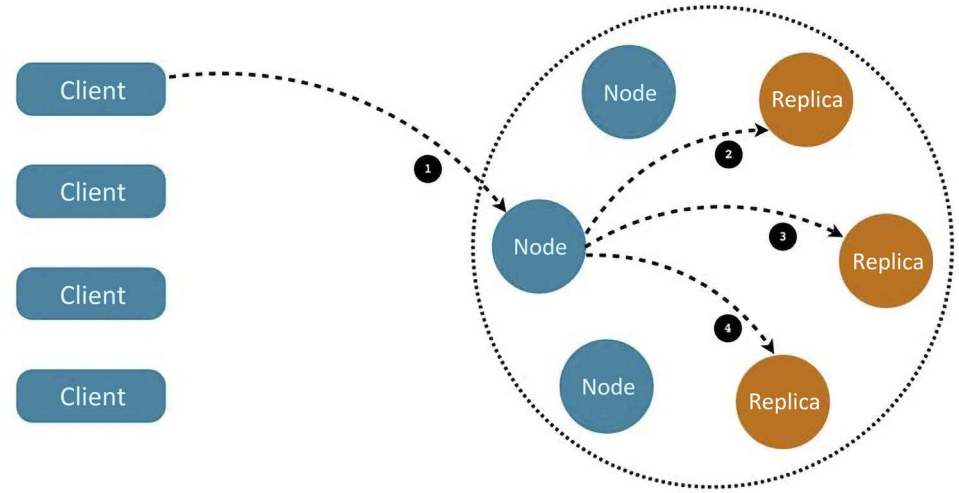


Figure 7.1: Request interacting with a Cassandra cluster

The Cassandra Docker image

Deploying Cassandra on Kubernetes as opposed to a standalone Cassandra cluster deployment requires a special Docker image. This is an important step because it means we can use Kubernetes to keep track of our Cassandra pods. The Dockerfile for an image is available here: `https://github.com/kubernetes/examples/blob/master/cassandra/image/Dockerfile`.

See below the Dockerfile that builds the Cassandra image. The base image is a flavor of Debian designed for use in containers (see `https://github.com/kubernetes/kubernetes/tree/master/build/debian-base`).

The Cassandra Dockerfile defines some build arguments that must be set when the image is built, creates a bunch of labels, defines many environment variables, adds all the files to the root directory inside the container, runs the `build.sh` script, declares the Cassandra data volume (where the data is stored), exposes a bunch of ports, and finally, uses `dumb-init` to execute the `run.sh` scripts:

```
FROM k8s.gcr.io/debian-base-amd64:0.3

ARG BUILD_DATE
ARG VCS_REF
ARG CASSANDRA_VERSION
ARG DEV_CONTAINER

LABEL \
    org.label-schema.build-date=$BUILD_DATE \
    org.label-schema.docker.dockerfile="/Dockerfile" \
    org.label-schema.license="Apache License 2.0" \
    org.label-schema.name="k8s-for-greeks/docker-cassandra-k8s" \
    org.label-schema.url="https://github.com/k8s-for-greeks/" \
    org.label-schema.vcs-ref=$VCS_REF \
    org.label-schema.vcs-type="Git" \
    org.label-schema.vcs-url="https://github.com/k8s-for-greeks/docker-
cassandra-k8s"

ENV CASSANDRA_HOME=/usr/local/apache-cassandra-${CASSANDRA_VERSION} \
    CASSANDRA_CONF=/etc/cassandra \
    CASSANDRA_DATA=/cassandra_data \
    CASSANDRA_LOGS=/var/log/cassandra \
    JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64 \
    PATH=${PATH}:/usr/lib/jvm/java-8-openjdk-amd64/bin:/usr/local/apache-
cassandra-${CASSANDRA_VERSION}/bin

ADD files /

RUN clean-install bash \
    && /build.sh \
    && rm /build.sh

VOLUME ["/$CASSANDRA_DATA"]

# 7000: intra-node communication
# 7001: TLS intra-node communication
# 7199: JMX
# 9042: CQL
# 9160: thrift service
EXPOSE 7000 7001 7199 9042 9160

CMD ["/usr/bin/dumb-init", "/bin/bash", "/run.sh"]
```

Here are all the files used by the Dockerfile:

build.sh
cassandra-seed.h
cassandra.yaml
jvm.options
kubernetes-cassandra.jar
logback.xml
ready-probe.sh
run.sh

We will not cover all of them, but will focus on a couple of interesting scripts: the `build.sh` and `run.sh` scripts.

Exploring the build.sh script

Cassandra is a Java program. The build script installs the Java runtime environment and a few necessary libraries and tools. It then sets a few variables that will be used later, such as `CASSANDRA_PATH`.

It downloads the correct version of Cassandra from the Apache organization (Cassandra is an Apache open source project), creates the `/cassandra_data/data` directory where Cassandra will store its `SSTables` and the `/etc/cassandra` configuration directory, copies files into the configuration directory, adds a Cassandra user, sets the readiness probe, installs Python, moves the Cassandra JAR file and the seed shared library to their target destination, and then cleans up all the intermediate files generated during this process:

```
...

clean-install \
    openjdk-8-jre-headless \
    libjemalloc1 \
    localepurge \
    dumb-init \
    wget

CASSANDRA_PATH="cassandra/${CASSANDRA_VERSION}/apache-cassandra-${CASSANDRA_
VERSION}-bin.tar.gz"
CASSANDRA_DOWNLOAD="http://www.apache.org/dyn/closer.cgi?path=/${CASSANDRA_
PATH}&as_json=1"
CASSANDRA_MIRROR=`wget -q -O - ${CASSANDRA_DOWNLOAD} | grep -oP
"(?<=\"preferred\": \")[^\"]+"`

echo "Downloading Apache Cassandra from $CASSANDRA_MIRROR$CASSANDRA_PATH..."
wget -q -O - $CASSANDRA_MIRROR$CASSANDRA_PATH \
    | tar -xzf - -C /usr/local
```

```
mkdir -p /cassandra_data/data
mkdir -p /etc/cassandra

mv /logback.xml /cassandra.yaml /jvm.options /etc/cassandra/
mv /usr/local/apache-cassandra-${CASSANDRA_VERSION}/conf/cassandra-env.sh /etc/
cassandra/

adduser --disabled-password --no-create-home --gecos '' --disabled-login
cassandra
chmod +x /ready-probe.sh
chown cassandra: /ready-probe.sh

DEV_IMAGE=${DEV_CONTAINER:-}
if [ ! -z "$DEV_IMAGE" ]; then
    clean-install python;
else
    rm -rf  $CASSANDRA_HOME/pylib;
fi

mv /kubernetes-cassandra.jar /usr/local/apache-cassandra-${CASSANDRA_VERSION}/
lib
mv /cassandra-seed.so /etc/cassandra/
mv /cassandra-seed.h /usr/local/lib/include

apt-get -y purge localepurge
apt-get -y autoremove
apt-get clean

rm -rf <many files>
```

Exploring the run.sh script

The `run.sh` script requires some shell skills and knowledge of Cassandra to understand, but it's worth the effort. First, some local variables are set for the Cassandra configuration file at `/etc/cassandra/cassandra.yaml`. The `CASSANDRA_CFG` variable will be used in the rest of the script:

```
set -e
CASSANDRA_CONF_DIR=/etc/cassandra
CASSANDRA_CFG=$CASSANDRA_CONF_DIR/cassandra.yaml
```

If no `CASSANDRA_SEEDS` were specified, then set the `HOSTNAME`, which is used by the `StatefulSet` later:

```
# we are doing StatefulSet or just setting our seeds
```

```
if [ -z "$CASSANDRA_SEEDS" ]; then
  HOSTNAME=$(hostname -f)
  CASSANDRA_SEEDS=$(hostname -f)
fi
```

Then comes a long list of environment variables with defaults. The syntax, `${VAR_NAME:-}`, uses the `VAR_NAME` environment variable, if it's defined, or the default value.

A similar syntax, `${VAR_NAME:=}`, does the same thing but also assigns the default value to the environment variable if it's not defined. This is a subtle but important difference.

Both variations are used here:

```
# The following vars relate to their counter parts in $CASSANDRA_CFG
# for instance rpc_address
CASSANDRA_RPC_ADDRESS="${CASSANDRA_RPC_ADDRESS:-0.0.0.0}"
CASSANDRA_NUM_TOKENS="${CASSANDRA_NUM_TOKENS:-32}"
CASSANDRA_CLUSTER_NAME="${CASSANDRA_CLUSTER_NAME:='Test Cluster'}"
CASSANDRA_LISTEN_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_BROADCAST_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_BROADCAST_RPC_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_DISK_OPTIMIZATION_STRATEGY="${CASSANDRA_DISK_OPTIMIZATION_STRATEGY:-
ssd}"
CASSANDRA_MIGRATION_WAIT="${CASSANDRA_MIGRATION_WAIT:-1}"
CASSANDRA_ENDPOINT_SNITCH="${CASSANDRA_ENDPOINT_SNITCH:-SimpleSnitch}"
CASSANDRA_DC="${CASSANDRA_DC}"
CASSANDRA_RACK="${CASSANDRA_RACK}"
CASSANDRA_RING_DELAY="${CASSANDRA_RING_DELAY:-30000}"
CASSANDRA_AUTO_BOOTSTRAP="${CASSANDRA_AUTO_BOOTSTRAP:-true}"
CASSANDRA_SEEDS="${CASSANDRA_SEEDS:false}"
CASSANDRA_SEED_PROVIDER="${CASSANDRA_SEED_PROVIDER:-org.apache.cassandra.
locator.SimpleSeedProvider}"
CASSANDRA_AUTO_BOOTSTRAP="${CASSANDRA_AUTO_BOOTSTRAP:false}"
```

By the way, I contributed my part to Kubernetes by opening a PR to fix a minor typo here. See `https://github.com/kubernetes/examples/pull/348`.

The next part configures monitoring JMX and controls garbage collection output:

```
# Turn off JMX auth
CASSANDRA_OPEN_JMX="${CASSANDRA_OPEN_JMX:-false}"
# send GC to STDOUT
CASSANDRA_GC_STDOUT="${CASSANDRA_GC_STDOUT:-false}"
```

Then comes a section where all the variables are printed on the screen. Let's skip most of it:

```
echo Starting Cassandra on ${CASSANDRA_LISTEN_ADDRESS}
echo CASSANDRA_CONF_DIR ${CASSANDRA_CONF_DIR}
echo CASSANDRA_CFG ${CASSANDRA_CFG}
echo CASSANDRA_AUTO_BOOTSTRAP ${CASSANDRA_AUTO_BOOTSTRAP}
...
```

The next section is very important. By default, Cassandra uses a simple snitch, which is unaware of racks and data centers. This is not optimal when the cluster spans multiple data centers and racks.

Cassandra is rack-aware and data center-aware and can optimize both for redundancy and high availability while limiting communication across data centers appropriately:

```
# if DC and RACK are set, use GossipingPropertyFileSnitch
if [[ $CASSANDRA_DC && $CASSANDRA_RACK ]]; then
  echo "dc=$CASSANDRA_DC" > $CASSANDRA_CONF_DIR/cassandra-rackdc.properties
  echo "rack=$CASSANDRA_RACK" >> $CASSANDRA_CONF_DIR/cassandra-rackdc.
properties
  CASSANDRA_ENDPOINT_SNITCH="GossipingPropertyFileSnitch"
fi
```

Memory management is also important, and you can control the maximum heap size to ensure Cassandra doesn't start thrashing and swapping to disk:

```
if [ -n "$CASSANDRA_MAX_HEAP" ]; then
  sed -ri "s/^(#)?-Xmx[0-9]+.*/-Xmx$CASSANDRA_MAX_HEAP/" "$CASSANDRA_CONF_DIR/
jvm.options"
  sed -ri "s/^(#)?-Xms[0-9]+.*/-Xms$CASSANDRA_MAX_HEAP/" "$CASSANDRA_CONF_DIR/
jvm.options"
fi

if [ -n "$CASSANDRA_REPLACE_NODE" ]; then
   echo "-Dcassandra.replace_address=$CASSANDRA_REPLACE_NODE/" >> "$CASSANDRA_
CONF_DIR/jvm.options"
fi
```

The rack and data center information is stored in a simple Java `propertiesfile`:

```
for rackdc in dc rack; do
  var="CASSANDRA_${rackdc^^}"
  val="${!var}"
  if [ "$val" ]; then
    sed -ri 's/^('"$rackdc"'=).*/\1 '"$val"'/' "$CASSANDRA_CONF_DIR/cassandra-
rackdc.properties"
  fi
done
```

The next section loops over all the variables defined earlier, finds the corresponding key in the `Cassandra.yaml` configuration files, and overwrites them. That ensures that each configuration file is customized on the fly just before it launches Cassandra itself:

```
for yaml in \
  broadcast_address \
  broadcast_rpc_address \
  cluster_name \
  disk_optimization_strategy \
  endpoint_snitch \
  listen_address \
  num_tokens \
  rpc_address \
  start_rpc \
  key_cache_size_in_mb \
  concurrent_reads \
  concurrent_writes \
  memtable_cleanup_threshold \
  memtable_allocation_type \
  memtable_flush_writers \
  concurrent_compactors \
  compaction_throughput_mb_per_sec \
  counter_cache_size_in_mb \
  internode_compression \
  endpoint_snitch \
  gc_warn_threshold_in_ms \
  listen_interface \
  rpc_interface \
  ; do
  var="CASSANDRA_${yaml^^}"
  val="${!var}"
  if [ "$val" ]; then
    sed -ri 's/^(# )?('"$yaml"':).*/\2 '"$val"'/' "$CASSANDRA_CFG"
  fi
done

echo "auto_bootstrap: ${CASSANDRA_AUTO_BOOTSTRAP}" >> $CASSANDRA_CFG
```

The next section is all about setting the seeds or seed provider depending on the deployment solution (`StatefulSet` or not). There is a little trick for the first pod to bootstrap as its own seed:

```
# set the seed to itself.  This is only for the first pod, otherwise
# it will be able to get seeds from the seed provider
```

```
if [[ $CASSANDRA_SEEDS == 'false' ]]; then
  sed -ri 's/- seeds:.*/- seeds: "'"$POD_IP"'"/' $CASSANDRA_CFG
else # if we have seeds set them.  Probably StatefulSet
  sed -ri 's/- seeds:.*/- seeds: "'"$CASSANDRA_SEEDS"'"/' $CASSANDRA_CFG
fi

sed -ri 's/- class_name: SEED_PROVIDER/- class_name: '"$CASSANDRA_SEED_
PROVIDER"'/' $CASSANDRA_CFG
```

The following section sets up various options for remote management and JMX monitoring. It's critical in complicated distributed systems to have proper administration tools. Cassandra has deep support for the ubiquitous **JMX** standard:

```
# send gc to stdout
if [[ $CASSANDRA_GC_STDOUT == 'true' ]]; then
  sed -ri 's/ -Xloggc:\/var\/log\/cassandra\/gc\.log//' $CASSANDRA_CONF_DIR/
cassandra-env.sh
fi

# enable RMI and JMX to work on one port
echo "JVM_OPTS=\"\$JVM_OPTS -Djava.rmi.server.hostname=$POD_IP\"" >>
$CASSANDRA_CONF_DIR/cassandra-env.sh

# getting WARNING messages with Migration Service
echo "-Dcassandra.migration_task_wait_in_seconds=${CASSANDRA_MIGRATION_WAIT}"
>> $CASSANDRA_CONF_DIR/jvm.options
echo "-Dcassandra.ring_delay_ms=${CASSANDRA_RING_DELAY}" >> $CASSANDRA_CONF_
DIR/jvm.options

if [[ $CASSANDRA_OPEN_JMX == 'true' ]]; then
  export LOCAL_JMX=no
  sed -ri 's/ -Dcom\.sun\.management\.jmxremote\.authenticate=true/ -Dcom\.
sun\.management\.jmxremote\.authenticate=false/' $CASSANDRA_CONF_DIR/cassandra-
env.sh
  sed -ri 's/ -Dcom\.sun\.management\.jmxremote\.password\.file=\/etc\/
cassandra\/jmxremote\.password//' $CASSANDRA_CONF_DIR/cassandra-env.sh
fi
```

Finally, it protects the data directory such that only the `cassandra` user can access it, the `CLASSPATH` is set to the Cassandra `JAR` file, and it launches Cassandra in the foreground (not daemonized) as the `cassandra` user:

```
chmod 700 "${CASSANDRA_DATA}"
chown -c -R cassandra "${CASSANDRA_DATA}" "${CASSANDRA_CONF_DIR}"
```

```
export CLASSPATH=/kubernetes-cassandra.jar

su cassandra -c "$CASSANDRA_HOME/bin/cassandra -f"
```

Hooking up Kubernetes and Cassandra

Connecting Kubernetes and Cassandra takes some work because Cassandra was designed to be very self-sufficient, but we want to let it hook into Kubernetes at the right time to provide capabilities such as automatically restarting failed nodes, monitoring, allocating Cassandra pods, and providing a unified view of the Cassandra pods side by side with other pods.

Cassandra is a complicated beast and has many knobs to control it. It comes with a `Cassandra.yaml` configuration file, and you can override all the options with environment variables.

Digging into the Cassandra configuration file

There are two settings that are particularly relevant: the seed provider and the snitch. The seed provider is responsible for publishing a list of IP addresses (seeds) for nodes in the cluster. Each node that starts running connects to the seeds (there are usually at least three) and if it successfully reaches one of them, they immediately exchange information about all the nodes in the cluster. This information is updated constantly for each node as the nodes gossip with each other.

The default seed provider configured in `Cassandra.yaml` is just a static list of IP addresses, in this case, just the loopback interface:

```
# any class that implements the SeedProvider interface and has a
# constructor that takes a Map<String, String> of parameters will do.
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other and learn
    # the topology of the ring.  You must change this if you are running
    # multiple nodes!
    #- class_name: io.k8s.cassandra.KubernetesSeedProvider
    - class_name: SEED_PROVIDER
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "127.0.0.1"
```

The other important setting is the snitch. It has two roles:

- Cassandra utilizes the snitch to gain valuable insights into your network topology, enabling it to effectively route requests.
- Cassandra employs this knowledge to strategically distribute replicas across your cluster, mitigating the risk of correlated failures. To achieve this, Cassandra organizes machines into data centers and racks, ensuring that replicas are not concentrated on a single rack, even if it doesn't necessarily correspond to a physical location.

Cassandra comes pre-loaded with several snitch classes, but none of them are Kubernetes-aware. The default is `SimpleSnitch`, but it can be overridden:

```
# You can use a custom Snitch by setting this to the full class
# name of the snitch, which will be assumed to be on your classpath.
endpoint_snitch: SimpleSnitch
```

Other snitches are:

- `GossipingPropertyFileSnitch`
- `PropertyFileSnitch`
- `Ec2Snitch`
- `Ec2MultiRegionSnitch`
- `RackInferringSnitch`

The custom seed provider

When running Cassandra nodes as pods in Kubernetes, Kubernetes may move pods around, including seeds. To accommodate that, a Cassandra seed provider needs to interact with the Kubernetes API server.

Here is a short snippet from the custom `KubernetesSeedProvider` (a Java class that implements the Cassandra `SeedProvider` API):

```
public class KubernetesSeedProvider implements SeedProvider {

...

    /**
     * Call Kubernetes API to collect a list of seed providers
     *
     * @return list of seed providers
     */
    public List<InetAddress> getSeeds() {
        GoInterface go = (GoInterface) Native.loadLibrary("cassandra-seed.so",
GoInterface.class);

        String service = getEnvOrDefault("CASSANDRA_SERVICE", "cassandra");
        String namespace = getEnvOrDefault("POD_NAMESPACE", "default");

        String initialSeeds = getEnvOrDefault("CASSANDRA_SEEDS", "");

        if ("".equals(initialSeeds)) {
            initialSeeds = getEnvOrDefault("POD_IP", "");
        }
```

```
        String seedSizeVar = getEnvOrDefault("CASSANDRA_SERVICE_NUM_SEEDS",
"8");
        Integer seedSize = Integer.valueOf(seedSizeVar);

        String data = go.GetEndpoints(namespace, service, initialSeeds);
        ObjectMapper mapper = new ObjectMapper();

        try {
            Endpoints endpoints = mapper.readValue(data, Endpoints.class);
            logger.info("cassandra seeds: {}", endpoints.ips.toString());
            return Collections.unmodifiableList(endpoints.ips);
        } catch (IOException e) {
            // This should not happen
            logger.error("unexpected error building cassandra seeds: {}" ,
e.getMessage());
            return Collections.emptyList();
        }
    }
```

Creating a Cassandra headless service

The role of the headless service is to allow clients in the Kubernetes cluster to connect to the Cassandra cluster through a standard Kubernetes service instead of keeping track of the network identities of the nodes or putting a dedicated load balancer in front of all the nodes. Kubernetes provides all that out of the box through its services.

Here is the `Service` manifest:

```
apiVersion:  v1
kind: Service
metadata:
  labels:
    app: cassandra
    name: cassandra
spec:
  clusterIP: None
  ports:
    - port: 9042
  selector:
    app: Cassandra
```

The `app: cassandra` label will group all the pods that participate in the service. Kubernetes will create endpoint records and the DNS will return a record for discovery. The `clusterIP` is `None`, which means the service is headless and Kubernetes will not do any load-balancing or proxying. This is important because Cassandra nodes do their own communication directly.

The `9042` port is used by Cassandra to serve CQL requests. Those can be queries, inserts/updates (it's always an upsert with Cassandra), or deletes.

Using StatefulSet to create the Cassandra cluster

Declaring a `StatefulSet` is not trivial. It is arguably the most complex Kubernetes resource. It has a lot of moving parts: standard metadata, the StatefulSet spec, the pod template (which is often pretty complex itself), and volume claim templates.

Dissecting the StatefulSet YAML file

Let's go methodically over this example StatefulSet YAML file that declares a three-node Cassandra cluster.

Here is the basic metadata. Note the `apiVersion` string is `apps/v1` (`StatefulSet` became generally available in Kubernetes 1.9):

```
apiVersion: "apps/v1"
kind: StatefulSet
metadata:
  name: cassandra
  labels:
     app: cassandra
```

The StatefulSet spec defines the headless service name, the label selector (`app: cassandra`), how many pods there are in the StatefulSet, and the pod template (explained later). The `replicas` field specifies how many pods are in the StatefulSet:

```
spec:
  serviceName: cassandra
  replicas: 3
  selector:
    matchLabels:
      app: cassandra
  template:
      ...
```

The term "replicas" for the pods is an unfortunate choice because the pods are not replicas of each other. They share the same pod template, but they have a unique identity, and they are responsible for different subsets of the state in general. This is even more confusing in the case of Cassandra, which uses the same term, "replicas," to refer to groups of nodes that redundantly duplicate some subset of the state (but are not identical, because each can manage an additional state too).

I opened a GitHub issue with the Kubernetes project to change the term from replicas to members:

`https://github.com/kubernetes/kubernetes.github.io/issues/2103`

The pod template contains a single container based on the custom Cassandra image. It also sets the termination grace period to 30 minutes. This means that when Kubernetes needs to terminate the pod, it will send the containers a `SIGTERM` signal notifying them they should exit, giving them a chance to do so gracefully. Any container that is still running after the grace period will be killed via `SIGKILL`.

Here is the pod template with the `app: cassandra` label:

```
    template:
      metadata:
        labels:
          app: cassandra
      spec:
        terminationGracePeriodSeconds: 1800
        containers:
        ...
```

The `containers` section has multiple important parts. It starts with a name and the image we looked at earlier:

```
        containers:
        - name: cassandra
          image: gcr.io/google-samples/cassandra:v14
          imagePullPolicy: Always
```

Then, it defines multiple container ports needed for external and internal communication by Cassandra nodes:

```
          ports:
          - containerPort: 7000
            name: intra-node
          - containerPort: 7001
            name: tls-intra-node
          - containerPort: 7199
            name: jmx
          - containerPort: 9042
            name: cql
```

The `resources` section specifies the CPU and memory needed by the container. This is critical because the storage management layer should never be a performance bottleneck due to CPU or memory. Note that it follows the best practice of identical requests and limits to ensure the resources are always available once allocated:

```
        resources:
          limits:
            cpu: "500m"
            memory: 1Gi
          requests:
           cpu: "500m"
           memory: 1Gi
```

Cassandra needs access to **inter-process communication** (**IPC**), which the container requests through the security context's capabilities:

```
        securityContext:
          capabilities:
            add:
              - IPC_LOCK
```

The `lifecycle` section runs the Cassandra `nodetool drain` command to make sure data on the node is transferred to other nodes in the Cassandra cluster when the container needs to shut down. This is the reason a 30-minute grace period is needed. Node draining involves moving a lot of data around:

```
        lifecycle:
          preStop:
            exec:
              command:
              - /bin/sh
              - -c
              - nodetool drain
```

The env section specifies the environment variables that will be available inside the container. The following is a partial list of the necessary variables. The `CASSANDRA_SEEDS` variable is set to the headless service, so a Cassandra node can talk to seed nodes on startup and discover the whole cluster. Note that in this configuration we don't use the special Kubernetes seed provider. `POD_IP` is interesting because it utilizes the Downward API to populate its value via the field reference to `status.podIP`:

```
        env:
          - name: MAX_HEAP_SIZE
            value: 512M
          - name: HEAP_NEWSIZE
            value: 100M
          - name: CASSANDRA_SEEDS
            value: "cassandra-0.cassandra.default.svc.cluster.local"
          - name: CASSANDRA_CLUSTER_NAME
            value: "K8Demo"
          - name: CASSANDRA_DC
            value: "DC1-K8Demo"
```

```
            - name: CASSANDRA_RACK
              value: "Rack1-K8Demo"
            - name: CASSANDRA_SEED_PROVIDER
              value: io.k8s.cassandra.KubernetesSeedProvider
            - name: POD_IP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
```

The readiness probe makes sure that requests are not sent to the node until it is actually ready to service them. The `ready-probe.sh` script utilizes Cassandra's `nodetool status` command:

```
          readinessProbe:
            exec:
              command:
              - /bin/bash
              - -c
              - /ready-probe.sh
            initialDelaySeconds: 15
            timeoutSeconds: 5
```

The last part of the container spec is the volume mount, which must match a persistent volume claim:

```
          volumeMounts:
          - name: cassandra-data
            mountPath: /var/lib/cassandra
```

That's it for the container spec. The last part is the volume claim templates. In this case, dynamic provisioning is used. It's highly recommended to use SSD drives for Cassandra storage, especially its journal. The requested storage in this example is 1 GiB. I discovered through experimentation that 1–2 TB is ideal for a single Cassandra node. The reason is that Cassandra does a lot of data shuffling under the covers, compacting and rebalancing the data. If a node leaves the cluster or a new one joins the cluster, you have to wait until the data is properly rebalanced before the data from the node that left is properly redistributed or a new node is populated.

Note that Cassandra needs a lot of disk space to do all this shuffling. It is recommended to have 50% free disk space. When you consider that you also need replication (typically 3x), then the required storage space can be 6x your data size. You can get by with 30% free space if you're adventurous and maybe use just 2x replication depending on your use case. But don't get below 10% free disk space, even on a single node. I learned the hard way that Cassandra will simply get stuck and will be unable to compact and rebalance such nodes without extreme measures.

The storage class `fast` must be defined in this case. Usually, for Cassandra, you need a special storage class and can't use the Kubernetes cluster default storage class.

The access mode is, of course, `ReadWriteOnce`:

```
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
    spec:
      storageClassName: fast
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

When deploying a StatefulSet, Kubernetes creates the pods in order per their index number. When scaling up or down, it also does so in order. For Cassandra, this is not important because it can handle nodes joining or leaving the cluster in any order. When a Cassandra pod is destroyed (ungracefully), the persistent volume remains. If a pod with the same index is created later, the original persistent volume will be mounted into it. This stable connection between a particular pod and its storage enables Cassandra to manage the state properly.

Summary

In this chapter, we covered the topic of stateful applications and how to integrate them with Kubernetes. We discovered that stateful applications are complicated and considered several mechanisms for discovery, such as DNS and environment variables. We also discussed several state management solutions, such as in-memory redundant storage, local storage, and persistent storage. The bulk of the chapter revolved around deploying a Cassandra cluster inside a Kubernetes cluster using a StatefulSet. We drilled down into the low-level details in order to appreciate what it really takes to integrate a third-party complex distributed system like Cassandra into Kubernetes. At this point, you should have a thorough understanding of stateful applications and how to apply them within your Kubernetes-based system. You are armed with multiple methods for various use cases, and maybe you've even learned a little bit about Cassandra.

In the next chapter, we will continue our journey and explore the important topic of scalability, in particular auto-scalability, and how to deploy and do live upgrades and updates as the cluster dynamically grows. These issues are very intricate, especially when the cluster has stateful apps running on it.

8

Deploying and Updating Applications

In this chapter, we will explore the automated pod scalability that Kubernetes provides, how it affects rolling updates, and how it interacts with quotas. We will touch on the important topic of provisioning and how to choose and manage the size of the cluster. Finally, we will look into CI/CD pipelines and infrastructure provisioning. Here are the main points we will cover:

- Live cluster updates
- Horizontal pod autoscaling
- Performing rolling updates with autoscaling
- Handling scarce resources with quotas and limits
- Continuous integration and deployment
- Provisioning infrastructure with Terraform, Pulumi, custom operators, and Crossplane

By the end of this chapter, you will have the ability to plan a large-scale cluster, provision it economically, and make informed decisions about the various trade-offs between performance, cost, and availability. You will also understand how to set up horizontal pod auto-scaling and use resource quotas intelligently to let Kubernetes automatically handle intermittent fluctuations in volume as well as deploy software safely to your cluster.

Live cluster updates

One of the most complicated and risky tasks involved in running a Kubernetes cluster is a live upgrade. The interactions between different parts of the system when some parts have different versions are often difficult to predict, but in many situations, they are required. Large clusters with many users can't afford to be offline for maintenance. The best way to attack complexity is to divide and conquer. Microservice architecture helps a lot here. You never upgrade your entire system. You just constantly upgrade several sets of related microservices, and if APIs have changed, then you upgrade their clients, too. A properly designed upgrade will preserve backward compatibility at least until all clients have been upgraded, and then deprecate old APIs across several releases.

In this section, we will discuss how to go about updating your cluster using various strategies, such as rolling updates, blue-green deployments, and canary deployments. We will also discuss when it's appropriate to introduce breaking upgrades versus backward-compatible upgrades. Then we will get into the critical topic of schema and data migrations.

Rolling updates

Rolling updates are updates where you gradually update components from the current version to the next. This means that your cluster will run current and new components at the same time. There are two cases to consider here:

- New components are backward-compatible
- New components are not backward-compatible

If the new components are backward-compatible, then the upgrade should be very easy. In earlier versions of Kubernetes, you had to manage rolling updates very carefully with labels and change the number of replicas gradually for both the old and new versions (although `kubectl rolling-update` is a convenient shortcut for replication controllers). But, the `Deployment` resource introduced in Kubernetes 1.2 makes it much easier and supports replica sets. It has the following capabilities built-in:

- Running server-side (it keeps going if your machine disconnects)
- Versioning
- Multiple concurrent rollouts
- Updating deployments
- Aggregating status across all pods
- Rollbacks
- Canary deployments
- Multiple upgrade strategies (rolling upgrade is the default)

Here is a sample manifest for a deployment that deploys three Nginx pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
```

```
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

The resource kind is `Deployment` and it's got the name `nginx-deployment`, which you can use to refer to this deployment later (for example, for updates or rollbacks). The most important part is, of course, the `spec`, which contains a pod template. The replicas determine how many pods will be in the cluster, and the template spec has the configuration for each container. In this case, just a single container.

To start the rolling update, create the deployment resource and check that it rolled out successfully:

```
$ k create -f nginx-deployment.yaml
deployment.apps/nginx-deployment created

$ k rollout status deployment/nginx-deployment
deployment "nginx-deployment" successfully rolled out
Deployments have an update strategy, which defaults to rollingUpdate:
$ k get deployment nginx-deployment -o yaml | grep strategy -A 4
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
```

The following diagram illustrates how a rolling update works:

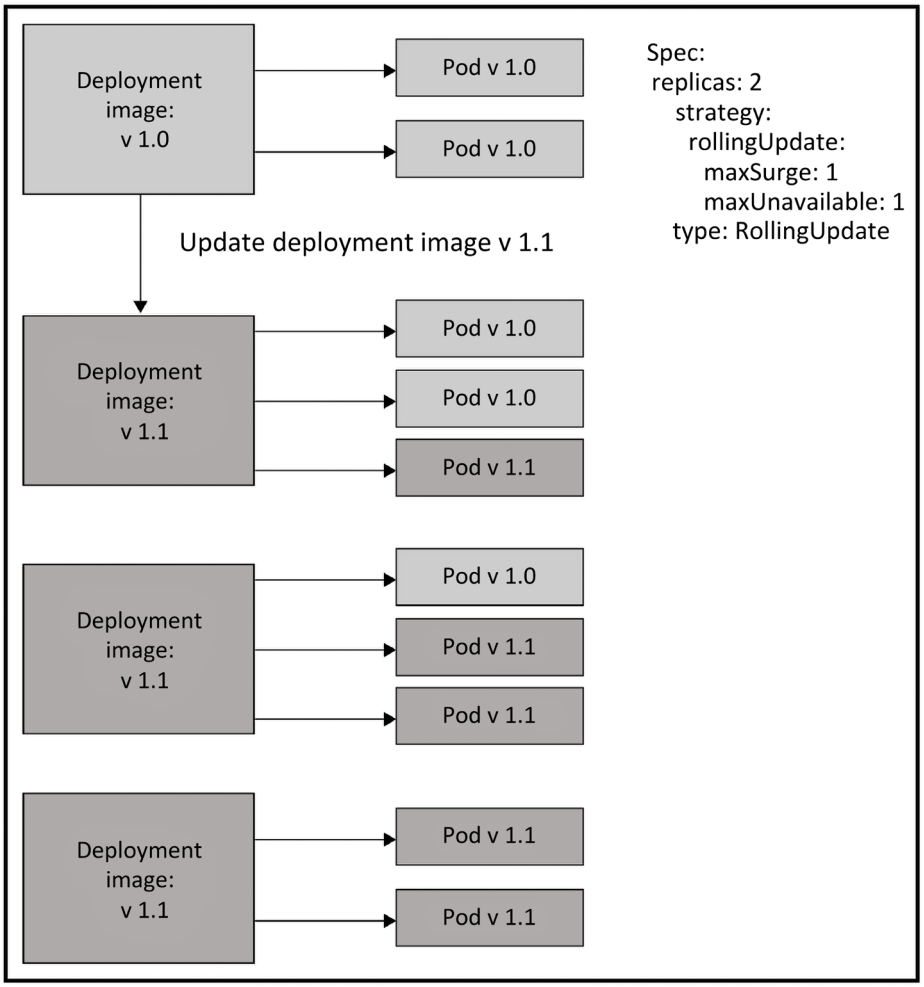


Figure 8.1: Kubernetes rolling update

Complex deployments

The `Deployment` resource is great when you just want to upgrade one pod, but you may often need to upgrade multiple pods, and those pods sometimes have version inter-dependencies. In those situations, you sometimes must forgo a rolling update or introduce a temporary compatibility layer.

For example, suppose service A depends on service B. Service B now has a breaking change. The v1 pods of service A can't interoperate with the pods from service B v2. It is also undesirable from a reliability and change management point of view to make the v2 pods of service B support the old and new APIs. In this case, the solution may be to introduce an adapter service that implements the v1 API of the B service. This service will sit between A and B and will translate requests and responses across versions.

This adds complexity to the deployment process and requires several steps, but the benefit is that the A and B services themselves are simple. You can do rolling updates across incompatible versions, and all indirection can go away once everybody upgrades to v2 (all A pods and all B pods).

But, rolling updates are not always the answer.

Blue-green deployments

Rolling updates are great for availability, but sometimes the complexity involved in managing a proper rolling update is considered too high, or it adds a significant amount of work, which pushes back more important projects. In these cases, blue-green upgrades provide a great alternative. With a blue-green release, you prepare a full copy of your production environment with the new version. Now you have two copies, old (blue) and new (green). It doesn't matter which one is blue and which one is green. The important thing is that you have two fully independent production environments. Currently, blue is active and services all requests. You can run all your tests on green. Once you're happy, you flip the switch and green becomes active. If something goes wrong, rolling back is just as easy; just switch back from green to blue.

The following diagram illustrates how blue-green deployments work using two deployments, two labels, and a single service, which uses a label selector to switch from the blue deployment to the green deployment:

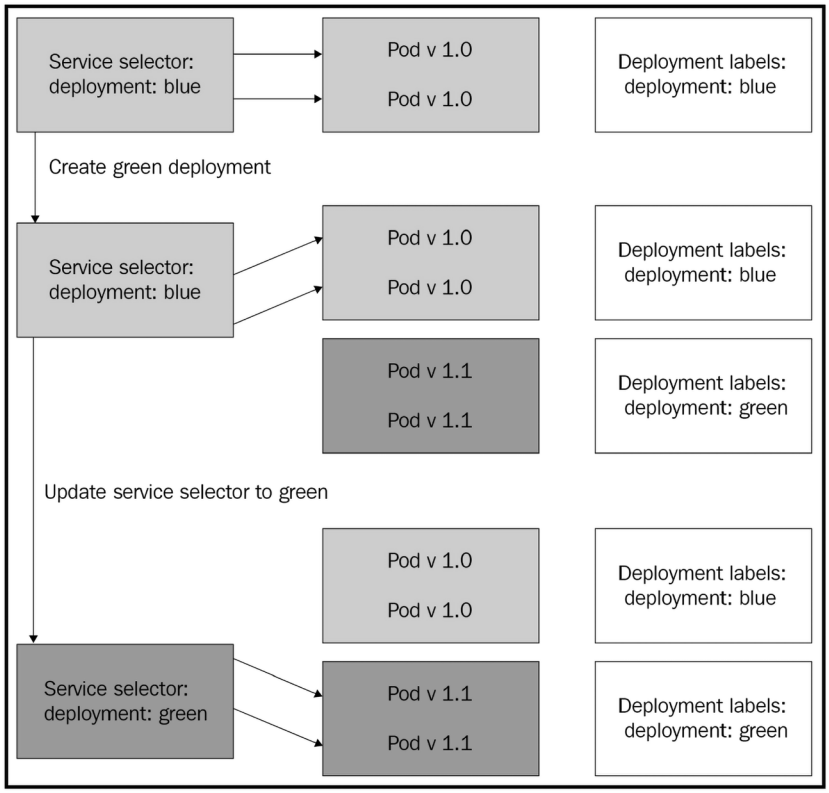


Figure 8.2: Blue-green deployment

I totally ignored the storage and in-memory state in the previous discussion. This immediate switch assumes that blue and green are composed of stateless components only and share a common persistence layer.

If there were storage changes or breaking changes to the API accessible to external clients, then additional steps would need to be taken. For example, if blue and green have their own storage, then all incoming requests may need to be sent to both blue and green, and green may need to ingest historical data from blue to get in sync before switching.

Canary deployments

Blue-green deployments are cool. However, there are times when a more nuanced approach is needed. Suppose you are responsible for a large distributed system with many users. The developers plan to deploy a new version of their service. They tested the new version of the service in the test and staging environment. But, the production environment is too complicated to be replicated one to one for testing purposes. This means there is a risk that the service will misbehave in production. That's where canary deployments shine.

The basic idea is to test the service in production but in a limited capacity. This way, if something is wrong with the new version, only a small fraction of your users or a small fraction of requests will be impacted. This can be implemented very easily in Kubernetes at the pod level. If a service is backed up by 10 pods and you deploy the new version to one pod, then only 10% of the requests will be routed by the service load balancer to the canary pod, while 90% of the requests are still serviced by the current version.

The following diagram illustrates this approach:

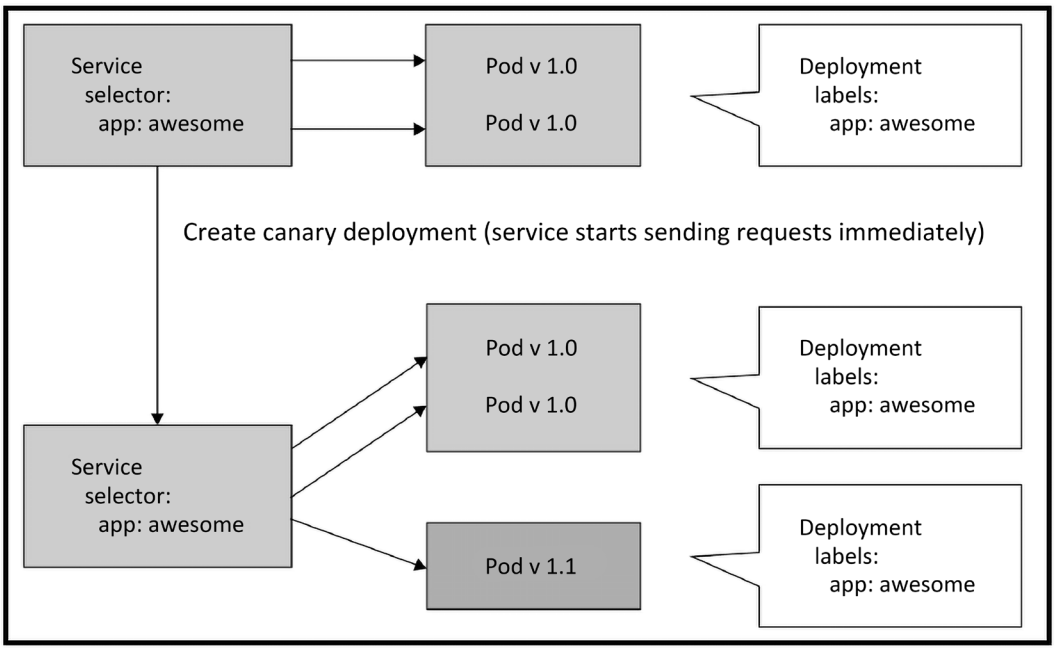


Figure 8.3: Canary deployment

There are more sophisticated ways to route traffic to a canary deployment using a service mesh. We will examine this later in *Chapter 14*, *Utilizing Service Meshes*.

We have discussed different ways to perform live cluster updates. Let's now address the hard problem of managing data-contract changes.

Managing data-contract changes

Data contracts describe how the data is organized. It's an umbrella term for structure metadata. The most common example is a relational database schema. Other examples include network payloads, file formats, and even the content of string arguments or responses. If you have a configuration file, then this configuration file has both a file format (JSON, YAML, TOML, XML, INI, or custom format) and some internal structure that describes what kind of hierarchy, keys, values, and data types are valid. Sometimes the data contract is explicit and sometimes it's implicit. Either way, you need to manage it carefully, or else you'll get runtime errors when code that's reading, parsing, or validating encounters data with an unfamiliar structure.

Migrating data

Data migration is a big deal. Many systems these days manage staggering amounts of data measured in terabytes, petabytes, or more. The amount of collected and managed data will continue to increase for the foreseeable future. The pace of data collection exceeds the pace of hardware innovation. The essential point is that if you have a lot of data, and you need to migrate it, it can take a while. In a previous company, I oversaw a project to migrate close to 100 terabytes of data from one Cassandra cluster of a legacy system to another Cassandra cluster.

The second Cassandra cluster had a different schema and was accessed by a Kubernetes cluster 24/7. The project was very complicated, and thus it kept getting pushed back when urgent issues popped up. The legacy system was still in place side by side with the next-gen system long after the original estimate.

There were a lot of mechanisms in place to split the data and send it to both clusters, but then we ran into scalability issues with the new system, and we had to address those before we could continue. The historical data was important, but it didn't have to be accessed with the same service level as recent hot data. So, we embarked on yet another project to send historical data to cheaper storage. That meant, of course, that client libraries or frontend services had to know how to query both stores and merge the results. When you deal with a lot of data, you can't take anything for granted. You run into scalability issues with your tools, your infrastructure, your third-party dependencies, and your processes. Large scale is not just a quantity change; it is often a qualitative change as well. Don't expect it to go smoothly. It is much more than copying some files from A to B.

Deprecating APIs

API deprecation comes in two flavors: internal and external. Internal APIs are APIs used by components that are fully controlled by you and your team or organization. You can be sure that all API users will upgrade to the new API within a short time. External APIs are used by users or services outside your direct sphere of influence.

There are a few gray-area situations where you work for a huge organization (think Google), and even internal APIs may need to be treated as external APIs. If you're lucky, all your external APIs are used by self-updating applications or through a web interface you control. In those cases, the API is practically hidden and you don't even need to publish it.

If you have a lot of users (or a few very important users) using your API, you should consider deprecation very carefully. Deprecating an API means you force your users to change their application to work with you or stay locked into an earlier version.

There are a few ways you can mitigate the pain:

- Don't deprecate. Extend the existing API or keep the previous API active. It is sometimes pretty simple, although it adds a testing burden.
- Provide client libraries in all relevant programming languages to your target audience. This is always good practice. It allows you to make many changes to the underlying API without disrupting users (as long as you keep the programming language interface stable).
- If you have to deprecate, explain why, allow ample time for users to upgrade, and provide as much support as possible (for example, an upgrade guide with examples). Your users will appreciate it.

We covered different ways to deploy and upgrade workloads and discussed how to manage data migrations and deprecating APIs. Let's take a look at another staple of Kubernetes – horizontal pod autoscaling – which allows our workloads to efficiently handle different volumes of requests and dynamically adjust the number of pods used to process these requests.

Horizontal pod autoscaling

Kubernetes can watch over your pods and scale them when the CPU utilization, memory, or some other metric crosses a threshold. The autoscaling resource specifies the details (the percentage of CPU and how often to check) and the corresponding autoscaling controller adjusts the number of replicas if needed.

The following diagram illustrates the different players and their relationships:

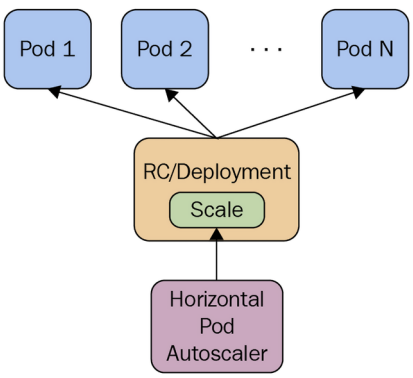


Figure 8.4: Horizontal pod autoscaling

As you can see, the horizontal pod autoscaler doesn't create or destroy pods directly. It adjusts the number of replicas in a `Deployment` or `StatefulSet` resource and its corresponding controllers take care of actually creating and destroying pods. This is very smart because you don't need to deal with situations where autoscaling conflicts with the normal operation of those controllers, unaware of the autoscaler efforts.

The autoscaler automatically does what we had to do ourselves before. Without the autoscaler, if we had a deployment with replicas set to 3, but we determined that, based on average CPU utilization, we actually needed 4, then we would have to update the deployment from 3 to 4 and keep monitoring the CPU utilization manually in all pods. However, instead, the autoscaler will do it for us.

Creating a horizontal pod autoscaler

To declare a horizontal pod autoscaler, we need a workload resource (`Deployment` or `StatefulSet`), and a `HorizontalPodAutoscaler` resource. Here is a simple deployment configured to maintain 3 Nginx pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: nginx
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: 400m
        ports:
        - containerPort: 80
```

```
$ k apply -f nginx-deployment.yaml
deployment.apps/nginx created
```

Note that in order to participate in autoscaling, the containers must request a specific amount of CPU.

The horizontal pod autoscaler references the Nginx deployment in `scaleTargetRef`:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  maxReplicas: 4
  minReplicas: 2

  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 90
```

```
$ k apply -f nginx-hpa.yaml
horizontalpodautoscaler.autoscaling/nginx created
```

The `minReplicas` and `maxReplicas` specify the range of scaling. This is needed to avoid runaway situations that could occur because of some problem. Imagine that, due to some bug, every pod immediately uses 100% of the CPU regardless of the actual load. Without the `maxReplicas` limit, Kubernetes will keep creating more and more pods until all cluster resources are exhausted. If we are running in a cloud environment with autoscaling of VMs, then we will incur a significant cost. The other side of this problem is that if there is no `minReplicas` and there is a lull in activity, then all pods could be terminated, and when new requests come in a new pod will have to be created and scheduled again, which could take several minutes if a new node needs to be provisioned too, and if the pod takes a while to get ready, it adds up. If there are patterns of on and off activity, then this cycle can repeat multiple times. Keeping the minimum number of replicas running can smooth this phenomenon. In the preceding example, `minReplicas` is set to 2, and `maxReplicas` is set to 4. Kubernetes will ensure that there are always between 2 to 4 Nginx instances running.

The **target CPU utilization percentage** is a mouthful. Let's abbreviate it to **TCUP**. You specify a single number like 80%, but Kubernetes doesn't start scaling up and down immediately when the threshold is crossed. This could lead to constant thrashing if the average load hovers around the TCUP. Kubernetes will alternate frequently between adding more replicas and removing replicas. This is often not a desired behavior. To address this concern, you can specify a delay for either scaling up or scaling down.

There are two flags for the `kube-controller-manager` to support this:

- `--horizontal-pod-autoscaler-downscale-delay`: The provided option requires a duration value that determines the waiting period for the autoscaler before initiating another downscale operation once the current one has finished. The default duration is set to 5 minutes (5m0s).
- `--horizontal-pod-autoscaler-upscale-delay`: This option expects a duration value that determines the waiting period for the autoscaler before initiating another upscale operation once the current one has finished. By default, the duration is set to 3 minutes (3m0s).

Let's check the HPA:

```
$ k get hpa
NAME    REFERENCE          TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
nginx   Deployment/nginx   <unknown>/90%   2         4         3          70s
```

As you can see, the targets are unknown. The HPA requires a metrics server to measure the CPU percentage. One of the easiest ways to install the metrics server is using Helm. We installed Helm in *Chapter 2*, *Creating Kubernetes Clusters*, already. Here is the command to install the Kubernetes metrics server into the monitoring namespace:

```
$ helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
"metrics-server" has been added to your repositories

$ helm upgrade --install metrics-server metrics-server/metrics-server \
               --namespace monitoring \
               --create-namespace

Release "metrics-server" does not exist. Installing it now.
NAME: metrics-server
LAST DEPLOYED: Sat Jul 30 23:16:09 2022
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
***********************************************************************
* Metrics Server                                                      *
***********************************************************************
  Chart version: 3.8.2
  App version:   0.6.1
  Image tag:     k8s.gcr.io/metrics-server/metrics-server:v0.6.1
***********************************************************************
```

Unfortunately, the `metrics-server` can't run on a KinD cluster out of the box due to certificate issues.

This is easy to fix with the following command:

```
$ k patch -n monitoring deployment metrics-server --type=json \
  -p '[{"op":"add","path":"/spec/template/spec/containers/0/args/-","value":"--
kubelet-insecure-tls"}]'
```

We may need to wait for the metrics server to be ready. A good way to do that is using `kubectl wait`:

```
kubectl wait deployment metrics-server -n monitoring --for=condition=Available
deployment.apps/metrics-server condition met
```

Now that kubectl has returned, we can also take advantage of the `kubectl top` command, which shows metrics about nodes and pods:

```
$ k top no
NAME                 CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
kind-control-plane   213m         5%     15Mi            0%


$ k top po
NAME                     CPU(cores)   MEMORY(bytes)
nginx-64f97b4d86-gqmjj   0m           3Mi
nginx-64f97b4d86-sj8cz   0m           3Mi
nginx-64f97b4d86-xc99j   0m           3Mi
```

After redeploying Nginx and the HPA, you can see the utilization and that the replica count is 3, which is within the range of 2-4:

```
$ k get hpa
NAME    REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx   Deployment/nginx   0%/90%    2         4         3          26s
```

Since the CPU utilization is below the utilization target, after a few minutes, the HPA will scale down Nginx to the minimum 2 replicas:

```
$ k get hpa
NAME    REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx   Deployment/nginx   0%/90%    2         4         2          6m57s
```

Custom metrics

CPU utilization is an important metric to gauge if pods that are bombarded with too many requests should be scaled up, or if they are mostly idle and can be scaled down. But CPU is not the only, and sometimes not even the best, metric to keep track of. Memory may be the limiting factor, or even more specialized metrics, such as the number of concurrent threads, the depth of a pod's internal on-disk queue, the average latency on a request, or the average number of service timeouts.

The horizontal pod custom metrics were added as an alpha extension in version 1.2. In version 1.6 they were upgraded to beta status, and in version 1.23, they became stable. You can now autoscale your pods based on multiple custom metrics.

The autoscaler will evaluate all the metrics and will autoscale based on the largest number of replicas required, so the requirements of all the metrics are respected.

Using the horizontal pod autoscaler with custom metrics requires some configuration when launching your cluster. First, you need to enable the API aggregation layer. Then you need to register your resource metrics API and your custom metrics API. This is not trivial. Enter Keda.

Keda

Keda stands for **Kubernetes Event-Driven Autoscaling**. It is an impressive project that packages everything you need to implement custom metrics for horizontal pod autoscaling. Typically, you would want to scale Deployments, StatefulSets, or Jobs, but Keda can also scale CRDs as long as they have a `/scale` subresource. Keda is deployed as an operator that watches several custom resources:

- `scaledobjects.keda.sh`
- `scaledjobs.keda.sh`
- `triggerauthentications.keda.sh`
- `clustertriggerauthentications.keda.sh`

Keda also has a metrics server, which supports a large number of event sources and scalers and can collect metrics from all these sources to inform the scaling process. Event sources include all the popular databases, message queues, cloud data stores, and various monitoring APIs. For example, if you rely on Prometheus for your metrics, you can use Keda to scale your workloads based on any metric or combination of metrics you push to Prometheus.

The following diagram depicts Keda's architecture:

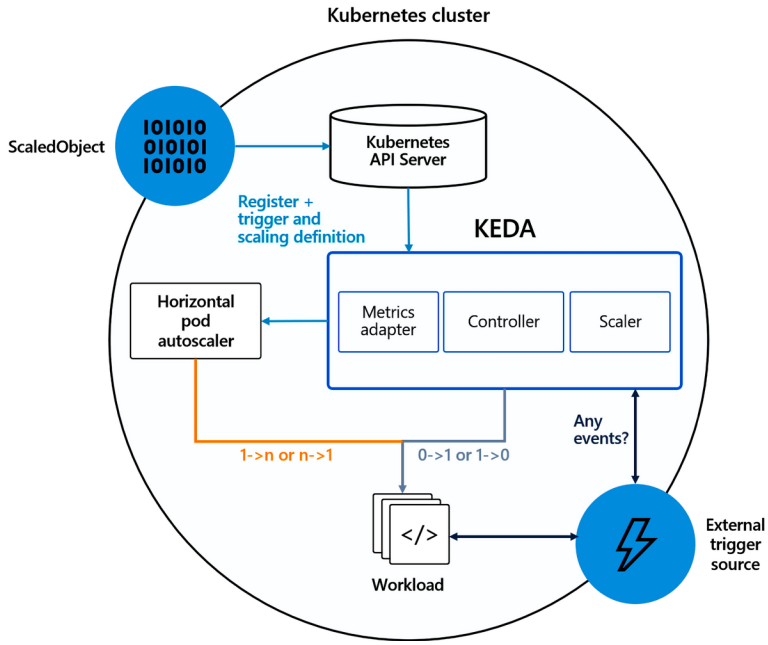


Figure 8.5: Keda architecture

See `https://keda.sh` for more details.

Autoscaling with kubectl

kubectl can create an autoscale resource using the standard `create` command accepting a configuration file. But kubectl also has a special command, `autoscale`, which lets you easily set an autoscaler in one command without a special configuration file.

First, let's start a deployment that makes sure there are three replicas of a simple pod and that just runs an infinite bash loop:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bash-loop
spec:
  replicas: 3
  selector:
    matchLabels:
      name: bash-loop
  template:
    metadata:
      labels:
        name: bash-loop
    spec:
      containers:
        - name: bash-loop
          image: g1g1/py-kube:0.3
          resources:
            requests:
              cpu: 100m
          command: ["/bin/bash", "-c", "while true; do sleep 10; done"]
```

```
$ k apply -f bash-loop-deployment.yaml
deployment.apps/bash-loop created
```

Here is the resulting deployment:

```
$ k get deployment
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
bash-loop   3/3     3            3           35s
```

You can see that the desired count and current count are both 3, meaning three pods are running. Let's make sure:

```
$ k get pods
NAME                          READY   STATUS    RESTARTS   AGE
bash-loop-8496f889f8-9khjs    1/1     Running   0          106s
bash-loop-8496f889f8-frhb7    1/1     Running   0          105s
bash-loop-8496f889f8-hcd2d    1/1     Running   0          105s
```

Now, let's create an autoscaler. To make it interesting, we'll set the minimum number of replicas to 4 and the maximum number to 6:

```
$ k autoscale deployment bash-loop --min=4 --max=6 --cpu-percent=50
horizontalpodautoscaler.autoscaling/bash-loop autoscaled
```

Here is the resulting horizontal pod autoscaler (you can use `hpa`). It shows the referenced deployment, the target and current CPU percentage, and the min/max pods. The name matches the referenced deployment `bash-loop`:

```
$ k get hpa
NAME        REFERENCE               TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
bash-loop   Deployment/bash-loop    2%/50%    4         6         4          36s
```

Originally, the deployment was set to have three replicas, but the autoscaler has a minimum of four pods. What's the effect on the deployment? Now the desired number of replicas is four. If the average CPU utilization goes above 50%, then it will climb to five or even six, but never below four:

```
$ k get deployment
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
bash-loop   4/4     4            4           4m11s
```

When we delete the horizontal pod autoscaler, the deployment retains the last desired number of replicas (4, in this case). Nobody remembers that the deployment was created initially with three replicas:

```
$ k delete hpa bash-loop
horizontalpodautoscaler.autoscaling "bash-loop" deleted
```

As you can see, the deployment wasn't reset and still maintains four pods, even when the autoscaler is gone:

```
$ k get deployment
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
bash-loop   4/4     4            4           5m17s
```

This makes sense because the horizontal pod autoscaler modified the spec of the deployment to have 4 replicas:

```
$ k get deploy bash-loop -o jsonpath='{.spec.replicas}'
4
```

Let's try something else. What happens if we create a new horizontal pod autoscaler with a range of 2 to 6 and the same CPU target of 50%?

```
$ k autoscale deployment bash-loop --min=2 --max=6 --cpu-percent=50
horizontalpodautoscaler.autoscaling/bash-loop autoscaled
```

Well, the deployment still maintains its four replicas, which is within the range:

```
$ k get deployment
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
bash-loop   4/4     4            4           8m18s
```

However, the actual CPU utilization is just 2%. The deployment will eventually be scaled down to two replicas, but because the horizontal pod autoscaler doesn't scale down immediately, we have to wait a few minutes (5 minutes by default):

```
$ k get deployment
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
bash-loop   2/2     2            2           28m
```

Let's check out the horizontal pod autoscaler itself:

```
$ k get hpa
NAME        REFERENCE                TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
bash-loop   Deployment/bash-loop     2%/50%    2         6         2          21m
```

Now, that you understand what horizontal pod autoscaling is all about, let's look at performing rolling updates with autoscaling.

Performing rolling updates with autoscaling

Rolling updates are a cornerstone of managing workloads in large clusters. When you do a rolling update of a deployment controlled by an HPA, the deployment will create a new replica set with the new image and start increasing its replicas, while reducing the replicas of the old replica set. At the same time, the HPA may change the total replica count of the deployment. This is not an issue. Everything will reconcile eventually.

Here is a deployment configuration file we've used in *Chapter 5*, *Using Kubernetes Resources in Practice*, for deploying the `hue-reminders` service:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hue-reminders
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
      app: hue
      service: reminders
  template:
    metadata:
      name: hue-reminders
      labels:
        app: hue
        service: reminders
    spec:
      containers:
      - name: hue-reminders
        image: g1g1/hue-reminders:2.2
        resources:
          requests:
            cpu: 100m
        ports:
        - containerPort: 80
```

```
$ k apply -f hue-reminders-deployment.yaml
deployment.apps/hue-reminders created
```

To support it with autoscaling and ensure we always have between 10 to 15 instances running, we can create an autoscaler configuration file:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hue-reminders
spec:
  maxReplicas: 15
  minReplicas: 10
  targetCPUUtilizationPercentage: 90
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: hue-reminders
```

Alternatively, we can use the kubectl `autoscale` command:

```
$ k autoscale deployment hue-reminders --min=10 --max=15 --cpu-percent=90
horizontalpodautoscaler.autoscaling/hue-reminders autoscaled
```

Let's perform a rolling update from version 2.2 to 3.0:

```
$ k set image deployment/hue-reminders hue-reminders=g1g1/hue-reminders:3.0
```

We can check the status using the `rollout status`:

```
$ k rollout status deployment hue-reminders

Waiting for deployment "hue-reminders" rollout to finish: 9 out of 10 new replicas
have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 9 out of 10 new replicas
have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 9 out of 10 new replicas
have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 9 out of 10 new replicas
have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 3 old replicas are pending
termination...
Waiting for deployment "hue-reminders" rollout to finish: 3 old replicas are pending
termination...
Waiting for deployment "hue-reminders" rollout to finish: 2 old replicas are pending
termination...
Waiting for deployment "hue-reminders" rollout to finish: 2 old replicas are pending
termination...
Waiting for deployment "hue-reminders" rollout to finish: 1 old replicas are pending
termination...
Waiting for deployment "hue-reminders" rollout to finish: 1 old replicas are pending
termination...
Waiting for deployment "hue-reminders" rollout to finish: 1 old replicas are pending
termination...
Waiting for deployment "hue-reminders" rollout to finish: 8 of 10 updated replicas
are available...
Waiting for deployment "hue-reminders" rollout to finish: 9 of 10 updated replicas
are available...
deployment "hue-reminders" successfully rolled out
```

Finally, we review the history of the deployment:

```
$ k rollout history deployment hue-reminders
deployment.apps/hue-reminders
REVISION  CHANGE-CAUSE
3         kubectl1.23.4 set image deployment/hue-reminders hue-reminders=g1g1/hue-
reminders:3.0 --record=true
4         kubectl1.23.4 set image deployment/hue-reminders hue-reminders=g1g1/hue-
reminders:3.0 --record=true
```

Autoscaling works based on resource usage and thresholds. In the next section, we will explore how Kubernetes lets us control and manage the resources of each workload using requests and limits.

Handling scarce resources with limits and quotas

With the horizontal pod autoscaler creating pods on the fly, we need to think about managing our resources. Scheduling can easily get out of control, and inefficient use of resources is a real concern. There are several factors, which can interact with each other in subtle ways:

- Overall cluster capacity
- Resource granularity per node
- Division of workloads per namespace
- Daemon sets
- Stateful sets
- Affinity, anti-affinity, taints, and tolerations

First, let's understand the core issue. The Kubernetes scheduler has to take into account all these factors when it schedules pods. If there are conflicts or a lot of overlapping requirements, then Kubernetes may have a problem finding room to schedule new pods. For example, a very extreme yet simple scenario is that a daemon set runs a pod on every node that requires 50% of the available memory. Now, Kubernetes can't schedule any other pod that needs more than 50% memory because the daemon set's pod gets priority. Even if you provision new nodes, the daemon set will immediately commandeer half of the memory.

Stateful sets are similar to daemon sets in that they require new nodes to expand. The trigger for adding new members to the stateful set is growth in data, but the impact is taking resources from the pool available for Kubernetes to schedule other workloads. In a multi-tenant situation, the noisy neighbor problem can rear its head in a provisioning or resource allocation context. You may plan exact rations meticulously in your namespace between different pods and their resource requirements, but you share the actual nodes with your neighbors from other namespaces that you may not even have visibility into.

Most of these problems can be mitigated by judiciously using namespace resource quotas and careful management of the cluster capacity across multiple resource types such as CPU, memory, and storage. In addition, if you control node provisioning, you may carve out dedicated nodes for your workloads by tainting them.

But, in most situations, a more robust and dynamic approach is to take advantage of the cluster autoscaler, which can add capacity to the cluster when needed (until the quota is exhausted).

Enabling resource quotas

Most Kubernetes distributions support `ResourceQuota` out of the box. The API server's `–admission-control` flag must have `ResourceQuota` as one of its arguments. You will also have to create a `ResourceQuota` object to enforce it. Note that there may be at most one `ResourceQuota` object per namespace to prevent potential conflicts. This is enforced by Kubernetes.

Resource quota types

There are different types of quotas we can manage and control. The categories are compute, storage, and objects.

Compute resource quota

Compute resources are CPU and memory. For each one, you can specify a limit or request a certain amount. Here is the list of compute-related fields. Note that `requests.cpu` can be specified as just `cpu`, and `requests.memory` can be specified as just `memory`:

- `limits.cpu`: The total CPU limits, considering all pods in a non-terminal state, must not exceed this value.
- `limits.memory`: The combined memory limits, considering all pods in a non-terminal state, must not surpass this value.
- `requests.cpu`: The total CPU requests, considering all pods in a non-terminal state, should not go beyond this value.
- `requests.memory`: The combined memory requests, considering all pods in a non-terminal state, should not exceed this value.
- `hugepages-`: The maximum allowed number of huge page requests of the specified size, considering all pods in a non-terminal state, must not surpass this value.

Since Kubernetes 1.10, you can also specify a quota for extended resources such as GPU resources. Here is an example:

```
requests.nvidia.com/gpu: 10
```

Storage resource quota

The storage resource quota type is a little more complicated. There are two entities you can restrict per namespace: the amount of storage and the number of persistent volume claims. However, in addition to just globally setting the quota on total storage or the total number of persistent volume claims, you can also do that per storage class. The notation for storage class resource quota is a little verbose, but it gets the job done:

- `requests.storage`: The total amount of requested storage across all persistent volume claims
- `persistentvolumeclaims`: The maximum number of persistent volume claims allowed in the namespace
- `.storageclass.storage.k8s.io/requests.storage`: The total amount of requested storage across all persistent volume claims associated with the `storage-class-name`
- `.storageclass.storage.k8s.io/persistentvolumeclaims`: The maximum number of persistent volume claims allowed in the namespace that are associated with the `storage-class-name`

Kubernetes 1.8 added alpha support for ephemeral storage quotas too:

- `requests.ephemeral-storage`: The total amount of requested ephemeral storage across all pods in the namespace claims
- `limits.ephemeral-storage`: The total amount of limits for ephemeral storage across all pods in the namespace claims

One of the problems with provisioning storage is that disk capacity is not the only factor. Disk I/O is an important resource too. For example, consider a pod that keeps updating the same small file. It will not use a lot of capacity, but it will perform a lot of I/O operations.

Object count quota

Kubernetes has another category of resource quotas, which is API objects. My guess is that the goal is to protect the Kubernetes API server from having to manage too many objects. Remember that Kubernetes does a lot of work under the hood. It often has to query multiple objects to authenticate, authorize, and ensure that an operation doesn't violate any of the many policies that may be in place. A simple example is pod scheduling based on replication controllers. Imagine that you have a million replica set objects. Maybe you just have three pods and most of the replica sets have zero replicas. Still, Kubernetes will spend all its time just verifying that indeed all those million replica sets have no replicas of their pod template and that they don't need to kill any pods. This is an extreme example, but the concept applies. Having too many API objects means a lot of work for Kubernetes.

In addition, it's a problem that clients use the discovery cache like kubectl itself. See this issue: `https://github.com/kubernetes/kubectl/issues/1126`.

Since Kubernetes 1.9, you can restrict the number of any namespaced resource (prior to that, coverage of objects that could be restricted was a little spotty). The syntax is interesting, `count/<resource type>.<group>`. Typically in YAML files and kubectl, you identify objects by group first, as in `<group>/<resource type>`.

Here are some objects you may want to limit (note that deployments can be limited for two separate API groups):

- `count/configmaps`
- `count/deployments.apps`
- `count/deployments.extensions`
- `count/persistentvolumeclaims`
- `count/replicasets.apps`
- `count/replicationcontrollers`
- `count/secrets`
- `count/services`
- `count/statefulsets.apps`
- `count/jobs.batch`
- `count/cronjobs.batch`

Since Kubernetes 1.5, you can restrict the number of custom resources too. Note that while the custom resource definition is cluster-wide, this allows you to restrict the actual number of the custom resources in each namespace. For example:

```
count/awesome.custom.resource
```

The most glaring omission is namespaces. There is no limit to the number of namespaces. Since all limits are per namespace, you can easily overwhelm Kubernetes by creating too many namespaces, where each namespace has only a small number of API objects. But, the ability to create namespaces should be reserved to cluster administrators only, who don't need resource quotas to constrain them.

Quota scopes

Some resources, such as pods, may be in different states, and it is useful to have different quotas for these different states. For example, if there are many pods that are terminating (this happens a lot during rolling updates), then it is OK to create more pods, even if the total number exceeds the quota. This can be achieved by only applying a pod object count quota to non-terminating pods. Here are the existing scopes:

- `Terminating`: Select pods in which the value of `activeDeadlineSeconds` is greater than or equal to `0`.
- `NotTerminating`: Select pods where `activeDeadlineSeconds` is not specified (nil).
- `BestEffort`: Select pods with a best effort quality of service, meaning pods that do not specify resource requests and limits.
- `NotBestEffort`: Select pods that do not have a best effort quality of service, indicating pods that specify resource requests and limits.
- `PriorityClass`: Select pods that define a priority class.
- `CrossNamespacePodAffinity`: Select pods with cross-namespace affinity or anti-affinity terms for pod scheduling.

While the `BestEffort` scope applies only to pods, the `Terminating`, `NotTerminating`, and `NotBestEffort` scopes apply to CPU and memory too. This is interesting because a resource quota limit can prevent a pod from terminating. Here are the supported objects:

- CPU
- Memory
- `limits.cpu`
- `limits.memory`
- `requests.cpu`
- `requests.memory`
- Pods

Resource quotas and priority classes

Kubernetes 1.9 introduced priority classes as a way to prioritize scheduling pods when resources are scarce. In Kubernetes 1.14, priority classes became stable. However, as of Kubernetes 1.12, resource quotas support separate resource quotas per priority class (in beta). That means that with priority classes, you can sculpt your resource quotas in a very fine-grained manner, even within a namespace.

For more details, check out `https://kubernetes.io/docs/concepts/policy/resource-quotas/#resource-quota-per-priorityclass`.

Requests and limits

The meaning of requests and limits in the context of resource quotas is that it requires the containers to explicitly specify the target attribute. This way, Kubernetes can manage the total quota because it knows exactly what range of resources is allocated to each container.

Working with quotas

That was a lot of theory. It's time to get hands-on. Let's create a namespace first:

```
$ k create namespace ns
namespace/ns created
```

Using namespace-specific context

When working with a namespace other than the default, I prefer to set the namespace of the current context, so I don't have to keep typing `--namespace=ns` for every command:

```
$ k config set-context --current --namespace ns
Context "kind-kind" modified.
```

Creating quotas

Here is a quota for compute:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
spec:
  hard:
    pods: 2
    requests.cpu: 1
    requests.memory: 200Mi
    limits.cpu: 2
    limits.memory: 2Gi
```

We create it by typing:

```
$ k apply -f compute-quota.yaml
resourcequota/compute-quota created
```

And here is a count quota:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts-quota
spec:
```

```
    hard:
      count/configmaps: 10
      count/persistentvolumeclaims: 4
      count/jobs.batch: 20
      count/secrets: 3
```

We create it by typing:

```
$ k apply -f object-count-quota.yaml
resourcequota/object-counts-quota created
```

We can observe all the quotas:

```
$ k get quota
NAME                  AGE   REQUEST
LIMIT
compute-quota         32s   pods: 0/2, requests.cpu: 0/1, requests.memory: 0/200Mi
limits.cpu: 0/2, limits.memory: 0/2Gi
object-counts-quota   13s   count/configmaps: 1/10, count/jobs.batch: 0/20, count/
persistentvolumeclaims: 0/4, count/secrets: 1/3
```

We can drill down to get all the information for both resource quotas in a more visually pleasing manner using `kubectl describe`:

```
$ k describe quota compute-quota
Name:            compute-quota
Namespace:       ns
Resource         Used  Hard
--------         ----  ----
limits.cpu       0     2
limits.memory    0     2Gi
pods             0     2
requests.cpu     0     1
requests.memory  0     200Mi


$ k describe quota object-counts-quota
Name:                         object-counts-quota
Namespace:                    ns
Resource                      Used  Hard
--------                      ----  ----
count/configmaps              1     10
count/jobs.batch              0     20
count/persistentvolumeclaims  0     4
count/secrets                 1     3
```

As you can see, it reflects exactly the specification, and it is defined in the `ns` namespace.

This view gives us an instant understanding of the global resource usage of important resources across the cluster without diving into too many separate objects.

Let's add an Nginx server to our namespace:

```
$ k create -f nginx-deployment.yaml
deployment.apps/nginx created
```

Let's check the pods:

```
$ k get po
No resources found in ns namespace.
```

Uh-oh. No resources were found. But, there was no error when the deployment was created. Let's check out the deployment then:

```
$ k describe deployment nginx
Name:                   nginx
Namespace:              ns
CreationTimestamp:      Sun, 31 Jul 2022 13:49:24 -0700
Labels:                 <none>
Annotations:            deployment.kubernetes.io/revision: 1
kind-kind | ns
Selector:               run=nginx
Replicas:               3 desired | 0 updated | 0 total | 0 available | 3 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  run=nginx
  Containers:
   nginx:
    Image:      nginx
    Port:       80/TCP
    Host Port:  0/TCP
    Requests:
      cpu:        400m
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type             Status  Reason
  ----             ------  ------
```

```
  Progressing      True    NewReplicaSetCreated
  Available        False   MinimumReplicasUnavailable
  ReplicaFailure   True    FailedCreate
OldReplicaSets:    <none>
NewReplicaSet:     nginx-64f97b4d86 (0/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  65s   deployment-controller  Scaled up replica set
nginx-64f97b4d86 to 3
```

There it is, in the `Conditions` section – the `ReplicaFailure` status is `True` and the reason is `FailedCreate`. You can see that the deployment created a new replica set called `nginx-64f97b4d86`, but it couldn't create the pods it was supposed to create. We still don't know why.

Let's check out the replica set. I use the JSON output format (`-o json`) and pipe it to `jq` for its nice layout, which is much better than the `jsonpath` output format that kubectl supports natively:

```
$ k get rs nginx-64f97b4d86 -o json | jq .status.conditions
[
  {
    "lastTransitionTime": "2022-07-31T20:49:24Z",
    "message": "pods \"nginx-64f97b4d86-ks7d6\" is forbidden: failed quota: compute-
quota: must specify limits.cpu,limits.memory,requests.memory",
    "reason": "FailedCreate",
    "status": "True",
    "type": "ReplicaFailure"
  }
]
```

The message is crystal clear. Since there is a compute quota in the namespace, every container must specify its CPU, memory requests, and limit. The quota controller must account for every container's compute resource usage to ensure the total namespace quota is respected.

OK. We understand the problem, but how to resolve it? We can create a dedicated deployment object for each pod type we want to use and carefully set the CPU and memory requests and limit.

For example, we can define Nginx deployment with resources. Since the resource quota specifies a hard limit of 2 pods, let's reduce the number of replicas from 3 to 2 as well:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
```

```
  selector:
    matchLabels:
      run: nginx
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: 400m
            memory: 60Mi
          limits:
            cpu: 400m
            memory: 60Mi
        ports:
        - containerPort: 80
```

Let's create it and check the pods:

```
$ k apply -f nginx-deployment-with-resources.yaml
deployment.apps/nginx created

$ k get po
NAME                     READY   STATUS    RESTARTS   AGE
nginx-5d68f45c5f-6h9w9   1/1     Running   0          21s
nginx-5d68f45c5f-b8htm   1/1     Running   0          21s
```

Yeah, it works! However, specifying the limit and resources for each pod type can be exhausting. Is there an easier or better way?

Using limit ranges for default compute quotas

A better way is to specify default compute limits. Enter limit ranges. Here is a configuration file that sets some defaults for containers:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
spec:
  limits:
```

```
    - default:
        cpu: 400m
        memory: 50Mi
      defaultRequest:
        cpu: 400m
        memory: 50Mi
      type: Container
```

Let's create it and observe the default limits:

```
$ k apply -f limits.yaml
limitrange/limits created

$ k describe limits
Name:       limits
Namespace:  ns
Type        Resource  Min  Max  Default Request  Default Limit  Max Limit/Request
Ratio
----        --------  ---  ---  ---------------  -------------  -----------------
---
Container   cpu       -    -    400m             400m           -
Container   memory    -    -    50Mi             50Mi           -
```

To test it, let's delete our current Nginx deployment with the explicit limits and deploy our original Nginx again:

```
$ k delete deployment nginx
deployment.apps "nginx" deleted

$ k apply -f nginx-deployment.yaml
deployment.apps/nginx created

$ k get deployment
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   2/3     2            2           16s
```

As you can see, only 2 out of 3 pods are ready. What happened? The default limits worked but, if you recall, the compute quota had a hard limit of 2 pods for the namespace. There is no way to override it with the `RangeLimit` object, so the deployment was able to create only two Nginx pods. This is exactly the desired result based on the current configuration. If the deployment really requires 3 pods, then the compute quota for the namespace should be updated to allow 3 pods.

This concludes our discussion of resource management using requests, limits, and quotas. The next section explores how to automate the deployment and configuration of multiple workloads at scale on Kubernetes.

Continuous integration and deployment

Kubernetes is a great platform for running your microservice-based applications. But, at the end of the day, it is an implementation detail. Users, and often most developers, may not be aware that the system is deployed on Kubernetes. But Kubernetes can change the game and make things that were too difficult before possible.

In this section, we'll explore the CI/CD pipeline and what Kubernetes brings to the table. At the end of this section, you'll be able to design CI/CD pipelines that take advantage of Kubernetes properties such as easy scaling and development-production parity to improve the productivity and robustness of day-to-day development and deployment.

What is a CI/CD pipeline?

A CI/CD pipeline is a set of tools and steps that takes a set of changes by developers or operators that modify the code, data, or configuration of a system, tests them, and deploys them to production (and possibly other environments). Some pipelines are fully automated and some are semi-automated with human checks. In large organizations, it is common to deploy changes automatically to test and staging environments. Release to production requires manual intervention with human approval. The following diagram depicts a typical CI/CD pipeline that follows this practice:

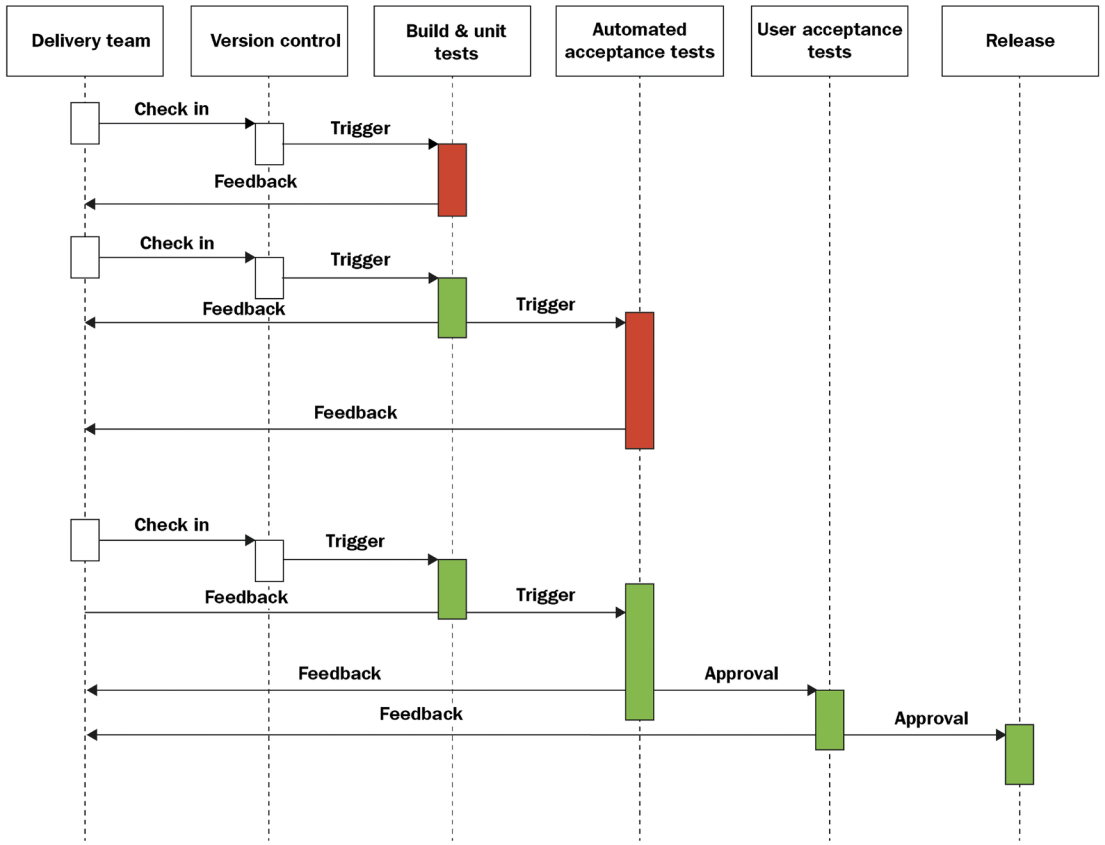


Figure 8.6: CI/CD pipeline

It may be worth mentioning that developers can be completely isolated from production infrastructure. Their interface is just a Git workflow, where a good example is Deis Workflow (PaaS on Kubernetes, similar to Heroku).

Designing a CI/CD pipeline for Kubernetes

When your deployment target is a Kubernetes cluster, you should rethink some traditional practices. For starters, the packaging is different. You need to bake images for your containers. Reverting code changes is super easy and instantaneous by using smart labeling. It gives you a lot of confidence that if a bad change slips through the testing net somehow, you'll be able to revert to the previous version immediately. But you want to be careful there. Schema changes and data migrations can't be automatically rolled back without coordination.

Another unique capability of Kubernetes is that developers can run a whole cluster locally. That takes some work when you design your cluster, but since the microservices that comprise your system run in containers, and those containers interact via APIs, it is possible and practical to do. As always, if your system is very data-driven, you will need to accommodate that and provide data snapshots and synthetic data that your developers can use. Also, if your services access external systems or cloud provider services, then fully local clusters may not be ideal.

Your CI/CD pipeline should allow the cluster administrator to quickly adjust quotas and limits to accommodate scaling and business growth. In addition, you should be able to easily deploy most of your workloads into different environments. For example, if your staging environment diverges from your production environment, it reduces the confidence that changes that worked well in the staging environment will not harm the production environment. By making sure that all environment changes go through CI/CD, it becomes possible to keep different environments in sync.

There are many commercial CI/CD solutions that support Kubernetes, but there are also several Kubernetes-native solutions, such as Tekton, Argo CD, Flux CD, and Jenkins X.

A Kubernetes-native CI/CD solution runs inside your cluster, is specified using Kubernetes CRDs, and uses containers to execute the steps. By using a Kubernetes-native CI/CD solution, you get the benefit of Kubernetes managing and easily scaling your CI/CD pipelines, which is often a non-trivial task.

Provisioning infrastructure for your applications

CI/CD pipelines are used for deploying workloads on Kubernetes. However, these services often require you to operate against infrastructures such as cloud resources, databases, and even the Kubernetes cluster itself. There are different ways to provision this infrastructure. Let's review some of the common solutions.

Cloud provider APIs and tooling

If you are fully committed to a single cloud provider and have no intentions of using multiple cloud providers or mixing cloud-based clusters with on-prem clusters, you may prefer to use your cloud provider's APIs tooling (e.g., AWS CloudFormation). There are several benefits to this approach:

- Deep integration with your cloud provider infrastructure
- Best support from your cloud provider
- No layer of indirection

However, this means that your view of the system will be split. Some information will be available through Kubernetes and stored in etcd. Other information will be stored and accessible through your cloud provider.

The lack of a Kubernetes-native view of infrastructure means that it may be challenging to run local clusters, and incorporating other cloud providers or on-prem will definitely take a lot of work.

Terraform

Terraform (`https://terraform.io`) by HashiCorp is a tool for **IaC** (**infrastructure as code**). It is the incumbent leader. You define your infrastructure using Terraform's HCL language and you can structure your infrastructure configuration using modules. It was initially focused on AWS, but over time it became a general-purpose tool for provisioning infrastructure on any cloud as well as other types of infrastructure via provider plugins.

Check out all the available providers in the Terraform registry: `https://registry.terraform.io/browse/providers`.

Since Terraform defines infrastructure declaratively, it naturally supports the GitOps life cycle, where changes to infrastructure must be checked into code control and can be reviewed, and the history is recorded.

You typically interact with Terraform through its CLI. You can run a `terraform plan` command to see what changes Terraform will make, and if you're happy with the result, you apply these changes via `terraform apply`.

The following diagram demonstrates the Terraform workflow:

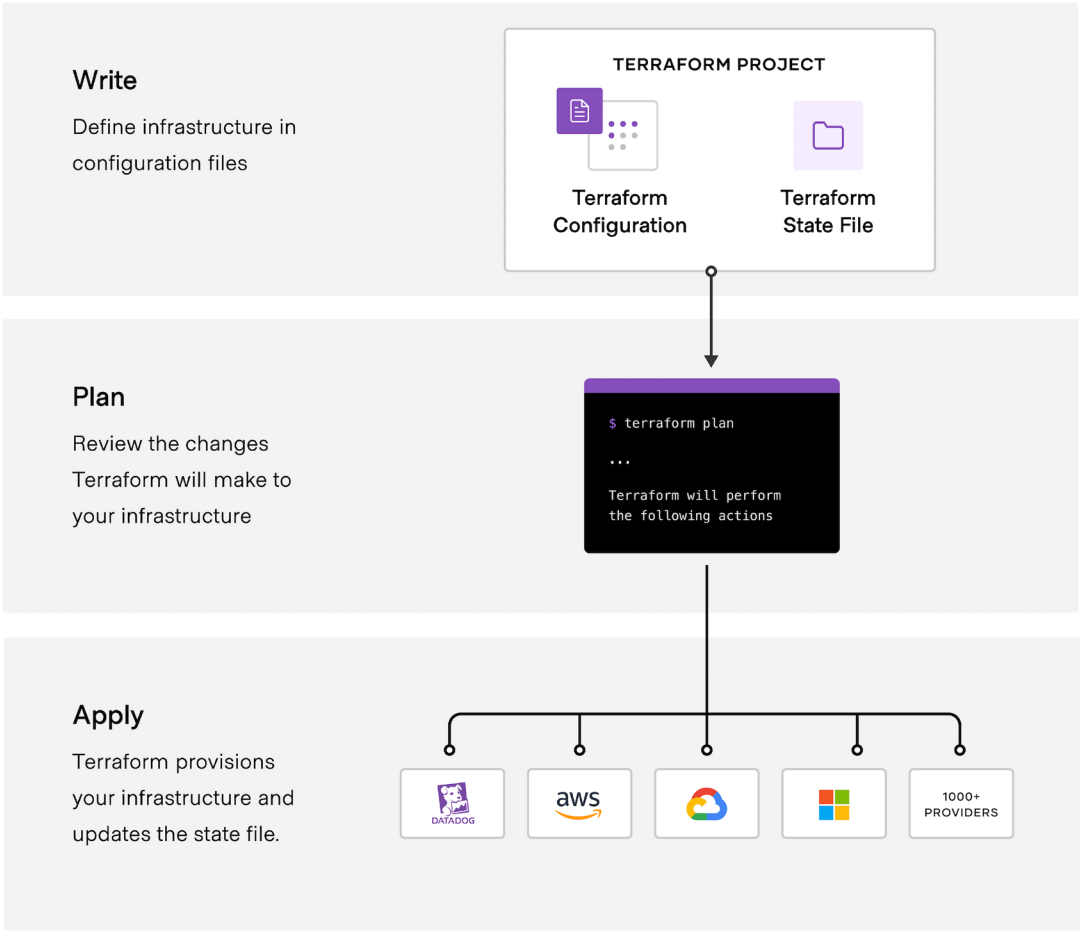


Figure 8.7: Terraform workflow

I have used Terraform extensively to provision infrastructure for large-scale systems on AWS, GCP, and Azure. It can definitely get the job done, but it suffers from several issues:

- Its managed state can get out of sync with the real-world
- Its design and language make it difficult to use at scale
- It can't detect and reconcile external changes to infrastructure automatically

Pulumi

Pulumi is a more modern tool for IaC. Conceptually, it is similar to Terraform, but you can use multiple programming languages to define infrastructure instead of a custom DSL. This gives you a full-fledged ecosystem of languages like TypeScript, Python, or Go, including testing and packaging to manage your infrastructure.

Pulumi also boasts of having dynamic providers that get updated on the same day to support cloud provider resources. It can also wrap Terraform providers to achieve full coverage of your infrastructure needs.

The Pulumi programming model is based on the concepts of stacks, resources, and inputs/outputs:

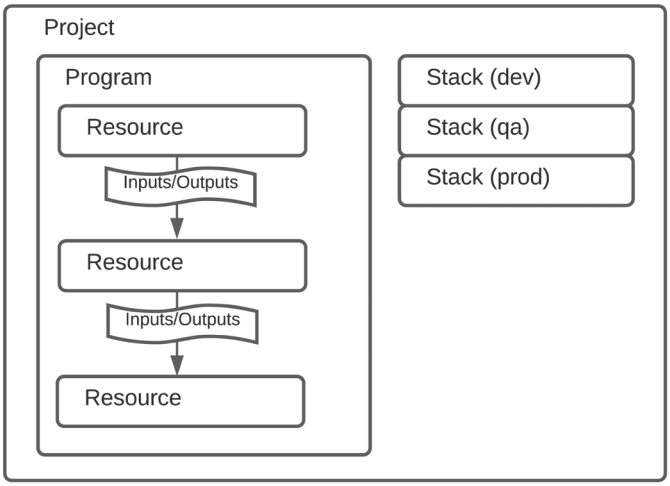


Figure 8.8: Pulumi programming model

Here is a simple example for provisioning an EC2 instance in Python using Pulumi:

```
import pulumi
import pulumi_aws as aws

group = aws.ec2.SecurityGroup('web-sg',
    description='Enable HTTP access',
    ingress=[
        { 'protocol': 'tcp', 'from_port': 80, 'to_port': 80, 'cidr_blocks':
['0.0.0.0/0'] }
    ])

server = aws.ec2.Instance('web-server',
    ami='ami-6869aa05',
    instance_type='t2.micro',
    vpc_security_group_ids=[group.name] # reference the security group resource
above
)

pulumi.export('public_ip', server.public_ip)
pulumi.export('public_dns', server.public_dns)
```

Custom operators

Both Terraform and Pulumi support Kubernetes and can provision clusters, but they are not cloud-native. They also don't allow dynamic reconciliation, which goes against the grain of the Kubernetes model. This means that if someone deletes or modifies some infrastructure provisioned by Terraform or Pulumi, it will not be detected until the next time you run Terraform/Pulumi.

Writing a custom Kubernetes operator gives you full control. You can expose as much of the configuration surface of the target infrastructure as you want and can enforce rules and default configurations. For example, in my current company, we used to manage a large number of Cloudflare DNS domains via Terraform. That caused a significant performance issue as Terraform tried to refresh all these domains by making API calls to Cloudflare for any change to the infrastructure (even unrelated to Cloudflare). We decided to write a custom Kubernetes operator to manage those domains. The operator defined several CRDs to represent zones, domains, and records and interacted with Cloudflare through their APIs.

In addition to the total control and the performance benefits, the operator automatically reconciled any outside changes, to avoid unintentional manual changes.

Using Crossplane

Custom operators are very powerful, but it takes a lot of work to write and maintain an operator. Crossplane (`https://crossplane.io`) styles itself as a control plane for your infrastructure. In practice, it means that you configure everything (providers, certs, resources, and composite resources) via CRDs. Infrastructure credentials like DB connection info are written to Kubernetes secrets, which can be consumed by workloads later. The Crossplane operator watches all the custom resources that define the infrastructure and reconciles them with the infrastructure providers.

Here is an example of defining an AWS RDS PostgresSQL instance:

```
apiVersion: database.example.org/v1alpha1
kind: PostgreSQLInstance
metadata:
  name: the-db
  namespace: data
spec:
  parameters:
    storageGB: 20
  compositionSelector:
    matchLabels:
      provider: aws
      vpc: default
  writeConnectionSecretToRef:
    name: db-conn
```

Crossplane extends kubectl with its own CLI to provide support for building, pushing, and installing packages.

In this section, we covered the concepts behind CI/CD pipelines and different methods to provision infrastructure on Kubernetes.

Summary

In this chapter, we've covered many topics related to deploying and updating applications, scaling Kubernetes clusters, managing resources, CI/CD pipelines, and provisioning infrastructure. We discussed live cluster updates, different deployment models, how the horizontal pod autoscaler can automatically manage the number of running pods, how to perform rolling updates correctly and safely in the context of autoscaling, and how to handle scarce resources via resource quotas. Then we discussed CI/CD pipelines and how to provision infrastructure on Kubernetes using tools like Terraform, Pulumi, custom operators, and Crossplane.

At this point, you have a good understanding of all the factors that come into play when a Kubernetes cluster faces dynamic and growing workloads. You have multiple tools to choose from for planning and designing your own release and scaling strategy.

In the next chapter, we will learn how to package applications for deployment on Kubernetes. We will discuss Helm as well as Kustomize and other solutions.

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`



9

Packaging Applications

In this chapter, we are going to look into Helm, the popular Kubernetes package manager. Every successful and non-trivial platform must have a good packaging system. Helm was developed by Deis (acquired by Microsoft in April 2017) and later contributed to the Kubernetes project directly. It became a CNCF project in 2018. We will start by understanding the motivation for Helm, its architecture, and its components. Then, we'll get hands-on and see how to use Helm and its charts within Kubernetes. That includes finding, installing, customizing, deleting, and managing charts. Last but not least, we'll cover how to create your own charts and handle versioning, dependencies, and templating.

The topics we will cover are as follows:

- Understanding Helm
- Using Helm
- Creating your own charts
- Helm alternatives

Understanding Helm

Kubernetes provides many ways to organize and orchestrate your containers at runtime, but it lacks a higher-level organization of grouping sets of images together. This is where Helm comes in. In this section, we'll go over the motivation for Helm, its architecture, and its components. We will discuss Helm 3. You might still find Helm 2 in the wild, but its end of life was at the end of 2020.

As you might recall, Kubernetes means helmsman or navigator in Greek. The Helm project took the nautical theme very seriously, as the project's name implies. The main Helm concept is the chart. Just as nautical charts describe in detail an area in the sea or a coastal region, a Helm chart describes in detail all the parts of an application.

Helm is designed to perform the following:

- Build charts from the ground up
- Bundle charts into archive files (`.tgz`)
- Interact with repositories containing charts

- Deploy and remove charts in an existing Kubernetes cluster
- Handle the lifecycle of installed charts

The motivation for Helm

Helm provides support for several important use cases:

- Managing complexity
- Easy upgrades
- Simple sharing
- Safe rollbacks

Charts can define even the most intricate applications, offer consistent application installation, and act as a central source of authority. In-place upgrades and custom hooks allow for easy updates. It's simple to share charts that can be versioned and hosted on public or private servers. When you need to roll back recent upgrades, Helm provides a single command to roll back a cohesive set of changes to your infrastructure.

The Helm 3 architecture

The Helm 3 architecture relies fully on client-side tooling and keeps its state as Kubernetes secrets. Helm 3 has several components: release secrets, client, and library.

The client is the command-line interface, and often a CI/CD pipeline is used to package and install applications. The client utilizes the Helm library to perform the requested operations, and the state of each deployed application is stored in a release secret.

Let's review the components.

Helm release secrets

Helm stores its releases as Kubernetes secrets in the target namespace. This means you can have multiple releases with the same name as long they are stored in different namespaces. Here is what a release secret looks like:

```
$ kubectl describe secret sh.helm.release.v1.prometheus.v1 -n monitoring
Name:         sh.helm.release.v1.prometheus.v1
Namespace:    monitoring
Labels:       modifiedAt=1659855458
name=prometheus
owner=helm
status=deployed
version=1
Annotations:  <none>

Type:  helm.sh/release.v1
```

```
Data
====
release:  51716 bytes
```

The data is Base64-encoded twice and then GZIP-compressed.

The Helm client

You install the Helm client on your machine. Helm carries out the following tasks:

- Facilitating local chart development
- Managing repositories
- Overseeing releases
- Interacting with the Helm library for:
    - Deployment of new releases
    - Upgrade of existing releases
    - Removal of existing releases

The Helm library

The Helm library is the component at the heart of Helm and is responsible for performing all the heavy lifting. The Helm library communicates with the Kubernetes API server and provides the following capabilities:

- Combining Helm charts, templates, and values files to build a release
- Installing the releases into Kubernetes
- Creating a release object
- Upgrading and uninstalling charts

Helm 2 vs Helm 3

Helm 2 was great and played a very important role in the Kubernetes ecosystem. But, there was a lot of criticism about Tiller, its server-side component. Helm 2 was designed and implemented before RBAC became the official access-control method. In the interest of usability, Tiller is installed by default with a very open set of permissions. It wasn't easy to lock it down for production usage. This is especially challenging in multi-tenant clusters.

The Helm team listened to the criticisms and came up with the Helm 3 design. Instead of the Tiller in-cluster component, Helm 3 utilizes the Kubernetes API server itself via CRDs to manage the state of releases. The bottom line is that Helm 3 is a client-only program. It can still manage releases and perform the same tasks as Helm 2, but without installing a server-side component.

This approach is more Kubernetes-native and less complicated, and the security concerns are gone. Helm users can perform via Helm only as much as their kube config allows.

Using Helm

Helm is a rich package management system that lets you perform all the necessary steps to manage the applications installed on your cluster. Let's roll up our sleeves and get going. We'll look at installing both Helm 2 and Helm 3, but we will use Helm 3 for all of our hands-on experiments and demonstrations.

Installing Helm

Installing Helm involves installing the client and the server. Helm is implemented in Go. The Helm 2 executable can serve as either the client or server. Helm 3, as mentioned before, is a client-only program.

Installing the Helm client

You must have Kubectl configured properly to talk to your Kubernetes cluster because the Helm client uses the Kubectl configuration to talk to the Kubernetes API server.

Helm provides binary releases for all platforms here: `https://github.com/helm/helm/releases`.

For Windows, the Chocolatey (`https://chocolatey.org`) package manager is the best option (usually up to date):

```
choco install kubernetes-helm
```

For macOS and Linux, you can install the client from a script:

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/
scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

On macOS, you can also use Homebrew (`https://brew.sh`):

```
brew install helm
$ helm version


version.BuildInfo{Version:"v3.9.2",
GitCommit:"1addefbfe665c350f4daf868a9adc5600cc064fd", GitTreeState:"clean",
GoVersion:"go1.18.4"}
```

Finding charts

To install useful applications and software with Helm, you need to find their charts first. Helm was designed to work with multiple repositories of charts. Helm 2 was configured to search the stable repository by default, but you could add additional repositories. Helm 3 comes with no default, but you can search the Helm Hub (`https://artifacthub.io`) or specific repositories. The Helm Hub was launched in December 2018 and it was designed to make it easy to discover charts and repositories hosted outside the stable or incubator repositories.

This is where the `helm search` command comes in. Helm can search the Helm Hub for a specific repository.

The hub contains 9,053 charts at the moment:

```
$ helm search hub | wc -l
     9053
```

We can search the hub for a specific keyword like `mariadb`. Here are the first 10 charts (there are 38):

```
$ helm search hub mariadb --max-col-width 60 | head -n 10
URL                                                 CHART VERSION   APP
VERSION DESCRIPTION
https://artifacthub.io/packages/helm/cloudnativeapp/mariadb     6.1.0
10.3.15     Fast, reliable, scalable, and easy to use open-source rel...
https://artifacthub.io/packages/helm/riftbit/mariadb            9.6.0
10.5.12     Fast, reliable, scalable, and easy to use open-source rel...
https://artifacthub.io/packages/helm/bitnami/mariadb            11.1.6
10.6.8      MariaDB is an open source, community-developed SQL databa...
https://artifacthub.io/packages/helm/bitnami-aks/mariadb        11.1.5
10.6.8      MariaDB is an open source, community-developed SQL databa...
https://artifacthub.io/packages/helm/camptocamp3/mariadb        1.0.0
Fast, reliable, scalable, and easy to use open-source rel...
https://artifacthub.io/packages/helm/openinfradev/mariadb       0.1.1
OpenStack-Helm MariaDB
https://artifacthub.io/packages/helm/sitepilot/mariadb          1.0.3           10.6
MariaDB chart for the Sitepilot platform.
https://artifacthub.io/packages/helm/groundhog2k/mariadb        0.5.0
10.8.3      A Helm chart for MariaDB on Kubernetes
https://artifacthub.io/packages/helm/nicholaswilde/mariadb      1.0.6
110.4.21    The open source relational database
```

As you can see, there are several charts that match the keyword `mariadb`. You can investigate those further and find the best one for your use case.

Adding repositories

By default, Helm 3 comes with no repositories set up, so you can search only the hub. In the past, the `stable` repo hosted by the CNCF was a good option to look for charts. But, CNCF didn't want to pay for hosting it, so now it just contains a lot of deprecated charts.

Instead, you can either install charts from the hub or do some research and add individual repositories. For example, for Prometheus, there is the `prometheus-community` Helm repository. Let's add it:

```
$ helm repo add prometheus-community https://prometheus-community.github.io/helm-
charts
"prometheus-community" has been added to your repositories
```

Now, we can search the `prometheus` repo:

```
$ helm search repo prometheus
```

```
NAME                                                CHART
VERSION   APP VERSION DESCRIPTION
test | default
prometheus-community/kube-prometheus-stack          39.4.1          0.58.0      kube-
prometheus-stack collects Kubernetes manif...
prometheus-community/prometheus                     15.12.0         2.36.2
Prometheus is a monitoring system and time seri...
prometheus-community/prometheus-adapter             3.3.1           v0.9.1      A
Helm chart for k8s prometheus adapter
prometheus-community/prometheus-blackbox-exporter   6.0.0           0.20.0
Prometheus Blackbox Exporter
prometheus-community/prometheus-cloudwatch-expo...  0.19.2          0.14.3      A
Helm chart for prometheus cloudwatch-exporter
prometheus-community/prometheus-conntrack-stats...  0.2.1           v0.3.0      A
Helm chart for conntrack-stats-exporter
prometheus-community/prometheus-consul-exporter     0.5.0           0.4.0       A
Helm chart for the Prometheus Consul Exporter
prometheus-community/prometheus-couchdb-exporter    0.2.0           1.0         A
Helm chart to export the metrics from couchdb...
prometheus-community/prometheus-druid-exporter      0.11.0          v0.8.0      Druid
exporter to monitor druid metrics with Pr...
prometheus-community/prometheus-elasticsearch-e...  4.14.0          1.5.0
Elasticsearch stats exporter for Prometheus
prometheus-community/prometheus-json-exporter       0.2.3           v0.3.0
Install prometheus-json-exporter
prometheus-community/prometheus-kafka-exporter      1.6.0           v1.4.2      A
Helm chart to export the metrics from Kafka i...
prometheus-community/prometheus-mongodb-exporter    3.1.0           0.31.0      A
Prometheus exporter for MongoDB metrics
prometheus-community/prometheus-mysql-exporter      1.9.0           v0.14.0     A
Helm chart for prometheus mysql exporter with...
prometheus-community/prometheus-nats-exporter       2.9.3           0.9.3       A
Helm chart for prometheus-nats-exporter
prometheus-community/prometheus-node-exporter       3.3.1           1.3.1       A
Helm chart for prometheus node-exporter
prometheus-community/prometheus-operator            9.3.2           0.38.1
DEPRECATED - This chart will be renamed. See ht...
prometheus-community/prometheus-pingdom-exporter    2.4.1           20190610-1  A
Helm chart for Prometheus Pingdom Exporter
prometheus-community/prometheus-postgres-exporter   3.1.0           0.10.1      A
Helm chart for prometheus postgres-exporter
prometheus-community/prometheus-pushgateway         1.18.2          1.4.2       A
Helm chart for prometheus pushgateway
```

```
prometheus-community/prometheus-rabbitmq-exporter   1.3.0          v0.29.0
Rabbitmq metrics exporter for prometheus
prometheus-community/prometheus-redis-exporter      5.0.0          1.43.0
Prometheus exporter for Redis metrics
prometheus-community/prometheus-snmp-exporter       1.1.0          0.19.0
Prometheus SNMP Exporter
prometheus-community/prometheus-stackdriver-exp...  4.0.0          0.12.0
Stackdriver exporter for Prometheus
prometheus-community/prometheus-statsd-exporter     0.5.0          0.22.7      A
Helm chart for prometheus stats-exporter
prometheus-community/prometheus-to-sd               0.4.0          0.5.2
Scrape metrics stored in prometheus format and ...
prometheus-community/alertmanager                   0.19.0         v0.23.0     The
Alertmanager handles alerts sent by client ...
prometheus-community/kube-state-metrics             4.15.0         2.5.0
Install kube-state-metrics to generate and expo...
prometheus-community/prom-label-proxy               0.1.0          v0.5.0      A
proxy that enforces a given label in a given ...
```

There are quite a few charts there. To get more information about a specific chart, we can use the `show` command (you can use the `inspectalias` command too). Let's check out `prometheus-community/prometheus`:

```
$ helm show chart prometheus-community/prometheus
apiVersion: v2
appVersion: 2.36.2
dependencies:
- condition: kubeStateMetrics.enabled
  name: kube-state-metrics
  repository: https://prometheus-community.github.io/helm-charts
  version: 4.13.*
description: Prometheus is a monitoring system and time series database.
home: https://prometheus.io/
icon: https://raw.githubusercontent.com/prometheus/prometheus.github.io/master/
assets/prometheus_logo-cb55bb5c346.png
maintainers:
- email: gianrubio@gmail.com
  name: gianrubio
- email: zanhsieh@gmail.com
  name: zanhsieh
- email: miroslav.hadzhiev@gmail.com
  name: Xtigyro
- email: naseem@transit.app
```

```
  name: naseemkullah
name: prometheus
sources:
- https://github.com/prometheus/alertmanager
- https://github.com/prometheus/prometheus
- https://github.com/prometheus/pushgateway
- https://github.com/prometheus/node_exporter
- https://github.com/kubernetes/kube-state-metrics
type: application
version: 15.12.0
```

You can also ask Helm to show you the `README` file, the values, or all the information associated with a chart. This can be overwhelming at times.

Installing packages

OK. You've found the package of your dreams. Now, you probably want to install it on your Kubernetes cluster. When you install a package, Helm creates a release that you can use to keep track of the installation progress. We install `prometheus` using the `helm install` command in the monitoring namespace and instruct Helm to create the namespace for us:

```
$ helm install prometheus prometheus-community/prometheus -n monitoring --create-
namespace
```

Let's go over the output. The first part of the output lists the name of the release that we provided, `prometheus`, when it was deployed, the namespace, and the revision:

```
NAME: prometheus
LAST DEPLOYED: Sat Aug  6 23:54:50 2022
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

The next part is custom notes, which can be pretty wordy. There is a lot of good information here about how to connect to the Prometheus server, the alert manager, and the `Pushgateway`:

```
NOTES:
The Prometheus server can be accessed via port 80 on the following DNS name from
within your cluster:
prometheus-server.default.svc.cluster.local


Get the Prometheus server URL by running these commands in the same shell:
test | default
```

```
  export POD_NAME=$(kubectl get pods --namespace default -l
"app=prometheus,component=server" -o jsonpath="{.items[0].metadata.name}")
  kubectl --namespace default port-forward $POD_NAME 9090


The Prometheus alertmanager can be accessed via port 80 on the following DNS name
from within your cluster:
prometheus-alertmanager.default.svc.cluster.local


Get the Alertmanager URL by running these commands in the same shell:
  export POD_NAME=$(kubectl get pods --namespace default -l
"app=prometheus,component=alertmanager" -o jsonpath="{.items[0].metadata.name}")
  kubectl --namespace default port-forward $POD_NAME 9093
#################################################################################
######   WARNING: Pod Security Policy has been moved to a global property.  #####
######            use .Values.podSecurityPolicy.enabled with pod-based      #####
######            annotations                                               #####
######            (e.g. .Values.nodeExporter.podSecurityPolicy.annotations) #####
#################################################################################


The Prometheus PushGateway can be accessed via port 9091 on the following DNS name
from within your cluster:
prometheus-pushgateway.default.svc.cluster.local


Get the PushGateway URL by running these commands in the same shell:
  export POD_NAME=$(kubectl get pods --namespace default -l
"app=prometheus,component=pushgateway" -o jsonpath="{.items[0].metadata.name}")
  kubectl --namespace default port-forward $POD_NAME 9091

For more information on running Prometheus, visit:
https://prometheus.io/
```

Checking the installation status

Helm doesn't wait for the installation to complete because it may take a while. The `helm status` command displays the latest information on a release in the same format as the output of the initial `helm install` command.

If you just care about the status without all the extra information, you can just `grep` for the `STATUS` line:

```
$ helm status -n monitoring prometheus | grep STATUS
STATUS: deployed
```

Let's list all the Helm releases in the `monitoring` namespace and verify that `prometheus` is listed:

```
$ helm list -n monitoring
NAME        NAMESPACE   REVISION    UPDATED                                 STATUS
CHART               APP VERSION
prometheus  monitoring  1           2022-08-06 23:57:34.124225 -0700 PDT    deployed
prometheus-15.12.0  2.36.2
```

As you recall, Helm stores the release information in a secret:

```
$ kubectl describe secret sh.helm.release.v1.prometheus.v1 -n monitoring
Name:         sh.helm.release.v1.prometheus.v1
Namespace:    monitoring
Labels:       modifiedAt=1659855458
              name=prometheus
              owner=helm
              status=deployed
              version=1
Annotations:  <none>

Type:  helm.sh/release.v1

Data
====
release:  51716 bytes
```

If you want to find all the Helm releases across all namespaces, use:

```
$ helm list -A
```

If you want to go low-level, you can list all the secrets that have the `owner=helm` label:

```
$ kubectl get secret -A -l owner=helm
```

To actually extract the release data from a secret, you need to jump through some hoops as it is Base64-encoded twice (why?) and GZIP-compressed. The final result is JSON:

```
kubectl get secret sh.helm.release.v1.prometheus.v1 -n monitoring -o jsonpath='{.
data.release}' | base64 --decode | base64 --decode | gunzip > prometheus.v1.json
```

You may also be interested in extracting just the manifests using the following command:

```
kubectl get secret sh.helm.release.v1.prometheus.v1 -n monitoring -o jsonpath='{.
data.release}' | base64 --decode | base64 --decode | gunzip | jq .manifest -r
```

Customizing a chart

Very often as a user, you want to customize or configure the charts you install. Helm fully supports customization via config files. To learn about possible customizations, you can use the `helm show` command again, but this time, focus on the values. For a complex project like Prometheus, the `values` file can be pretty large:

```
$ helm show values prometheus-community/prometheus | wc -l
    1901
```

Here is a partial output:

```
$ helm show values prometheus-community/prometheus | head -n 20
rbac:
  create: true

podSecurityPolicy:
  enabled: false

imagePullSecrets:
# - name: "image-pull-secret"

## Define serviceAccount names for components. Defaults to component's fully
qualified name.
##
serviceAccounts:
  alertmanager:
    create: true
    name:
    annotations: {}
  nodeExporter:
    create: true
    name:
    annotations: {}
```

Commented-out lines often contain default values like the name of the `imagePullSecrets`:

```
imagePullSecrets:
# - name: "image-pull-secret"
```

If you want to customize any part of the Prometheus installation, then save the values to a file, make any modifications you like, and then install Prometheus using the custom values file:

```
$ helm install prometheus prometheus-community/prometheus --create-namespace -n
monitoring -f custom-values.yaml
```

You can also set individual values on the command line with `--set`. If both `-f` and `--set` try to set the same values, then `--set` takes precedence. You can specify multiple values using comma-separated lists: `--set a=1,b=2`. Nested values can be set as `--set outer.inner=value`.

Additional installation options

The `helm install` command can work with a variety of sources:

- A chart repository (as demonstrated)
- A local chart archive (`helm install foo-0.1.1.tgz`)
- An extracted chart directory (`helm install path/to/foo`)
- A complete URL (`helm install https://example.com/charts/foo-1.2.3.tgz`)

Upgrading and rolling back a release

You may want to upgrade a package you installed to the latest and greatest version. Helm provides the `upgrade` command, which operates intelligently and only updates things that have changed. For example, let's check the current values of our `prometheus` installation:

```
$ helm get values prometheus -n monitoring
USER-SUPPLIED VALUES:
null
```

So far, we haven't provided any user values. As part of the default installation, `prometheus` installed an alert manager component:

```
$ k get deploy prometheus-alertmanager -n monitoring
NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
prometheus-alertmanager    1/1     1            1           19h
```

Let's disable the alert manager by upgrading and passing a new value:

```
$ helm upgrade --set alertmanager.enabled=false \
     prometheus prometheus-community/prometheus \
     -n monitoring
Release "prometheus" has been upgraded. Happy Helming!
NAME: prometheus
LAST DEPLOYED: Sun Aug  7 19:55:52 2022
NAMESPACE: monitoring
STATUS: deployed
REVISION: 2
TEST SUITE: None
NOTES:
...
```

The upgrade completed successfully. We can see that the output doesn't mention how to get the URL of the alert manager anymore. Let's verify that the alert manager deployment was removed:

```
$ k get deployment -n monitoring
NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
prometheus-kube-state-metrics   1/1     1            1           20h
prometheus-pushgateway          1/1     1            1           20h
prometheus-server               1/1     1            1           20h
```

Now, if we check the custom values, we can see our modification:

```
$ helm get values prometheus -n monitoring
USER-SUPPLIED VALUES:
alertmanager:
  enabled: false
```

Suppose we decide that alerts are kind of important and, actually, we want to have the Prometheus alert manager. No problem, we can roll back to our original installation. The `helm history` command shows us all the available revisions we can roll back to:

```
$ helm history prometheus -n monitoring
REVISION    UPDATED                     STATUS      CHART               APP VERSION
DESCRIPTION
1           Sat Aug  6 23:57:34 2022    superseded  prometheus-15.12.0  2.36.2
Install complete
2           Sun Aug  7 19:55:52 2022    deployed    prometheus-15.12.0  2.36.2
Upgrade complete
```

Let's roll back to revision 1:

```
$ helm rollback prometheus 1 -n monitoring
Rollback was a success! Happy Helming!

$ helm history prometheus -n monitoring
REVISION    UPDATED                     STATUS      CHART           APP VERSION
DESCRIPTION
REVISION    UPDATED                     STATUS      CHART               APP VERSION
DESCRIPTION
1           Sat Aug  6 23:57:34 2022    superseded  prometheus-15.12.0  2.36.2
Install complete
2           Sun Aug  7 19:55:52 2022    superseded  prometheus-15.12.0  2.36.2
Upgrade complete
3           Sun Aug  7 20:02:30 2022    deployed    prometheus-15.12.0  2.36.2
Rollback to 1
```

As you can see, the rollback actually created a new revision number 3. Revision 2 is still there in case we want to go back to it.

Let's verify that our changes were rolled back:

```
$ k get deployment -n monitoring
NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
prometheus-alertmanager         1/1     1            1           152m
prometheus-kube-state-metrics   1/1     1            1           22h
prometheus-pushgateway          1/1     1            1           22h
prometheus-server               1/1     1            1           22h
```

Yep. The alert manager is back.

Deleting a release

You can, of course, uninstall a release too by using the `helm uninstall` command.

First, let's examine the list of releases. We have only the `prometheus` release in the monitoring namespace:

```
$ helm list -n monitoring
NAME        NAMESPACE   REVISION    UPDATED                                 STATUS
CHART               APP VERSION
prometheus  monitoring  3           2022-08-07 20:02:30.270229 -0700 PDT    deployed
prometheus-15.12.0  2.36.2
```

Now, let's uninstall it. You can use any of the following equivalent commands:

- `uninstall`
- `un`
- `delete`
- `del`

Here we are using the `uninstall` command:

```
$ helm uninstall prometheus -n monitoring
release "prometheus" uninstalled
```

With that, there are no more releases:

```
$ helm list -n monitoring
NAME    NAMESPACE   REVISION    UPDATED STATUS  CHART   APP VERSION
```

Helm can keep track of uninstalled releases too. If you provide `--keep-history` when you uninstall, then you'll be able to see uninstalled releases by adding the `--all` or `--uninstalled` flags to `helm list`.

Note that the monitoring namespace remained even though it was created by Helm as part of installing Prometheus, but it is empty now:

```
$ k get all -n monitoring
No resources found in monitoring namespace.
```

Working with repositories

Helm stores charts in repositories that are simple HTTP servers. Any standard HTTP server can host a Helm repository. In the cloud, the Helm team verified that AWS S3 and Google Cloud Storage can both serve as Helm repositories in web-enabled mode. You can even store Helm repositories on GitHub pages.

Note that Helm doesn't provide tools for uploading charts to remote repositories because that would require the remote server to understand Helm, know where to put the chart, and know how to update the `index.yaml` file.



Note that Helm recently added experimental support for storing Helm charts in OCI registries. Check out `https://helm.sh/docs/topics/registries/` for more details.

On the client side, the `helm repo` command lets you list, add, remove, index, and update:

```
$ helm repo

This command consists of multiple subcommands to interact with chart repositories.

It can be used to add, remove, list, and index chart repositories.

Usage:
  helm repo [command]

Available Commands:
  add         add a chart repository
  index       generate an index file given a directory containing packaged charts
  list        list chart repositories
  remove      remove one or more chart repositories
  update      update information of available charts locally from chart repositories
```

We already used the `helm repo add` and `helm repo list` commands earlier. Let's see how to create our own charts and manage them.

Managing charts with Helm

Helm provides several commands to manage charts.

It can create a new chart for you:

```
$ helm create cool-chart
Creating cool-chart
```

Helm will create the following files and directories under `cool-chart`:

```
$ tree cool-chart
cool-chart
├── Chart.yaml
├── charts
├── templates
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── service.yaml
│   ├── serviceaccount.yaml
│   └── tests
│       └── test-connection.yaml
```

Once you have edited your chart, you can package it into a `tar.gz` archive:

```
    $ helm package cool-chart
Successfully packaged chart and saved it to: cool-chart-0.1.0.tgz
```

Helm will create an archive called `cool-chart-0.1.0.tgz` and store it in the local directory.

You can also use `helm lint` to help you find issues with your chart's formatting or information:

```
$ helm lint cool-chart
==> Linting cool-chart
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

Taking advantage of starter packs

The `helm create` command offers an optional `--starter` flag, allowing you to specify a starter chart. Starters are regular charts located in `$XDG_DATA_HOME/helm/starters`. As a chart developer, you can create charts explicitly intended to serve as starter templates for creating new charts. When developing such charts, please keep the following considerations in mind:

- The YAML content within a starter chart will be overwritten by the generator.
- Users will typically modify the contents of a starter chart, so it is crucial to provide clear documentation explaining how users can make modifications.

Presently, there is no built-in mechanism for installing starter charts. The only way to add a chart to `$XDG_DATA_HOME/helm/starters` is through manual copying. If you create starter pack charts, ensure that your chart's documentation explicitly mentions this requirement.

Creating your own charts

A chart represents a group of files that define a cohesive set of Kubernetes resources. It can range from a simple deployment of a Memcached pod to a complex configuration of a complete web application stack, including HTTP servers, databases, caches, queues, and more.

To organize a chart, its files are structured within a specific directory tree. These files can then be bundled into versioned archives, which can be easily deployed and managed. The key file is `Chart.yaml`.

The Chart.yaml file

The `Chart.yaml` file is the main file of a Helm chart. It requires a name and version fields:

- `apiVersion`: The API version of the chart.
- `name`: The name of the chart, which should match the directory name.
- `version`: The version of the chart using the `SemVer` 2 format.

Additionally, there are several optional fields that can be included in the `Chart.yaml` file:

- `kubeVersion`: A range of compatible Kubernetes versions specified in `SemVer` format.
- `description`: A brief description of the project in a single sentence.
- `keywords`: A list of keywords associated with the project.
- `home`: The URL of the project's homepage.
- `sources`: A list of URLs to the project's source code.
- `dependencies`: A list of dependencies for the chart, including the name, version, repository, condition, tags, and alias.
- `maintainers`: A list of maintainers for the chart, including the name, email, and URL.
- `icon`: The URL to an SVG or PNG image that can be used as an icon.
- `appVersion`: The version of the application contained within the chart. It does not have to follow `SemVer`.
- `deprecated`: A boolean value indicating whether the chart is deprecated.
- `annotations`: Additional key-value pairs that provide extra information.

Versioning charts

The `version` field in the `Chart.yaml` file plays a crucial role for various Helm tools. It is used by the `helm package` command when creating a package, as it constructs the package name based on the version specified in the `Chart.yaml`. It is important to ensure that the version number in the package name matches the version number in the `Chart.yaml` file. Deviating from this expectation can result in an error, as the system assumes the consistency of these version numbers. Therefore, it is essential to maintain the coherence between the version field in the `Chart.yaml` file and the generated package name to avoid any issues.

The appVersion field

The optional `appVersion` field is not related to the `version` field. It is not used by Helm and serves as metadata or documentation for users that want to understand what they are deploying. Helm ignores it.

Deprecating charts

From time to time, you may want to deprecate a chart. You can mark a chart as deprecated by setting the optional `deprecated` field in `Chart.yaml` to `true`. It's enough to deprecate the latest version of a chart. You can later reuse the chart name and publish a newer version that is not deprecated. The workflow for deprecating charts typically involves the following steps:

1. **Update the Chart.yaml file**: Modify the `Chart.yaml` file of the chart to indicate that it is deprecated. This can be done by adding a `deprecated` field and setting it to `true`. Additionally, it is common practice to bump the version of the chart to indicate that a new version with deprecation information has been released.
2. **Release the new version**: Package and release the updated chart with the deprecation information to the chart repository. This ensures that users are aware of the deprecation when they try to install or upgrade the chart.
3. **Communicate the deprecation**: It is important to communicate the deprecation to users and provide information on alternative options or recommended migration paths. This can be done through documentation, release notes, or other channels to ensure that users are informed about the deprecation and can plan accordingly.
4. **Remove the chart from the source repository**: Once the deprecated chart has been released and communicated to users, it is recommended to remove the chart from the source repository, such as a Git repository, to avoid confusion and ensure that users are directed to the latest version in the chart repository.

By following these steps, you can effectively deprecate a chart and provide a clear process for users to transition to newer versions or alternative solutions.

Chart metadata files

Charts can include several metadata files, such as `README.md`, `LICENSE`, and `NOTES.txt`, which provide important information about the chart. The `README.md` file, formatted as markdown, is particularly crucial and should contain the following details:

- **Application or service description**: Provide a clear and concise description of the application or service that the chart represents. This description should help users understand the purpose and functionality of the chart.
- **Prerequisites and requirements**: Specify any prerequisites or requirements that need to be met before using the chart. This could include specific versions of Kubernetes, required dependencies, or other conditions that must be satisfied.
- **YAML options and default values**: Document the available options that users can configure in the chart's YAML files. Describe each option, its purpose, accepted values or format, and the default values. This information empowers users to customize the chart according to their needs.
- **Installation and configuration instructions**: Provide clear instructions on how to install and configure the chart. This may involve specifying the command-line options or Helm commands to deploy the chart and any additional steps or considerations during the configuration process.

- **Additional information**: Include any other relevant information that may assist users during the installation or configuration of the chart. This could involve best practices, troubleshooting tips, or known limitations.

By including these details in the `README.md` file, chart users can easily understand the chart's purpose, requirements, and how to effectively install and configure it for their specific use case.

If the chart contains a template or `NOTES.txt` file, then the file will be displayed, printed out after installation and when viewing the release status, or upgraded. The notes should be concise to avoid clutter and point to the `README.md` file for detailed explanations. It's common to put usage notes and next steps in `NOTES.txt`. Remember that the file is evaluated as a template. The notes are printed on the screen when you run `helm install` as well as `helm status`.

Managing chart dependencies

In Helm, a chart may depend on other charts. These dependencies are expressed explicitly by listing them in the `dependencies` field of the `Chart.yaml` file or copied directly to the `charts/` subdirectory. This provides a great way to benefit from and reuse the knowledge and work of others. A dependency in Helm can take the form of either a chart archive (e.g., `foo-1.2.3.tgz`) or an unpacked chart directory. However, it's important to note that the name of a dependency should not begin with an underscore (_) or a period (`.`), as these files are ignored by the chart loader. Therefore, it's recommended to avoid starting dependency names with these characters to ensure they are properly recognized and loaded by Helm.

Let's add `kube-state-metrics` from the `prometheus-community` repo as a dependency to our `cool-chart`'s `Chart.yaml` file:

```
dependencies:
  - name: kube-state-metrics
    version: "4.13.*"
    repository: https://prometheus-community.github.io/helm-charts
    condition: kubeStateMetrics.enabled
```

The `name` field represents the desired name of the chart you want to install. It should match the name of the chart as it is defined in the repository.

The `version` field specifies the specific version of the chart you want to install. It helps to ensure that you get the desired version of the chart.

The `repository` field contains the complete URL of the chart repository that the chart will be fetched from. It points to the location where the chart and its versions are stored and can be accessed.

The `condition` field is discussed in the subsequent section.

If the repository is not added yet use `helm repo` to add it locally.

Once your dependencies are defined, you can run the `helm dependency update` command. Helm will download all of the specified charts into the charts subdirectory for you:

```
$ helm dep up cool-chart
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. ⎈Happy Helming!⎈
Saving 1 charts
Downloading kube-state-metrics from repo https://prometheus-community.github.io/helm-charts
Deleting outdated charts
```

Helm stores the dependency charts as archives in the `charts/` directory:

```
$ ls cool-chart/charts
kube-state-metrics-4.13.0.tgz
```

Managing charts and their dependencies in the `Chart.yaml` dependencies field (as opposed to just copying charts into the `charts/` subdirectory) is a best practice. It explicitly documents dependencies, facilitates sharing across the team, and supports automated pipelines.

Utilizing additional subfields of the dependencies field

Each entry in the `requirements.yaml` file's `requirements` entry may include optional fields such as `tags` and `condition`.

These fields can be used to dynamically control the loading of charts (if not specified all charts will be loaded). If `tags` or `condition` fields are present, Helm will evaluate them and determine if the target chart should be loaded or not.

- The `condition` field in chart dependencies holds one or more comma-delimited YAML paths. These paths refer to values in the top parent's values file. If a path exists and evaluates to a Boolean value, it determines whether the chart will be enabled or disabled. If multiple paths are provided, only the first valid path encountered is evaluated. If no paths exist, the condition has no effect, and the chart will be loaded regardless.
- The `tags` field allows you to associate labels with the chart. It is a YAML list where you can specify one or more tags. In the top parent's values file, you can enable or disable all charts with specific tags by specifying the tag and a corresponding Boolean value. This provides a convenient way to manage and control charts based on their associated tags.

Here is an example `dependencies` field and a `values.yaml` that makes good use of conditions and tags to enable and disable the installation of dependencies. The `dependencies` field defines two conditions for installing its dependencies based on the value of the global enabled field and the specific subchart's enabled field:

```yaml
dependencies:
  - name: subchart1
    repository: http://localhost:10191
```

```
    version: 0.1.0
    condition: subchart1.enabled, global.subchart1.enabled
    tags:
      - front-end
      - subchart1
  - name: subchart2
    repository: http://localhost:10191
    version: 0.1.0
    condition: subchart2.enabled,global.subchart2.enabled
    tags:
      - back-end
      - subchart2
```

The `values.yaml` file assigns values to some of the `condition` variables. The `subchart2` tag doesn't get a value, so it is enabled automatically:

```
# parentchart/values.yaml
subchart1:
  enabled: true
tags:
  front-end: false
  back-end: true
```

You can set `tags` and `condition` values from the command line too when installing a chart, and they'll take precedence over the `values.yaml` file:

```
$ helm install --set subchart2.enabled=false
```

The resolution of tags and conditions is as follows:

- Conditions that are set in values override tags. The first `condition` path that exists per chart takes effect, and other conditions are ignored.
- If any of the tags associated with a chart are set to `true` in the top parent's values, the chart is considered enabled.
- The `tags` and `condition` values must be set at the top level of the values file.
- Nested tags tables or tags within global configurations are not currently supported. This means that the tags should be directly under the top parent's values and not nested within other structures.

Using templates and values

Any non-trivial application will require configuration and adaptation to the specific use case. Helm charts are templates that use the Go template language to populate placeholders. Helm supports additional functions from the Sprig library, which contains a lot of useful helpers as well as several other specialized functions. The template files are stored in the `templates/` subdirectory of the chart. Helm will use the template engine to render all files in this directory and apply the provided value files.

Writing template files

Template files are just text files that follow the Go template language rules. They can generate Kubernetes configuration files as well as any other file. Here is the service template file of the Prometheus server's `service.yaml` template from the `prometheus-community` repo:

```
{{- if and .Values.server.enabled .Values.server.service.enabled -}}
apiVersion: v1
kind: Service
metadata:
{{- if .Values.server.service.annotations }}
  annotations:
{{ toYaml .Values.server.service.annotations | indent 4 }}
{{- end }}
  labels:
    {{- include "prometheus.server.labels" . | nindent 4 }}
{{- if .Values.server.service.labels }}
{{ toYaml .Values.server.service.labels | indent 4 }}
{{- end }}
  name: {{ template "prometheus.server.fullname" . }}
{{ include "prometheus.namespace" . | indent 2 }}
spec:
{{- if .Values.server.service.clusterIP }}
  clusterIP: {{ .Values.server.service.clusterIP }}
{{- end }}
{{- if .Values.server.service.externalIPs }}
  externalIPs:
{{ toYaml .Values.server.service.externalIPs | indent 4 }}
{{- end }}
{{- if .Values.server.service.loadBalancerIP }}
  loadBalancerIP: {{ .Values.server.service.loadBalancerIP }}
{{- end }}
{{- if .Values.server.service.loadBalancerSourceRanges }}
  loadBalancerSourceRanges:
  {{- range $cidr := .Values.server.service.loadBalancerSourceRanges }}
    - {{ $cidr }}
  {{- end }}
{{- end }}
  ports:
    - name: http
      port: {{ .Values.server.service.servicePort }}
      protocol: TCP
```

```
      targetPort: 9090
    {{- if .Values.server.service.nodePort }}
      nodePort: {{ .Values.server.service.nodePort }}
    {{- end }}
    {{- if .Values.server.service.gRPC.enabled }}
    - name: grpc
      port: {{ .Values.server.service.gRPC.servicePort }}
      protocol: TCP
      targetPort: 10901
    {{- if .Values.server.service.gRPC.nodePort }}
      nodePort: {{ .Values.server.service.gRPC.nodePort }}
    {{- end }}
    {{- end }}
  selector:
  {{- if and .Values.server.statefulSet.enabled .Values.server.service.
statefulsetReplica.enabled }}
    statefulset.kubernetes.io/pod-name: {{ template "prometheus.server.
fullname" . }}-{{ .Values.server.service.statefulsetReplica.replica }}
  {{- else -}}
    {{- include "prometheus.server.matchLabels" . | nindent 4 }}
{{- if .Values.server.service.sessionAffinity }}
  sessionAffinity: {{ .Values.server.service.sessionAffinity }}
{{- end }}
  {{- end }}
  type: "{{ .Values.server.service.type }}"
{{- end -}}
```

It is available here: `https://github.com/prometheus-community/helm-charts/blob/main/charts/prometheus/templates/service.yaml`.

Don't worry if it looks confusing. The basic idea is that you have a simple text file with placeholders for values that can be populated later in various ways as well as conditions, some functions, and pipelines that can be applied to those values.

Using pipelines and functions

Helm allows rich and sophisticated syntax in the template files via the built-in Go template functions, Sprig functions, and pipelines. Here is an example template that takes advantage of these capabilities. It uses the `repeat`, `quote`, and `upper` functions for the `food` and `drink` keys, and it uses pipelines to chain multiple functions together:

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
    name: {{ .Release.Name }}-configmap
  data:
    greeting: "Hello World"
    drink: {{ .Values.favorite.drink | repeat 3 | quote }}
    food: {{ .Values.favorite.food | upper }}
```

Let's add a `values.yaml` file:

```
  favorite:
    drink: coffee
    food: pizza
```

Testing and troubleshooting your charts

Now, we can use the `helm template` command to see the result:

```
$ helm template food food-chart
---
# Source: food-chart/templates/config-map.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: food-configmap
data:
  greeting: "Hello World"
  drink: "coffeecoffeecoffee"
  food: PIZZA
```

As you can see, our templating worked. The drink `coffee` was repeated 3 times and quoted. The food `pizza` became uppercase `PIZZA` (unquoted).

Another good way of debugging is to run `install` with the `--dry-run` flag. It provides additional information:

```
$ helm install food food-chart --dry-run -n monitoring
NAME: food
LAST DEPLOYED: Mon Aug  8 00:24:03 2022
NAMESPACE: monitoring
STATUS: pending-install
REVISION: 1
TEST SUITE: None
HOOKS:
MANIFEST:
---
# Source: food-chart/templates/config-map.yaml
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: food-configmap
data:
  greeting: "Hello World"
  drink: "coffeecoffeecoffee"
  food: PIZZA
```

You can also override values on the command line:

```
$ helm template food food-chart --set favorite.drink=water
---
# Source: food-chart/templates/config-map.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: food-configmap
data:
  greeting: "Hello World"
  drink: "waterwaterwater"
  food: PIZZA
```

The ultimate test is, of course, to install your chart into your cluster. You don't need to upload your chart to a chart repository for testing; just run `helm install` locally:

```
$ helm install food food-chart -n monitoring
NAME: food
LAST DEPLOYED: Mon Aug  8 00:25:53 2022
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

There is now a Helm release called `food`:

```
$ helm list -n monitoring
NAME    NAMESPACE   REVISION    UPDATED                                 STATUS
CHART               APP VERSION
food    monitoring  1           2022-08-08 00:25:53.587342 -0700 PDT    deployed
food-chart-0.1.0    1.16.0
```

Most importantly, the `food-configmap` config map was created with the correct data:

```
$ k get cm food-configmap -o yaml -n monitoring
apiVersion: v1
data:
```

```
  drink: coffeecoffeecoffee
  food: PIZZA
  greeting: Hello World
kind: ConfigMap
metadata:
  annotations:
    meta.helm.sh/release-name: food
    meta.helm.sh/release-namespace: monitoring
  creationTimestamp: "2022-08-08T07:25:54Z"
  labels:
    app.kubernetes.io/managed-by: Helm
  name: food-configmap
  namespace: monitoring
  resourceVersion: "4247163"
  uid: ada4957d-bd6d-4c2e-8b2c-1499ca74a3c3
```

Embedding built-in objects

Helm provides some built-in objects you can use in your templates. In the Prometheus chart template above, `Release.Name`, `Release.Service`, `Chart.Name`, and `Chart.Version` are examples of Helm predefined values. Other objects are:

- `Values`
- `Chart`
- `Template`
- `Files`
- `Capabilities`

The `Values` object contains all the values defined in the `values` file or on the command line. The `Chart` object is the content of `Chart.yaml`. The `Template` object contains information about the current template. `Files` and `Capabilities` are map-like objects that allow access via various functions to the non-specialized files and general information about the Kubernetes cluster.

Note that unknown fields in `Chart.yaml` are ignored by the template engine and cannot be used to pass arbitrary structured data to templates.

Feeding values from a file

Here is part of the Prometheus server's default `values` file. The values from this file are used to populate multiple templates. The values represent defaults that you can override by copying the file and modifying it to fit your needs. Note the useful comments that explain the purpose and various options for each value:

```
server:
  ## Prometheus server container name
  ##
```

```
  enabled: true

  ## Use a ClusterRole (and ClusterRoleBinding)
  ## - If set to false - we define a RoleBinding in the defined namespaces ONLY
  ##
  ## NB: because we need a Role with nonResourceURL's ("/metrics") - you must get
someone with Cluster-admin privileges to define this role for you, before running
with this setting enabled.
  ##     This makes prometheus work - for users who do not have ClusterAdmin privs,
but wants prometheus to operate on their own namespaces, instead of clusterwide.
  ##
  ## You MUST also set namespaces to the ones you have access to and want monitored
by Prometheus.
  ##
  # useExistingClusterRoleName: nameofclusterrole

  ## namespaces to monitor (instead of monitoring all - clusterwide). Needed if you
want to run without Cluster-admin privileges.
  # namespaces:
  #   - yournamespace

  name: server

  # sidecarContainers - add more containers to prometheus server
  # Key/Value where Key is the sidecar `- name: <Key>`
  # Example:
  #   sidecarContainers:
  #     webserver:
  #       image: nginx
  sidecarContainers: {}
```

That was a deep dive into creating your own charts with Helm. Well, Helm is used widely and extensively to package and deploy Kubernetes applications. However, Helm is not the only game in town. There are several good alternatives that you may prefer. In the next section, we will review some of the most promising Helm alternatives.

Helm alternatives

Helm is battle tested and very common in the Kubernetes world, but it has its downsides and critics, especially when you develop your own charts. A lot of the criticism was about Helm 2 and its server-side component, Tiller. However, Helm 3 is not a panacea either. On a large scale, when you develop your own charts and complex templates with lots of conditional logic and massive `values` files, it can become very challenging to manage.

If you feel the pain, you may want to investigate some alternatives. Note that most of these projects focus on the deployment aspect. Helm's dependency management is still a strength. Let's look at some interesting projects that you may want to consider.

Kustomize

Kustomize is an alternative to YAML templating by using the concept of overlays on top of raw YAML files. It was added to kubectl in Kubernetes 1.14.

See `https://github.com/kubernetes-sigs/kustomize`.

Cue

Cue is a very interesting project. Its data validation language and inference were strongly inspired by logic programming. It is not a general-purpose programming language. It is focused on data validation, data templating, configuration, querying, and code generation, but has some scripting too. The main concept of Cue is the unification of types and data. That gives Cue a lot of expressive power and obviates the need for constructs like enums and generics.

See `https://cuelang.org`.

See the specific discussion about replacing Helm with Cue here: `https://github.com/cue-lang/cue/discussions/1159`.

kapp-controller

kapp-controller provides continuous delivery and package management capabilities for Kubernetes.

Its declarative APIs and layered approach allow you to build, deploy, and manage your applications effectively. With Kapp-controller, you can package your software into distributable packages and empower users to discover, configure, and install these packages on a Kubernetes cluster seamlessly.

See `https://carvel.dev/kapp-controller/`.

That concludes our quick review of Helm alternatives.

Summary

In this chapter, we took a look at Helm, a popular Kubernetes package manager. Helm gives Kubernetes the ability to manage complicated software composed of many Kubernetes resources with inter-dependencies. It serves the same purpose as an OS package manager. It organizes packages and lets you search charts, install and upgrade charts, and share charts with collaborators. You can develop your own charts and store them in repositories. Helm 3 is a client-side-only solution that uses Kubernetes secrets to manage the state of releases in your cluster. We also looked at some Helm alternatives.

At this point, you should understand the important role that Helm serves in the Kubernetes ecosystem and community. You should be able to use it productively and even develop and share your own charts.

In the next chapter, we will look at how Kubernetes does networking at a pretty low level.

10

Exploring Kubernetes Networking

In this chapter, we will examine the important topic of networking. Kubernetes as an orchestration platform manages containers/pods running on different machines (physical or virtual) and requires an explicit networking model. We will look at the following topics:

- Understanding the Kubernetes networking model
- Kubernetes network plugins
- Kubernetes and eBPF
- Kubernetes networking solutions
- Using network policies effectively
- Load balancing options

By the end of this chapter, you will understand the Kubernetes approach to networking and be familiar with the solution space for aspects such as standard interfaces, networking implementations, and load balancing. You will even be able to write your very own **Container Networking Interface** (**CNI**) plugin if you wish.

Understanding the Kubernetes networking model

The Kubernetes networking model is based on a flat address space. All pods in a cluster can directly see each other. Each pod has its own IP address. There is no need to configure any **Network Address Translation** (**NAT**). In addition, containers in the same pod share their pod's IP address and can communicate with each other through `localhost`. This model is pretty opinionated, but once set up, it simplifies life considerably both for developers and administrators. It makes it particularly easy to migrate traditional network applications to Kubernetes. A pod represents a traditional node and each container represents a traditional process.

We will cover the following:

- Intra-pod communication
- Pod-to-service communication
- External access

- Lookup and discovery
- DNS in Kubernetes

Intra-pod communication (container to container)

A running pod is always scheduled on one (physical or virtual) node. That means that all the containers run on the same node and can talk to each other in various ways, such as via the local filesystem, any IPC mechanism, or using `localhost` and well-known ports. There is no danger of port collision between different pods because each pod has its own IP address and when a container in the pod uses `localhost`, it applies to the pod's IP address only. So if container 1 in pod 1 connects to port `1234`, which container 2 listens to on pod 1, it will not conflict with another container in pod 2 running on the same node that also listens on port `1234`. The only caveat is that if you're exposing ports to the host, then you should be careful about pod-to-node affinity. This can be handled using several mechanisms, such as Daemonsets and pod anti-affinity.

Inter-pod communication (pod to pod)

Pods in Kubernetes are allocated a network-visible IP address (not private to the node). Pods can communicate directly without the aid of NAT, tunnels, proxies, or any other obfuscating layer. Well-known port numbers can be used for a configuration-free communication scheme. The pod's internal IP address is the same as its external IP address that other pods see (within the cluster network; not exposed to the outside world). That means that standard naming and discovery mechanisms such as a **Domain Name System** (**DNS**) work out of the box.

Pod-to-service communication

Pods can talk to each other directly using their IP addresses and well-known ports, but that requires the pods to know each other's IP addresses. In a Kubernetes cluster, pods can be destroyed and created constantly. There may also be multiple replicas of the same pod spec, each with its own IP address. The Kubernetes service resource provides a layer of indirection that is very useful because the service is stable even if the set of actual pods that responds to requests is ever-changing. In addition, you get automatic, highly available load balancing because the kube-proxy on each node takes care of redirecting traffic to the correct pod:

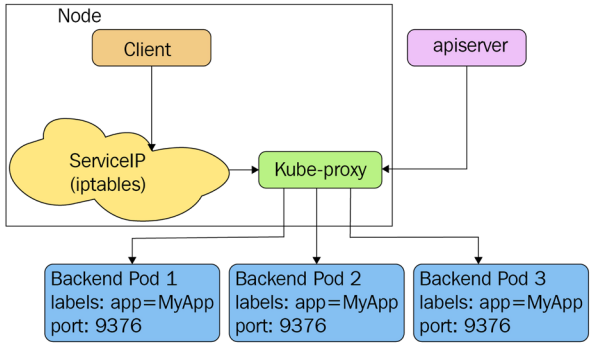


Figure 10.1: Internal load balancing using a serviceExternal access

Eventually, some containers need to be accessible from the outside world. The pod IP addresses are not visible externally. The service is the right vehicle, but external access typically requires two redirects. For example, cloud provider load balancers are not Kubernetes-aware, so they can't direct traffic to a particular service directly to a node that runs a pod that can process the request. Instead, the public load balancer just directs traffic to any node in the cluster and the kube-proxy on that node will redirect it again to an appropriate pod if the current node doesn't run the necessary pod.

The following diagram shows how the external load balancer just sends traffic to an arbitrary node, where the kube-proxy takes care of further routing if needed:

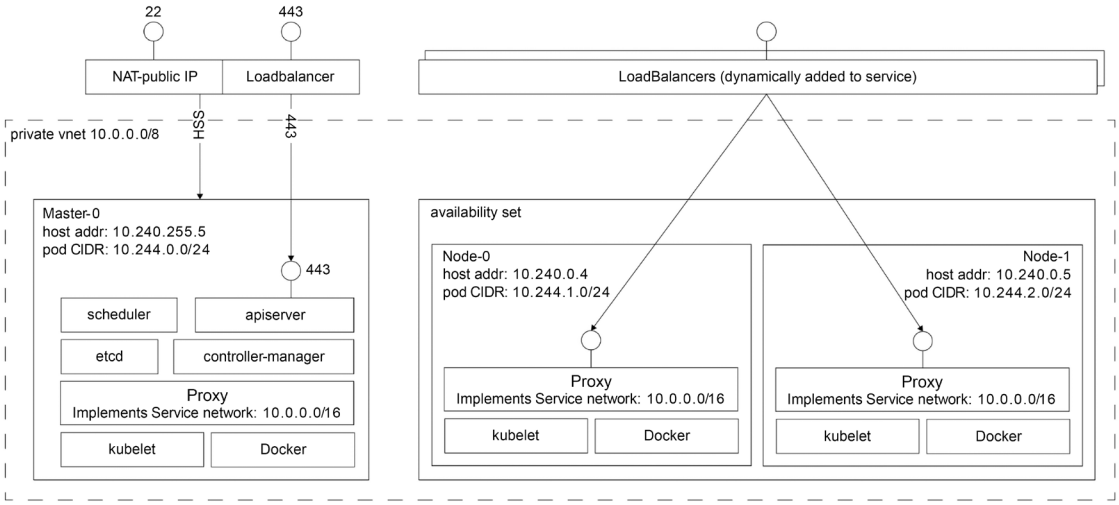


Figure 10.2: External load balancer sending traffic to an arbitrary node and the kube-proxy

Lookup and discovery

In order for pods and containers to communicate with each other, they need to find each other. There are several ways for containers to locate other containers or announce themselves, which we will discuss in the following subsections. Each approach has its own pros and cons.

Self-registration

We've mentioned self-registration several times. Let's understand what it means exactly. When a container runs, it knows its pod's IP address. Every container that wants to be accessible to other containers in the cluster can connect to some registration service and register its IP address and port. Other containers can query the registration service for the IP addresses and ports of all registered containers and connect to them. When a container is destroyed (gracefully), it will unregister itself. If a container dies ungracefully, then some mechanism needs to be established to detect that. For example, the registration service can periodically ping all registered containers, or the containers can be required periodically to send a keep-alive message to the registration service.

The benefit of self-registration is that once the generic registration service is in place (no need to customize it for different purposes), there is no need to worry about keeping track of containers. Another huge benefit is that containers can employ sophisticated policies and decide to unregister temporarily if they are unavailable based on local conditions; for example, if a container is busy and doesn't want to receive any more requests at the moment. This sort of smart and decentralized dynamic load balancing can be very difficult to achieve globally without a registration service. The downside is that the registration service is yet another non-standard component that containers need to know about in order to locate other containers.

Services and endpoints

Kubernetes services can be considered standard registration services. Pods that belong to a service are registered automatically based on their labels. Other pods can look up the endpoints to find all the service pods or take advantage of the service itself and directly send a message to the service that will get routed to one of the backend pods. Although, most of the time, pods will just send their message to the service itself, which will forward it to one of the backing pods. Dynamic membership can be achieved using a combination of the replica count of deployments, health checks, readiness checks, and horizontal pod autoscaling.

Loosely coupled connectivity with queues

What if containers can talk to each other without knowing their IP addresses and ports or even service IP addresses or network names? What if most of the communication can be asynchronous and decoupled? In many cases, systems can be composed of loosely coupled components that are not only unaware of the identities of other components but are also unaware that other components even exist. Queues facilitate such loosely coupled systems. Components (containers) listen to messages from the queue, respond to messages, perform their jobs, and post messages to the queue, such as progress messages, completion status, and errors. Queues have many benefits:

- Easy to add processing capacity without coordination just by adding more containers that listen to the queue
- Easy to keep track of the overall load based on the queue depth
- Easy to have multiple versions of components running side by side by versioning messages and/or queue topics
- Easy to implement load balancing as well as redundancy by having multiple consumers process requests in different modes
- Easy to add or remove other types of listeners dynamically

The downsides of queues are the following:

- You need to make sure that the queue provides appropriate durability and high availability so it doesn't become a critical **single point of failure** (**SPOF**)
- Containers need to work with the async queue API (could be abstracted away)
- Implementing a request-response requires somewhat cumbersome listening on response queues

Overall, queues are an excellent mechanism for large-scale systems and they can be utilized in large Kubernetes clusters to ease coordination.

Loosely coupled connectivity with data stores

Another loosely coupled method is to use a data store (for example, Redis) to store messages and then other containers can read them. While possible, this is not the design objective of data stores, and the result is often cumbersome, fragile, and doesn't have the best performance. Data stores are optimized for data storage and access and not for communication. That being said, data stores can be used in conjunction with queues, where a component stores some data in a data store and then sends a message to the queue saying that the data is ready for processing. Multiple components listen to the message and all start processing the data in parallel.

Kubernetes ingress

Kubernetes offers an ingress resource and controller that is designed to expose Kubernetes services to the outside world. You can do it yourself, of course, but many tasks involved in defining an ingress are common across most applications for a particular type of ingress, such as a web application, CDN, or DDoS protector. You can also write your own ingress objects.

The ingress object is often used for smart load balancing and TLS termination. Instead of configuring and deploying your own Nginx server, you can benefit from the built-in ingress controller. If you need a refresher, check out *Chapter 5*, *Using Kubernetes Resources in Practice*, where we discussed the ingress resource with examples.

DNS in Kubernetes

A DNS is a cornerstone technology in networking. Hosts that are reachable on IP networks have IP addresses. DNS is a hierarchical and decentralized naming system that provides a layer of indirection on top of IP addresses. This is important for several use cases, such as:

- Load balancing
- Dynamically replacing hosts with different IP addresses
- Providing human-friendly names to well-known access points

DNS is a vast topic and a full discussion is outside the scope of this book. Just to give you a sense, there are tens of different RFC standards that cover DNS: `https://en.wikipedia.org/wiki/Domain_Name_System#Standards`.

In Kubernetes, the main addressable resources are pods and services. Each pod and service has a unique internal (private) IP address within the cluster. The kubelet configures the pods with a `resolve.conf` file that points them to the internal DNS server. Here is what it looks like:

```
$ k run -it --image g1g1/py-kube:0.3 -- bash
If you don't see a command prompt, try pressing enter.
root@bash:/#

root@bash:/# cat /etc/resolv.conf
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.0.10
options ndots:5
```

The nameserver IP address `10.96.0.10` is the address of the `kube-dns` service:

```
$ k get svc -n kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   19m
```

A pod's hostname is, by default, just its metadata name. If you want pods to have a fully qualified domain name inside the cluster, you can create a headless service and also set the hostname explicitly, as well as a subdomain to the service name. Here is how to set up a DNS for two pods called `py-kube1` and `py-kube2` with hostnames of `trouble1` and `trouble2`, as well as a subdomain called `maker`, which matches the headless service:

```
apiVersion: v1
kind: Service
metadata:
  name: maker
spec:
  selector:
    app: py-kube
  clusterIP: None # headless service
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: py-kube1
  labels:
    app: py-kube
spec:
  hostname: trouble
  subdomain: maker
  containers:
  - image: g1g1/py-kube:0.3
    command:
      - sleep
      - "9999"
    name: trouble
---
apiVersion: v1
kind: Pod
metadata:
  name: py-kube2
  labels:
    app: py-kube
spec:
  hostname: trouble2
  subdomain: maker
  containers:
    - image: g1g1/py-kube:0.3
      command:
        - sleep
        - "9999"
      name: trouble
```

Let's create the pods and service:

```
$ k apply -f pod-with-dns.yaml
service/maker created
pod/py-kube1 created
pod/py-kube2 created
```

Now, we can check the hostnames and the DNS resolution inside the pod. First, we will connect to `py-kube2` and verify that its hostname is `trouble2` and the **fully qualified domain name** (**FQDN**) is `trouble2.maker.default.svc.cluster.local`.

Then, we can resolve the FQDN of both `trouble` and `trouble2`:

```
$ k exec -it py-kube2 -- bash
root@trouble2:/# hostname
trouble2

root@trouble2:/# hostname --fqdn
trouble2.maker.default.svc.cluster.local

root@trouble2:/# dig +short trouble.maker.default.svc.cluster.local
10.244.0.10

root@trouble2:/# dig +short trouble2.maker.default.svc.cluster.local
10.244.0.9
```

To close the loop, let's confirm that the IP addresses `10.244.0.10` and `10.244.0.9` actually belong to the `py-kube1` and `py-kube2` pods:

```
$ k get po -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP            NODE
NOMINATED NODE   READINESS GATES
py-kube1   1/1     Running   0          10m   10.244.0.10   kind-control-plane
<none>           <none>
py-kube2   1/1     Running   0          18m   10.244.0.9    kind-control-plane
<none>           <none>
```

There are additional configuration options and DNS policies you can apply. See `https://kubernetes.io/docs/concepts/services-networking/dns-pod-service`.

CoreDNS

Earlier, we mentioned that the kubelet uses a `resolve.conf` file to configure pods by pointing them to the internal DNS server, but where is this internal DNS server hiding? You can find it in the `kube-system` namespace. The service is called `kube-dns`:

```
$ k describe svc -n kube-system kube-dns
Name:              kube-dns
Namespace:         kube-system
Labels:            k8s-app=kube-dns
                   kubernetes.io/cluster-service=true
                   kubernetes.io/name=CoreDNS
Annotations:       prometheus.io/port: 9153
                   prometheus.io/scrape: true
Selector:          k8s-app=kube-dns
Type:              ClusterIP
IP Family Policy:  SingleStack
```

```
IP Families:       IPv4
IP:                10.96.0.10
IPs:               10.96.0.10
Port:              dns  53/UDP
TargetPort:        53/UDP
Endpoints:         10.244.0.2:53,10.244.0.3:53
Port:              dns-tcp  53/TCP
TargetPort:        53/TCP
Endpoints:         10.244.0.2:53,10.244.0.3:53
Port:              metrics  9153/TCP
TargetPort:        9153/TCP
Endpoints:         10.244.0.2:9153,10.244.0.3:9153
Session Affinity:  None
Events:            <none>
```

Note that selector: `k8s-app=kube-dns`. Let's find the pods that back this service:

```
$ k get po -n kube-system -l k8s-app=kube-dns
NAME                       READY   STATUS    RESTARTS   AGE
coredns-64897985d-n4x5b    1/1     Running   0          97m
coredns-64897985d-nqtwk    1/1     Running   0          97m
```

The service is called `kube-dns`, but the pods have a prefix of `coredns`. Interesting. Let's check the image the deployment uses:

```
$ k get deploy coredns -n kube-system -o jsonpath='{.spec.template.spec.
containers[0]}' | jq .image
"k8s.gcr.io/coredns/coredns:v1.8.6"
```

The reason for this mismatch is that, initially, the default Kubernetes DNS server was called `kube-dns`. Then, `CoreDNS` replaced it as the mainstream DNS server due to its simplified architecture and better performance.

We have covered a lot of information about the Kubernetes networking model and its components. In the next section, we will cover the Kubernetes network plugins that implement this model with standard interfaces such as CNI and Kubenet.

Kubernetes network plugins

Kubernetes has a network plugin system since networking is so diverse and different people would like to implement it in different ways. Kubernetes is flexible enough to support any scenario. The primary network plugin is CNI, which we will discuss in depth. But Kubernetes also comes with a simpler network plugin called Kubenet. Before we go over the details, let's get on the same page with the basics of Linux networking (just the tip of the iceberg). This is important because Kubernetes networking is built on top of standard Linux networking and you need this foundation to understand how Kubernetes networking works.

Basic Linux networking

Linux, by default, has a single shared network space. The physical network interfaces are all accessible in this namespace. But the physical namespace can be divided into multiple logical namespaces, which is very relevant to container networking.

IP addresses and ports

Network entities are identified by their IP address. Servers can listen to incoming connections on multiple ports. Clients can connect (TCP) or send/receive data (UDP) to servers within their network.

Network namespaces

Namespaces group a bunch of network devices such that they can reach other servers in the same namespace, but not *other* servers, even if they are physically on the same network. Linking networks or network segments can be done via bridges, switches, gateways, and routing.

Subnets, netmasks, and CIDRs

A granular division of networks segments is very useful when designing and maintaining networks. Dividing networks into smaller subnets with a common prefix is a common practice. These subnets can be defined by bitmasks that represent the size of the subnet (how many hosts it can contain). For example, a netmask of 255.255.255.0 means that the first 3 octets are used for routing and only 256 (actually 254) individual hosts are available. The **Classless Inter-Domain Routing** (**CIDR**) notation is often used for this purpose because it is more concise, encodes more information, and also allows combining hosts from multiple legacy classes (A, B, C, D, E). For example, 172.27.15.0/24 means that the first 24 bits (3 octets) are used for routing.

Virtual Ethernet devices

Virtual Ethernet (**veth**) devices represent physical network devices. When you create a veth that's linked to a physical device, you can assign that veth (and by extension, the physical device) into a namespace where devices from other namespaces can't reach it directly, even if, physically, they are on the same local network.

Bridges

Bridges connect multiple network segments to an aggregate network, so all the nodes can communicate with each other. Bridging is done at layer 2 (the data link) of the OSI network model.

Routing

Routing connects separate networks, typically based on routing tables that instruct network devices how to forward packets to their destinations. Routing is done through various network devices, such as routers, gateways, switches, and firewalls, including regular Linux boxes.

Maximum transmission unit

The **maximum transmission unit** (**MTU**) determines how big packets can be. On Ethernet networks, for example, the MTU is 1,500 bytes. The bigger the MTU, the better the ratio between payload and headers, which is a good thing. But the downside is that minimum latency is reduced because you have to wait for the entire packet to arrive and, furthermore, in case of failure, you have to retransmit the entire big packet.

Pod networking

Here is a diagram that describes the relationship between pod, host, and the global internet at the networking level via `veth0`:

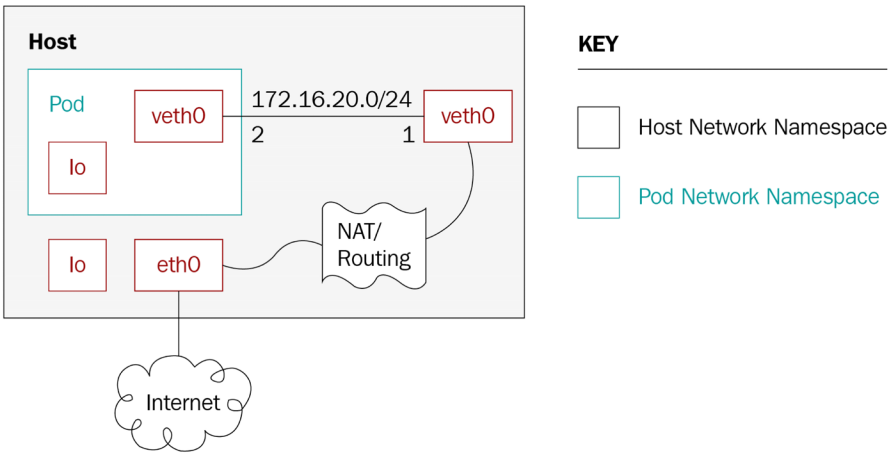


Figure 10.3: Pod networking

Kubenet

Back to Kubernetes. Kubenet is a network plugin. It's very rudimentary: it establishes a Linux bridge named `cbr0` and creates a veth interface for each pod. This is commonly used by cloud providers to configure routing rules for communication between nodes, or in single-node environments. The veth pair connects each pod to its host node using an IP address from the host's IP address' range.

Requirements

The Kubenet plugin has the following requirements:

- The node must be assigned a subnet to allocate IP addresses to its pods
- The standard CNI bridge, `lo`, and host-local plugins must be installed at version 0.2.0 or higher
- The kubelet must be executed with the `--network-plugin=kubenet` flag
- The kubelet must be executed with the `--non-masquerade-cidr=<clusterCidr>` flag
- The kubelet must be run with `--pod-cidr` or the kube-controller-manager must be run with `--allocate-node-cidrs=true --cluster-cidr=<cidr>`

Setting the MTU

The MTU is critical for network performance. Kubernetes network plugins such as Kubenet make their best efforts to deduce the optimal MTU, but sometimes they need help. If an existing network interface (for example, the `docker0` bridge) sets a small MTU, then Kubenet will reuse it. Another example is IPsec, which requires lowering the MTU due to the extra overhead from IPsec encapsulation, but the Kubenet network plugin doesn't take it into consideration. The solution is to avoid relying on the automatic calculation of the MTU and just tell the kubelet what MTU should be used for network plugins via the `--network-plugin-mtu` command-line switch that is provided to all network plugins. However, at the moment, only the Kubenet network plugin accounts for this command-line switch.

The Kubenet network plugin is mostly around for backward compatibility reasons. The CNI is the primary network interface that all modern network solution providers implement to integrate with Kubernetes. Let's see what it's all about.

The CNI

The CNI is a specification as well as a set of libraries for writing network plugins to configure network interfaces in Linux containers. The specification actually evolved from the rkt network proposal. CNI is an established industry standard now even beyond Kubernetes. Some of the organizations that use CNI are:

- Kubernetes
- OpenShift
- Mesos
- Kurma
- Cloud Foundry
- Nuage
- IBM
- AWS EKS and ECS
- Lyft

The CNI team maintains some core plugins, but there are a lot of third-party plugins too that contribute to the success of CNI. Here is a non-exhaustive list:

- Project Calico: A layer 3 virtual network for Kubernetes
- Weave: A virtual network to connect multiple Docker containers across multiple hosts
- Contiv networking: Policy-based networking
- Cilium: ePBF for containers
- Flannel: Layer 3 network fabric for Kubernetes
- Infoblox: Enterprise-grade IP address management
- Silk: A CNI plugin for Cloud Foundry

- OVN-kubernetes: A CNI plugin based on OVS and Open Virtual Networking (OVN)
- DANM: Nokia's solution for Telco workloads on Kubernetes

CNI plugins provide a standard networking interface for arbitrary networking solutions.

The container runtime

CNI defines a plugin spec for networking application containers, but the plugin must be plugged into a container runtime that provides some services. In the context of CNI, an application container is a network-addressable entity (has its own IP address). For Docker, each container has its own IP address. For Kubernetes, each pod has its own IP address and the pod is considered the CNI container, and the containers within the pod are invisible to CNI.

The container runtime's job is to configure a network and then execute one or more CNI plugins, passing them the network configuration in JSON format.

The following diagram shows a container runtime using the CNI plugin interface to communicate with multiple CNI plugins:

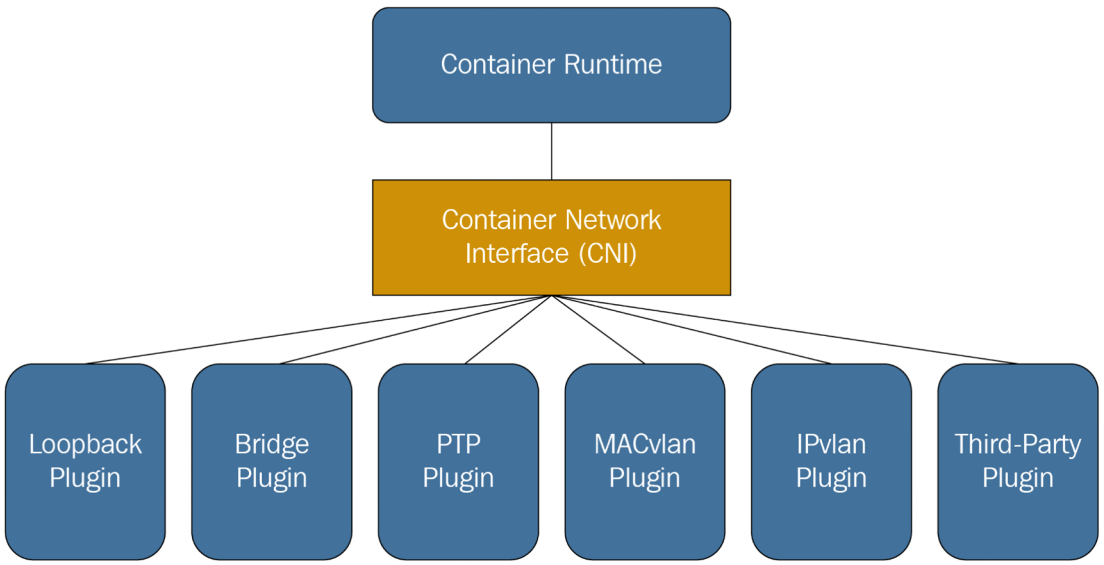


Figure 10.4: Container runtime with CNI

The CNI plugin

The CNI plugin's job is to add a network interface into the container network namespace and bridge the container to the host via a veth pair. It should then assign an IP address via an **IP address management** (**IPAM**) plugin and set up routes.

The container runtime (any CRI-compliant runtime) invokes the CNI plugin as an executable. The plugin needs to support the following operations:

- Add a container to the network
- Remove a container from the network
- Report version

The plugin uses a simple command-line interface, standard input/output, and environment variables. The network configuration in JSON format is passed to the plugin through standard input. The other arguments are defined as environment variables:

- `CNI_COMMAND`: Specifies the desired operation, such as `ADD`, `DEL`, or `VERSION`.
- `CNI_CONTAINERID`: Represents the ID of the container.
- `CNI_NETNS`: Points to the path of the network namespace file.
- `CNI_IFNAME`: Specifies the name of the interface to be set up. The CNI plugin should use this name or return an error.
- `CNI_ARGS`: Contains additional arguments passed in by the user during invocation. It consists of alphanumeric key-value pairs separated by semicolons, such as `FOO=BAR;ABC=123`.
- `CNI_PATH`: Indicates a list of paths to search for CNI plugin executables. The paths are separated by an OS-specific list separator, such as ":" on Linux and ";" on Windows.

If the command succeeds, the plugin returns a zero exit code and the generated interfaces (in the case of the `ADD` command) are streamed to standard output as JSON. This low-tech interface is smart in the sense that it doesn't require any specific programming language or component technology or binary API. CNI plugin writers can use their favorite programming language too.

The result of invoking the CNI plugin with the `ADD` command looks as follows:

```
{
  "cniVersion": "0.3.0",
  "interfaces": [              (this key omitted by IPAM plugins)
      {
          "name": "<name>",
          "mac": "<MAC address>", (required if L2 addresses are meaningful)
          "sandbox": "<netns path or hypervisor identifier>" (required for
container/hypervisor interfaces, empty/omitted for host interfaces)
      }
  ],
  "ip": [
      {
          "version": "<4-or-6>",
          "address": "<ip-and-prefix-in-CIDR>",
          "gateway": "<ip-address-of-the-gateway>",     (optional)
          "interface": <numeric index into 'interfaces' list>
```

```
        },
        ...
    ],
    "routes": [                                          (optional)
        {
            "dst": "<ip-and-prefix-in-cidr>",
            "gw": "<ip-of-next-hop>"                     (optional)
        },
        ...
    ]
    "dns": {
      "nameservers": <list-of-nameservers>               (optional)
      "domain": <name-of-local-domain>                   (optional)
      "search": <list-of-additional-search-domains>      (optional)
      "options": <list-of-options>                       (optional)
    }
}
```

The input network configuration contains a lot of information: `cniVersion`, `name`, `type`, `args` (optional), `ipMasq` (optional), `ipam`, and `dns`. The `ipam` and `dns` parameters are dictionaries with their own specified keys. Here is an example of a network configuration:

```
{
  "cniVersion": "0.3.0",
  "name": "dbnet",
  "type": "bridge",
  // type (plugin) specific
  "bridge": "cni0",
  "ipam": {
    "type": "host-local",
    // ipam specific
    "subnet": "10.1.0.0/16",
    "gateway": "10.1.0.1"
  },
  "dns": {
    "nameservers": ["10.1.0.1"]
  }
}
```

Note that additional plugin-specific elements can be added. In this case, the `bridge: cni0` element is a custom one that the specific bridge plugin understands.

The CNI spec also supports network configuration lists where multiple CNI plugins can be invoked in order.

That concludes the conceptual discussion of Kubernetes network plugins, which are built on top of basic Linux networking, allowing multiple network solution providers to integrate smoothly with Kubernetes.

Later in this chapter, we will dig into a full-fledged implementation of a CNI plugin. First, let's talk about one of the most exciting prospects in the Kubernetes networking world – **extended Berkeley Packet Filter** (**eBPF**).

Kubernetes and eBPF

Kubernetes, as you know very well, is a very versatile and flexible platform. The Kubernetes developers, in their wisdom, avoided making many assumptions and decisions that could later paint them into a corner. For example, Kubernetes networking operates at the IP and DNS levels only. There is no concept of a network or subnets. Those are left for networking solutions that integrate with Kubernetes through very narrow and generic interfaces like CNI.

That opens the door to a lot of innovation because Kubernetes doesn't constrain the choices of implementors.

Enter ePBF. It is a technology that allows running sandboxed programs safely in the Linux kernel without compromising the system's security or requiring you to make changes to the kernel itself or even kernel modules. These programs execute in response to events. This is a big deal for software-defined networking, observability, and security. Brendan Gregg calls it the Linux super-power.

The original BPF technology could be attached only to sockets for packet filtering (hence the name Berkeley Packet Filter). With ePBF, you can attach to additional objects, such as:

- Kprobes
- Tracepoints
- Network schedulers or qdiscs for classification or action
- XDP

Traditional Kubernetes routing is done by the kube-proxy. It is a user space process that runs on every node. It's responsible for setting up `iptable` rules and does UDP, TCP, and STCP forwarding as well as load balancing (based on Kubernetes services). At large scale, kube-proxy becomes a liability. The `iptable` rules are processed sequentially and the frequent user space to kernel space transitions are unnecessary overhead. It is possible to completely remove kube-proxy and replace it with an eBPF-based approach that performs the same function much more efficiently. We will discuss one of these solutions – Cilium – in the next section.

Here is an overview of eBPF:

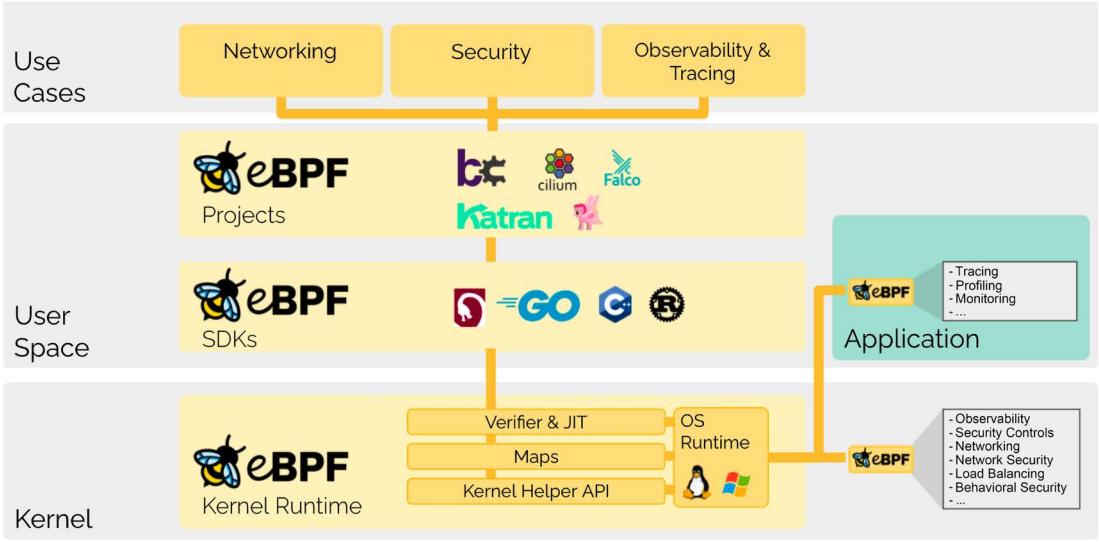


Figure 10.5: eBPF overview

For more details, check out `https://ebpf.io`.

Kubernetes networking solutions

Networking is a vast topic. There are many ways to set up networks and connect devices, pods, and containers. Kubernetes can't be opinionated about it. The high-level networking model of a flat address space for Pods is all that Kubernetes prescribes. Within that space, many valid solutions are possible, with various capabilities and policies for different environments. In this section, we'll examine some of the available solutions and understand how they map to the Kubernetes networking model.

Bridging on bare-metal clusters

The most basic environment is a raw bare-metal cluster with just an L2 physical network. You can connect your containers to the physical network with a Linux bridge device. The procedure is quite involved and requires familiarity with low-level Linux network commands such as `brctl`, `ipaddr`, `iproute`, `iplink`, and `nsenter`. If you plan to implement it, this guide can serve as a good start (search for the *With Linux Bridge devices* section): `http://blog.oddbit.com/2014/08/11/four-ways-to-connect-a-docker/`.

The Calico project

Calico is a versatile virtual networking and network security solution for containers. Calico can integrate with all the primary container orchestration frameworks and runtimes:

- Kubernetes (CNI plugin)
- Mesos (CNI plugin)
- Docker (libnetwork plugin)
- OpenStack (Neutron plugin)

Calico can also be deployed on-premises or on public clouds with its full feature set. Calico's network policy enforcement can be specialized for each workload and makes sure that traffic is controlled precisely and packets always go from their source to vetted destinations. Calico can automatically map network policy concepts from orchestration platforms to its own network policy. The reference implementation of Kubernetes' network policy is Calico. Calico can be deployed together with Flannel, utilizing Flannel's networking layer and Calico's network policy facilities.

Weave Net

Weave Net is all about ease of use and zero configuration. It uses VXLAN encapsulation under the hood and micro DNS on each node. As a developer, you operate at a higher abstraction level. You name your containers and Weave Net lets you connect to them and use standard ports for services. That helps migrate existing applications into containerized applications and microservices. Weave Net has a CNI plugin for interfacing with Kubernetes (and Mesos). On Kubernetes 1.4 and higher, you can integrate Weave Net with Kubernetes by running a single command that deploys a `Daemonset`:

```
kubectl apply -f https://github.com/weaveworks/weave/releases/download/v2.8.1/weave-daemonset-k8s.yaml
```

The Weave Net pods on every node will take care of attaching any new pod you create to the Weave network. Weave Net supports the network policy API, as well providing a complete, yet easy-to-set-up solution.

Cilium

Cilium is a CNCF incubator project that is focused on eBPF-based networking, security, and observability (via its Hubble project).

Let's take a look at the capabilities Cilium provides.

Efficient IP allocation and routing

Cilium allows a flat Layer 3 network that covers multiple clusters and connects all application containers. Host scope allocators can allocate IP addresses without coordination with other hosts. Cilium supports multiple networking models:

- **Overlay**: This model utilizes encapsulation-based virtual networks that span across all hosts. It supports encapsulation formats like VXLAN and Geneve, as well as other formats supported by Linux. Overlay mode works with almost any network infrastructure as long as the hosts have IP connectivity. It provides a flexible and scalable solution.

- **Native routing**: In this model, Kubernetes leverages the regular routing table of the Linux host. The network infrastructure must be capable of routing the IP addresses used by the application containers. Native Routing mode is considered more advanced and requires knowledge of the underlying networking infrastructure. It works well with native IPv6 networks, cloud network routers, or when using custom routing daemons.

Identity-based service-to-service communication

Cilium provides a security management feature that assigns a security identity to groups of application containers with the same security policies. This identity is then associated with all network packets generated by those application containers. By doing this, Cilium enables the validation of the identity at the receiving node. The management of security identities is handled through a key-value store, which allows for efficient and secure management of identities within the Cilium networking solution.

Load balancing

Cilium offers distributed load balancing for traffic between application containers and external services as an alternative to kube-proxy. This load balancing functionality is implemented using efficient hashtables in eBPF, providing a scalable approach compared to the traditional iptables method. With Cilium, you can achieve high-performance load balancing while ensuring efficient utilization of network resources.

When it comes to east-west load balancing, Cilium excels in performing efficient service-to-backend translation directly within the Linux kernel's socket layer. This approach eliminates the need for per-packet NAT operations, resulting in lower overhead and improved performance.

For north-south load balancing, Cilium's eBPF implementation is highly optimized for maximum performance. It can be seamlessly integrated with **XDP** (**eXpress Data Path**) and supports advanced load balancing techniques like **Direct Server Return** (**DSR**) and Maglev consistent hashing. This allows load balancing operations to be efficiently offloaded from the source host, further enhancing performance and scalability.

Bandwidth management

Cilium implements bandwidth management through efficient **Earliest Departure Time** (**EDT**)-based rate-limiting with eBPF for egress traffic. This significantly reduces transmission tail latencies for applications.

Observability

Cilium offers comprehensive event monitoring with rich metadata. In addition to capturing the source and destination IP addresses of dropped packets, it also provides detailed label information for both the sender and receiver. This metadata enables enhanced visibility and troubleshooting capabilities. Furthermore, Cilium exports metrics through Prometheus, allowing for easy monitoring and analysis of network performance.

To further enhance observability, the Hubble observability platform provides additional features such as service dependency maps, operational monitoring, alerting, and comprehensive visibility into application and security aspects. By leveraging flow logs, Hubble enables administrators to gain valuable insights into the behavior and interactions of services within the network.

Cilium is a large project with a very broad scope. Here, we just scratched the surface. See `https://cilium.io` for more details.

There are many good networking solutions. Which network solution is the best for you? If you're running in the cloud, I recommend using the native CNI plugin from your cloud provider. If you're on your own, Calico is a solid choice, and if you're adventurous and need to heavily optimize your network, consider Cilium.

In the next section, we will cover network policies that let you get a handle on the traffic in your cluster.

Using network policies effectively

The Kubernetes network policy is about managing network traffic to selected pods and namespaces. In a world of hundreds of microservices deployed and orchestrated, as is often the case with Kubernetes, managing networking and connectivity between pods is essential. It's important to understand that it is not primarily a security mechanism. If an attacker can reach the internal network, they will probably be able to create their own pods that comply with the network policy in place and communicate freely with other pods. In the previous section, we looked at different Kubernetes networking solutions and focused on the container networking interface. In this section, the focus is on the network policy, although there are strong connections between the networking solution and how the network policy is implemented on top of it.

Understanding the Kubernetes network policy design

A network policy defines the communication rules for pods and other network endpoints within a Kubernetes cluster. It uses labels to select specific pods and applies whitelist rules to control traffic access to the selected pods. These rules complement the isolation policy defined at the namespace level by allowing additional traffic based on the defined criteria. By configuring network policies, administrators can fine-tune and restrict the communication between pods, enhancing security and network segmentation within the cluster.

Network policies and CNI plugins

There is an intricate relationship between network policies and CNI plugins. Some CNI plugins implement both network connectivity and a network policy, while others implement just one aspect, but they can collaborate with another CNI plugin that implements the other aspect (for example, Calico and Flannel).

Configuring network policies

Network policies are configured via the `NetworkPolicy` resource. You can define ingress and/or egress policies. Here is a sample network policy that specifies both ingress and egress:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: awesome-project
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              project: awesome-project
        - podSelector:
            matchLabels:
              role: frontend
      ports:
       - protocol: TCP
         port: 6379
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/24
      ports:
        - protocol: TCP
          port: 7777
```

Implementing network policies

While the network policy API itself is generic and is part of the Kubernetes API, the implementation is tightly coupled to the networking solution. That means that on each node, there is a special agent or gatekeeper (Cilium implements it via eBPF in the kernel) that does the following:

- Intercepts all traffic coming into the node
- Verifies that it adheres to the network policy
- Forwards or rejects each request

Kubernetes provides the facilities to define and store network policies through the API. Enforcing the network policy is left to the networking solution or a dedicated network policy solution that is tightly integrated with the specific networking solution.

Calico is a good example of this approach. Calico has its own networking solution and a network policy solution, which work together. In both cases, there is tight integration between the two pieces. The following diagram shows how the Kubernetes policy controller manages the network policies and how agents on the nodes execute them:

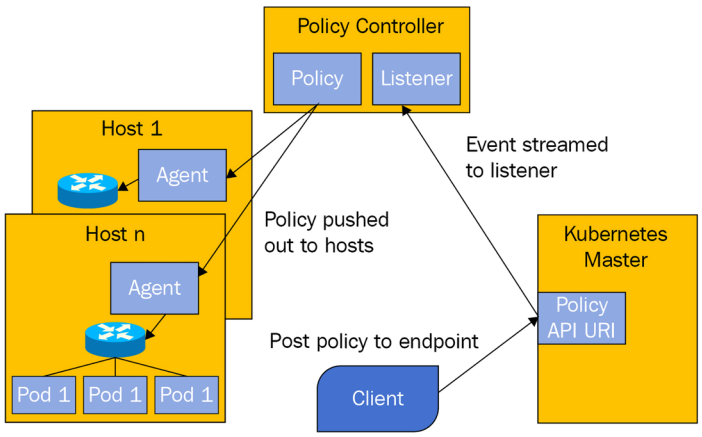


Figure 10.6: Kubernetes network policy management

In this section, we covered various networking solutions, as well as network policies, and we briefly discussed load balancing. However, load balancing is a wide subject and the next section will explore it.

Load balancing options

Load balancing is a critical capability in dynamic systems such as a Kubernetes cluster. Nodes, VMs, and pods come and go, but the clients typically can't keep track of which individual entities can service their requests. Even if they could, it requires a complicated dance of managing a dynamic map of the cluster, refreshing it frequently, and handling disconnected, unresponsive, or just slow nodes. This so-called client-side load balancing is appropriate in special cases only. Server-side load balancing is a battle-tested and well-understood mechanism that adds a layer of indirection that hides the internal turmoil from the clients or consumers outside the cluster. There are options for external as well as internal load balancers. You can also mix and match and use both. The hybrid approach has its own particular pros and cons, such as performance versus flexibility. We will cover the following options:

- External load balancer
- Service load balancer
- Ingress
- HA Proxy
- MetalLB
- Traefik
- Kubernetes Gateway API

External load balancers

An external load balancer is a load balancer that runs outside the Kubernetes cluster. There must be an external load balancer provider that Kubernetes can interact with to configure the external load balancer with health checks and firewall rules and get the external IP address of the load balancer.

The following diagram shows the connection between the load balancer (in the cloud), the Kubernetes API server, and the cluster nodes. The external load balancer has an up-to-date picture of which pods run on which nodes and it can direct external service traffic to the right pods:

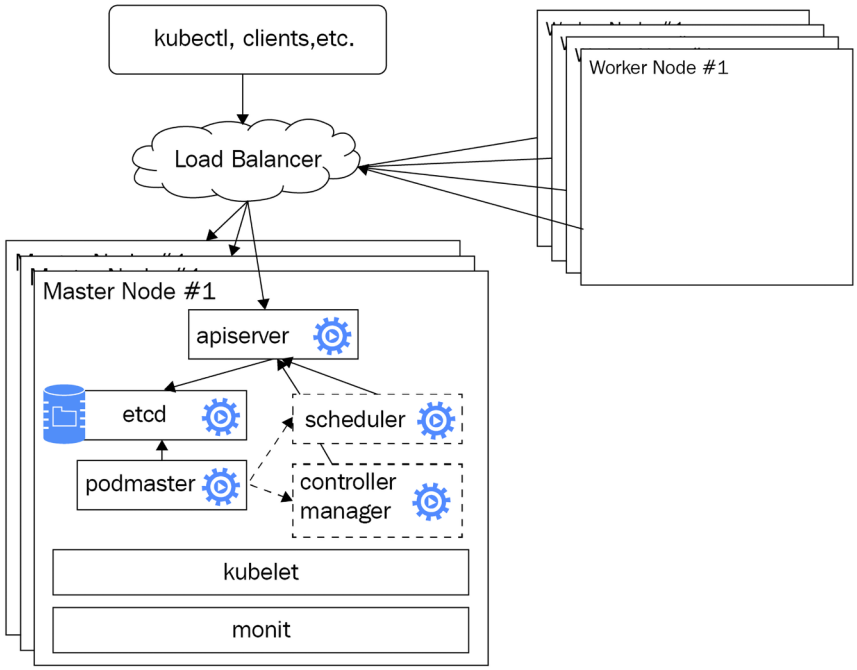


Figure 10.7: The connection between the load balancer, the Kubernetes API server, and the cluster nodes

Configuring an external load balancer

The external load balancer is configured via the service configuration file or directly through kubectl. We use a service type of `LoadBalancer` instead of using a service type of `ClusterIP`, which directly exposes a Kubernetes node as a load balancer. This depends on an external load balancer provider properly installed and configured in the cluster.

Via manifest file

Here is an example service manifest file that accomplishes this goal:

```
apiVersion: v1
kind: Service
metadata:
  name: api-gateway
spec:
  type: LoadBalancer
  ports:
  - port:  80
    targetPort: 5000
  selector:
    svc: api-gateway
    app: delinkcious
```

Via kubectl

You may also accomplish the same result using a direct `kubectl` command:

```
$ kubectl expose deployment api-gateway --port=80 --target-port=5000 --name=api-
gateway --type=LoadBalancer
```

The decision whether to use a service configuration file or `kubectl` command is usually determined by the way you set up the rest of your infrastructure and deploy your system. Manifest files are more declarative and more appropriate for production usage where you want a versioned, auditable, and repeatable way to manage your infrastructure. Typically, this will be part of a GitOps-based CI/CD pipeline.

Finding the load balancer IP addresses

The load balancer will have two IP addresses of interest. The internal IP address can be used inside the cluster to access the service. Clients outside the cluster will use the external IP address. It's a good practice to create a DNS entry for the external IP address. It is particularly important if you want to use TLS/SSL, which requires stable hostnames. To get both addresses, use the `kubectl describe service` command. The `IP` field denotes the internal IP address and the `LoadBalancer Ingress` field denotes the external IP address:

```
$ kubectl describe services example-service
Name: example-service
Selector: app=example
Type: LoadBalancer
IP: 10.67.252.103
LoadBalancer Ingress: 123.45.678.9
Port: <unnamed> 80/TCP
NodePort: <unnamed> 32445/TCP
```

```
Endpoints: 10.64.0.4:80,10.64.1.5:80,10.64.2.4:80
Session Affinity: None
No events.
```

Preserving client IP addresses

Sometimes, the service may be interested in the source IP address of the clients. Up until Kubernetes 1.5, this information wasn't available. In Kubernetes 1.7, the capability to preserve the original client IP was added to the API.

Specifying original client IP address preservation

You need to configure the following two fields of the `service` spec:

- `service.spec.externalTrafficPolicy`: This field determines whether the service should route external traffic to a node-local endpoint or a cluster-wide endpoint, which is the default. The `Cluster` option doesn't reveal the client source IP and might add a hop to a different node, but spreads the load well. The `Local` option keeps the client source IP and doesn't add an extra hop as long as the service type is `LoadBalancer` or `NodePort`. Its downside is it might not balance the load very well.
- `service.spec.healthCheckNodePort`: This field is optional. If used, then the service health check will use this port number. The default is the allocated node port. It has an effect on services of the `LoadBalancer` type whose `externalTrafficPolicy` is set to `Local`.

Here is an example:

```
apiVersion: v1
kind: Service
metadata:
  name: api-gateway
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  ports:
  - port:  80
    targetPort: 5000
  selector:
    svc: api-gateway
    app: delinkcious
```

Understanding even external load balancing

External load balancers operate at the node level; while they direct traffic to a particular pod, the load distribution is done at the node level. That means that if your service has four pods, and three of them are on node A and the last one is on node B, then an external load balancer is likely to divide the load evenly between node A and node B.

This will have the 3 pods on node A handle half of the load (1/6 each) and the single pod on node B handle the other half of the load on its own. Weights may be added in the future to address this issue. You can avoid the issue of too many pods unevenly distributed between nodes by using pod anti-affinity or topology spread constraints.

Service load balancers

Service load balancing is designed for funneling internal traffic within the Kubernetes cluster and not for external load balancing. This is done by using a service type of `clusterIP`. It is possible to expose a service load balancer directly via a pre-allocated port by using a service type of `NodePort` and using it as an external load balancer, but it requires curating all Node ports across the cluster to avoid conflicts and might not be appropriate for production. Desirable features such as SSL termination and HTTP caching will not be readily available.

The following diagram shows how the service load balancer (the yellow cloud) can route traffic to one of the backend pods it manages (via labels of course):

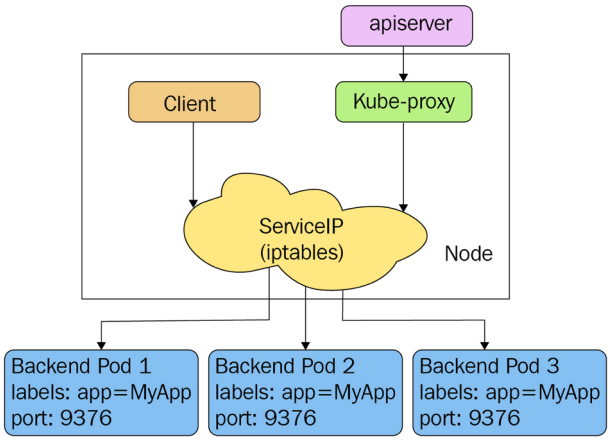


Figure 10.8: The service load balancer routing traffic to a backend pod

Ingress

Ingress in Kubernetes is, at its core, a set of rules that allow inbound HTTP/S traffic to reach cluster services. In addition, some ingress controllers support the following:

- Connection algorithms
- Request limits
- URL rewrites and redirects
- TCP/UDP load balancing
- SSL termination
- Access control and authorization

Ingress is specified using an `Ingress` resource and is serviced by an ingress controller. The `Ingress` resource was in beta since Kubernetes 1.1 and finally, in Kubernetes 1.19, it became GA. Here is an example of an ingress resource that manages traffic into two services. The rules map the externally visible `http://foo.bar.com/foo` to the `s1` service, and `http://foo.bar.com/bar` to the `s2` service:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test
spec:
  ingressClassName: cool-ingress
  rules:
  - host: foo.bar.com
    http:
      paths:

- path: /foo
  backend:
    service:
      name: s1
      port: 80
- path: /bar
  backend:
    service:
      name: s2
      port: 80
```

The `ingressClassname` specifies an `IngressClass` resource, which contains additional information about the ingress. If it's omitted, a default ingress class must be defined.

Here is what it looks like:

```
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  labels:
    app.kubernetes.io/component: controller
  name: cool-ingress
  annotations:
    ingressclass.kubernetes.io/is-default-class: "true"
spec:
  controller: k8s.io/ingress-nginx
```

Ingress controllers often require annotations to be added to the `Ingress` resource in order to customize its behavior.

The following diagram demonstrates how `Ingress` works:

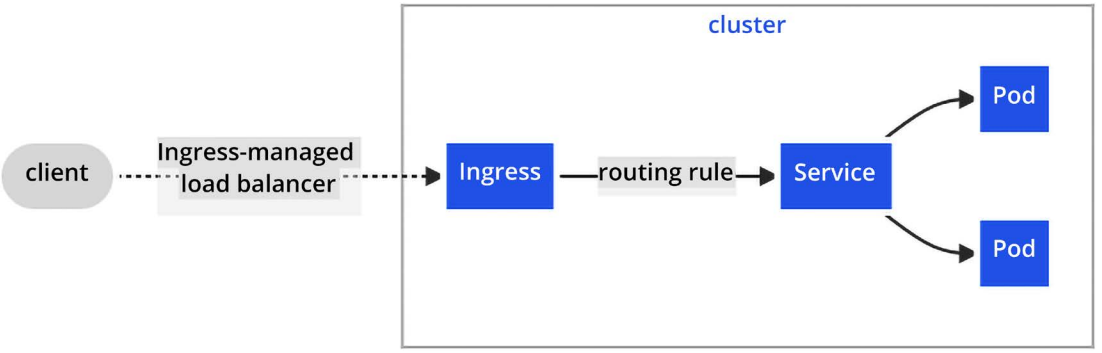


Figure 10.9: Demonstration of ingress

There are two official ingress controllers right now in the main Kubernetes repository. One of them is an L7 ingress controller for GCE only, the other is a more general-purpose Nginx ingress controller that lets you configure the Nginx web server through a `ConfigMap`. The Nginx ingress controller is very sophisticated and brings a lot of features that are not available yet through the ingress resource directly. It uses the Endpoints API to directly forward traffic to pods. It supports Minikube, GCE, AWS, Azure, and bare-metal clusters. For more details, check out `https://github.com/kubernetes/ingress-nginx`.

However, there are many more ingress controllers that may be better for your use case, such as:

- Ambassador
- Traefik
- Contour
- Gloo

For even more ingress controllers, see `https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/`.

HAProxy

We discussed using a cloud provider external load balancer using the service type `LoadBalancer` and using the internal service load balancer inside the cluster using `ClusterIP`. If we want a custom external load balancer, we can create a custom external load balancer provider and use `LoadBalancer` or use the third service type, `NodePort`. **High-Availability** (**HA**) **Proxy** is a mature and battle-tested load balancing solution. It is considered one of the best choices for implementing external load balancing with on-premises clusters. This can be done in several ways:

- Utilize `NodePort` and carefully manage port allocations
- Implement a custom load balancer provider interface

- Run HAProxy inside your cluster as the only target of your frontend servers at the edge of the cluster (load balanced or not)

You can use all these approaches with HAProxy. Regardless, it is still recommended to use ingress objects. The service-loadbalancer project is a community project that implemented a load balancing solution on top of HAProxy. You can find it here: https://github.com/kubernetes/contrib/tree/master/service-loadbalancer. Let's look into how to use HAProxy in a bit more detail.

Utilizing the NodePort

Each service will be allocated a dedicated port from a predefined range. This usually is a high range such as 30,000 and above to avoid clashing with other applications using ports that are not well known. HAProxy will run outside the cluster in this case and it will be configured with the correct port for each service. Then, it can just forward any traffic to any nodes and Kubernetes via the internal service, and the load balancer will route it to a proper pod (double load balancing). This is, of course, sub-optimal because it introduces another hop. The way to circumvent it is to query the Endpoints API and dynamically manage for each service the list of its backend pods and directly forward traffic to the pods.

A custom load balancer provider using HAProxy

This approach is a little more complicated, but the benefit is that it is better integrated with Kubernetes and can make the transition to/from on-premises and the cloud easier.

Running HAProxy inside the Kubernetes cluster

In this approach, we use the internal HAProxy load balancer inside the cluster. There may be multiple nodes running HAProxy and they will share the same configuration to map incoming requests and load-balance them across the backend servers (the Apache servers in the following diagram):

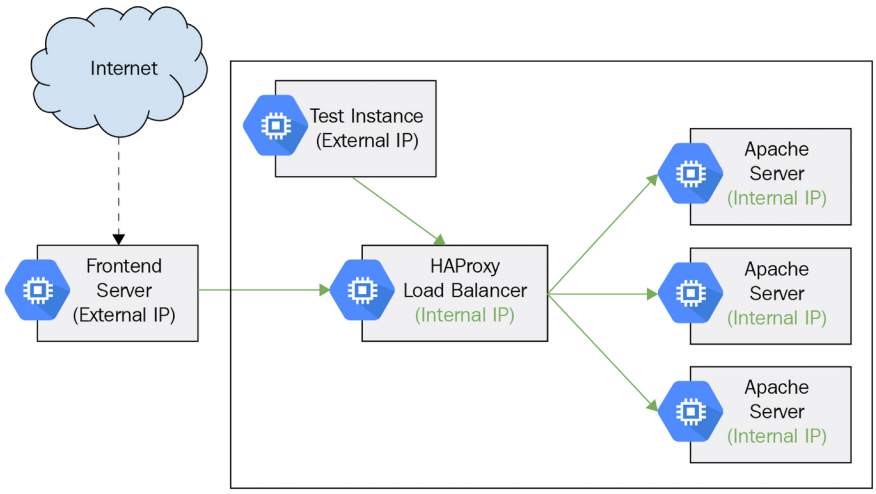


Figure 10.10: Multiple nodes running HAProxy for incoming requests and to load-balance the backend servers

HAProxy also developed its own ingress controller, which is Kubernetes-aware. This is arguably the most streamlined way to utilize HAProxy in your Kubernetes cluster. Here are some of the capabilities you gain when using the HAProxy ingress controller:

- Streamlined integration with the HAProxy load balancer
- SSL termination
- Rate limiting
- IP whitelisting
- Multiple load balancing algorithms: round-robin, least connections, URL hash, and random
- A dashboard that shows the health of your pods, current request rates, response times, etc.
- Traffic overload protection

MetalLB

MetalLB also provides a load balancer solution for bare-metal clusters. It is highly configurable and supports multiple modes such as L2 and BGP. I had success configuring it even for minikube. For more details, check out `https://metallb.universe.tf`.

Traefik

Traefik is a modern HTTP reverse proxy and load balancer. It was designed to support microservices. It works with many backends, including Kubernetes, to manage its configuration automatically and dynamically. This is a game-changer compared to traditional load balancers. It has an impressive list of features:

- It's fast
- Single Go executable
- Tiny official Docker image: The solution provides a lightweight and official Docker image, ensuring efficient resource utilization.
- Rest API: It offers a RESTful API for easy integration and interaction with the solution.
- Hot-reloading of configuration: Configuration changes can be applied dynamically without requiring a process restart, ensuring seamless updates.
- Circuit breakers and retry: The solution includes circuit breakers and retry mechanisms to handle network failures and ensure robust communication.
- Round-robin and rebalancer load balancers: It supports load balancing algorithms like round-robin and rebalancer to distribute traffic across multiple instances.
- Metrics support: The solution provides various options for metrics collection, including REST, Prometheus, Datadog, statsd, and InfluxDB.
- Clean AngularJS web UI: It offers a user-friendly web UI powered by AngularJS for easy configuration and monitoring.

- Websocket, HTTP/2, and GRPC support: The solution is capable of handling Websocket, HTTP/2, and GRPC protocols, enabling efficient communication.
- Access logs: It provides access logs in both JSON and Common Log Format (CLF) for monitoring and troubleshooting.
- Let's Encrypt support: The solution seamlessly integrates with Let's Encrypt for automatic HTTPS certificate generation and renewal.
- High availability with cluster mode: It supports high availability by running in cluster mode, ensuring redundancy and fault tolerance.

Overall, this solution offers a comprehensive set of features for deploying and managing applications in a scalable and reliable manner.

See `https://traefik.io/traefik/` to learn more about Traefik.

Kubernetes Gateway API

Kubernetes Gateway API is a set of resources that model service networking in Kubernetes. You can think of it as the evolution of the ingress API. While there are no intentions to remove the ingress API, its limitations couldn't be addressed by improving it, so the Gateway API project was born.

Where the ingress API consists of a single `Ingress` resource and an optional `IngressClass`, Gateway API is more granular and breaks the definition of traffic management and routing into different resources. Gateway API defines the following resources:

- `GatewayClass`
- `Gateway`
- `HTTPRoute`
- `TLSRoute`
- `TCPRoute`
- `UDPRoute`

Gateway API resources

The role of the `GatewayClass` is to define common configurations and behavior that can be used by multiple similar gateways.

The role of the gateway is to define an endpoint and a collection of routes where traffic can enter the cluster and be routed to backend services. Eventually, the gateway configures an underlying load balancer or proxy.

The role of the routes is to map specific requests that match the route to a specific backend service.

The following diagram demonstrates the resources and organization of Gateway API:

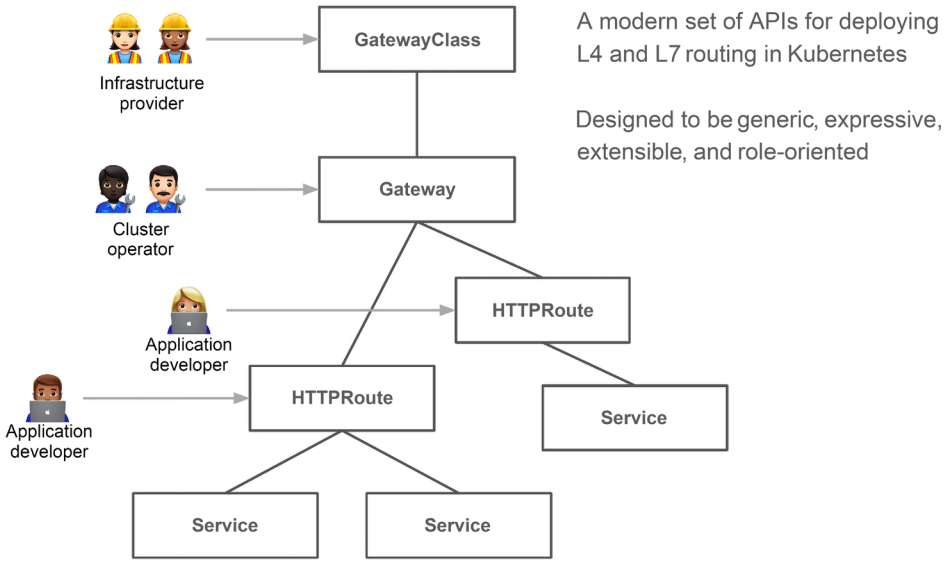


Figure 10.11: Gateway API resources

Attaching routes to gateways

Gateways and routes can be associated in different ways:

- One-to-one: A gateway may have a single route from a single owner that isn't associated with any other gateway
- One-to-many: A gateway may have multiple routes associated with it from multiple owners
- Many-to-many: A route may be associated with multiple gateways (each may have additional routes)

Gateway API in action

Let's see how all the pieces of Gateway API fit together with a simple example. Here is a Gateway resource:

```
apiVersion: gateway.networking.k8s.io/v1beta1
kind: Gateway
metadata:
  name: cool-gateway
  namespace: ns1
spec:
  gatewayClassName: cool-gateway-class
  listeners:
  - name: cool-service
```

```
    port: 80
    protocol: HTTP
    allowedRoutes:
      kinds:
        - kind: HTTPRoute
      namespaces:
        from: Selector
        selector:
          matchLabels:
            # This label is added automatically as of K8s 1.22
            # to all namespaces
            kubernetes.io/metadata.name: ns2
```

Note that the gateway is defined in namespace `ns1`, but it allows only HTTP routes that are defined in namespace `ns2`. Let's see a route that attaches to this gateway:

```
apiVersion: gateway.networking.k8s.io/v1beta1
kind: HTTPRoute
metadata:
  name: cool-route
  namespace: ns2
spec:
  parentRefs:
  - kind: Gateway
    name: cool-gateway
    namespace: ns1
  rules:
  - backendRefs:
    - name: cool-service
      port: 8080
```

The route `cool-route` is properly defined in namespace `ns2`; it is an HTTP route, so it matches. To close the loop, the route defines a parent reference to the `cool-gateway` gateway in namespace `ns1`.

See `https://gateway-api.sigs.k8s.io` to learn more about Gateway API.

Load balancing on Kubernetes is an exciting area. It offers many options for both north-south and east-west load balancing. Now that we have covered load balancing in detail, let's dive deep into the CNI plugins and how they are implemented.

Writing your own CNI plugin

In this section, we will look at what it takes to actually write your own CNI plugin. First, we will look at the simplest plugin possible – the loopback plugin. Then, we will examine the plugin skeleton that implements most of the boilerplate associated with writing a CNI plugin.

Finally, we will review the implementation of the bridge plugin. Before we dive in, here is a quick reminder of what a CNI plugin is:

- A CNI plugin is an executable
- It is responsible for connecting new containers to the network, assigning unique IP addresses to CNI containers, and taking care of routing
- A container is a network namespace (in Kubernetes, a pod is a CNI container)
- Network definitions are managed as JSON files, but are streamed to the plugin via standard input (no files are being read by the plugin)
- Auxiliary information can be provided via environment variables

First look at the loopback plugin

The loopback plugin simply adds the loopback interface. It is so simple that it doesn't require any network configuration information. Most CNI plugins are implemented in Golang and the loopback CNI plugin is no exception. The full source code is available here: `https://github.com/containernetworking/plugins/blob/master/plugins/main/loopback`.

There are multiple packages from the container networking project on GitHub that provide many of the building blocks necessary to implement CNI plugins, as well as the netlink package for adding interfaces, removing interfaces, setting IP addresses, and setting routes. Let's look at the imports of the `loopback.go` file first:

```
package main

import (
    "encoding/json"
    "errors"
    "fmt"
    "net"

    "github.com/vishvananda/netlink"

    "github.com/containernetworking/cni/pkg/skel"
    "github.com/containernetworking/cni/pkg/types"
    current "github.com/containernetworking/cni/pkg/types/100"
    "github.com/containernetworking/cni/pkg/version"

    "github.com/containernetworking/plugins/pkg/ns"
    bv "github.com/containernetworking/plugins/pkg/utils/buildversion"
)
```

Then, the plugin implements two commands, `cmdAdd` and `cmdDel`, which are called when a container is added to or removed from the network. Here is the `add` command, which does all the heavy lifting:

```
func cmdAdd(args *skel.CmdArgs) error {
    conf, err := parseNetConf(args.StdinData)
    if err != nil {
        return err
    }

    var v4Addr, v6Addr *net.IPNet

    args.IfName = "lo" // ignore config, this only works for loopback
    err = ns.WithNetNSPath(args.Netns, func(_ ns.NetNS) error {
        link, err := netlink.LinkByName(args.IfName)
        if err != nil {
            return err // not tested
        }

        err = netlink.LinkSetUp(link)
        if err != nil {
            return err // not tested
        }

        v4Addrs, err := netlink.AddrList(link, netlink.FAMILY_V4)
        if err != nil {
            return err // not tested
        }
        if len(v4Addrs) != 0 {
            v4Addr = v4Addrs[0].IPNet
            // sanity check that this is a loopback address
            for _, addr := range v4Addrs {
                if !addr.IP.IsLoopback() {
                    return fmt.Errorf("loopback interface found with non-
loopback address %q", addr.IP)
                }
            }
        }

        v6Addrs, err := netlink.AddrList(link, netlink.FAMILY_V6)
        if err != nil {
            return err // not tested
        }
```

```
        if len(v6Addrs) != 0 {
            v6Addr = v6Addrs[0].IPNet
            // sanity check that this is a loopback address
            for _, addr := range v6Addrs {
                if !addr.IP.IsLoopback() {
                    return fmt.Errorf("loopback interface found with non-
loopback address %q", addr.IP)
                }
            }
        }

        return nil
    })
    if err != nil {
        return err // not tested
    }

    var result types.Result
    if conf.PrevResult != nil {
        // If loopback has previous result which passes from previous CNI
plugin,
        // loopback should pass it transparently
        result = conf.PrevResult
    } else {
        r := &current.Result{
            CNIVersion: conf.CNIVersion,
            Interfaces: []*current.Interface{
                &current.Interface{
                    Name:    args.IfName,
                    Mac:     "00:00:00:00:00:00",
                    Sandbox: args.Netns,
                },
            },
        }

        if v4Addr != nil {
            r.IPs = append(r.IPs, &current.IPConfig{
                Interface: current.Int(0),
                Address:   *v4Addr,
            })
        }
```

```
        if v6Addr != nil {
            r.IPs = append(r.IPs, &current.IPConfig{
                Interface: current.Int(0),
                Address:   *v6Addr,
            })
        }

        result = r
    }

    return types.PrintResult(result, conf.CNIVersion)
}
```

The core of this function is setting the interface name to `lo` (for loopback) and adding the link to the container's network namespace. It supports both IPv4 and IPv6.

The `del` command does the opposite and is much simpler:

```
func cmdDel(args *skel.CmdArgs) error {
    if args.Netns == "" {
        return nil
    }
    args.IfName = "lo" // ignore config, this only works for loopback
    err := ns.WithNetNSPath(args.Netns, func(ns.NetNS) error {
        link, err := netlink.LinkByName(args.IfName)
        if err != nil {
            return err // not tested
        }

        err = netlink.LinkSetDown(link)
        if err != nil {
            return err // not tested
        }

        return nil
    })
    if err != nil {
        //  if NetNs is passed down by the Cloud Orchestration Engine, or if it
called multiple times
        // so don't return an error if the device is already removed.
        // https://github.com/kubernetes/kubernetes/
issues/43014#issuecomment-287164444
```

```
        _, ok := err.(ns.NSPathNotExistErr)
        if ok {
            return nil
        }
        return err
    }

    return nil
}
```

The `main` function simply calls the `PluginMain()` function of the `skel` package, passing the command functions. The `skel` package will take care of running the CNI plugin executable and will invoke the `cmdAdd` and `delCmd` functions at the right time:

```
func main() {
    skel.PluginMain(cmdAdd, cmdCheck, cmdDel, version.All,
bv.BuildString("loopback"))
}
```

Building on the CNI plugin skeleton

Let's explore the `skel` package and see what it does under the covers. The `PluginMain()` entry point, is responsible for invoking `PluginMainWithError()`, catching errors, printing them to standard output, and exiting:

```
func PluginMain(cmdAdd, cmdCheck, cmdDel func(_ *CmdArgs) error, versionInfo
version.PluginInfo, about string) {
    if e := PluginMainWithError(cmdAdd, cmdCheck, cmdDel, versionInfo, about);
e != nil {
        if err := e.Print(); err != nil {
            log.Print("Error writing error JSON to stdout: ", err)
        }
        os.Exit(1)
    }
}
```

The `PluginErrorWithMain()` function instantiates a dispatcher, sets it up with all the I/O streams and the environment, and invokes its internal `pluginMain()` method:

```
func PluginMainWithError(cmdAdd, cmdCheck, cmdDel func(_ *CmdArgs) error,
versionInfo version.PluginInfo, about string) *types.Error {
    return (&dispatcher{
        Getenv: os.Getenv,
        Stdin:  os.Stdin,
        Stdout: os.Stdout,
```

```
        Stderr: os.Stderr,
    }).pluginMain(cmdAdd, cmdCheck, cmdDel, versionInfo, about)
}
```

Here, finally, is the main logic of the skeleton. It gets the `cmd` arguments from the environment (which includes the configuration from standard input), detects which `cmd` is invoked, and calls the appropriate plugin function (`cmdAdd` or `cmdDel`). It can also return version information:

```
func (t *dispatcher) pluginMain(cmdAdd, cmdCheck, cmdDel func(_ *CmdArgs)
error, versionInfo version.PluginInfo, about string) *types.Error {
    cmd, cmdArgs, err := t.getCmdArgsFromEnv()
    if err != nil {
        // Print the about string to stderr when no command is set
        if err.Code == types.ErrInvalidEnvironmentVariables && t.Getenv("CNI_
COMMAND") == "" && about != "" {
            _, _ = fmt.Fprintln(t.Stderr, about)
            _, _ = fmt.Fprintf(t.Stderr, "CNI protocol versions supported:
%s\n", strings.Join(versionInfo.SupportedVersions(), ", "))
            return nil
        }
        return err
    }

    if cmd != "VERSION" {
        if err = validateConfig(cmdArgs.StdinData); err != nil {
            return err
        }
        if err = utils.ValidateContainerID(cmdArgs.ContainerID); err != nil {
            return err
        }
        if err = utils.ValidateInterfaceName(cmdArgs.IfName); err != nil {
            return err
        }
    }

    switch cmd {
    case "ADD":
        err = t.checkVersionAndCall(cmdArgs, versionInfo, cmdAdd)
    case "CHECK":
        configVersion, err := t.ConfVersionDecoder.Decode(cmdArgs.StdinData)
        if err != nil {
            return types.NewError(types.ErrDecodingFailure, err.Error(), "")
        }
```

```
        if gtet, err := version.GreaterThanOrEqualTo(configVersion, "0.4.0");
err != nil {
            return types.NewError(types.ErrDecodingFailure, err.Error(), "")
        } else if !gtet {
            return types.NewError(types.ErrIncompatibleCNIVersion, "config
version does not allow CHECK", "")
        }
        for _, pluginVersion := range versionInfo.SupportedVersions() {
            gtet, err := version.GreaterThanOrEqualTo(pluginVersion,
configVersion)
            if err != nil {
                return types.NewError(types.ErrDecodingFailure, err.Error(),
"")
            } else if gtet {
                if err := t.checkVersionAndCall(cmdArgs, versionInfo,
cmdCheck); err != nil {
                    return err
                }
                return nil
            }
        }
        return types.NewError(types.ErrIncompatibleCNIVersion, "plugin version
does not allow CHECK", "")
    case "DEL":
        err = t.checkVersionAndCall(cmdArgs, versionInfo, cmdDel)
    case "VERSION":
        if err := versionInfo.Encode(t.Stdout); err != nil {
            return types.NewError(types.ErrIOFailure, err.Error(), "")
        }
    default:
        return types.NewError(types.ErrInvalidEnvironmentVariables, fmt.
Sprintf("unknown CNI_COMMAND: %v", cmd), "")
    }

    return err
}
```

The loopback plugin is one of the simplest CNI plugins. Let's check out the bridge plugin.

Reviewing the bridge plugin

The bridge plugin is more substantial. Let's look at some key parts of its implementation. The full source code is available here: `https://github.com/containernetworking/plugins/tree/main/plugins/main/bridge`.

The plugin defines in the `bridge.go` file a network configuration struct with the following fields:

```
type NetConf struct {
    types.NetConf
    BrName       string `json:"bridge"`
    IsGW         bool   `json:"isGateway"`
    IsDefaultGW  bool   `json:"isDefaultGateway"`
    ForceAddress bool   `json:"forceAddress"`
    IPMasq       bool   `json:"ipMasq"`
    MTU          int    `json:"mtu"`
    HairpinMode  bool   `json:"hairpinMode"`
    PromiscMode  bool   `json:"promiscMode"`
    Vlan         int    `json:"vlan"`
    MacSpoofChk  bool   `json:"macspoofchk,omitempty"`
    EnableDad    bool   `json:"enabledad,omitempty"`

    Args struct {
        Cni BridgeArgs `json:"cni,omitempty"`
    } `json:"args,omitempty"`
    RuntimeConfig struct {
        Mac string `json:"mac,omitempty"`
    } `json:"runtimeConfig,omitempty"`

    mac string
}
```

We will not cover what each parameter does and how it interacts with the other parameters due to space limitations. The goal is to understand the flow and have a starting point if you want to implement your own CNI plugin. The configuration is loaded from JSON via the `loadNetConf()` function. It is called at the beginning of the `cmdAdd()` and `cmdDel()` functions:

```
n, cniVersion, err := loadNetConf(args.StdinData, args.Args)
```

Here is the core of the `cmdAdd()` that uses information from network configuration, sets up the bridge, and sets up a veth:

```
br, brInterface, err := setupBridge(n)
    if err != nil {
        return err
    }

    netns, err := ns.GetNS(args.Netns)
    if err != nil {
        return fmt.Errorf("failed to open netns %q: %v", args.Netns, err)
```

```
    }
    defer netns.Close()

    hostInterface, containerInterface, err := setupVeth(netns, br, args.IfName,
n.MTU, n.HairpinMode, n.Vlan)
    if err != nil {
        return err
    }
```

Later, the function handles the L3 mode with its multiple cases:

```
    // Assume L2 interface only
    result := &current.Result{
        CNIVersion: current.ImplementedSpecVersion,
        Interfaces: []*current.Interface{
            brInterface,
            hostInterface,
            containerInterface,
        },
    }

    if n.MacSpoofChk {
        ...
    }

    if isLayer3 {
        // run the IPAM plugin and get back the config to apply
        r, err := ipam.ExecAdd(n.IPAM.Type, args.StdinData)
        if err != nil {
            return err
        }

        // release IP in case of failure
        defer func() {
            if !success {
                ipam.ExecDel(n.IPAM.Type, args.StdinData)
            }
        }()

        // Convert whatever the IPAM result was into the current Result type
        ipamResult, err := current.NewResultFromResult(r)
        if err != nil {
```

```
            return err
        }

        result.IPs = ipamResult.IPs
        result.Routes = ipamResult.Routes
        result.DNS = ipamResult.DNS

        if len(result.IPs) == 0 {
            return errors.New("IPAM plugin returned missing IP config")
        }

        // Gather gateway information for each IP family
        gwsV4, gwsV6, err := calcGateways(result, n)
        if err != nil {
            return err
        }

        // Configure the container hardware address and IP address(es)
        if err := netns.Do(func(_ ns.NetNS) error {
            ...
        }

        // check bridge port state
        retries := []int{0, 50, 500, 1000, 1000}
        for idx, sleep := range retries {
            ...
        }

        if n.IsGW {
            ...
        }

        if n.IPMasq {
            ...
        }
    } else {
        ...
    }
```

Finally, it updates the MAC address that may have changed and returns the results:

```
    // Refetch the bridge since its MAC address may change when the first
    // veth is added or after its IP address is set
    br, err = bridgeByName(n.BrName)
    if err != nil {
        return err
    }
    brInterface.Mac = br.Attrs().HardwareAddr.String()

    // Return an error requested by testcases, if any
    if debugPostIPAMError != nil {
        return debugPostIPAMError
    }

    // Use incoming DNS settings if provided, otherwise use the
    // settings that were already configued by the IPAM plugin
    if dnsConfSet(n.DNS) {
        result.DNS = n.DNS
    }

    success = true

    return types.PrintResult(result, cniVersion)
```

This is just part of the full implementation. There is also route setting and hardware IP allocation. If you plan to write your own CNI plugin, I encourage you to pursue the full source code, which is quite extensive, to get the full picture: `https://github.com/containernetworking/plugins/tree/main/plugins/main/bridge`.

Let's summarize what we have learned.

Summary

In this chapter, we covered a lot of ground. Networking is such a vast topic as there are so many combinations of hardware, software, operating environments, and user skills. It is a very complicated endeavor to come up with a comprehensive networking solution that is both robust, secure, performs well, and is easy to maintain. For Kubernetes clusters, the cloud providers mostly solve these issues. But if you run on-premises clusters or need a tailor-made solution, you get a lot of options to choose from. Kubernetes is a very flexible platform, designed for extension. Networking in particular is highly pluggable.

The main topics we discussed were the Kubernetes networking model (a flat address space where pods can reach other), how lookup and discovery work, the Kubernetes network plugins, various networking solutions at different levels of abstraction (a lot of interesting variations), using network policies effectively to control the traffic inside the cluster, ingress and Gateway APIs, the spectrum of load balancing solutions, and, finally, we looked at how to write a CNI plugin by dissecting a real-world implementation.

At this point, you are probably overwhelmed, especially if you're not a subject matter expert. However, you should have a solid grasp of the internals of Kubernetes networking, be aware of all the interlocking pieces required to implement a full-fledged solution, and be able to craft your own solution based on trade-offs that make sense for your system and your skill level.

In *Chapter 11*, *Running Kubernetes on Multiple Clusters*, we will go even bigger and look at running Kubernetes on multiple clusters with federation. This is an important part of the Kubernetes story for geo-distributed deployments and ultimate scalability. Federated Kubernetes clusters can exceed local limitations, but they bring a whole slew of challenges too.

11

Running Kubernetes on Multiple Clusters

In this chapter, we'll take it to the next level and consider options for running Kubernetes and deploying workloads on multiple clouds and multiple clusters. Since a single Kubernetes cluster has limits, once you exceed these limits you must run multiple clusters. A typical Kubernetes cluster is a closely-knit unit where all the components run in relative proximity and are connected by a fast network (typically, a physical data center or cloud provider availability zone). This is great for many use cases, but there are several important use cases where systems need to scale beyond a single cluster or a cluster needs to be stretched across multiple availability zones.

This is a very active area in Kubernetes these days. In the previous edition of the book, this chapter covered Kubernetes Federation and Gardener. Since then, the Kubernetes Federation project was abandoned. There are now many projects that provide different flavors of multi-cluster solutions, such as direct management, Virtual Kubelet solutions, and the gardener.cloud project, which is pretty unique.

The topics we will cover include the following:

- Stretched clusters vs multiple clusters
- The history of cluster federation in Kubernetes
- Cluster API
- Karmada
- Clusternet
- Clusterpedia
- Open Cluster Management
- Virtual Kubelet solutions
- Introduction to the Gardener project

Stretched Kubernetes clusters versus multi-cluster Kubernetes

There are several reasons to run multiple Kubernetes clusters:

- You want redundancy in case the geographical zone your cluster runs in has some issues
- You need more nodes or pods than a single Kubernetes cluster supports
- You want to isolate workloads across different clusters for security reasons

For the first reason it is possible to use a stretched cluster; for the other reasons, you must run multiple clusters.

Understanding stretched Kubernetes clusters

A stretched cluster (AKA wide cluster) is a single Kubernetes cluster where the control plane nodes and the work nodes are provisioned across multiple geographical availability zones or regions. Cloud providers offer this model for HA-managed Kubernetes clusters.

Pros of a stretched cluster

There are several benefits to the stretched cluster model:

- Your cluster, with proper redundancy, is protected from data center failures as a **SPOF** (**single point of failure**)
- The simplicity of operating against a single Kubernetes cluster is a huge win (logging, metrics, and upgrades)
- When you run your own unmanaged stretched cluster you can extend it transparently to additional locations (on-prem, edge, and other cloud providers)

Cons of a stretched cluster

However, the stretched model has its downsides too:

- You can't exceed the limits of a single Kubernetes cluster
- Degraded performance due to cross-zone networking
- On the cloud cross-zone, networking costs might be substantial
- Cluster upgrades are an all-or-nothing affair

In short, it's good to have the option for stretched clusters, but be prepared to switch to the multi-cluster model if some of the downsides are unacceptable.

Understanding multi-cluster Kubernetes

Multi-cluster Kubernetes means provisioning multiple Kubernetes clusters. Large-scale systems often can't be deployed on a single cluster for various reasons mentioned earlier. That means you need to provision multiple Kubernetes clusters and then figure out how to deploy your workloads on all these clusters and how to handle various use cases, such as some clusters being unavailable or having degraded performance. There are many more degrees of freedom.

Pros of multi-cluster Kubernetes

Here are some of the benefits of the multi-cluster model:

- Scale your system arbitrarily – there are no inherent limits on the number of clusters
- Provide cluster-level isolation to sensitive workloads at the RBAC level
- Utilize multiple cloud providers without incurring excessive costs (as long as most traffic remains within the same cloud provider region)
- Upgrade and perform incremental operations, even for cluster-wide operations

Cons of multi-cluster Kubernetes

However, there are some non-trivial downsides to the multi-cluster level:

- The very high complexity of provisioning and managing a fleet of clusters
- Need to figure out how to connect all the clusters
- Need to figure out how to store and provide access to data across all the clusters
- A lot of options to shoot yourself in the foot when designing multi-cluster deployments
- Need to work hard to provide centralized observability across all clusters

There are solutions out there for some of these problems, but at this point in time, there is no clear winner you can just adopt and easily configure for your needs. Instead, you will need to adapt and solve problems depending on the specific issues raised with your organization's multi-cluster structure.

The history of cluster federation in Kubernetes

In the previous editions of the book, we discussed Kubernetes Cluster Federation as a solution to managing multiple Kubernetes clusters as a single conceptual cluster. Unfortunately, this project has been inactive since 2019, and the Kubernetes multi-cluster **Special Interest Group** (**SIG**) is considering archiving it. Before we describe more modern approaches let's get some historical context. It's funny to talk about the history of a project like Kubernetes that didn't even exist before 2014, but the pace of development and the large number of contributors took Kubernetes through an accelerated evolution. This is especially relevant for the Kubernetes Federation.

In March 2015, the first revision of the Kubernetes Cluster Federation proposal was published. It was fondly nicknamed "Ubernetes" back then. The basic idea was to reuse the existing Kubernetes APIs to manage multiple clusters. This proposal, now called Federation V1, went through several rounds of revision and implementation but never reached general availability, and the main repo has been retired: `https://github.com/kubernetes-retired/federation`.

The SIG multi-cluster workgroup realized that the multi-cluster problem is more complicated than initially perceived. There are many ways to skin this particular cat and there is no one-size-fits-all solution. The new direction for cluster federation was to use dedicated APIs for federation. A new project and a set of tools were created and implemented as Kubernetes Federation V2: `https://github.com/kubernetes-sigs/kubefed`.

Unfortunately, this didn't take off either, and the consensus of the multi-cluster SIG is that since the project is not being maintained, it needs to be archived.

See the notes for the meeting from 2022-08-09: `https://tinyurl.com/sig-multicluster-notes`.

There are a lot of projects out there moving fast to try to solve the multi-cluster problem, and they all operate at different levels. Let's look at some of the prominent ones. The goal here is just to introduce these projects and what makes them unique. It is beyond the scope of this chapter to fully explore each one. However, we will dive deeper into one of the projects – the Cluster API – in *Chapter 17*, *Running Kubernetes in Production*.

Cluster API

The Cluster API (AKA CAPI) is a project from the Cluster Lifecycle SIG. Its goal is to make provisioning, upgrading, and operating multiple Kubernetes clusters easy. It supports both kubeadm-based clusters as well as managed clusters via dedicated providers. It has a cool logo inspired by the famous "It's turtles all the way down" story. The idea is that the Cluster API uses Kubernetes to manage Kubernetes clusters.

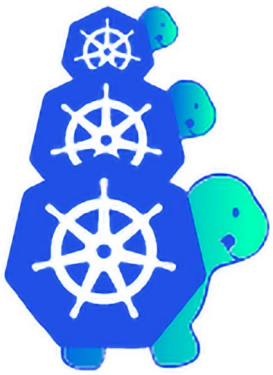


Figure 11.1: The Cluster API logo

Cluster API architecture

The Cluster API has a very clean and extensible architecture. The primary components are:

- The management cluster
- The work cluster
- The bootstrap provider
- The infrastructure provider
- The control plane
- Custom resources

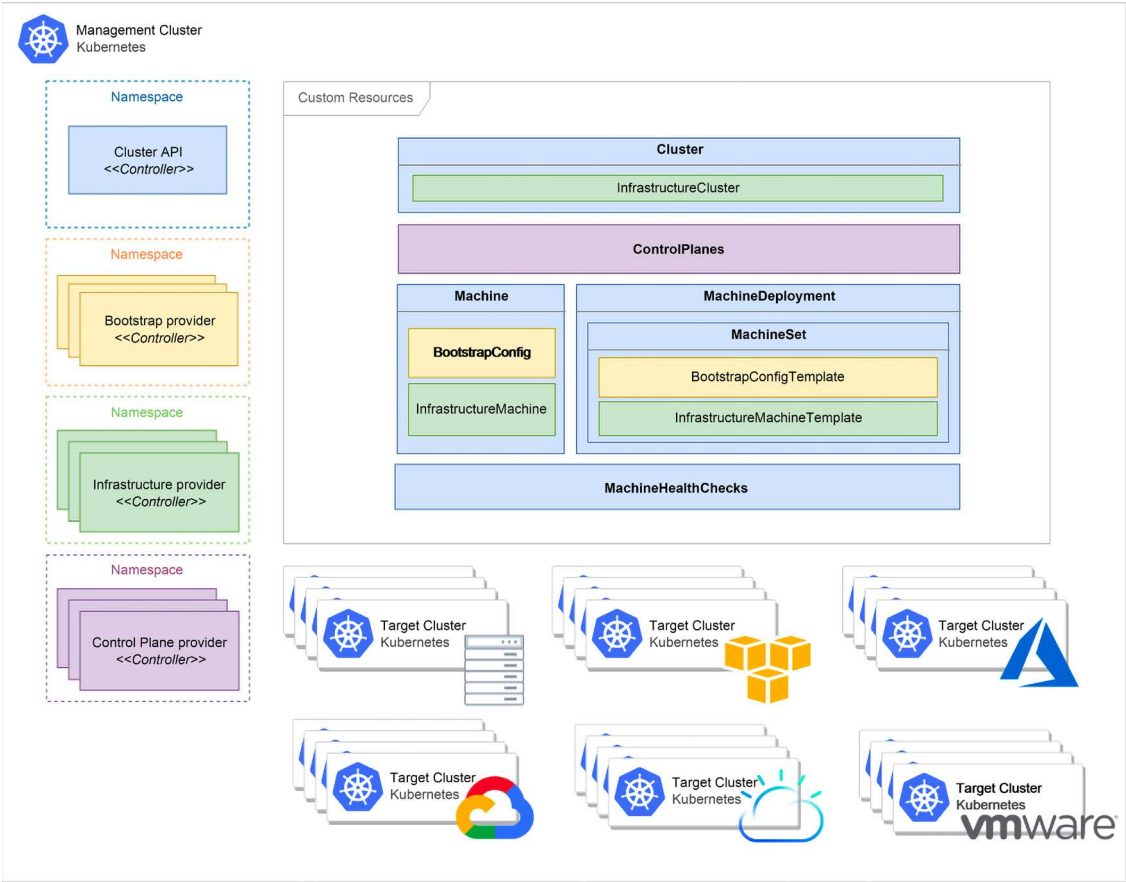


Figure 11.2: Cluster API architecture

Let's understand the role of each one of these components and how they interact with each other.

Management cluster

The management cluster is a Kubernetes cluster that is responsible for managing other Kubernetes clusters (work clusters). It runs the Cluster API control plane and providers, and it hosts the Cluster API custom resources that represent the other clusters.

The clusterctl CLI can be used to work with the management cluster. The clusterctl CLI is a command-line tool with a lot of commands and options, if you want to experiment with the Cluster API through its CLI, visit `https://cluster-api.sigs.k8s.io/clusterctl/overview.html`.

Work cluster

A work cluster is just a regular Kubernetes cluster. These are the clusters that developers use to deploy their workloads. The work clusters don't need to be aware that they are managed by the Cluster API.

Bootstrap provider

When CAPI creates a new Kubernetes cluster, it needs certificates before it can create the work cluster's control plane and, finally, the worker nodes. This is the job of the bootstrap provider. It ensures all the requirements are met and eventually joins the worker nodes to the control plane.

Infrastructure provider

The infrastructure provider is a pluggable component that allows CAPI to work in different infrastructure environments, such as cloud providers or bare-metal infrastructure providers. The infrastructure provider implements a set of interfaces as defined by CAPI to provide access to compute and network resources.

Check out the current providers' list here: `https://cluster-api.sigs.k8s.io/reference/providers.html`.

Control plane

The control plane of a Kubernetes cluster consists of the API server, the etcd stat store, the scheduler, and the controllers that run the control loops to reconcile the resources in the cluster. The control plane of the work clusters can be provisioned in various ways. CAPI supports the following modes:

- Machine-based – the control plane components are deployed as static pods on dedicated machines
- Pod-based – the control plane components are deployed via `Deployments` and `StatefulSet`, and the API server is exposed as a `Service`
- External – the control plane is provisioned and managed by an external provider (typically, a cloud provider)

Custom resources

The custom resources represent the Kubernetes clusters and machines managed by CAPI as well as additional auxiliary resources. There are a lot of custom resources, and some of them are still considered experimental. The primary CRDs are:

- `Cluster`
- `ControlPlane` (represents control plane machines)
- `MachineSet` (represents worker machines)
- `MachineDeployment`
- `Machine`
- `MachineHealthCheck`

Some of these generic resources have references to corresponding resources offered by the infrastructure provider.

The following diagram illustrates the relationships between the control plane resources that represent the clusters and machine sets:

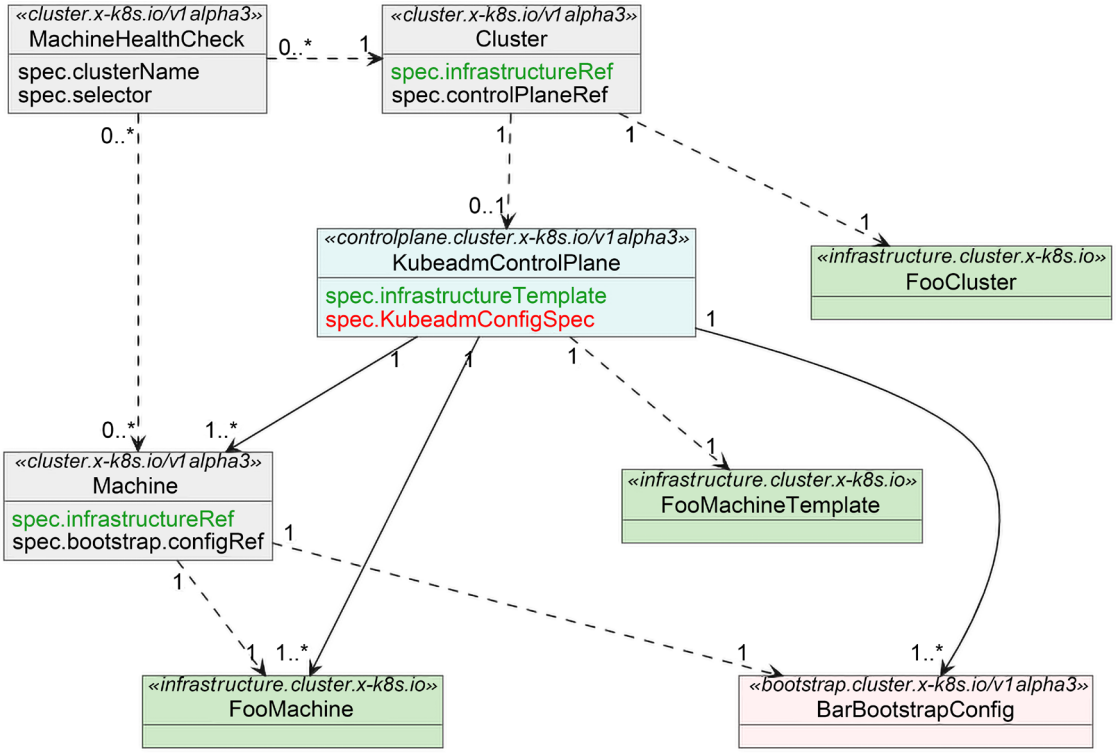


Figure 11.3: Cluster API control plane resources

CAPI also has an additional set of experimental resources that represent a managed cloud provider environment:

- `MachinePool`
- `ClusterResourceSet`
- `ClusterClass`

See `https://github.com/kubernetes-sigs/cluster-api` for more details.

Karmada

Karmada is a CNCF sandbox project that focuses on deploying and running workloads across multiple Kubernetes clusters. Its claim to fame is that you don't need to make changes to your application configuration. While CAPI was focused on the lifecycle management of clusters, Karmada picks up when you already have a set of Kubernetes clusters and you want to deploy workloads across all of them. Conceptually, Karmada is a modern take on the abandoned Kubernetes Federation project.

It can work with Kubernetes in the cloud, on-prem, and on the edge.

See `https://github.com/karmada-io/karmada`.

Let's look at Karmada's architecture.

Karmada architecture

Karmada is heavily inspired by Kubernetes. It provides a multi-cluster control plane with similar components to the Kubernetes control plane:

- Karmada API server
- Karmada controller manager
- Karmada scheduler

If you understand how Kubernetes works, then it is pretty easy to understand how Karmada extends it 1:1 to multiple clusters.

The following diagram illustrates the Karmada architecture:

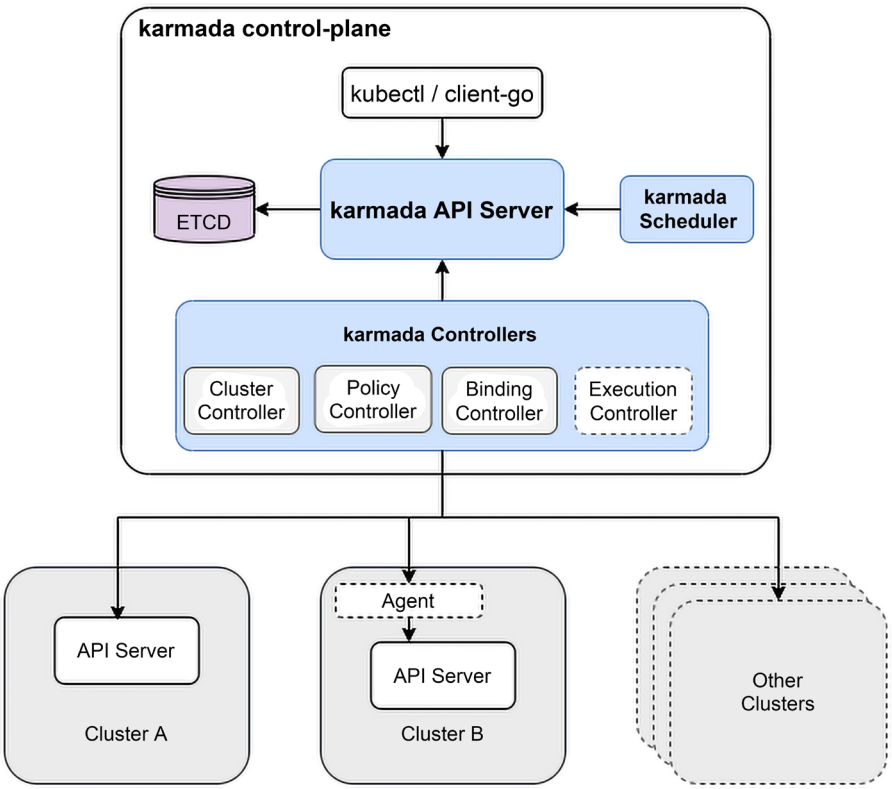


Figure 11.4: Karmada architecture

Karmada concepts

Karmada is centered around several concepts implemented as Kubernetes CRDs. You define and update your applications and services using these concepts and Karmada ensures that your workloads are deployed and run in the right place across your multi-cluster system.

Let's look at these concepts.

ResourceTemplate

The resource template looks just like a regular Kubernetes resource such as `Deployment` or `StatefulSet`, but it doesn't actually get deployed to the Karmada control plane. It only serves as a blueprint that will eventually be deployed to member clusters.

PropagationPolicy

The propagation policy determines where a resource template should be deployed. Here is a simple propagation policy that will place the `nginx` Deployment into two clusters, called `member1` and `member2`:

```
apiVersion: policy.karmada.io/v1alpha1
kind: PropagationPolicy
metadata:
  name: cool-policy
spec:
  resourceSelectors:
    - apiVersion: apps/v1
      kind: Deployment
      name: nginx
  placement:
    clusterAffinity:
      clusterNames:
        - member1
        - member2
```

OverridePolicy

Propagation policies operate across multiple clusters, but sometimes, there are exceptions. The override policy lets you apply fine-grained rules to override existing propagation policies. There are several types of rules:

- `ImageOverrider`: Dedicated to overriding images for workloads
- `CommandOverrider`: Dedicated to overriding commands for workloads
- `ArgsOverrider`: Dedicated to overriding args for workloads
- `PlaintextOverrider`: A general-purpose tool to override any kind of resources

Additional capabilities

There is much more to Karmada, such as:

- Multi-cluster de-scheduling
- Re-scheduling
- Multi-cluster failover
- Multi-cluster service discovery

Check the Karmada documentation for more details: `https://karmada.io/docs/`.

Clusternet

Clusternet is an interesting project. It is centered around the idea of managing multiple Kubernetes clusters as "visiting the internet" (hence the name "Clusternet"). It supports cloud-based, on-prem, edge, and hybrid clusters. The core features of Clusternet are:

- Kubernetes multi-cluster management and governance
- Application coordination
- A CLI via the kubectl plugin
- Programmatic access via a wrapper to the Kubernetes Client-Go library

Clusternet architecture

The Clusternet architecture is similar to Karmada but simpler. There is a parent cluster that runs the Clusternet hub and Clusternet scheduler. On each child cluster, there is a Clusternet agent. The following diagram illustrates the structure and interactions between the components:

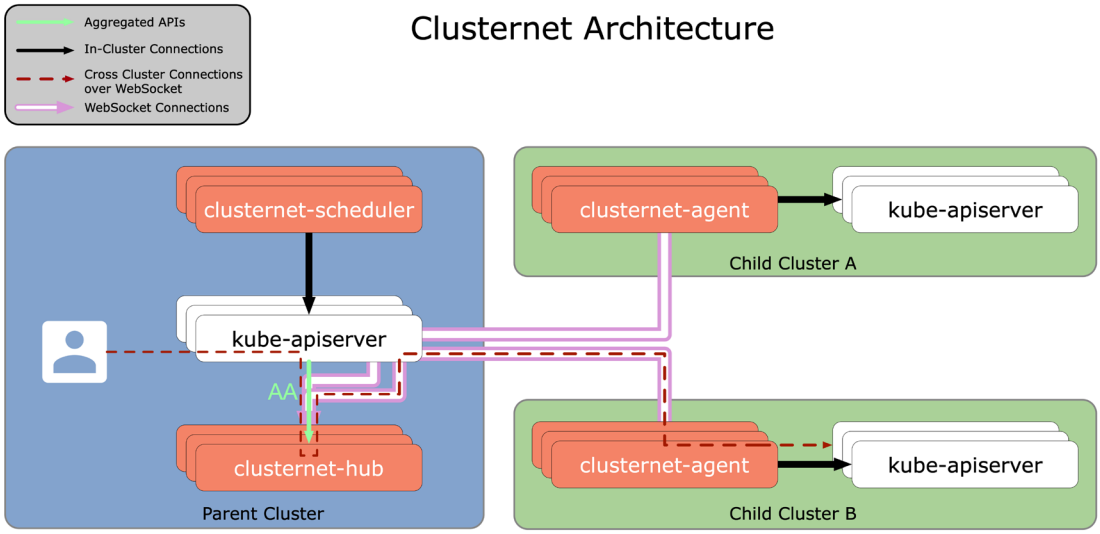


Figure 11.5: Clusternet architecture

Clusternet hub

The hub has multiple roles. It is responsible for approving cluster registration requests and creating namespaces, service accounts, and RBAC resources for all child clusters. It also serves as an aggregated API server that maintains WebSocket connections to the agent on child clusters. The hub also provides a Kubernetes-like API to proxy requests to each child cluster. Last but not least, the hub coordinates the deployment of applications and their dependencies to multiple clusters from a single set of resources.

Clusternet scheduler

The Clusternet scheduler is the component that is responsible for ensuring that resources (called feeds in Clusternet terminology) are deployed and balanced across all the child clusters according to policies called `SchedulingStrategy`.

Clusternet agent

The Clusternet agent runs on every child cluster and communicates with the hub. The agent on a child cluster is the equivalent of the kubelet on a node. It has several roles. The agent registers its child cluster with the parent cluster. The agent provides a heartbeat to the hub that includes a lot of information, such as the Kubernetes version, running platform, health, readiness, and liveness of workloads. The agent also sets up the WebSocket connection to the hub on the parent cluster to allow full-duplex communication channels over a single TCP connection.

Multi-cluster deployment

Clusternet models multi-cluster deployment as subscriptions and feeds. It provides a `Subscription` custom resource that can be used to deploy a set of resources (called feeds) to multiple clusters (called subscribers) based on different criteria. Here is an example of a `Subscription` that deploys a `Namespace`, a `Service`, and a `Deployment` to multiple clusters with a label of `clusters.clusternet.io/cluster-id`:

```
# examples/dynamic-dividing-scheduling/subscription.yaml
apiVersion: apps.clusternet.io/v1alpha1
kind: Subscription
metadata:
  name: dynamic-dividing-scheduling-demo
  namespace: default
spec:
  subscribers: # filter out a set of desired clusters
    - clusterAffinity:
```

```
        matchExpressions:
          - key: clusters.clusternet.io/cluster-id
            operator: Exists
  schedulingStrategy: Dividing
  dividingScheduling:
    type: Dynamic
    dynamicDividing:
      strategy: Spread # currently we only support Spread dividing strategy
  feeds: # defines all the resources to be deployed with
    - apiVersion: v1
      kind: Namespace
      name: qux
    - apiVersion: v1
      kind: Service
      name: my-nginx-svc
      namespace: qux
    - apiVersion: apps/v1 # with a total of 6 replicas
      kind: Deployment
      name: my-nginx
      namespace: qux
```

See `https://clusternet.io` for more details.

Clusterpedia

Clusterpedia is a CNCF sandbox project. Its central metaphor is Wikipedia for Kubernetes clusters. It has a lot of capabilities around multi-cluster search, filtering, field selection, and sorting. This is unusual because it is a read-only project. It doesn't offer to help with managing the clusters or deploying workloads. It is focused on observing your clusters.

Clusterpedia architecture

The architecture is similar to other multi-cluster projects. There is a control plane element that runs the Clusterpedia API server and ClusterSynchro manager components. For each observed cluster, there is a dedicated component called cluster syncro that synchronizes the state of the clusters into the storage layer of Clusterpedia. One of the most interesting aspects of the architecture is the Clusterpedia aggregated API server, which makes all your clusters seem like a single huge logical cluster. Note that the Clusterpedia API server and the ClusterSynchro manager are loosely coupled and don't interact directly with each other. They just read and write from a shared storage layer.

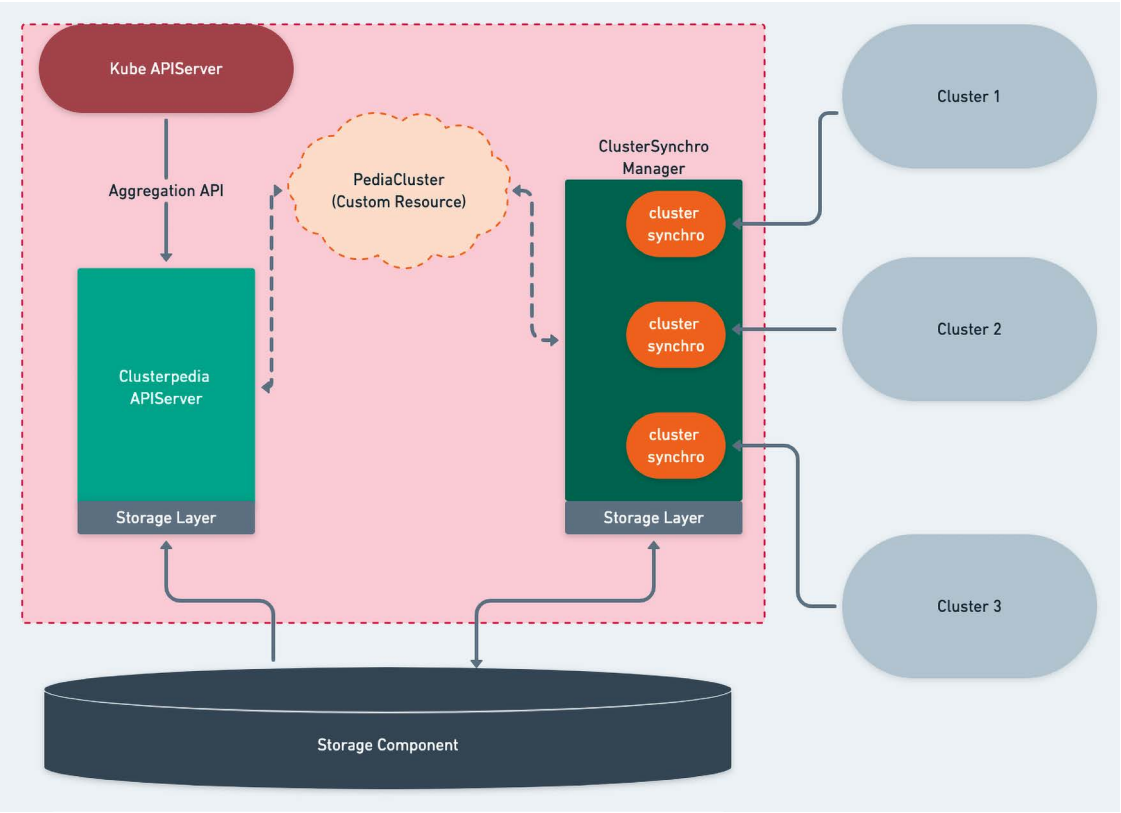


Figure 11.6: Clusterpedia architecture

Let's look at each of the components and understand what their purpose is.

Clusterpedia API server

The Clusterpedia API server is an aggregated API server. That means that it registers itself with the Kubernetes API server and, in practice, extends the standard Kubernetes API server via custom endpoints. When requests come to the Kubernetes API server, it forwards them to the Clusterpedia API server, which accesses the storage layer to satisfy them. The Kubernetes API server serves as a forwarding layer for the requests that Clusterpedia handles.

This is an advanced aspect of Kubernetes. We will discuss API server aggregation in *Chapter 15*, *Extending Kubernetes*.

ClusterSynchro manager

Clusterpedia observes multiple clusters to provide its search, filter, and aggregation features. One way to implement it is that whenever a request comes in, Clusterpedia would query all the observed clusters, collect the results, and return them. This approach is very problematic, as some clusters might be slow to respond and similar requests will require returning the same information, which is wasteful and costly. Instead, the ClusterSynchro manager collectively synchronizes the state of each observed cluster into Clusterpedia storage, where the Clusterpedia API server can respond quickly.

Storage layer

The storage layer is an abstraction layer that stores the state of all observed clusters. It provides a uniform interface that can be implemented by different storage components. The Clusterpedia API server and the ClusterSynchro manager interact with the storage layer interface and never talk to each other directly.

Storage component

The storage component is an actual data store that implements the storage layer interface and stores the state of observed clusters. Clusterpedia was designed to support different storage components to provide flexibility for their users. Currently, supported storage components include MySQL, Postgres, and Redis.

Importing clusters

To onboard clusters into Clusterpedia, you define a PediaCluster custom resource. It is pretty straightforward:

```
apiVersion: cluster.clusterpedia.io/v1alpha2
kind: PediaCluster
metadata:
  name: cluster-example
spec:
  apiserver: "https://10.30.43.43:6443"
  kubeconfig:
  caData:
  tokenData:
  certData:
  keyData:
  syncResources: []
```

You need to provide credentials to access the cluster, and then Clusterpedia will take over and sync its state.

Advanced multi-cluster search

This is where Clusterpedia shines. You can access the Clusterpedia cluster via an API or through kubectl. When accessing it through a URL it looks like you hit the aggregated API server endpoint:

```
kubectl get --raw="/apis/clusterpedia.io/v1beta1/resources/apis/apps/v1/
deployments?clusters=cluster-1,cluster-2"
```

You can specify the target cluster as a query parameter (in this case, `cluster-1` and `cluster-2`).

When accessing through kubectl, you specify the target clusters as a label (in this case, `"search.clusterpedia.io/clusters in (cluster-1,cluster-2)"`):

```
kubectl --cluster clusterpedia get deployments -l "search.clusterpedia.io/clusters in
(cluster-1,cluster-2)"
```

Other search labels and queries exist for namespaces and resource names:

- `search.clusterpedia.io/namespaces` (query parameter is `namespaces`)
- `search.clusterpedia.io/names` (query parameter is `names`)

There is also an experimental fuzzy search label, `internalstorage.clusterpedia.io/fuzzy-name`, for resource names, but no query parameter. This is useful as often, resources have generated names with random suffixes.

You can also search by creation time:

- `search.clusterpedia.io/before` (query parameter is `before`)
- `search.clusterpedia.io/since` (query parameter is `since`)

Other capabilities include filtering by resource labels or field selectors as well as organizing the results using `OrderBy` and `Paging`.

Resource collections

Another important concept is resource collections. The standard Kubernetes API offers a straightforward REST API where you can list or get one kind of resource at a time. However, often, users would like to get multiple types of resources at the same time. For example, the `Deployment`, `Service`, and `HorizontalPodAutoscaler` with a specific label. This requires multiple calls via the standard Kubernetes API, even if all these resources are available on one cluster.

Clusterpedia defines a `CollectionResource` that groups together resources that belong to the following categories:

- `any` (all resources)
- `workloads` (`Deployments`, `StatefulSets`, and `DaemonSets`)
- `kuberesources` (all resources other than workloads)

You can search for any combination of resources in one API call by passing API groups and resource kinds:

```
kubectl get --raw "/apis/clusterpedia.io/v1beta1/collectionresources/
any?onlyMetadata=true&groups=apps&resources=batch/jobs,batch/cronjobs"
```

See `https://github.com/clusterpedia-io/clusterpedia` for more details.

Open Cluster Management

Open Cluster Management (**OCM**) is a CNCF sandbox project for multi-cluster management, as well as multi-cluster scheduling and workload placement. Its claim to fame is closely following many Kubernetes concepts, extensibility via addons, and strong integration with other open source projects, such as:

- Submariner
- Clusternet (that we covered earlier)
- KubeVela

The scope of OCM covers cluster lifecycle, application lifecycle, and governance.

Let's look at OCM's architecture.

OCM architecture

OCM's architecture follows the hub and spokes model. It has a hub cluster, which is the OCM control plane that manages multiple other clusters (the spokes).

The control plane's hub cluster runs two controllers: the registration controller and the placement controller. In addition, the control plane runs multiple management addons, which are the foundation for OCM's extensibility. On each managed cluster, there is a so-called Klusterlet that has a registration-agent and work-agent that interact with the registration controller and placement controller on the hub cluster. Then, there are also addon agents that interact with the addons on the hub cluster.

The following diagram illustrates how the different components of OCM communicate:

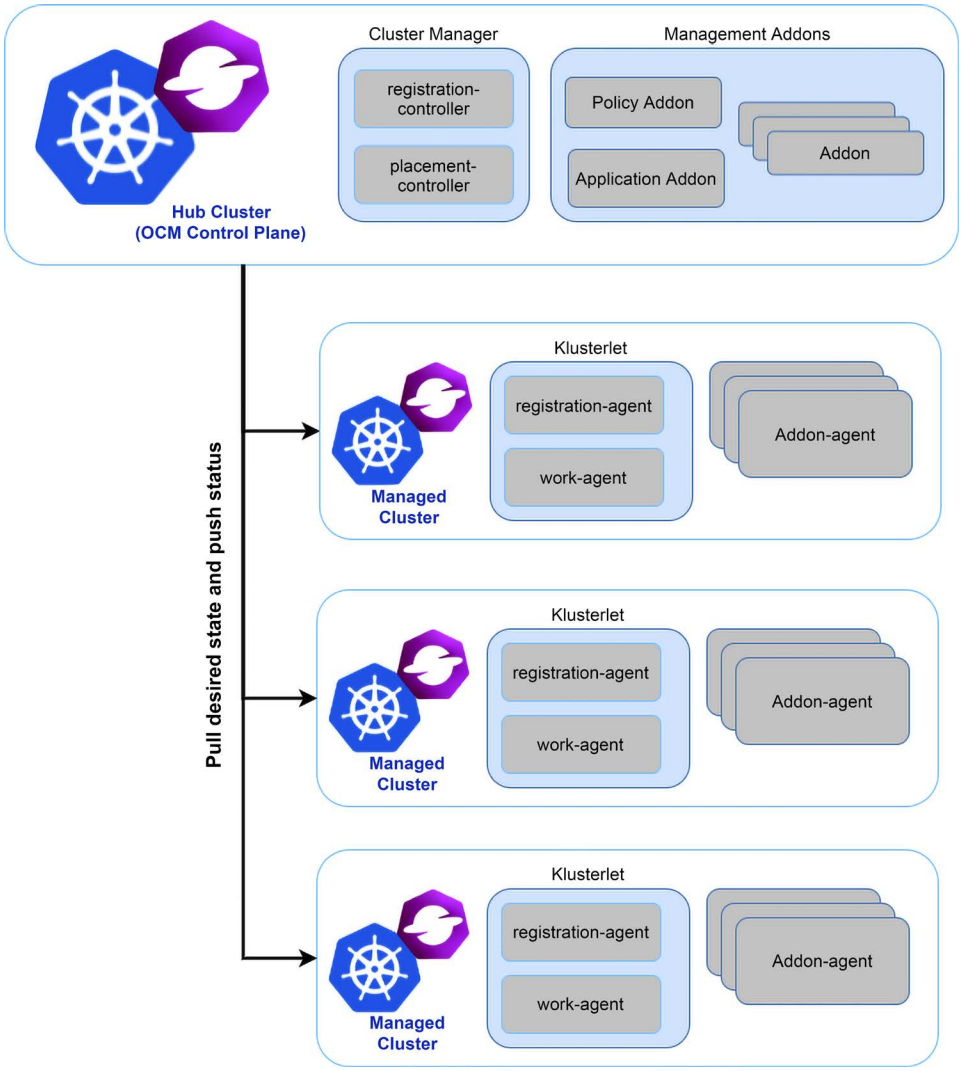


Figure 11.7: OCM architecture

Let's look at the different aspects of OCM.

OCM cluster lifecycle

Cluster registration is a big part of OCM's secure multi-cluster story. OCM prides itself on the secure double opt-in handshake registration. Since a hub-and-spoke cluster may have different administrators, this model provides protection for each side from undesired requests. Each side can terminate the relationship at any time.

The following diagram demonstrates the registration process (CSR means certificate signing request):

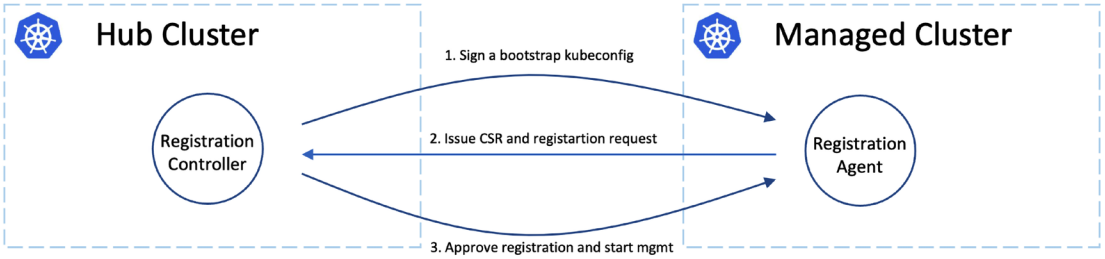


Figure 11.8: OCM registration process

OCM application lifecycle

The OCM application lifecycle supports creating, updating, and deleting resources across multiple clusters.

The primary building block is the `ManifestWork` custom resource that can define multiple resources. Here is an example that contains only a single `Deployment`:

```
apiVersion: work.open-cluster-management.io/v1
kind: ManifestWork
metadata:
  namespace: <target managed cluster>
  name: awesome-workload
spec:
  workload:
    manifests:
      - apiVersion: apps/v1
        kind: Deployment
        metadata:
          name: hello
          namespace: default
        spec:
          selector:
            matchLabels:
```

```
            app: hello
        template:
          metadata:
            labels:
              app: hello
          spec:
            containers:
              - name: hello
                image: quay.io/asmacdo/busybox
                command:
                  ["sh", "-c", 'echo "Hello, Kubernetes!" && sleep 3600']
```

The `ManifestWork` is created on the hub cluster and is deployed to the target cluster according to the namespace mapping. Each target cluster has a namespace representing it in the hub cluster. A work agent running on the target cluster will monitor all `ManifestWork` resources on the hub cluster in their namespace and sync changes.

OCM governance, risk, and compliance

OCM provides a governance model based on policies, policy templates, and policy controllers. The policies can be bound to a specific set of clusters for fine-grained control.

Here is a sample policy that requires the existence of a namespace called `Prod`:

```
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: policy-namespace
  namespace: policies
  annotations:
    policy.open-cluster-management.io/standards: NIST SP 800-53
    policy.open-cluster-management.io/categories: CM Configuration Management
    policy.open-cluster-management.io/controls: CM-2 Baseline Configuration
spec:
  remediationAction: enforce
  disabled: false
  policy-templates:
    - objectDefinition:
        apiVersion: policy.open-cluster-management.io/v1
        kind: ConfigurationPolicy
        metadata:
          name: policy-namespace-example
        spec:
          remediationAction: inform
```

```
          severity: low
          object-templates:
            - complianceType: MustHave
              objectDefinition:
                kind: Namespace # must have namespace 'prod'
                apiVersion: v1
                metadata:
                  name: prod
```

See `https://open-cluster-management.io/` for more details.

Virtual Kubelet

Virtual Kubelet is a fascinating project. It impersonates a kubelet to connect Kubernetes to other APIs such as AWS Fargate or Azure ACI. The Virtual Kubelet looks like a node to the Kubernetes cluster, but the compute resources backing it up are abstracted away. The Virtual Kubelet looks like just another node to the Kubernetes cluster:

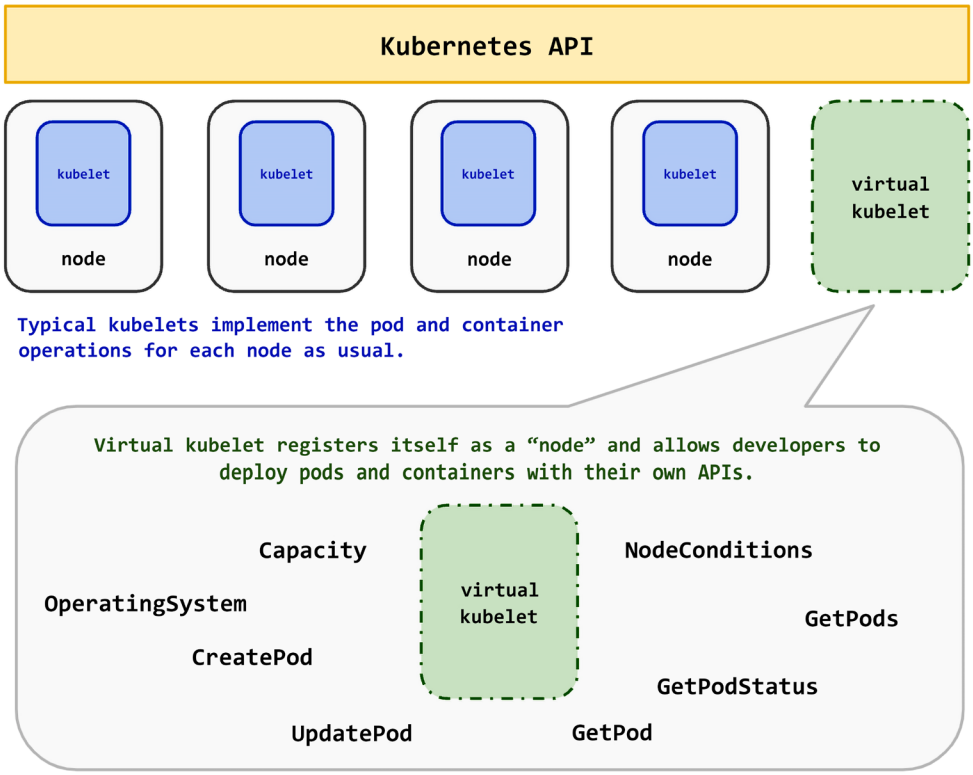


Figure 11.9: Virtual Kubelet, which looks like a regular node to the Kubernetes cluster

The features of the Virtual Kubelet are:

- Creating, updating, and deleting pods
- Accessing container logs and metrics
- Getting a pod, pods, and pod status
- Managing capacity
- Accessing node addresses, node capacity, and node daemon endpoints
- Choosing the operating system
- Supporting your own virtual network

See `https://github.com/virtual-kubelet/virtual-kubelet` for more details.

This concept can be used to connect multiple Kubernetes clusters too, and several projects follow this approach. Let's look briefly at some projects that use Virtual Kubelet for multi-cluster management such as tensile-kube, Admiralty, and Liqo.

Tensile-kube

Tensile-kube is a sub-project of the Virtual Kubelet organization on GitHub.

Tensile-kube brings the following to the table:

- Automatic discovery of cluster resources
- Async notification of pod modifications
- Full access to pod logs and kubectl exec
- Global scheduling of pods
- Re-scheduling of pods using descheduler
- PV/PVC
- Service

Tensile-kube uses the terminology of the upper cluster for the cluster that contains the Virtual Kubelets, and the lower clusters for the clusters that are exposed as virtual nodes in the upper cluster.

Here is the tensile-kube architecture:

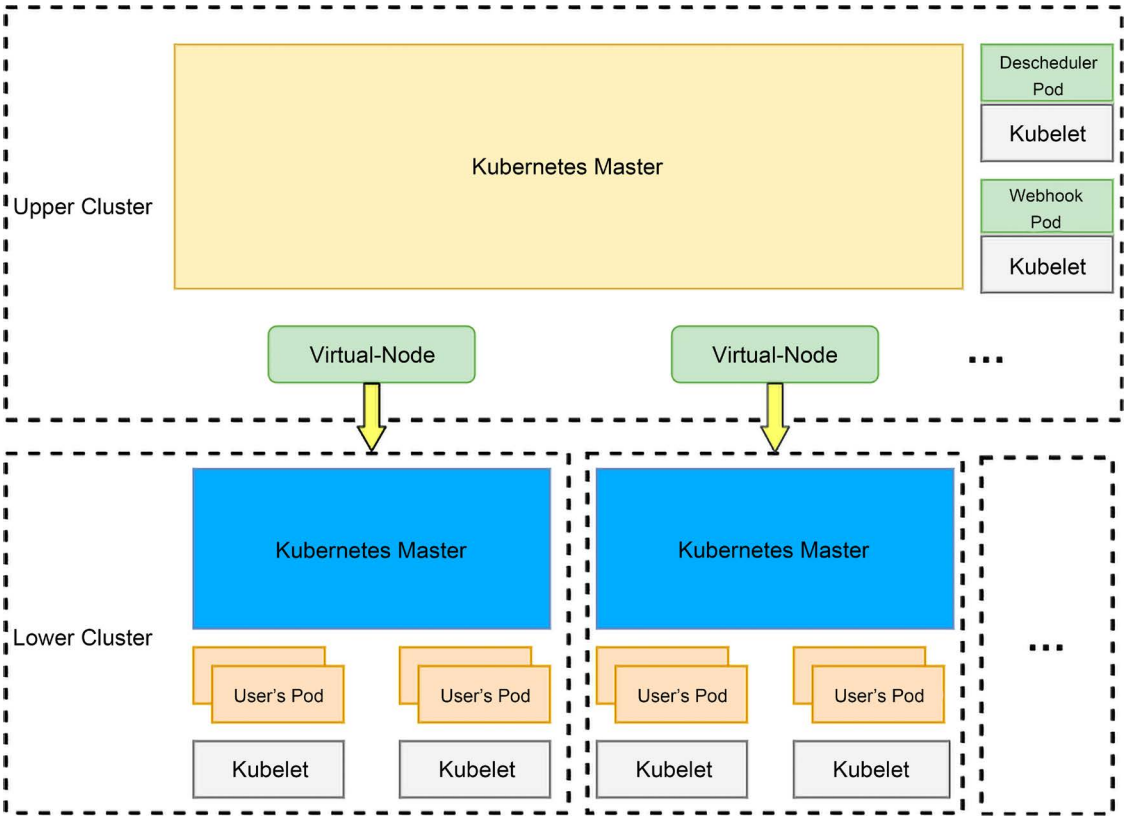


Figure 11.10: Tensile-kube architecture

See `https://github.com/virtual-kubelet/tensile-kube` for more details.

Admiralty

Admiralty is an open source project backed by a commercial company. Admiralty takes the Virtual Kubelet concept and builds a sophisticated solution for multi-cluster orchestration and scheduling. Target clusters are represented as virtual nodes in the source cluster. It has a pretty complicated architecture that involves three levels of scheduling. Whenever a pod is created on a proxy, pods are created on the source cluster, candidate pods are created on each target cluster, and eventually, one of the candidate pods is selected and becomes a delegate pod, which is a real pod that actually runs its containers. This is all supported by custom multi-cluster schedulers built on top of the Kubernetes scheduling framework. To schedule workloads on Admiralty, you need to annotate any pod template with `multicluster.admiralty.io/elect=""` and Admiralty will take it from there.

Here is a diagram that demonstrates the interplay between different components:

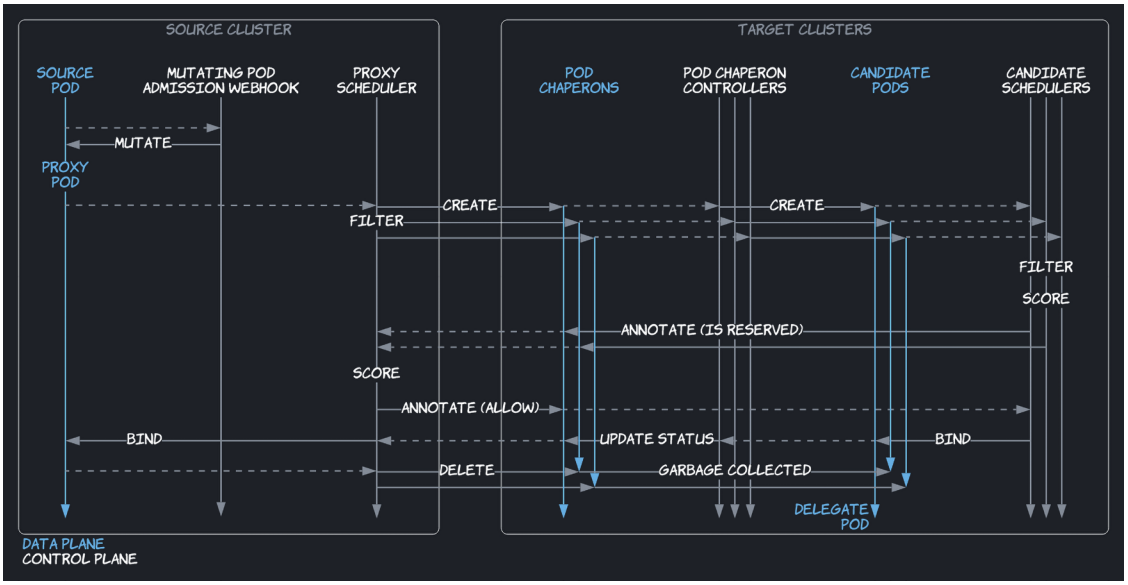


Figure 11.11: Admiralty architecture

Admiralty provides the following features:

- Highly available
- Live disaster recovery
- Dynamic **CDN** (**content delivery network**)
- Multi-cluster workflows
- Support for edge computing, IoT, and 5G
- Governance
- Cluster upgrades
- Clusters as cattle abstraction
- Global resource federation
- Cloud bursting and arbitrage

See `https://admiralty.io` for more details.

Liqo

Liqo is an open source project based on the liquid computing concept. Let your tasks and data float around and find the best place to run. Its scope is very impressive, as it targets not only the compute aspect of running pods across multiple clusters but also provides network fabric and storage fabric. These aspects of connecting clusters and managing data across clusters are often harder problems to solve than just running workloads.

In Liqo's terminology, the management cluster is called the home cluster and the target clusters are called foreign clusters. The virtual nodes in the home cluster are called "Big" nodes, and they represent the foreign clusters.

Liqo utilizes IP address mapping to achieve a flat IP address space across all foreign clusters that may have internal IP conflicts.

Liqo filters and batches events from the foreign clusters to reduce pressure on the home cluster.

Here is a diagram of the Liqo architecture:

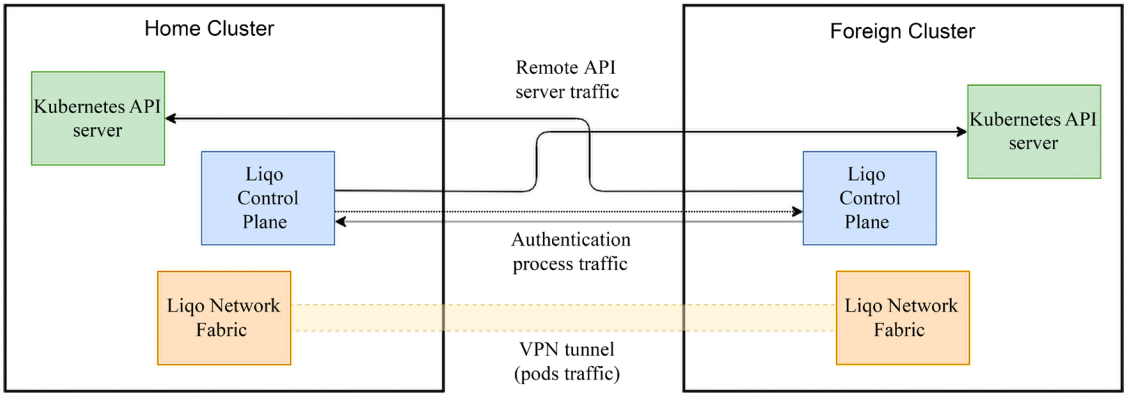


Figure 11.12: Liqo architecture

See `https://liqo.io` for more details.

Let's move on and take an in-depth look at the Gardener project, which takes a different approach.

Introducing the Gardener project

The Gardener project is an open source project developed by SAP. It lets you manage thousands (yes, thousands!) of Kubernetes clusters efficiently and economically. Gardener solves a very complex problem, and the solution is elegant but not simple. Gardener is the only project that addresses both the cluster lifecycle and application lifecycle.

In this section, we will cover the terminology of Gardener and its conceptual model, dive deep into its architecture, and learn about its extensibility features. The primary theme of Gardener is to use Kubernetes to manage Kubernetes clusters. A good way to think about Gardener is Kubernetes-control-plane-as-a-service.

See `https://gardener.cloud` for more details.

Understanding the terminology of Gardener

The Gardener project, as you may have guessed, uses botanical terminology to describe the world. There is a garden, which is a Kubernetes cluster responsible for managing seed clusters. A seed is a Kubernetes cluster responsible for managing a set of shoot clusters. A shoot cluster is a Kubernetes cluster that runs actual workloads.

The cool idea behind Gardener is that the shoot clusters contain only the worker nodes. The control planes of all the shoot clusters run as Kubernetes pods and services in the seed cluster.

The following diagram describes in detail the structure of Gardener and the relationships between its components:

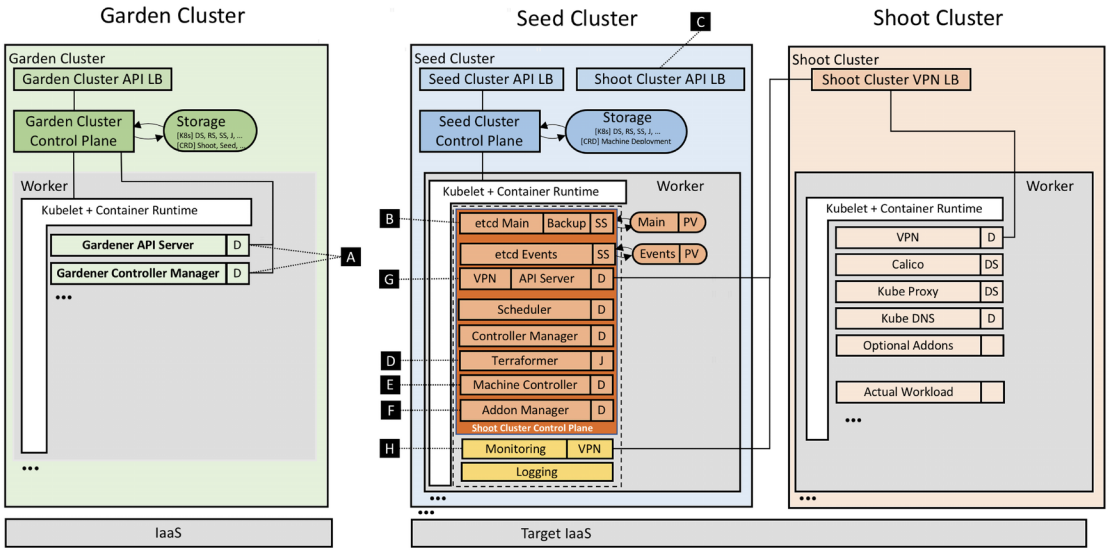


Figure 11.13: The Gardener project structure

Don't panic! Underlying all this complexity is a crystal clear conceptual model.

Understanding the conceptual model of Gardener

The architecture diagram of Gardener can be overwhelming. Let's unpack it slowly and surface the underlying principles. Gardener really embraces the spirit of Kubernetes and offloads a lot of the complexity of managing a large set of Kubernetes clusters to Kubernetes itself. At its heart, Gardener is an aggregated API server that manages a set of custom resources using various controllers. It embraces and takes full advantage of Kubernetes' extensibility. This approach is common in the Kubernetes community. Define a set of custom resources and let Kubernetes manage them for you. The novelty of Gardener is that it takes this approach to the extreme and abstracts away parts of Kubernetes infrastructure itself.

In a "normal" Kubernetes cluster, the control plane runs in the same cluster as the worker nodes. Typically, in large clusters, control plane components like the Kubernetes API server and etcd run on dedicated nodes and don't mix up with the worker nodes. Gardener thinks in terms of many clusters and it takes all the control planes of all the shoot clusters and has a seed cluster to manage them. So the Kubernetes control plane of the shoot clusters is managed in the seed cluster as regular Kubernetes `Deployments`, which automatically provides replication, monitoring, self-healing, and rolling updates by Kubernetes.

So, the control plane of a Kubernetes shoot cluster is analogous to a `Deployment`. The seed cluster, on the other hand, maps to a Kubernetes node. It manages multiple shoot clusters. It is recommended to have a seed cluster per cloud provider. The Gardener developers actually work on a gardenlet controller for seed clusters that is similar to the kubelet on nodes.

If the seed clusters are like Kubernetes nodes, then the Garden cluster that manages those seed clusters is like a Kubernetes cluster that manages its worker nodes.

By pushing the Kubernetes model this far, the Gardener project leverages the strengths of Kubernetes to achieve robustness and performance that would be very difficult to build from scratch.

Let's dive into the architecture.

Diving into the Gardener architecture

Gardener creates a Kubernetes namespace in the seed cluster for each shoot cluster. It manages the certificates of the shoot clusters as Kubernetes secrets in the seed cluster.

Managing the cluster state

The etcd data store for each cluster is deployed as a StatefulSet with one replica. In addition, events are stored in a separate etcd instance. The etcd data is periodically snapshotted and stored in remote storage for backup and restore purposes. This enables very fast recovery of clusters that lost their control plane (e.g., when an entire seed cluster becomes unreachable). Note that when a seed cluster goes down, the shoot cluster continues to run as usual.

Managing the control plane

As mentioned before, the control plane of a shoot cluster X runs in a separate seed cluster, while the worker nodes run in a shoot cluster. This means that pods in the shoot cluster can use internal DNS to locate each other, but communication to the Kubernetes API server running in the seed cluster must be done through an external DNS. This means the Kubernetes API server runs as a `Service` of the `LoadBalancer` type.

Preparing the infrastructure

When creating a new shoot cluster, it's important to provide the necessary infrastructure. Gardener uses Terraform for this task. A Terraform script is dynamically generated based on the shoot cluster specification and stored as a ConfigMap within the seed cluster. To facilitate this process, a dedicated component (Terraformer) runs as a job, performs all the provisioning, and then writes the state into a separate ConfigMap.

Using the Machine controller manager

To provision nodes in a provider-agnostic manner that can work for private clouds too, Gardener has several custom resources such as `MachineDeployment`, `MachineClass`, `MachineSet`, and `Machine`. They work with the Kubernetes Cluster Lifecycle group to unify their abstractions because there is a lot of overlap. In addition, Gardener takes advantage of the cluster auto-scaler to offload the complexity of scaling node pools up and down.

Networking across clusters

The seed cluster and shoot clusters can run on different cloud providers. The worker nodes in the shoot clusters are often deployed in private networks. Since the control plane needs to interact closely with the worker nodes (mostly the kubelet), Gardener creates a VPN for direct communication.

Monitoring clusters

Observability is a big part of operating complex distributed systems. Gardener provides a lot of monitoring out of the box using best-of-class open source projects like a central Prometheus server, deployed in the garden cluster that collects information about all seed clusters. In addition, each shoot cluster gets its own Prometheus instance in the seed cluster. To collect metrics, Gardener deploys two `kube-state-metrics` instances for each cluster (one for the control plane in the seed and one for the worker nodes in the shoot). The node-exporter is deployed too to provide additional information on the nodes. The Prometheus `AlertManager` is used to notify the operator when something goes wrong. Grafana is used to display dashboards with relevant data on the state of the system.

The gardenctl CLI

You can manage Gardener using only kubectl, but you will have to switch profiles and contexts a lot as you explore different clusters. Gardener provides the `gardenctl` command-line tool that offers higher-level abstractions and can operate on multiple clusters at the same time. Here is an example:

```
$ gardenctl ls shoots
projects:
- project: team-a
  shoots:
  - dev-eu1
  - prod-eu1

$ gardenctl target shoot prod-eu1
[prod-eu1]

$ gardenctl show prometheus
NAME           READY     STATUS    RESTARTS   AGE       IP              NODE
prometheus-0   3/3       Running   0          106d      10.241.241.42   ip-10-240-7-
72.eu-central-1.compute.internal

URL: https://user:password@p.prod-eu1.team-a.seed.aws-eu1.example.com
```

One of the most prominent features of Gardener is its extensibility. It has a large surface area and it supports many environments. Let's see how extensibility is built into its design.

Extending Gardener

Gardener supports the following environments:

- AliCloud
- AWS
- Azure
- Equinix Metal
- GCP
- OpenStack
- vSphere

It started, like Kubernetes itself, with a lot of provider-specific support in the primary Gardener repository. Over time, it followed the Kubernetes example that externalized cloud providers and migrated the providers to separate Gardener extensions. Providers can be specified using a CloudProfile CRD such as:

```
apiVersion: core.gardener.cloud/v1beta1
kind: CloudProfile
metadata:
  name: aws
spec:
  type: aws
  kubernetes:
    versions:
    - version: 1.24.3
    - version: 1.23.8
      expirationDate: "2022-10-31T23:59:59Z"
  machineImages:
  - name: coreos
    versions:
    - version: 2135.6.0
  machineTypes:
  - name: m5.large
    cpu: "2"
    gpu: "0"
    memory: 8Gi
    usable: true
  volumeTypes:
  - name: gp2
    class: standard
    usable: true
  - name: io1
```

```
    class: premium
    usable: true
  regions:
  - name: eu-central-1
    zones:
    - name: eu-central-1a
    - name: eu-central-1b
    - name: eu-central-1c
  providerConfig:
    apiVersion: aws.provider.extensions.gardener.cloud/v1alpha1
    kind: CloudProfileConfig
    machineImages:
    - name: coreos
      versions:
      - version: 2135.6.0
        regions:
        - name: eu-central-1
          ami: ami-034fd8c3f4026eb39
          # architecture: amd64 # optional
```

Then, a shoot cluster will choose a provider and configure it with the necessary information:

```
apiVersion: gardener.cloud/v1alpha1
kind: Shoot
metadata:
  name: johndoe-aws
  namespace: garden-dev
spec:
  cloudProfileName: aws
  secretBindingName: core-aws
  cloud:
    type: aws
    region: eu-west-1
    providerConfig:
      apiVersion: aws.cloud.gardener.cloud/v1alpha1
      kind: InfrastructureConfig
      networks:
        vpc: # specify either 'id' or 'cidr'
        # id: vpc-123456
          cidr: 10.250.0.0/16
        internal:
        - 10.250.112.0/22
```

```
        public:
        - 10.250.96.0/22
        workers:
        - 10.250.0.0/19
      zones:
      - eu-west-1a
    workerPools:
    - name: pool-01
    # Taints, labels, and annotations are not yet implemented. This requires
interaction with the machine-controller-manager, see
    # https://github.com/gardener/machine-controller-manager/issues/174. It is
only mentioned here as future proposal.
    # taints:
    # - key: foo
    #   value: bar
    #   effect: PreferNoSchedule
    # labels:
    # - key: bar
    #   value: baz
    # annotations:
    # - key: foo
    #   value: hugo
      machineType: m4.large
      volume: # optional, not needed in every environment, may only be
specified if the referenced CloudProfile contains the volumeTypes field
        type: gp2
        size: 20Gi
      providerConfig:
        apiVersion: aws.cloud.gardener.cloud/v1alpha1
        kind: WorkerPoolConfig
        machineImage:
          name: coreos
          ami: ami-d0dcef3
        zones:
        - eu-west-1a
      minimum: 2
      maximum: 2
      maxSurge: 1
      maxUnavailable: 0
  kubernetes:
    version: 1.11.0
```

```
    ...
  dns:
    provider: aws-route53
    domain: johndoe-aws.garden-dev.example.com
  maintenance:
    timeWindow:
      begin: 220000+0100
      end: 230000+0100
    autoUpdate:
      kubernetesVersion: true
  backup:
    schedule: "*/5 * * * *"
    maximum: 7
  addons:
    kube2iam:
      enabled: false
    kubernetes-dashboard:
      enabled: true
    cluster-autoscaler:
      enabled: true
    nginx-ingress:
      enabled: true
      loadBalancerSourceRanges: []
    kube-lego:
      enabled: true
      email: john.doe@example.com
```

But, the extensibility goals of Gardener go far beyond just being provider agnostic. The overall process of standing up a Kubernetes cluster involves many steps. The Gardener project aims to let the operator customize each and every step by defining custom resources and webhooks. Here is the general flow diagram with the CRDs, mutating/validating admission controllers, and webhooks associated with each step:

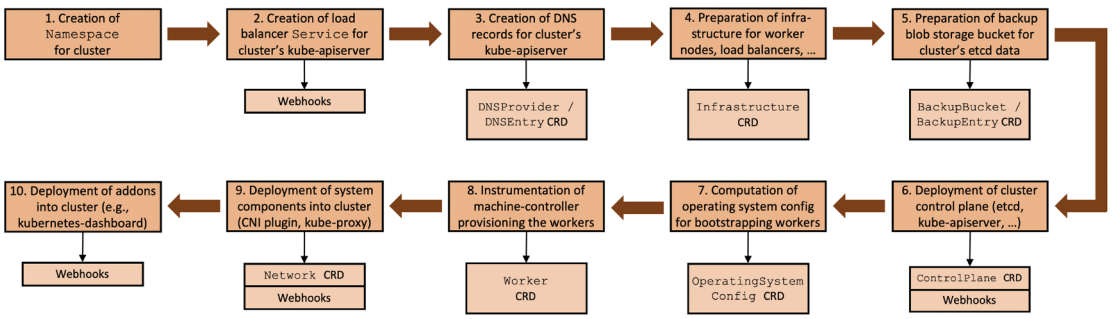


Figure 11.14: Flow diagram of CRDs mutating and validating admission controllers

Here are the CRD categories that comprise the extensibility space of Gardener:

- Providers for DNS management, such as Route53 and CloudDNS
- Providers for blob storage, including S3, GCS, and ABS
- Infrastructure providers like AWS, GCP, and Azure
- Support for various operating systems such as CoreOS Container Linux, Ubuntu, and FlatCar Linux
- Network plugins like Calico, Flannel, and Cilium
- Optional extensions, such as Let's Encrypt's certificate service

We have covered Gardener in depth, which brings us to the end of the chapter.

Summary

In this chapter, we've covered the exciting area of multi-cluster management. There are many projects that tackle this problem from different angles. The Cluster API project has a lot of momentum for solving the sub-problem of managing the lifecycle of multiple clusters. Many other projects take on the resource management and application lifecycle. These projects can be divided into two categories: projects that explicitly manage multiple clusters using a management cluster and managed clusters, and projects that utilize the Virtual Kubelet where whole clusters appear as virtual nodes in the main cluster.

The Gardener project has a very interesting approach and architecture. It tackles the problem of multiple clusters from a different perspective and focuses on the large-scale management of clusters. It is the only project that addresses both cluster lifecycle and application lifecycle.

At this point, you should have a clear understanding of the current state of multi-cluster management and what the different projects offer. You may decide that it's still too early or that you want to take the plunge.

In the next chapter, we will explore the exciting world of serverless computing on Kubernetes. Serverless can mean two different things: you don't have to manage servers for your long-running workloads, and also, running functions as a service. Both forms of serverless are available for Kubernetes, and both of them are extremely useful.

12

Serverless Computing on Kubernetes

In this chapter, we will explore the fascinating world of serverless computing in the cloud. The term "serverless" is getting a lot of attention, but it is a misnomer. A true serverless application runs as a web application in a user's browser or a mobile app and only interacts with external services. However, the types of serverless systems we build on Kubernetes are different. We will explain exactly what serverless means on Kubernetes and how it relates to other serverless solutions. We will cover serverless cloud solutions, introduce Knative - the Kubernetes foundation for functions as a service - and dive into Kubernetes **Function-as-a-Service** (**FaaS**) frameworks.

This chapter will cover the following main topics:

- Understanding serverless computing
- Serverless Kubernetes in the cloud
- Knative
- Kubernetes FaaS Frameworks

Let's start by clarifying what serverless is all about.

Understanding serverless computing

OK. Let's get it out of the way. Servers are still there. The term "serverless" means that you don't have to provision, configure, and manage the servers yourself. Public cloud platforms were a real paradigm shift by eliminating the need to deal with physical hardware, data centers, and networking. But even on the cloud it takes a lot of work and know-how to create machine images, provision instances, configure them, upgrade and patch operating systems, define network policies, and manage certificates and access control. With serverless computing large chunks of this important but tedious work go away. The allure of serverless is multi-pronged:

- A whole category of problems dealing with provisioning goes away
- Capacity planning is a non-issue
- You pay only for what you use

You lose some control because you have to live with the choices made by the cloud provider, but there is a lot of customization you can take advantage of for critical parts of a system. Of course, where you need total control you can still manage your own infrastructure by explicitly provisioning VMs and deploying workloads directly.

The bottom line is that the serverless approach is not just hype, it provides real benefits. Let's examine the two flavors of serverless.

Running long-running services on "serverless" infrastructure

Long-running services are the bread and butter of microservice-based distributed systems. These services must be always up, waiting to service requests, and can be scaled up and down to match the volume. In the traditional cloud you had to provision enough capacity to handle spikes and changing volumes, which often led to over-provisioning or increased delays in processing when requests were waiting for under-provisioned services.

Serverless services address this issue with zero effort from developers and relatively little effort from operators. The idea is that you just mark your service to run on the serverless infrastructure and configure it with some parameters, such as the expected CPU, memory, and limits to scaling. The service appears to other services and clients just like a traditional service that you deployed on infrastructure you provisioned yourself.

Services that fall into this category have the following characteristics:

- Always running (they never scale down to zero)
- Expose multiple endpoints (such as HTTP and gRPC)
- Require that you implement the request handling and routing yourself
- Can listen to events instead of, or in addition to, exposing endpoints
- Service instances can maintain in-memory caches, long-term connections, and sessions
- In Kubernetes, microservices are represented directly by the service resource

Now, let's look at FaaS.

Running functions as a service on "serverless" infrastructure

Even in the largest distributed systems not every workload handles multiple requests per second. There are always tasks that need to run in response to relatively infrequent events, whether on schedule or invoked in an ad hoc manner. It's possible to have a long-running service just sitting there twiddling its virtual thumbs and processing a request every now and then, but that's wasteful. You can try to hitch such tasks to other long-running services, but that creates very undesirable coupling, which goes against the philosophy of microservices.

A much better approach known as FaaS is to treat such tasks separately and provide different abstractions and tooling to address them.

FaaS is a computing model where a central authority (e.g., a cloud provider or Kubernetes) offers its users a way to run code (essentially functions) without worrying about where this code runs.

Kubernetes has the concept of a `Job` and a `CronJob` object. These address some issues that FaaS solutions tackle, but not completely.

A FaaS solution is often much simpler to get up and running compared to a traditional service. The developers may only need to write the code of a function; the FaaS solution will take care of the rest:

- Building and packaging
- Exposing as an endpoint
- A trigger based on events
- Provisioning and scaling automatically
- Monitoring and providing logs and metrics

Here are some of the characteristics of FaaS solutions:

- It runs on demand (can scale down to zero)
- It exposes a single endpoint (usually HTTP)
- It can be triggered by events or get an automatic endpoint
- It often has severe limitations on resource usage and maximum runtime
- Sometimes, it might have a cold start (that is, when scaling up from zero)

FaaS is indeed a form of serverless computing since the user doesn't need to provision servers in order to run their code, but it is used to run short-term functions. There is another form of serverless computing used for running long-running services too.

Serverless Kubernetes in the cloud

All the major cloud providers now support serverless long-running services for Kubernetes. Microsoft Azure was the first to offer it. Kubernetes interacts with nodes via the kubelet. The basic idea of serverless infrastructure is that instead of provisioning actual nodes (physical or VMs) a virtual node is created in some fashion. Different cloud providers use different solutions to accomplish this goal.



Don't forget the cluster auto scaler

Before jumping into cloud provider-specific solutions make sure to check out the Kubernetes-native option of the cluster autoscaler. The cluster autoscaler scales the nodes in your cluster and doesn't suffer from the limitations of some of the other solutions. All the Kubernetes scheduling and control mechanisms work out of the box with the cluster autoscaler because it just automates adding and removing regular nodes from your cluster. No exotic and provider-specific capabilities are used.

But you may have good reasons to prefer a more provider-integrated solution. For example, AWS Fargate runs inside Firecracker (see `https://github.com/firecracker-microvm/firecracker`), which is a lightweight VM with strong security boundaries (as a side note, Lambda functions run on Firecracker too). Similarly Google Cloud Run runs in gVisor. Azure has several different hosting solutions such as dedicated VMs, Kubernetes, and Arc.

Azure AKS and Azure Container Instances

Azure supported **Azure Container Instances** (**ACI**) for a long time. ACI is not Kubernetes-specific. It allows you to run on-demand containers on Azure in a managed environment. It is similar in some regards to Kubernetes but is Azure-specific. It even has the concept of a container group, which is similar to a pod. All containers in a container group will be scheduled to run on the same host machine.

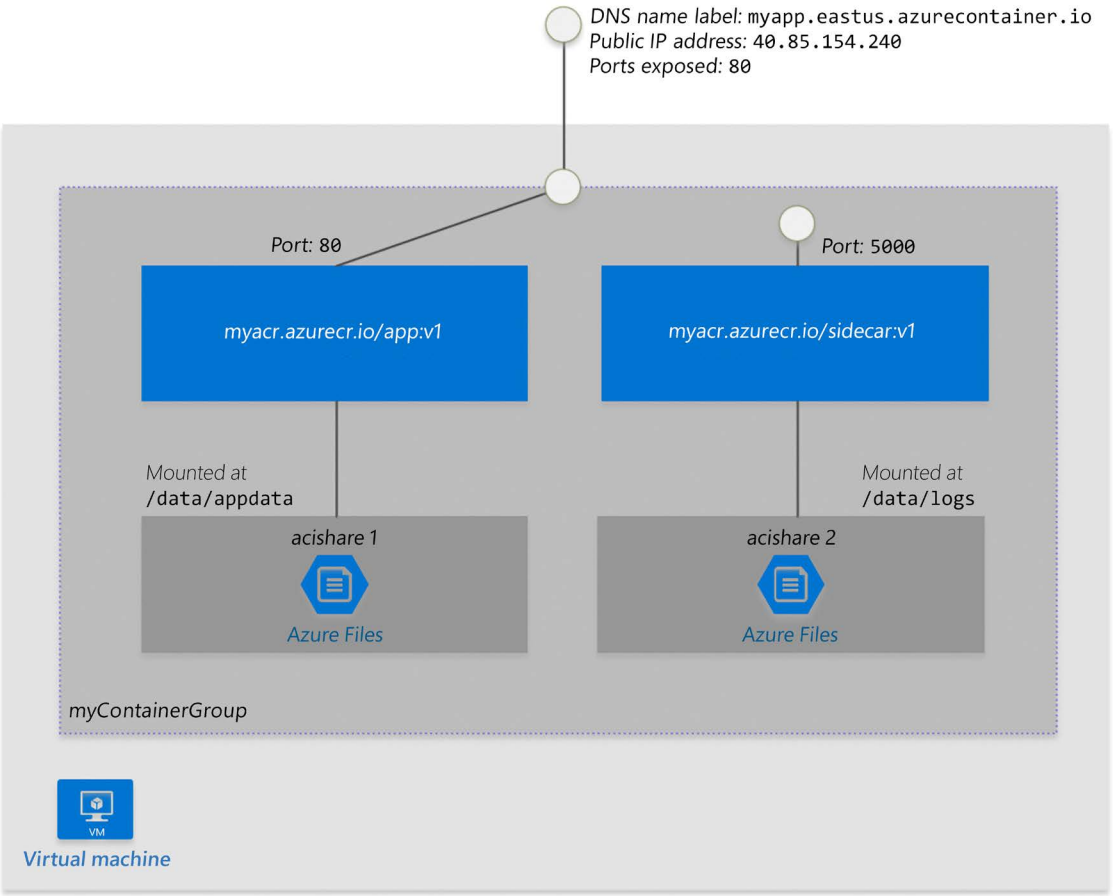


Figure 12.1: ACI architecture

The integration with Kubernetes/AKS is modeled as bursting from AKS to ACI. The guiding principle here is that for your known workloads you should provision your own nodes, but if there are spikes then the extra load will burst dynamically to ACI. This approach is considered more economical because running on ACI is more expensive than provisioning your own nodes. AKS uses the virtual kubelet CNCF project we explored in the previous chapter to integrate your Kubernetes cluster with the infinite capacity of ACI. It works by adding a virtual node to your cluster, backed by ACI that appears on the Kubernetes side as a single node with infinite resources.

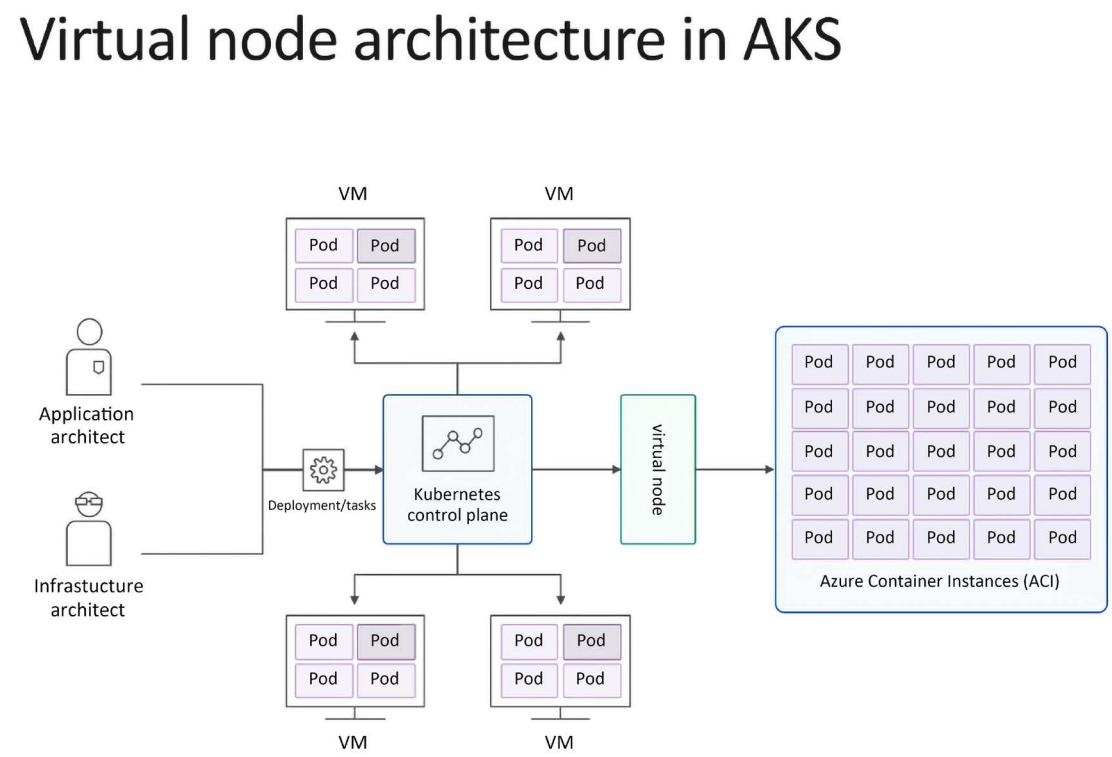


Figure 12.2: Virtual node architecture in AKS

Let's see how AWS does it with EKS and Fargate.

AWS EKS and Fargate

AWS released Fargate (`https://aws.amazon.com/fargate`) in 2018, which is similar to Azure ACI and lets you run containers in a managed environment. Originally, you could use Fargate on EC2 or ECS (AWS's proprietary container orchestration services). At the big AWS conference re:Invent 2019, Fargate became generally available on EKS too. That means that you now have a fully managed Kubernetes solution that is truly serverless. EKS takes care of the control plane and Fargate takes care of the worker nodes for you.

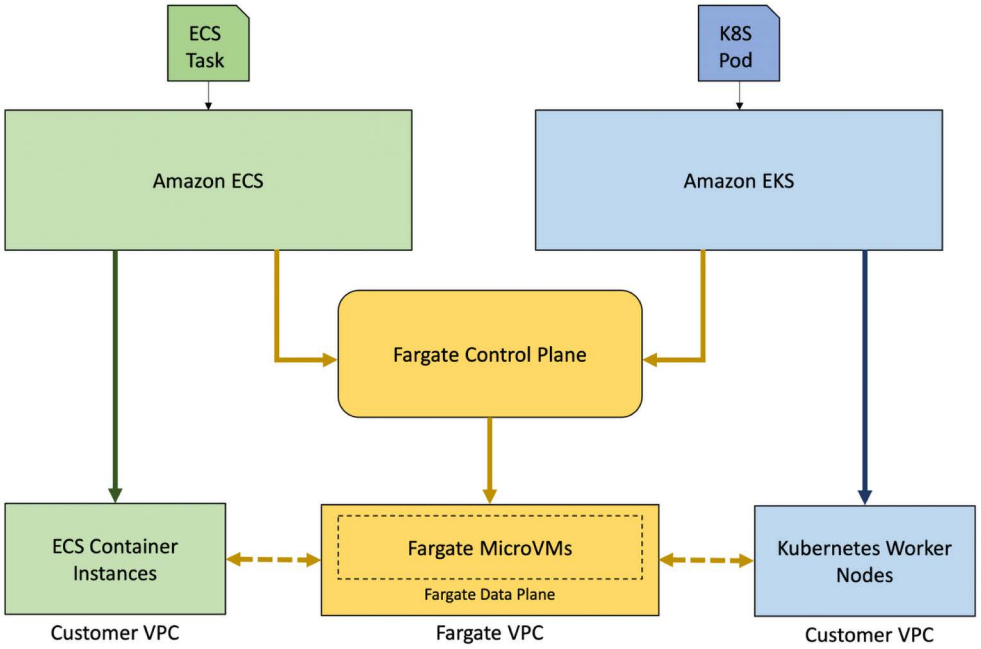


Figure 12.3: EKS and Fargate architecture

EKS and Fargate model the interaction between your Kubernetes cluster and Fargate differently than AKS and ACI. While on AKS a single infinite virtual node represents the entire capacity of ACI, on EKS each pod gets its own virtual node. But those nodes are not real nodes of course. Fargate has its own control plane and data plane that supports EC2, ECS, as well as EKS. The EKS-Fargate integration is done via a set of custom Kubernetes controllers that watch for pods that need to be deployed to a particular namespace or have specific labels, forwarding those pods to be scheduled by Fargate. The following diagram illustrates the workflow from EKS to Fargate.

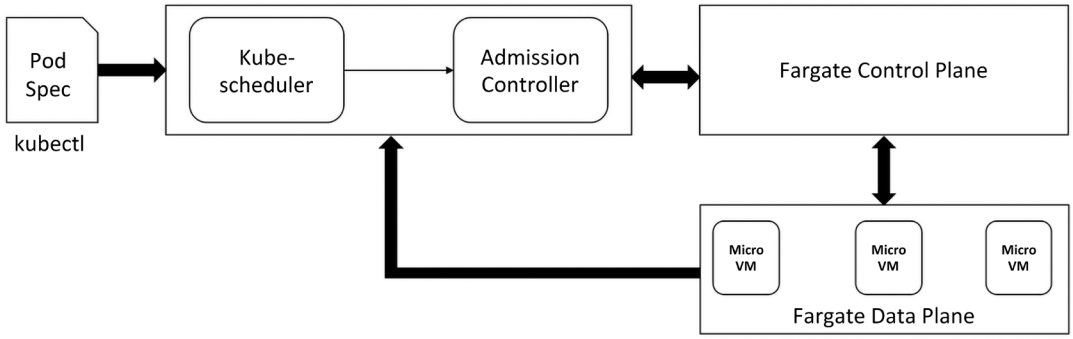


Figure 12.4: EKS to Fargate workflow

When working with Fargate there are several limitations you should be aware of:

- A maximum of 16 vCPU and 120 GB memory per pod
- 20 GiB of container image layer storage
- Stateful workloads that require persistent volumes or filesystems are not supported
- `Daemonsets`, privileged pods, or pods that use `HostNetwork` or `HostPort` are not supported
- You can use the Application Load Balancer or Network Load Balancer

If those limitations are too severe for you, you can try a more direct approach and utilize the virtual kubelet project to integrate Fargate into your cluster.

What about Google - the father of Kubernetes?

Google Cloud Run

It may come as a surprise, but Google is the Johnny-come-lately of serverless Kubernetes. Cloud Run is Google's serverless offering. It is based on Knative, which we will dissect in depth in the next section. The basic premise is that there are two flavors of Cloud Run. Plain Cloud Run is similar to ACI and Fargate. It lets you run containers in an environment fully managed by Google. Cloud Run for Anthos supports GKE, and on-premises lets you run containerized workloads in your GKE cluster.

Cloud Run for Anthos is currently the only serverless platform that allows you to run containers on custom machine types (including GPUs). Anthos Cloud Run services participate in the Istio service mesh and provide a streamlined Kubernetes-native experience. See `https://cloud.google.com/anthos/service-mesh` for more details.

Note that while managed Cloud Run uses gVisor isolation, the Anthos Cloud Run uses standard Kubernetes (container-based) isolation.

The following diagram shows both models and the layering of access methods and deployment options:

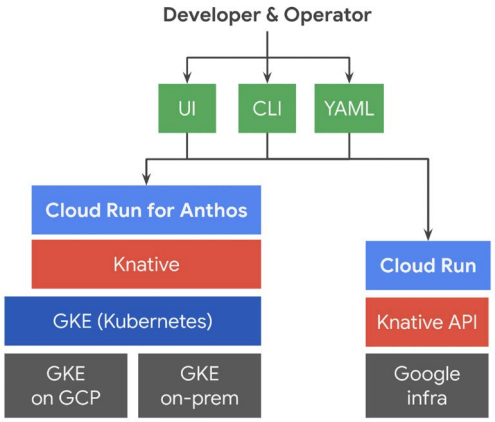


Figure 12.5: Cloud Run models

It's time to learn more about Knative.

Knative

Kubernetes doesn't have built-in support for FaaS. As a result many solutions were developed by the community and ecosystem. The goal of Knative is to provide building blocks that multiple FaaS solutions can utilize without reinventing the wheel.

But, that's not all! Knative also offers the unique capability to scale down long-running services all the way to zero. This is a big deal. There are many use cases where you may prefer to have a long-running service that can handle a lot of requests coming its way in rapid succession. In those situations it is not the best approach to fire a new function instance per request. But when there is no traffic coming in, it's great to scale the service to zero instances, pay nothing, and leave more capacity for other services that may need more resources at that time. Knative supports other important use cases like load balancing based on percentages, load balancing based on metrics, blue-green deployments, canary deployments, and advanced routing. It can even optionally do automatic TLS certificates as well as HTTP monitoring. Finally, Knative works with both HTTP and gRPC.

There are currently two Knative components: Knative serving and Knative eventing. There used to also be a Knative build component, but it was factored out to form the foundation of Tekton (`https://github.com/tektoncd/pipeline`) - a Kubernetes-native CD project.

Let's start with Knative serving.

Knative serving

The domain of Knative serving is running versioned services on Kubernetes and routing traffic to those services. This is above and beyond standard Kubernetes services. Knative serving defines several CRDs to model its domain: `Service`, `Route`, `Configuration`, and `Revision`. The `Service` manages a `Route` and a `Configuration`. A `Configuration` can have multiple revisions.

The `Route` can route service traffic to a particular revision. Here is a diagram that illustrates the relationship between the different objects:

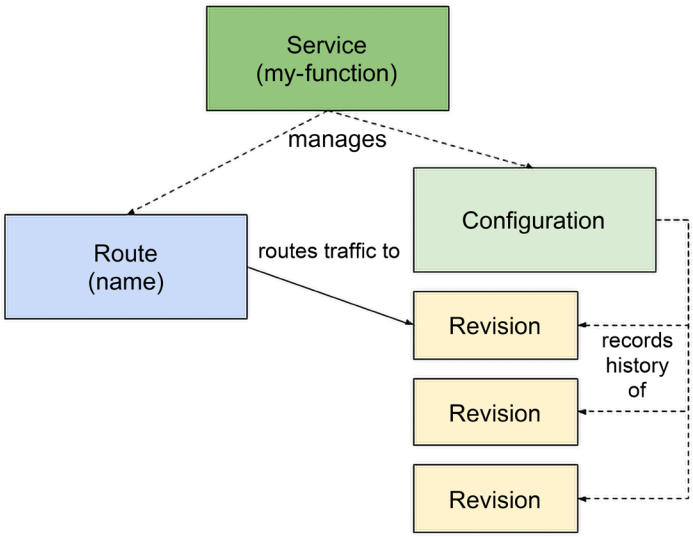


Figure 12.6: Knative serving CRDs

Let's try Knative serving in a local environment.

Install a quickstart environment

Knative provides an easy development setup. Let's install the `kn` CLI and quickstart plugin. Follow the instructions here: `https://knative.dev/docs/getting-started/quickstart-install`.

Now, we can run the plugin with KinD, which will provision a new KinD cluster and install multiple components such as the Knative-service, Kourier networking layer, and Knative eventing.

```
$ kn quickstart kind
Running Knative Quickstart using Kind
✅ Checking dependencies...
    Kind version is: 0.16.0

☸ Creating Kind cluster...
Creating cluster "knative" ...
 ✓ Ensuring node image (kindest/node:v1.24.3) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
```

```
 ✓ Waiting ≤ 2m0s for control-plane = Ready ⌛ kind-knative | default
 • Ready after 19s 💚
Set kubectl context to "kind-knative"
You can now use your cluster with:

kubectl cluster-info --context kind-knative

Have a nice day! 👋

🔥 Installing Knative Serving v1.6.0 ...
    CRDs installed...
    Core installed...
    Finished installing Knative Serving
🔘 Installing Kourier networking layer v1.6.0 ...
    Kourier installed...
    Ingress patched...
    Finished installing Kourier Networking layer
🕸 Configuring Kourier for Kind...
    Kourier service installed...
    Domain DNS set up...
    Finished configuring Kourier
🔥 Installing Knative Eventing v1.6.0 ...
    CRDs installed...
    Core installed...
    In-memory channel installed...
    Mt-channel broker installed...
    Example broker installed...
    Finished installing Knative Eventing
🚀 Knative install took: 2m22s
🎉 Now have some fun with Serverless and Event Driven Apps!
```

Let's install the sample `hello` service:

```
$ kn service create hello \
--image gcr.io/knative-samples/helloworld-go \
--port 8080 \
--env TARGET=World

Creating service 'hello' in namespace 'default':

  0.080s The Route is still working to reflect the latest desired specification.
```

```
  0.115s ...
  0.127s Configuration "hello" is waiting for a Revision to become ready.
 21.229s ...
 21.290s Ingress has not yet been reconciled.
 21.471s Waiting for load balancer to be ready
 21.665s Ready to serve.

Service 'hello' created to latest revision 'hello-00001' is available at URL:
http://hello.default.127.0.0.1.sslip.io
```

We can call the service using httpie and get the `Hello, World!` response:

```
$ http --body http://hello.default.127.0.0.1.sslip.io

Hello World!
```

Let's look at the `Service` object.

The Knative Service object

The Knative `Service` combines the Kubernetes `Deployment` and `Service` into a single object. That makes a lot of sense because except for the special case of headless services (`https://kubernetes.io/docs/concepts/services-networking/service/#headless-services`) there is always a deployment behind every service.

The Knative `Service` automatically manages the entire life cycle of its workload. It is responsible for creating the route and configuration and a new revision whenever the service is updated. This is very convenient because the user just needs to deal with the `Service` object.

Here is the metadata for the helloworld-go Knative service:

```
$ k get ksvc hello -o json | jq .metadata
{
  "annotations": {
    "serving.knative.dev/creator": "kubernetes-admin",
    "serving.knative.dev/lastModifier": "kubernetes-admin"
  },
  "creationTimestamp": "2022-09-25T21:11:21Z",
  "generation": 1,
  "name": "hello",
  "namespace": "default",
  "resourceVersion": "19380",
  "uid": "03b5c668-3934-4260-bdba-13357a48501e"
}
```

And here is the spec:

```
$ k get ksvc hello -o json | jq .spec
{
  "template": {
    "metadata": {
      "annotations": {
        "client.knative.dev/updateTimestamp": "2022-09-25T21:11:21Z",
        "client.knative.dev/user-image": "gcr.io/knative-samples/helloworld-go"
      },
      "creationTimestamp": null
    },
    "spec": {
      "containerConcurrency": 0,
      "containers": [
        {
          "env": [
            {
              "name": "TARGET",
              "value": "World"
            }
          ],
          "image": "gcr.io/knative-samples/helloworld-go",
          "name": "user-container",
          "ports": [
            {
              "containerPort": 8080,
              "protocol": "TCP"
            }
          ],
          "readinessProbe": {
            "successThreshold": 1,
            "tcpSocket": {
              "port": 0
            }
          },
          "resources": {}
        }
      ],
      "enableServiceLinks": false,
      "timeoutSeconds": 300
```

```
    }
  },
  "traffic": [
    {
      "latestRevision": true,
      "percent": 100
    }
  ]
}
```

Note the `traffic` section of the spec that directs 100% of requests to the latest revision. This is what determines the `Route` CRD.

Creating new revisions

Let's create a new revision of the `hello` service with a `TARGET` environment variable of `Knative`:

```
$ kn service update hello --env TARGET=Knative
Updating Service 'hello' in namespace 'default':

  0.097s The Configuration is still working to reflect the latest desired
specification.
  3.000s Traffic is not yet migrated to the latest revision.
  3.041s Ingress has not yet been reconciled.
  3.155s Waiting for load balancer to be ready
  3.415s Ready to serve.
```

Now, we have two revisions:

```
$ k get revisions
NAME          CONFIG NAME   K8S SERVICE NAME   GENERATION   READY   REASON   ACTUAL
REPLICAS   DESIRED REPLICAS
hello-00001   hello                            1            True             0
0
hello-00002   hello                            2            True             1
1
```

The `hello-00002` revision is the active one. Let's confirm:

```
$ http --body http://hello.default.127.0.0.1.sslip.io

Hello Knative!
```

The Knative Route object

The Knative `Route` object allows you to direct a percentage of incoming requests to particular revisions. The default is 100% to the latest revision, but you can change it. This allows advanced deployment scenarios, like blue-green deployments as well as canary deployments.

Here is the `hello` route that directs 100% of traffic to the latest revision:

```
apiVersion: serving.knative.dev/v1
kind: Route
metadata:
  annotations:
    serving.knative.dev/creator: kubernetes-admin
    serving.knative.dev/lastModifier: kubernetes-admin
  labels:
    serving.knative.dev/service: hello
  name: hello
  namespace: default
spec:
  traffic:
  - configurationName: hello
    latestRevision: true
    percent: 100
```

Let's direct 50% of the traffic to the previous revision:

```
$ kn service update hello \
--traffic hello-00001=50 \
--traffic @latest=50

Updating Service 'hello' in namespace 'default':

  0.078s The Route is still working to reflect the latest desired specification.
  0.124s Ingress has not yet been reconciled.
  0.192s Waiting for load balancer to be ready
  0.399s Ready to serve.

Service 'hello' with latest revision 'hello-00002' (unchanged) is available at URL:
http://hello.default.127.0.0.1.sslip.io
```

Now, if we call the service repeatedly, we see a mix of responses from the two revisions:

```
$ while true; do http --body http://hello.default.127.0.0.1.sslip.io; done
Hello World!
Hello World!
Hello World!
Hello Knative!
Hello Knative!
Hello Knative!
Hello Knative!
```

```
Hello World!
Hello Knative!
Hello World!
```

Let's look at the route using the neat kubectl plugin (`https://github.com/itaysk/kubectl-neat`):

```
$ k get route hello -o yaml | k neat
apiVersion: serving.knative.dev/v1
kind: Route
metadata:
  annotations:
    serving.knative.dev/creator: kubernetes-admin
    serving.knative.dev/lastModifier: kubernetes-admin
  labels:
    serving.knative.dev/service: hello
  name: hello
  namespace: default
spec:
  traffic:
  - configurationName: hello
    latestRevision: true
    percent: 50
  - latestRevision: false
    percent: 50
    revisionName: hello-00001
```

The Knative Configuration object

The `Configuration` CRD contains the latest version of the service and the number of generations. For example, if we update the service to version 2:

```
apiVersion: serving.knative.dev/v1 # Current version of Knative
kind: Service
metadata:
  name: helloworld-go # The name of the app
  namespace: default # The namespace the app will use
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-samples/helloworld-go # The URL to the image of
the app
          env:
```

```
            - name: TARGET # The environment variable printed out by the sample
  app
              value: "Yeah, it still works - version 2 !!!"
```

Knative also generates a configuration object that now points to the `hello-00002` revision:

```
$ k get configuration hello -o yaml
apiVersion: serving.knative.dev/v1
kind: Configuration
metadata:
  annotations:
    serving.knative.dev/creator: kubernetes-admin
    serving.knative.dev/lastModifier: kubernetes-admin
    serving.knative.dev/routes: hello
  creationTimestamp: "2022-09-25T21:11:21Z"
  generation: 2
  labels:
    serving.knative.dev/service: hello
    serving.knative.dev/serviceUID: 03b5c668-3934-4260-bdba-13357a48501e
  name: hello
  namespace: default
  ownerReferences:
  - apiVersion: serving.knative.dev/v1
    blockOwnerDeletion: true
    controller: true
    kind: Service
    name: hello
    uid: 03b5c668-3934-4260-bdba-13357a48501e
  resourceVersion: "22625"
  uid: fabfcb7c-e3bc-454e-a887-9f84057943f7
spec:
  template:
    metadata:
      annotations: kind-knative | default
        client.knative.dev/updateTimestamp: "2022-09-25T21:21:00Z"
        client.knative.dev/user-image: gcr.io/knative-samples/helloworld-go
      creationTimestamp: null
    spec:
      containerConcurrency: 0
      containers:
      - env:
        - name: TARGET
```

```
          value: Knative
        image: gcr.io/knative-samples/helloworld-go@
sha256:5ea96ba4b872685ff4ddb5cd8d1a97ec18c18fae79ee8df0d29f446c5efe5f50
        name: user-container
        ports:
        - containerPort: 8080
          protocol: TCP
        readinessProbe:
          successThreshold: 1
          tcpSocket:
            port: 0
        resources: {}
      enableServiceLinks: false
      timeoutSeconds: 300
status:
  conditions:
  - lastTransitionTime: "2022-09-25T21:21:03Z"
    status: "True"
    type: Ready
  latestCreatedRevisionName: hello-00002
  latestReadyRevisionName: hello-00002
  observedGeneration: 2
```

To summarize, Knative serving provides better deployment and networking for Kubernetes for long-running services and functions. Let's see what Knative eventing brings to the table.

Knative eventing

Traditional services on Kubernetes or other systems expose API endpoints that consumers can hit (often over HTTP) to send a request for processing. This pattern of request-response is very useful, hence why it is so popular. However, this is not the only pattern to invoke services or functions. Most distributed systems have some form of loosely-coupled interactions where events are published. It is often desirable to invoke some code when events occur.

Before Knative you had to build this capability yourself or use some third-party library that binds events to code. Knative eventing aims to provide a standard way to accomplish this task. It is compatible with the CNCF CloudEvents specification (`https://github.com/cloudevents/spec`).

Getting familiar with Knative eventing terminology

Before diving into the architecture let's define some terms and concepts we will use later.

Event consumer

There are two types of event consumers: Addressable and Callable. Addressable consumers can receive events over HTTP through their `status.address.url` field. The Kubernetes `Service` object doesn't have such a field, but it is also considered a special case of an Addressable consumer.

Callable consumers receive an event delivered over HTTP, and they may return another event in the response that will be consumed just like an external event. Callable consumers provide an effective way to transform events.

Event source

An event source is the originator of an event. Knative supports many common sources, and you can write your own custom event source too. Here are some of the supported event sources:

- AWS SQS
- Apache Camel
- Apache CouchDB
- Apache Kafka
- Bitbucket
- ContainerSource
- Cron Job
- GCP PubSub
- GitHub
- GitLab
- Google Cloud Scheduler
- Kubernetes (Kubernetes Events)

Check out the full list here: `https://knative.dev/docs/eventing/sources`.

Broker and Trigger

A broker mediates events identified by particular attributes and matches them with consumers via triggers. The trigger includes a filter of event attributes and an Addressable consumer. When the event arrives at the broker, it forwards it to consumers that have triggers with matching filters to the event attributes. The following diagram illustrates this workflow:

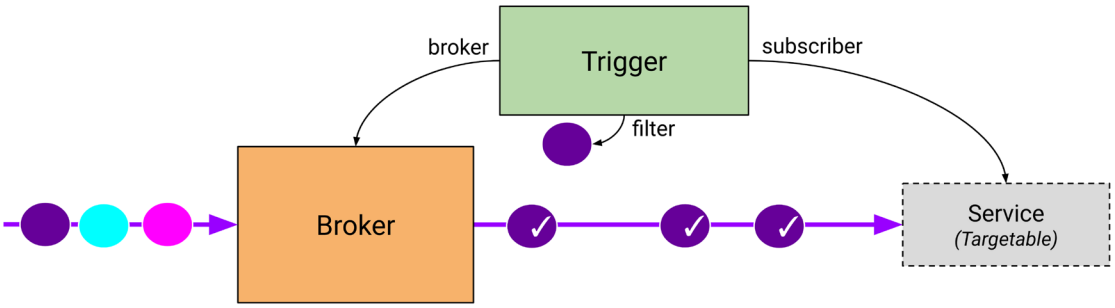


Figure 12.7: The workflow of brokers, triggers, and services

Event types and event registry

Events can have a type, which is modeled as the `EventType` CRD. The event registry stores all the event types. Triggers can use the event type as one of their filter criteria.

Channels and subscriptions

A channel is an optional persistence layer. Different event types may be routed to different channels with different backing stores. Some channels may store events in memory, while other channels may persist to disk via NATS streaming, Kafka, or similar. Subscribers (consumers) eventually receive and handle the events.

Now that we covered the various bits and pieces of Knative eventing let's understand its architecture.

The architecture of Knative eventing

The current architecture supports two modes of event delivery:

- Simple delivery
- Fan-out delivery

The simple delivery is just 1:1 source -> consumer. The consumer can be a core Kubernetes service or a Knative service. If the consumer is unreachable the source is responsible for handling the fact that the event can't be delivered. The source can retry, log an error, or take any other appropriate action.

The following diagram illustrates this simple concept:

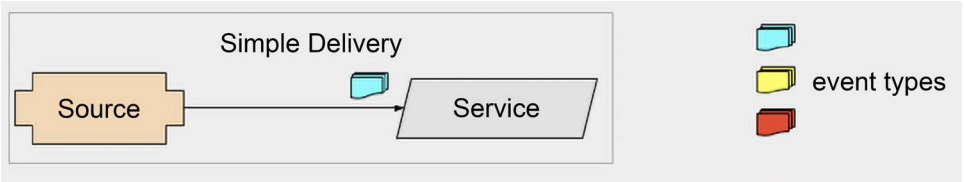


Figure 12.8: Simple delivery

The fan-out delivery can support arbitrarily complex processing where multiple consumers subscribe to the same event on a channel. Once an event is received by the channel the source is not responsible for the event anymore. This allows a more dynamic subscription of consumers because the source doesn't even know who the consumers are. In essence, there is a loose coupling between producers and consumers.

The following diagram illustrates the complex processing and subscriptions patterns that can arise when using channels:

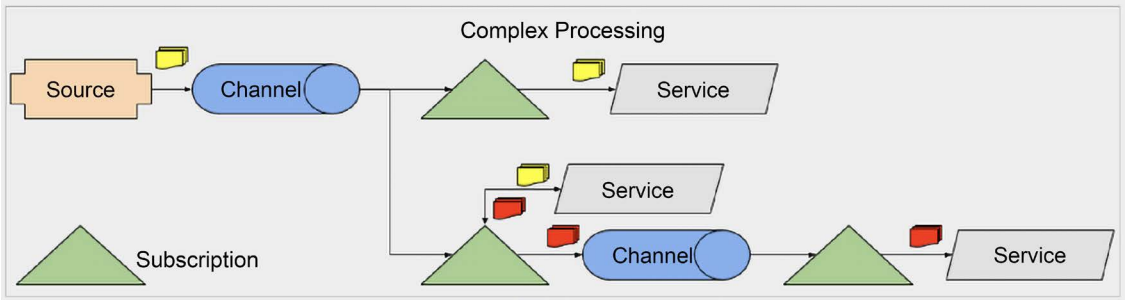


Figure 12.9: Fan-out delivery

At this point, you should have a decent understanding of the scope of Knative and how it establishes a solid serverless foundation for Kubernetes. Let's play around a little with Knative and see what it feels like.

Checking the scale to zero option of Knative

Knative is configured by default to scale to zero with a grace period of 30 seconds. That means that after 30 seconds of inactivity (no request coming in) all the pods will be terminated until a new request comes in. To verify that we can wait 30 seconds and check the pods in the default namespace:

```
$ kubectl get po
No resources found in default namespace.
```

Then, we can invoke the service and after a short time we get our response:

```
$ http --body http://hello.default.127.0.0.1.sslip.io
Hello World!
```

Let's watch when the pods disappear by using the `-w` flag:

```
$ k get po -w
NAME                                     READY   STATUS        RESTARTS   AGE
hello-00001-deployment-7c4b6cc4df-4j7bf  2/2     Running       0          46s
hello-00001-deployment-7c4b6cc4df-4j7bf  2/2     Terminating   0          98s
hello-00001-deployment-7c4b6cc4df-4j7bf  1/2     Terminating   0          2m
hello-00001-deployment-7c4b6cc4df-4j7bf  0/2     Terminating   0          2m9s
hello-00001-deployment-7c4b6cc4df-4j7bf  0/2     Terminating   0          2m9s
hello-00001-deployment-7c4b6cc4df-4j7bf  0/2     Terminating   0          2m9s
```

Now that we have had a little fun with Knative, we can move on to discussing FaaS solutions on Kubernetes.

Kubernetes Function-as-a-Service frameworks

Let's acknowledge the elephant in the room - FaaS. The Kubernetes Job and CronJob are great, and cluster autoscaling and cloud providers managing the infrastructure is awesome. Knative with its scale to zero and traffic routing is super cool. But, what about the actual FaaS? Fear not, Kubernetes has many options here - maybe too many options. There are many FaaS frameworks for Kubernetes:

- Fission
- Kubeless
- OpenFaaS
- OpenWhisk
- Riff (built on top of Knative)
- Nuclio
- BlueNimble
- Fn
- Rainbond

Some of these frameworks have a lot of traction and some of them don't. Two of the most prominent frameworks I discussed in the previous edition of the book, Kubeless and Riff, have been archived (Riff calls itself complete).

We will look into a few of the more popular options that are still active. In particular we will look at OpenFaaS and Fission.

OpenFaaS

OpenFaaS (`https://www.openfaas.com`) is one of the most mature, popular, and active FaaS projects. It was created in 2016 and has more than 30,000 stars on GitHub at the time of writing. OpenFaaS has a community edition and licensed Pro and Enterprise editions. Many production features, such as advanced autoscaling and scale to zero, are not available in the community edition. OpenFaaS comes with two additional components - Prometheus (for metrics) and NATS (asynchronous queue). Let's see how OpenFaaS provides a FaaS solution on Kubernetes.

Delivery pipeline

OpenFaaS provides a complete ecosystem and delivery mechanism to package and run your functions on Kubernetes. It can run on VMs too using fasstd, but this is a book about Kubernetes. The typical workflow looks like this:

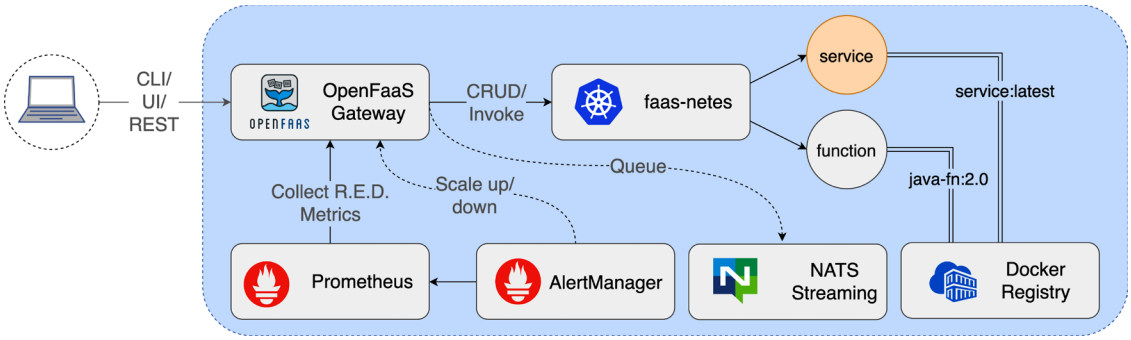


Figure 12.10: A typical OpenFaaS workflow

The `faas-cli` allows you to build, push and deploy your functions as Docker/OCI images. When you build your functions, you can use various templates as well as add your own. These steps can be incorporated into any CI/CD pipeline.

OpenFaaS features

OpenFaaS exposes its capabilities via a gateway. You interact with the gateway via a REST API, CLI, or web-based UI. The gateway exposes different endpoints.

The primary features of OpenFaaS are:

- Function management
- Function invocations and triggers
- Autoscaling
- Metrics
- A web-based UI

Function management

You manage functions by creating or building images, pushing these images, and deploying them. The `faas-cli` helps with these tasks. We will see an example later in the chapter.

Function invocations and triggers

OpenFaaS functions can be invoked as HTTP endpoints or via various triggers such as NATS events, other event systems, and directly through the CLI.

Metrics

OpenFaaS exposes a prometheus/metrics endpoint that can be used to scrape metrics. Some of the metrics are only available with the Pro version. See a complete list of metrics here: `https://docs.openfaas.com/architecture/metrics/`.

Autoscaling

One of OpenFaaS's claims to fame is that it scales up and down (including down to zero in the Pro version) based on various metrics. It doesn't use the Kubernetes **Horizontal Pod Autoscaler** (**HPA**) and supports different scaling modes, such as rps, capacity, and cpu (the same as Kubernetes HPA). You can achieve similar results with a project like Keda (`https://keda.sh`), but then you'd have to build it yourself, while OpenFaaS provides it out of the box.

Web-based UI

OpenFaaS provides a simple web-based UI, available on the API gateway as the `/ui` endpoint.

Here is what it looks like:

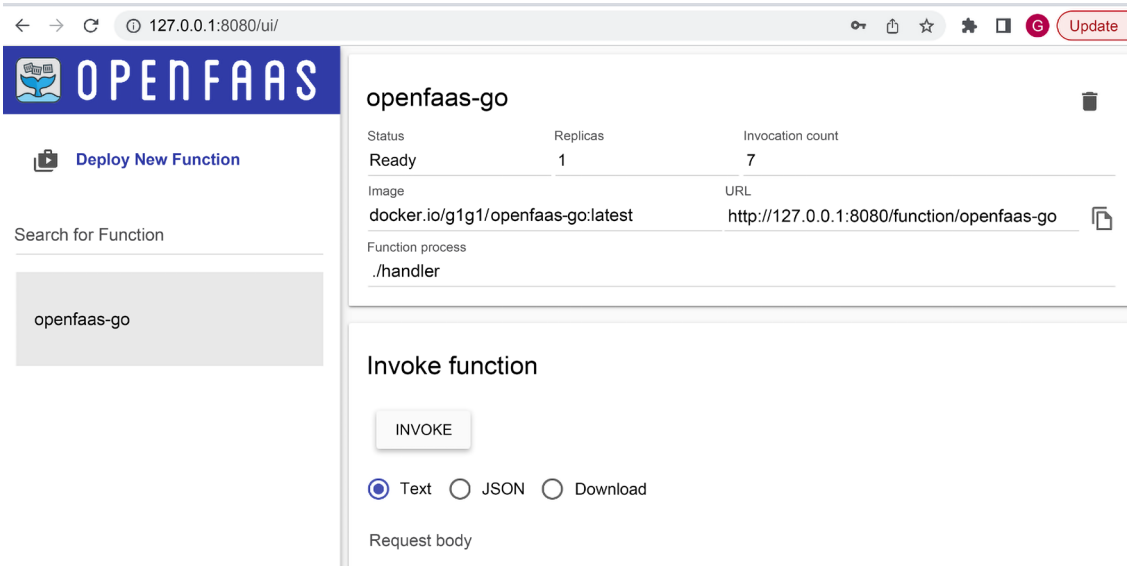


Figure 12.11: The OpenFaaS web-based UI

OpenFaaS architecture

OpenFaaS has multiple components that interact to provide all its capabilities in an extensible and Kubernetes-native way.

The main components are:

- OpenFaaS API gateway
- FaaS Provider
- Prometheus and Alert Manager
- OpenFaaS Operator
- The API gateway

Your functions are stored as CRDs. The OpenFaaS Operator watches these functions. The following diagram illustrates the various components and the relationships between them.

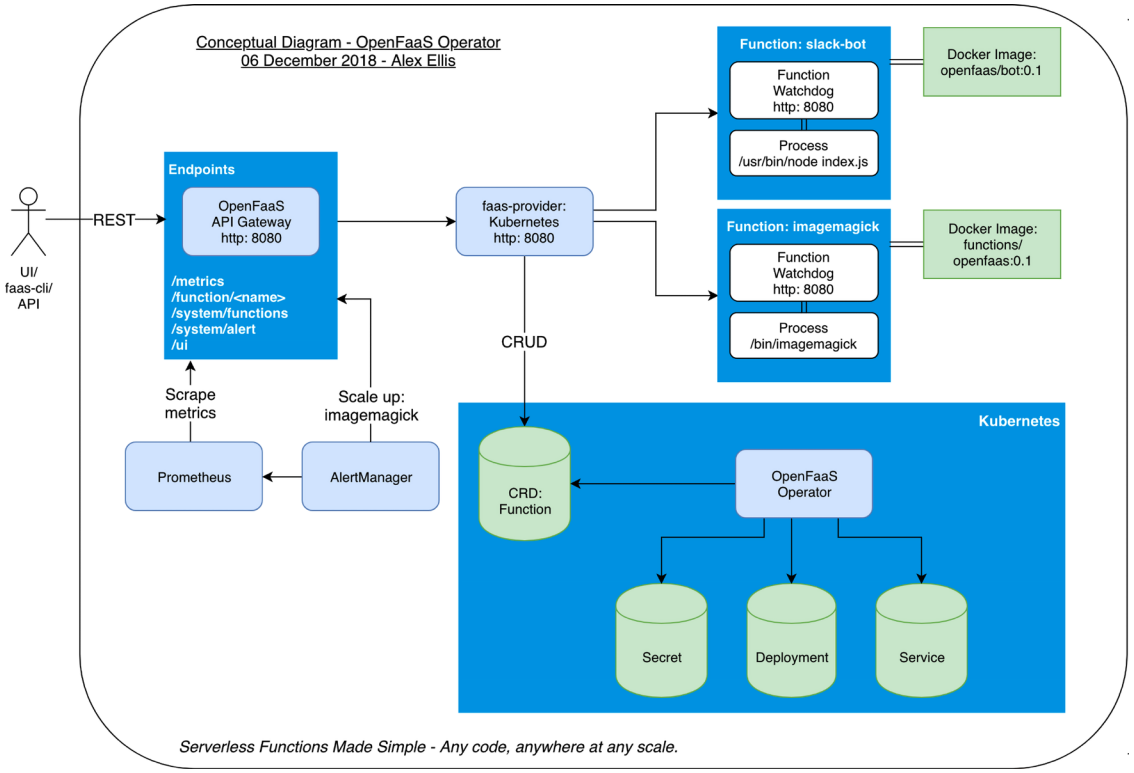


Figure 12.12: OpenFaaS architecture

Let's play with OpenFaaS to understand how everything works from a user perspective.

Taking OpenFaaS for a ride

Let's install OpenFaaS and the fass-cli CLI. We will use the recommended arkade package manager (`https://github.com/alexellis/arkade`), developed by the OpenFaaS founder. So, let's install arkade first. Arkade can install Kubernetes applications and various command-line tools.

On a Mac, you can use homebrew:

```
$ brew install arkade
```

On Windows, you need to install Git Bash (`https://git-scm.com/downloads`), and then from a Git Bash prompt:

```
$ curl -sLS https://get.arkade.dev | sh
```

Let's verify that arkade is available:

```
$ ark
```

```
            _                  _
  __ _ _ __| | ____ _  __| | ___
 / _` | '__| |/ / _` |/ _` |/ _ \
| (_| | |  |   < (_| | (_| |  __/
 \__,_|_|  |_|\_\__,_|\__,_|\___|

Open Source Marketplace For Developer Tools

Usage:
  arkade [flags]
  arkade [command]

Available Commands:
  chart       Chart utilities
  completion  Output shell completion for the given shell (bash or zsh)
  get         The get command downloads a tool
  help        Help about any command
  info        Find info about a Kubernetes app
  install     Install Kubernetes apps from helm charts or YAML files
  system      System apps
  uninstall   Uninstall apps installed with arkade
  update      Print update instructions
  version     Print the version

Flags:
  -h, --help   help for arkade

Use "arkade [command] --help" for more information about a command.
```

If you don't want to use arkade there are other options to install OpenFaaS. See `https://docs.openfaas.com/deployment/kubernetes/`.

Next, let's install OpenFaaS on our Kubernetes cluster:

```
$ ark install openfaas
Using Kubeconfig: /Users/gigi.sayfan/.kube/config
Client: arm64, Darwin
2022/10/01 11:29:14 User dir established as: /Users/gigi.sayfan/.arkade/
Downloading: https://get.helm.sh/helm-v3.9.3-darwin-amd64.tar.gz
```

```
/var/folders/qv/7l781jhs6j19gw3b89f4fcz40000gq/T/helm-v3.9.3-darwin-amd64.tar.gz
written.
2022/10/01 11:29:17 Extracted: /var/folders/qv/7l781jhs6j19gw3b89f4fcz40000gq/T/helm
2022/10/01 11:29:17 Copying /var/folders/qv/7l781jhs6j19gw3b89f4fcz40000gq/T/helm to
/Users/gigi.sayfan/.arkade/bin/helm
Downloaded to:  /Users/gigi.sayfan/.arkade/bin/helm helm
"openfaas" has been added to your repositories

Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "openfaas" chart repository
Update Complete. ⎈Happy Helming!⎈

VALUES values-arm64.yaml
Command: /Users/gigi.sayfan/.arkade/bin/helm [upgrade --install
openfaas openfaas/openfaas --namespace openfaas --values /var/folders/
qv/7l781jhs6j19gw3b89f4fcz40000gq/T/charts/openfaas/values-arm64.yaml --set
gateway.directFunctions=false --set openfaasImagePullPolicy=IfNotPresent
--set gateway.replicas=1 --set queueWorker.replicas=1 --set dashboard.
publicURL=http://127.0.0.1:8080 --set queueWorker.maxInflight=1 --set autoscaler.
enabled=false --set basic_auth=true --set faasnetes.imagePullPolicy=Always
--set basicAuthPlugin.replicas=1 --set clusterRole=false --set operator.
create=false --set ingressOperator.create=false --set dashboard.enabled=false --set
serviceType=NodePort]
Release "openfaas" does not exist. Installing it now.
NAME: openfaas
LAST DEPLOYED: Sat Oct  1 11:29:28 2022
NAMESPACE: openfaas
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
To verify that openfaas has started, run:

  kubectl -n openfaas get deployments -l "release=openfaas, app=openfaas"
=======================================================================
= OpenFaaS has been installed.                                        =
=======================================================================

# Get the faascli
kind-openfass | default
curl -SLsf https://cli.openfaas.com | sudo sh
```

```
# Forward the gateway to your machine
kubectl rollout status -n openfaas deploy/gateway
kubectl port-forward -n openfaas svc/gateway 8080:8080 &

# If basic auth is enabled, you can now log into your gateway:
PASSWORD=$(kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-auth-
password}" | base64 --decode; echo)
echo -n $PASSWORD | faas-cli login --username admin --password-stdin

faas-cli store deploy figlet
faas-cli list

# For Raspberry Pi
faas-cli store list \
 --platform armhf

faas-cli store deploy figlet \
 --platform armhf

# Find out more at:
# https://github.com/openfaas/faas

 arkade needs your support: https://github.com/sponsors/alexellis
```

OpenFaaS creates two namespaces: `openfaas` for itself and `openfass-fn` for your functions. There are several deployments in the `openfass` namespace:

```
$ k get deploy -n openfaas
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
alertmanager        1/1     1            1           6m2s
basic-auth-plugin   1/1     1            1           6m2s
gateway             1/1     1            1           6m2s
nats                1/1     1            1           6m2s
prometheus          1/1     1            1           6m2s
queue-worker        1/1     1            1           6m2s
```

The `openfass-fn` namespace is empty at the moment.

OK. Let's install the OpenFaaS CLI:

```
$ brew install faas-cli
==> Downloading https://ghcr.io/v2/homebrew/core/faas-cli/manifests/0.14.8
######################################################################## 100.0%
```

```
==> Downloading https://ghcr.io/v2/homebrew/core/faas-cli/blobs/
sha256:cf9460398c45ea401ac688e77a8884cbceaf255064a1d583f8113b6c2bd68450
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sha
256:cf9460398c45ea401ac688e77a8884cbceaf255064a1d583f8113b6c2bd68450?se=2022-10-
01T18%3A50%3A00Z&sig=V%
######################################################################## 100.0%
==> Pouring faas-cli--0.14.8.arm64_monterey.bottle.tar.gz
==> Caveats
zsh completions have been installed to:
  /opt/homebrew/share/zsh/site-functions
==> Summary
🍺  /opt/homebrew/Cellar/faas-cli/0.14.8: 9 files, 8.4MB
==> Running `brew cleanup faas-cli`...
Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).
```

First, we need to port-forward the gateway service so that faas-cli can access our cluster:

```
$ kubectl port-forward -n openfaas svc/gateway 8080:8080 &
[3] 76489
$ Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

The next step is to fetch the admin password from the secret called `basic-auth` and use it to log in as the admin user:

```
$ PASSWORD=$(kubectl get secret -n openfaas basic-auth -o jsonpath="{.data.basic-
auth-password}" | base64 --decode; echo)
echo -n $PASSWORD | faas-cli login --username admin --password-stdin
```

Now, we are ready to deploy and run functions on our cluster. Let's look at the function templates available in the store:

```
$ faas-cli template store list

NAME                    SOURCE          DESCRIPTION
csharp                  openfaas        Classic C# template
dockerfile              openfaas        Classic Dockerfile template
go                      openfaas        Classic Golang template
java11                  openfaas        Java 11 template
java11-vert-x           openfaas        Java 11 Vert.x template
node17                  openfaas        HTTP-based Node 17 template
node16                  openfaas        HTTP-based Node 16 template
```

```
node14                  openfaas           HTTP-based Node 14 template
node12                  openfaas           HTTP-based Node 12 template
node                    openfaas           Classic NodeJS 8 template
php7                    openfaas           Classic PHP 7 template
php8                    openfaas           Classic PHP 8 template
python                  openfaas           Classic Python 2.7 template
python3                 openfaas           Classic Python 3.6 template
python3-dlrs            intel              Deep Learning Reference Stack v0.4 for ML
workloads
ruby                    openfaas           Classic Ruby 2.5 template
ruby-http               openfaas           Ruby 2.4 HTTP template
python27-flask          openfaas           Python 2.7 Flask template
python3-flask           openfaas           Python 3.7 Flask template
python3-flask-debian    openfaas           Python 3.7 Flask template based on Debian
python3-http            openfaas           Python 3.7 with Flask and HTTP
python3-http-debian     openfaas           Python 3.7 with Flask and HTTP based on
Debian
golang-http             openfaas           Golang HTTP template
golang-middleware       openfaas           Golang Middleware template
python3-debian          openfaas           Python 3 Debian template
powershell-template     openfaas-incubator Powershell Core Ubuntu:16.04 template
powershell-http-template openfaas-incubator Powershell Core HTTP Ubuntu:16.04
template
rust                    booyaa             Rust template
crystal                 tpei               Crystal template
csharp-httprequest      distantcam         C# HTTP template
csharp-kestrel          burtonr            C# Kestrel HTTP template
vertx-native            pmlopes            Eclipse Vert.x native image template
swift                   affix              Swift 4.2 Template
lua53                   affix              Lua 5.3 Template
vala                    affix              Vala Template
vala-http               affix              Non-Forking Vala Template
quarkus-native          pmlopes            Quarkus.io native image template
perl-alpine             tmiklas            Perl language template based on Alpine
image
crystal-http            koffeinfrei        Crystal HTTP template
rust-http               openfaas-incubator Rust HTTP template
bash-streaming          openfaas-incubator Bash Streaming template
cobol                   devries            COBOL Template
```

What do you know! There is even a COBOL template if you're so inclined. For our purposes we will use Golang. There are several Golang templates. We will use the `golang-http` template. We need to pull the template for the first time:

```
$ faas-cli template store pull golang-http
Fetch templates from repository: https://github.com/openfaas/golang-http-template at
2022/10/02 14:48:38 Attempting to expand templates from https://github.com/openfaas/
golang-http-template
2022/10/02 14:48:39 Fetched 2 template(s) : [golang-http golang-middleware] from
https://github.com/openfaas/golang-http-template
```

The template has a lot of boilerplate code that takes care of all the ceremony needed to eventually produce a container that can be run on Kubernetes:

```
$ ls -la template/golang-http
total 64
drwxr-xr-x  11 gigi.sayfan  staff   352 Oct  2 14:48 .
drwxr-xr-x   4 gigi.sayfan  staff   128 Oct  2 14:52 ..
-rw-r--r--   1 gigi.sayfan  staff    52 Oct  2 14:48 .dockerignore
-rw-r--r--   1 gigi.sayfan  staff     9 Oct  2 14:48 .gitignore
-rw-r--r--   1 gigi.sayfan  staff  1738 Oct  2 14:48 Dockerfile
drwxr-xr-x   4 gigi.sayfan  staff   128 Oct  2 14:48 function
-rw-r--r--   1 gigi.sayfan  staff   110 Oct  2 14:48 go.mod
-rw-r--r--   1 gigi.sayfan  staff   257 Oct  2 14:48 go.sum
-rw-r--r--   1 gigi.sayfan  staff    32 Oct  2 14:48 go.work
-rw-r--r--   1 gigi.sayfan  staff  3017 Oct  2 14:48 main.go
-rw-r--r--   1 gigi.sayfan  staff   465 Oct  2 14:48 template.yml
```

Let's create our function:

```
$ faas-cli new --prefix docker.io/g1g1 --lang golang-http openfaas-go
Folder: openfaas-go created.
kind-openfaas | openfaas
  ___                   _____           ____
 / _ \ _ __   ___ _ __ |  ___|_ _  __ _/ ___|
| | | | '_ \ / _ \ '_ \| |_ / _` |/ _` \___ \
| |_| | |_) |  __/ | | |  _| (_| | (_| |___) |
 \___/| .__/ \___|_| |_|_|  \__,_|\__,_|____/
      |_|


Function created in folder: openfaas-go
Stack file written: openfaas-go.yml
```

```
Notes:
You have created a new function which uses Go 1.18 and Alpine
Linux as its base image.

To disable the go module, for private vendor code, please use
"--build-arg GO111MODULE=off" with faas-cli build or configure this
via your stack.yml file.

See more: https://docs.openfaas.com/cli/templates/

For the template's repo and more examples:
https://github.com/openfaas/golang-http-template
```

This command generated 3 files:

- `openfaas-go.yml`
- `openfaas-go/go.mod`
- `openfaas-go/handler.go`

Let's examine these files.

The `openfaas-go.yml` is our function manifest:

```
$ cat openfaas-go.yml
version: 1.0
provider:
  name: openfaas
  gateway: http://127.0.0.1:8080
functions:
  openfaas-go:
    lang: golang-http
    handler: ./openfaas-go
    image: docker.io/g1g1/openfaas-go:latest
```

Note that the image has the prefix of my Docker registry user account in case I want to push the image. Multiple functions can be defined in a single manifest file.

The `go.mod` is very basic:

```
$ cat openfaas-go/go.mod
module handler/function

go 1.18
```

The `handler.go` file is where we will write our code:

```
$ cat openfaas-go/handler.go
package function

import (
    "fmt"
    "net/http"

    handler "github.com/openfaas/templates-sdk/go-http"
)

// Handle a function invocation
func Handle(req handler.Request) (handler.Response, error) {
    var err error

    message := fmt.Sprintf("Body: %s", string(req.Body))

    return handler.Response{
        Body:       []byte(message),
        StatusCode: http.StatusOK,
    }, err
}
```

The default implementation is sort of an HTTP echo, where the response just returns the body of the request.

Let's build it. The default output is very verbose and shows a lot of output from Docker, so I will use the `--quiet` flag:

```
$ faas-cli build -f openfaas-go.yml --quiet
[0] > Building openfaas-go.
Clearing temporary build folder: ./build/openfaas-go/
Preparing: ./openfaas-go/ build/openfaas-go/function
Building: docker.io/g1g1/openfaas-go:latest with golang-http template. Please wait..
Image: docker.io/g1g1/openfaas-go:latest built.
[0] < Building openfaas-go done in 0.93s.
[0] Worker done.

Total build time: 0.93s
```

The result is a Docker image:

```
$ docker images | grep openfaas
g1g1/openfaas-go                    latest          215e95884a9b    3 minutes ago
18.3MB
```

We can push this image to the Docker registry (or other registries) if you have an account:

```
$ faas-cli push -f openfaas-go.yml
[0] > Pushing openfaas-go [docker.io/g1g1/openfaas-go:latest].
The push refers to repository [docker.io/g1g1/openfaas-go]
668bbc37657f: Pushed
185851557ef2: Pushed
1d14a6a345f2: Pushed
5f70bf18a086: Pushed
ecf2d64591ca: Pushed
f6b0a98cfe18: Pushed
5d3e392a13a0: Mounted from library/golang
latest: digest:
sha256:cb2b3051e2cac7c10ce78a844e331a5c55e9a2296c5c3ba9e0e8ee0523ceba84 size: 1780
[0] < Pushing openfaas-go [docker.io/g1g1/openfaas-go:latest] done.
```

The Docker image is now available on Docker Hub.

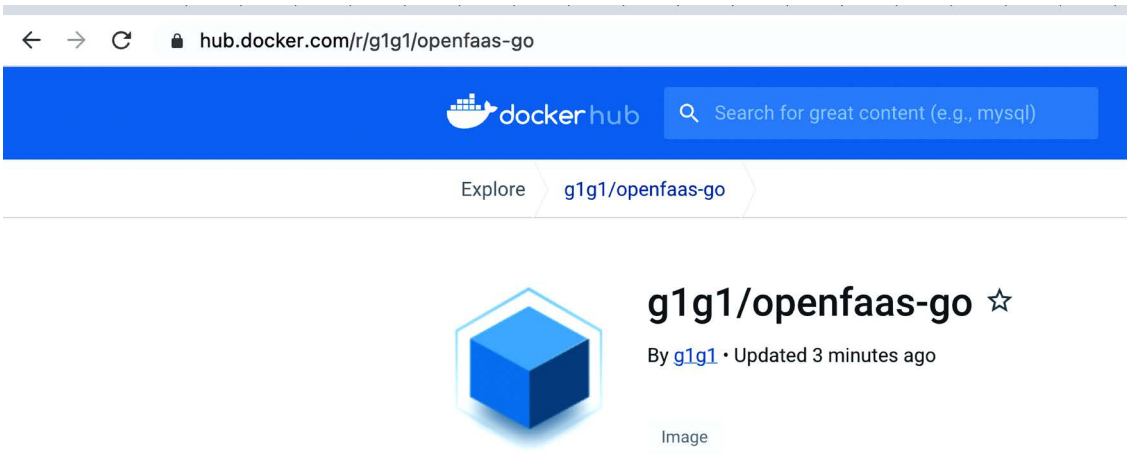


Figure 12.13: The Docker image is available on Docker Hub

The last step is to deploy the image to the cluster:

```
$ faas-cli deploy -f openfaas-go.yml
Deploying: openfaas-go.
Handling connection for 8080
```

```
Deployed. 202 Accepted.
URL: http://127.0.0.1:8080/function/openfaas-go
```

Let's invoke our function a few times with a different request body to see that the response is accurate:

```
$ http POST http://127.0.0.1:8080/function/openfaas-go body='yeah, it works!' -b
Handling connection for 8080
Body: {
    "body": "yeah, it works!"
}

$ http POST http://127.0.0.1:8080/function/openfaas-go body='awesome!' -b
Handling connection for 8080
Body: {
    "body": "awesome!"
}
```

Yeah, it works! Awesome!

We can see our function and some statistics, like the number of invocations and the number of replicas, using the `list` command:

```
$ faas-cli list
Handling connection for 8080
Function                        Invocations     Replicas
openfaas-go                     6               1
```

To summarize, OpenFaaS provides a mature and comprehensive solution for functions as a service on Kubernetes. It still requires you to build a Docker image, push it, and deploy it to the cluster in separate steps using its CLI. It is relatively simple to incorporate these steps into a CI/CD pipeline or a simple script.

Fission

Fission (`https://fission.io`) is a mature and well-documented framework. It models the FaaS world as environments, functions, and triggers. Environments are needed to build and run your function code for the specific languages. Each language environment contains an HTTP server and often a dynamic loader (for dynamic languages). Functions are the objects that represent the serverless functions and triggers are how the functions deployed in the cluster can be invoked. There are 4 kinds of triggers:

- **HTTP trigger**: Invoke a function via the HTTP endpoint.
- **Timer trigger**: Invoke a function at a certain time.
- **Message queue trigger**: Invoke a function when an event is pulled from the message queue (supports Kafka, NATS, and Azure queues).
- **Kubernetes watch trigger**: Invoke a function in response to a Kubernetes event in your cluster.

It's interesting that the message queue triggers are not just fire-and-forget. They support optional response and error queues. Here is a diagram that shows the flow:

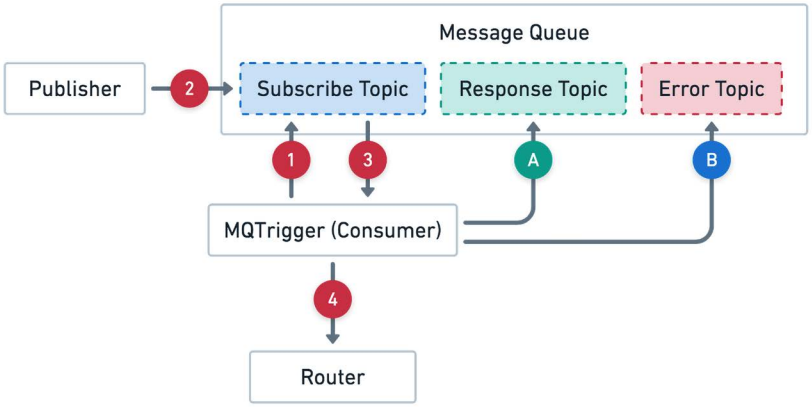


Figure 12.14: Fission mq trigger

Fission is proud of its 100 millisecond cold start. It achieves it by keeping a pool of "warm" containers with a small dynamic loader. When a function is first called there is a running container ready to go, and the code is sent to this container for execution. In a sense, Fission cheats because it never starts cold. The bottom line is that Fission doesn't scale to zero but is very fast for first-time calls.

Fission executor

Fission supports two types of executors - `NewDeploy` and `PoolManager`. The NewDeploy executor is very similar to OpenFaaS and creates a `Deployment`, `Service`, and HPA for each function. Here is what function invocation looks like with the NewDeploy executor:

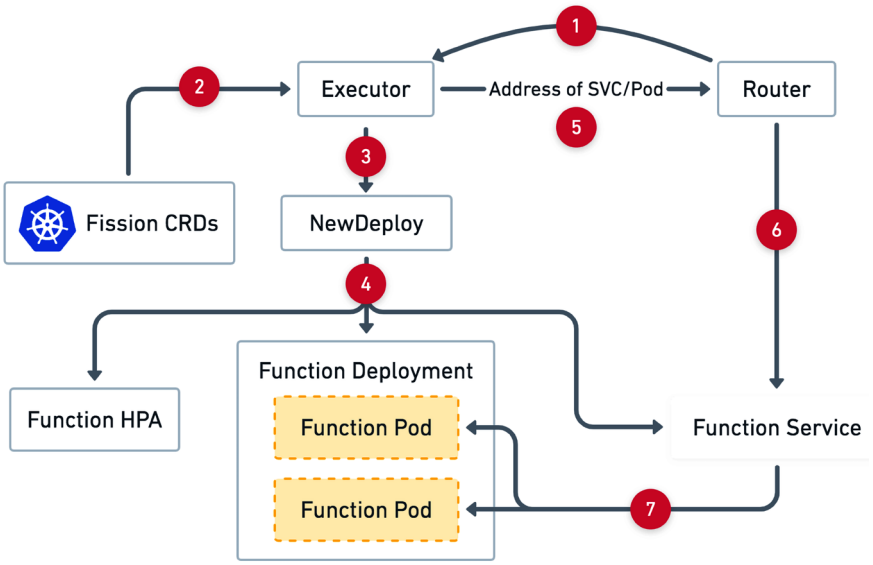


Figure 12.15: Fission function invocation

The `PoolManager` executor manages a pool of generic pods per environment. When a function for a particular environment is invoked, the `PoolManager` executor will run it on one of the available generic pools.

The `NewDeploy` executor allows fine-grained control over the resources needed to run a particular function, and it can also scale to zero. This comes at the cost of a higher cold start being needed to create a new pod for each function. Note that pods stick around, so if the same function is invoked again soon after the last invocation, it doesn't have to pay the cold start cost.

The `PoolManager` executor keeps generic pods around, so it is fast to invoke functions, but when no new functions need to be invoked, the pods in the pool just sit there idle. Also, functions can control the resources available to them.

You may use different executors for different functions depending on their usage patterns.

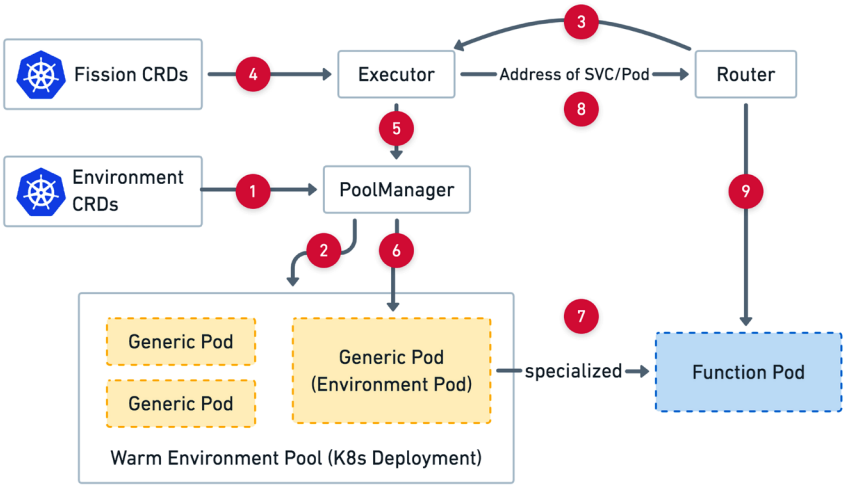


Figure 12.16: Fission executors

Fission workflows

Fission has one other claim to fame - Fission workflows. This is a separate project built on top of Fission. It allows you to build sophisticated workflows made of chains of Fission functions. It is currently in maintenance mode due to the time constraints of the core Fission team.

See the project's page for more details: `https://github.com/fission/fission-workflows`.

Here is a diagram that describes the architecture of Fission workflows:

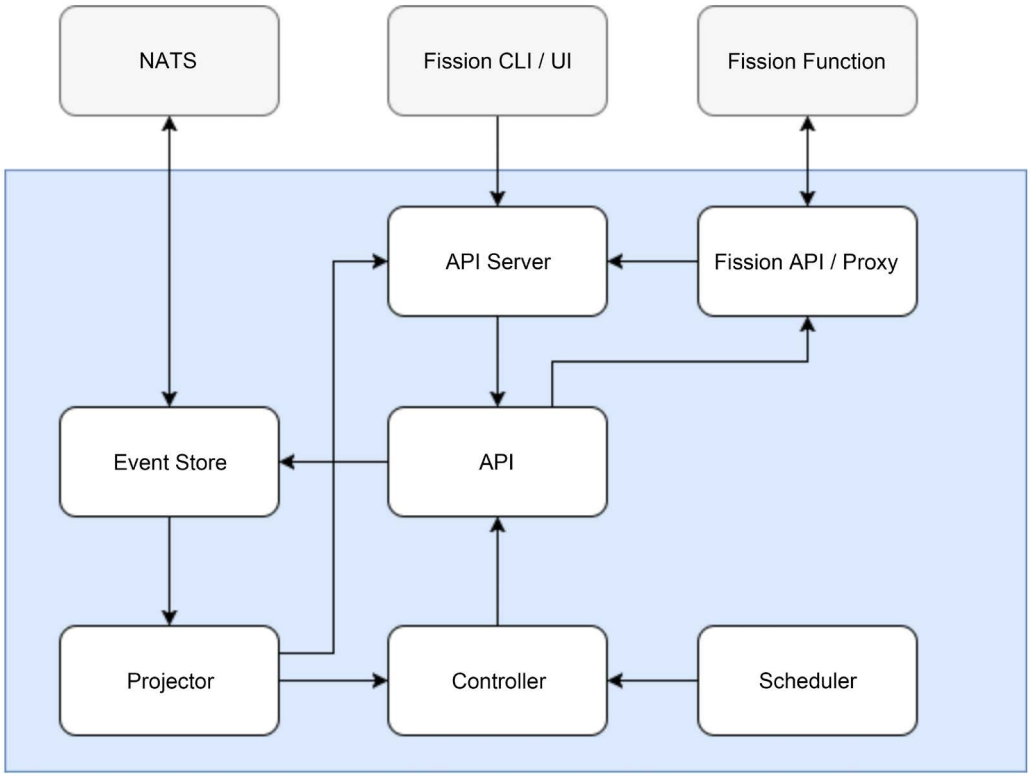


Figure 12.17: Fission workflows

You define workflows in YAML that specify tasks (often Fission functions), inputs, outputs, conditions, and delays. For example:

```
apiVersion: 1
description: Send a message to a slack channel when the temperature exceeds a
certain threshold
output: CreateResult
# Input: 'San Fransisco, CA'
tasks:
  # Fetch weather for input
  FetchWeather:
```

```
    run: wunderground-conditions
    inputs:
      default:
        apiKey: <API_KEY>
        state: "{$.Invocation.Inputs.default.substring($.Invocation.Inputs.
default.indexOf(',') + 1).trim()}"
        city: "{$.Invocation.Inputs.default.substring(0, $.Invocation.Inputs.
default.indexOf(',')).trim()}"

  ToCelsius:
    run: tempconv
    inputs:
      default:
        temperature: "{$.Tasks.FetchWeather.Output.current_observation.temp_f}"
        format: F
        target: C
    requires:
    - FetchWeather

  # Send a slack message if the temperature threshold has been exceeded
  CheckTemperatureThreshold:
    run: if
    inputs:
      if: "{$.Tasks.ToCelsius.Output.temperature > 25}"
      then:
        run: slack-post-message
        inputs:
          default:
            message: "{'It is ' + $.Tasks.ToCelsius.Output.temperature + 'C in
' + $.Invocation.Inputs.default + ' :fire:'}"
            path: <HOOK_URL>
    requires:
    - ToCelsius

    # Besides the potential Slack message, compose the response of this
workflow {location, celsius, fahrenheit}
  CreateResult:
    run: compose
    inputs:
      celsius: "{$.Tasks.ToCelsius.Output.temperature}"
      fahrenheit: "{$.Tasks.FetchWeather.Output.current_observation.temp_f}"
```

```
      location: "{$.Invocation.Inputs.default}"
      sentSlackMsg: "{$.Tasks.CheckTemperatureThreshold.Output}"
    requires:
    - ToCelsius
    - CheckTemperatureThreshold
```

Let's give Fission a try:

Experimenting with Fission

First, let's install it using Helm:

```
$ k create ns fission
$ k create -k "github.com/fission/fission/crds/v1?ref=v1.17.0"
$ helm repo add fission-charts https://fission.github.io/fission-charts/
$ helm repo update
$ helm install --version v1.17.0 --namespace fission fission \
  --set serviceType=NodePort,routerServiceType=NodePort \
  fission-charts/fission-all
```

Here are all the CRDs it created:

```
$ k get crd -o name | grep fission
customresourcedefinition.apiextensions.k8s.io/canaryconfigs.fission.io
customresourcedefinition.apiextensions.k8s.io/environments.fission.io
customresourcedefinition.apiextensions.k8s.io/functions.fission.io
customresourcedefinition.apiextensions.k8s.io/httptriggers.fission.io
customresourcedefinition.apiextensions.k8s.io/kuberneteswatchtriggers.fission.io
customresourcedefinition.apiextensions.k8s.io/messagequeuetriggers.fission.io
customresourcedefinition.apiextensions.k8s.io/packages.fission.io
customresourcedefinition.apiextensions.k8s.io/timetriggers.fission.io
```

The Fission CLI will come in handy too:

For Mac:

```
$ curl -Lo fission https://github.com/fission/fission/releases/download/v1.17.0/
fission-v1.17.0-darwin-amd64 && chmod +x fission && sudo mv fission /usr/local/bin/
```

For Linux or Windows on WSL:

```
$ curl -Lo fission https://github.com/fission/fission/releases/download/v1.17.0/
fission-v1.17.0-linux-amd64 && chmod +x fission && sudo mv fission /usr/local/bin/
```

We need to create an environment to be able to build our function. Let's go with a Python environment:

```
$ fission environment create --name python --image fission/python-env
poolsize setting default to 3
environment 'python' created
```

With a Python environment in place we can create a serverless function. First, save this code to `yeah.py`:

```
def main():
    return 'Yeah, it works!!!'
```

Then, we create the Fission function called "`yeah`":

```
$ fission function create --name yeah --env python --code yeah.py
Package 'yeah-b9d5d944-9c6e-4e67-81fb-96e047625b74' created
function 'yeah' created
```

We can test the function through the Fission CLI:

```
$ fission function test --name yeah
Yeah, it works!!!
```

The real deal is invoking it through an HTTP endpoint. We need to create a route for that:

```
$ fission route create --method GET --url /yeah --function yeah --name yeah
trigger 'yeah' created
```

With the route in place we still need to port-forward the service pod to expose it to the local environment:

```
$ k -n fission port-forward $(k -n fission get pod -l svc=router -o name) 8888:8888 &
$ export FISSION_ROUTER=127.0.0.1:8888
```

With all the preliminaries out of the way, let's test our function via httpie:

```
$ http http://${FISSION_ROUTER}/yeah -b
Handling connection for 8888
Yeah, it works!!!
```

You can skip the port-forwarding and test directly with the Fission CLI:

```
$ fission function test yeah --name yeah
Yeah, it works!!!
```

Fission is similar to OpenFaaS in its capabilities, but it feels a little bit more streamlined and easier to use. Both solutions are solid, and it's up to you to pick the one that you prefer.

Summary

In this chapter, we covered the hot topic of serverless computing. We explained the two meanings of serverless - eliminating the need to manage servers as well as deploying and running functions as a service. We explored in depth the aspects of serverless infrastructure in the cloud, especially in the context of Kubernetes. We compared the built-in cluster autoscaler as a Kubernetes-native serverless solution to the offerings of other cloud providers, like AWS EKS+Fargate, Azure AKS+ACI, and Google Cloud Run. We then switched gears and dove into the exciting and promising Knative project, with its scale-to-zero capabilities and advanced deployment options. Then, we moved to the wild world of FaaS on Kubernetes.

We discussed the plethora of solutions out there and examined them in detail, including hands-on experiments with two of the most prominent and battle-tested solutions out there: OpenFaaS and Fission. The bottom line is that both flavors of serverless computing bring real benefits in terms of operations and cost management. It's going to be fascinating to watch the evolution and consolidation of these technologies in the cloud and Kubernetes.

In the next chapter, our focus will be on monitoring and observability. Complex systems like large Kubernetes clusters with lots of different workloads, continuous delivery pipelines, and configuration changes must have excellent monitoring in place in order to keep all the balls in the air. Kubernetes has some great options that we should take advantage of.

13

Monitoring Kubernetes Clusters

In the previous chapter, we looked at serverless computing and its manifestations on Kubernetes. A lot of innovation happens in this space, and it is both super useful and fascinating to follow the evolution.

In this chapter, we're going to talk about how to make sure your systems are up and running and performing correctly, and how to respond when they're not. In *Chapter 3*, *High Availability and Reliability*, we discussed related topics. The focus here is on knowing what's going on in your system and what practices and tools you can use.

There are many aspects to monitoring, such as logging, metrics, distributed tracing, error reporting, and alerting. Practices like auto-scaling and self-healing depend on monitoring to detect that there is a need to scale or to heal.

The topics we will cover in this chapter include:

- Understanding observability
- Logging with Kubernetes
- Recording metrics with Kubernetes
- Distributed tracing with Jaeger
- Troubleshooting problems

The Kubernetes community recognizes the importance of monitoring and has put a lot of effort to make sure Kubernetes has a solid monitoring story. The **Cloud Native Computing Foundation** (**CNCF**) is the de-facto curator of cloud-native infrastructure projects. It has graduated twenty projects so far. Kubernetes was the first project to graduate, and out of the early graduated projects, three other projects that graduated more than two years ago are focused on monitoring: Prometheus, Fluentd, and Jaeger. This means that monitoring and observability are the foundation for a large-scale Kubernetes-based system. Before we dive into the ins and out of Kubernetes monitoring and specific projects and tools, we should get a better understanding of what monitoring is all about. A good framework for thinking about monitoring is how observable your system is.

Understanding observability

Observability is a big word. What does it mean in practice? There are different definitions out there and big debates about how monitoring and observability are similar and different. I take the stance that observability is the property of the system that defines what we can tell about the state and behavior of the system right now and historically. In particular, we are interested in the health of the system and its components. Monitoring is the collection of tools, processes, and techniques that we use to increase the observability of the system.

There are different facets of information that we need to collect, record, and aggregate in order to get a good sense of what our system is doing. Those facets include logs, metrics, distributed traces, and errors. The monitoring or observability data is multidimensional and crosses many levels. Just collecting it doesn't help much. We need to be able to query it, visualize it, and alert other systems when things go wrong. Let's review the various components of observability.

Logging

Logging is a key monitoring tool. Every self-respecting long-running software must have logs. Logs capture timestamped events. They are critical for many applications like business intelligence, security, compliance, audits, debugging, and troubleshooting. It's important to understand that a complicated distributed system will have different logs for different components, and extracting insights from logs is not a trivial undertaking.

There are several key attributes to logs: format, storage, and aggregation.

Log format

Logs may come in various formats. Plain text is very common and human-readable but requires a lot of work to parse and merge with other logs. Structured logs are better suitable for large systems because they can be processed at scale. Binary logs make sense for systems that generate a lot of logs as they are more space efficient, but require custom tools and processing to extract their information.

Log storage

Logs can be stored in memory, on the file system, in a database, in cloud storage, sent to a remote logging service, or any combination of these. In the cloud-native world, where software runs in containers, it's important to pay special attention to where logs are stored and how to fetch them when necessary.

Questions like durability come to mind when containers can come and go. In Kubernetes, the standard output and standard error streams of containers are automatically logged and available, even when the pod terminates. But, issues like having enough space for logs and log rotation are always relevant.

Log aggregation

In the end, the best practice is to send local logs to a centralized logging service that is designed to handle various log formats, persist them as necessary, and aggregate many types of logs in a way that can be queried and reasoned about.

Metrics

Metrics measure some aspects of the system over time. Metrics are time series of numerical values (typically floating point numbers). Each metric has a name and often a set of labels that help later in slicing and dicing. For example, the CPU utilization of a node or the error rate of a service are metrics.

Metrics are much more economical than logs. They require a fixed amount of space per time period that doesn't ebb and flow with incoming traffic like logs.

Also, since metrics are numerical in nature, they don't need parsing or transformations. Metrics can be easily combined and analyzed using statistical methods and serve as triggers for events and alerts.

A lot of metrics at different levels (node, container, process, network, and disk) are often collected for you automatically by the OS, cloud provider, or Kubernetes.

But you can also create custom metrics that map to high-level concerns of your system and can be configured with application-level policies.

Distributed tracing

Modern distributed systems often use microservice-based architecture, where an incoming request is bounced between multiple microservices, waits in queues, and triggers serverless functions. When you try to analyze errors, failures, data integrity issues, or performance issues, it is critical to be able to follow the path of a request. This is where distributed tracing comes in.

A distributed trace is a collection of spans and references. You can think of a trace as a **directed acyclic graph** (**DAG**) that represents a request's traversal through the components of a distributed system. Each span records the time the request spent in a given component and references are the edges of the graph that connect one span to the following spans.

Here is an example:

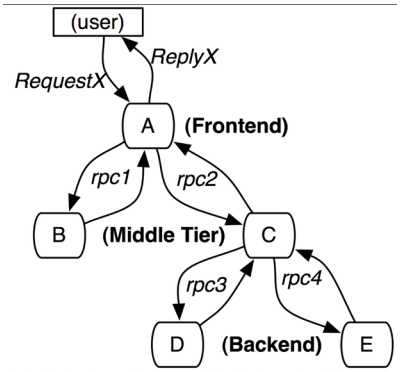


Figure 13.1: The path of a sample distributed trace

Distributed tracing is indispensable for understanding complex distributed systems.

Application error reporting

Error and exception reporting is sometimes done as part of logging. You definitely want to log errors, and looking at logs when things go wrong is a time-honored tradition. However, there are levels for capturing error information that go beyond logging. When an error occurs in one of your applications, it is useful to capture an error message, the location of the error in the code, and the stack trace. This is pretty standard and most programming languages can provide all this information, although stack traces are multi-line and don't fit well with line-based logs. A useful piece of additional information to capture is the local state in each level of the stack trace. This helps when a problem occurs in a central place but local states, like the number and size of entries in some lists can help identify the root cause.

A central error reporting service like Sentry or Rollbar provides a lot of value specific to errors beyond logging, such as rich error information, context, and user information.

Dashboards and visualization

OK. You've done a great job of collecting logs, defining metrics, tracing your requests, and reporting rich errors. Now, you want to figure out what your system or parts of it, are doing. What is the baseline? How does traffic fluctuate throughout the day, week, and on holidays? When the system is under stress, what parts are the most vulnerable?

In a complicated system that involves hundreds and thousands of services and data stores and integrates with external systems, you can't just look at the raw log files, metrics, and traces.

You need to be able to combine a lot of information and build system health dashboards, visualize your infrastructure, and create business-level reports and diagrams.

You may get some of it (especially for infrastructure) automatically if you're using cloud platforms. But you should expect to do some serious work around visualization and dashboards.

Alerting

Dashboards are great for humans that want to get a broad view of the system and be able to drill down and understand how it behaves. Alerting is all about detecting abnormal situations and triggering some action. Ideally, your system is self-healing and can recover on its own from most situations. But you should at least report it, so humans can review what happened at their leisure and decide if further action is needed.

Alerting can be integrated with emails, chat rooms, and on-call systems. It is often linked to metrics, and when certain conditions apply, an alert is raised.

Now that we have covered, in general, the different elements involved in monitoring complex systems, let's see how to do it with Kubernetes.

Logging with Kubernetes

We need to consider carefully our logging strategy with Kubernetes. There are several types of logs that are relevant for monitoring purposes. Our workloads run in containers, of course, and we care about these logs, but we also care about the logs of Kubernetes components like the API server, kubelet, and container runtime.

In addition, chasing logs across multiple nodes and containers is a non-starter. The best practice is to use central logging (a.k.a. log aggregation). There are several options here, which we will explore soon.

Container logs

Kubernetes stores the standard output and standard error of every container. They are available through the `kubectl logs` command.

Here is a pod manifest that prints the current date and time every 10 seconds:

```
apiVersion: v1
kind: Pod
metadata:
  name: now
spec:
  containers:
    - name: now
      image: g1g1/py-kube:0.3
      command: ["/bin/bash", "-c", "while true; do sleep 10; date; done"]
```

We can save it to a file called `now-pod.yaml` and create it:

```
$ k apply -f now-pod.yaml
pod/now created
```

To check out the logs, we use the `kubectl logs` command:

```
$ kubectl logs now
Sat Jan  4 00:32:38 UTC 2020
Sat Jan  4 00:32:48 UTC 2020
Sat Jan  4 00:32:58 UTC 2020
Sat Jan  4 00:33:08 UTC 2020
Sat Jan  4 00:33:18 UTC 2020
```

A few points about container logs. The `kubectl logs` command expects a pod name. If the pod has multiple containers, you need to specify the container name too:

```
$ k logs <pod name> -c <container name>
```

If a deployment or replica set creates multiple copies of the same pod, you can query the logs of all pods in a single call by using a shared label:

```
k logs -l <label>
```

If a container crashes for some reason, you can use the `kubectl logs -p` command to look at logs from the crashed container.

Kubernetes component logs

If you run Kubernetes in a managed environment like GKE, EKS, or AKS, you won't be able to access Kubernetes component logs directly, but this is expected. You're not responsible for the Kubernetes control plane. However, the logs of control plane components (like the API server and cluster autoscaler), as well as node components (like the kubelet and container runtime), may be important for troubleshooting issues. Cloud providers often offer proprietary ways to access these logs.

Here are the standard control plane components and their log location if you run your own Kubernetes control plane:

- API server: `/var/log/kube-apiserver.log`
- Scheduler: `/var/log/kube-scheduler.log`
- Controller manager: `/var/log/kube-controller-manager.log`

The worker node components and their log locations are:

- Kubelet: `/var/log/kubelet.log`
- Kube proxy: `/var/log/kube-proxy.log`

Note that on a systemd-based system, you'll need to use `journalctl` to view the worker node logs.

Centralized logging

Reading container logs is fine for quick and dirty troubleshooting problems in a single pod. To diagnose and debug system-wide issues, we need centralized logging (a.k.a. log aggregation). All the logs from our containers should be sent to a central repository and made accessible for slicing and dicing using filters and queries.

When deciding on your central logging approach, there are several important decisions:

- How to collect the logs
- Where to store the logs
- How to handle sensitive log information

We will answer these questions in the following sections.

Choosing a log collection strategy

Logs are collected typically by an agent that is running close to the process generating the logs and making sure to deliver them to the central logging service.

Let's look at the common approaches.

Direct logging to a remote logging service

In this approach, there is no log agent. It is the responsibility of each application container to send logs to the remote logging service. This is typically done through a client library. It is a high-touch approach and applications need to be aware of the logging target as well as be configured with proper credentials.

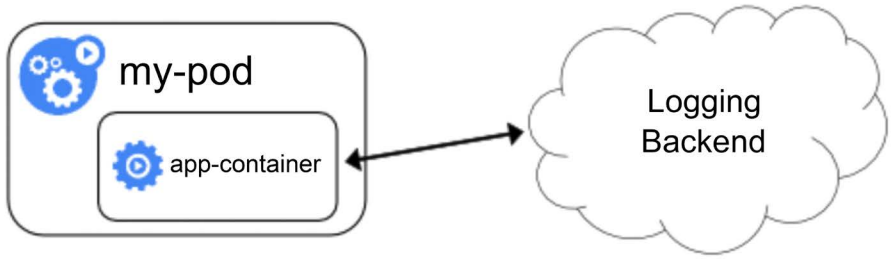


Figure 13.2: Direct logging

If you ever want to change your log collection strategy, it will require changes to each and every application (at least bumping to a new version of the library).

Node agent

The node agent approach is best when you control the worker nodes, and you want to abstract away the act of log aggregation from your applications. Each application container can simply write to standard output and standard error and the agent running on each node will intercept the logs and deliver them to the remote logging service.

Typically, you deploy the node agent as a DaemonSet so, as nodes are added or removed from the cluster, the log agent will always be present without additional work.

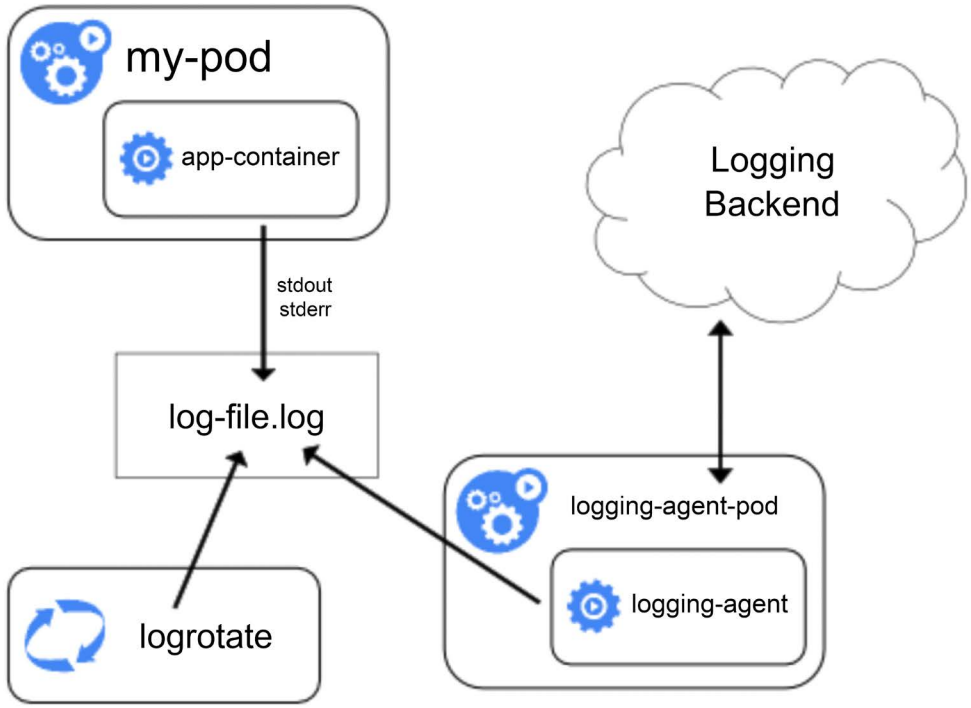


Figure 13.3: Using a node agent for logging

Sidecar container

The sidecar container is best when you don't have control over your cluster nodes, or if you use some serverless computing infrastructure to deploy containers but don't want to use the direct logging approach. The node agent approach is out of the question if you don't control the node and can't install the agent, but you can attach a sidecar container that will collect the logs and deliver them to the central logging service. It is not as efficient as the node agent approach because each container will need its own logging sidecar container, but it can be done at the deployment stage without requiring code changes and application knowledge.

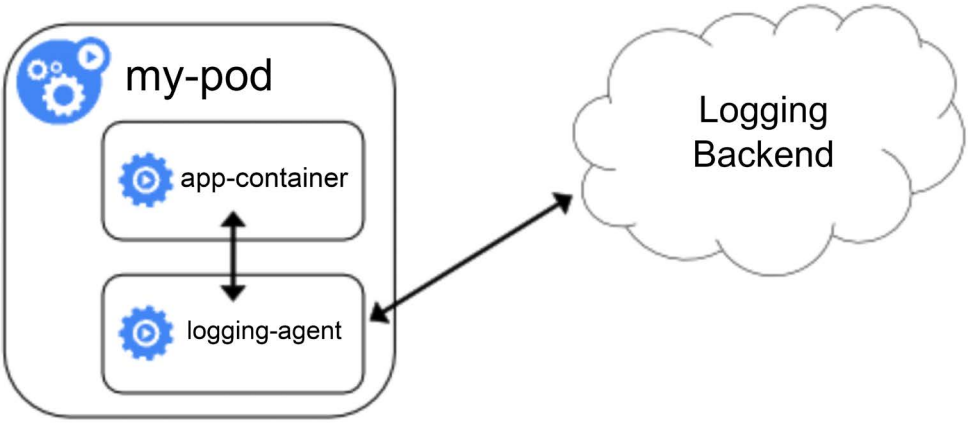


Figure 13.4: Using a sidecar container for logging

Now that we covered the topic of log collection, let's consider how to store and manage those logs centrally.

Cluster-level central logging

If your entire system is running in a single Kubernetes cluster, then cluster-level logging may be a great choice. You can install a central logging service like Grafana Loki, ElasticSearch, or Graylog in your cluster and enjoy a cohesive log aggregation experience without sending your log data elsewhere.

Remote central logging

There are situations where in-cluster central logging doesn't cut it for various reasons:

- Logs are used for audit purposes; it may be necessary to log to a separate and controlled location (e.g., on AWS, it is common to log to a separate account).
- Your system runs on multiple clusters and logging in each cluster is not really central.
- You run on a cloud provider and prefer to log into the cloud platform logging service (e.g., StackDriver on GCP or CloudWatch on AWS).
- You already work with a remote central logging service like SumoLogic or Splunk and you prefer to continue using them.
- You just don't want the hassle of collecting and storing log data.

- Cluster-wide issues can impact your log collection, storage, or access and impede your ability to troubleshoot them.

Logging to a remote central location can be done by all methods: direct logging, node agent logging, or sidecar logging. In all cases, the endpoint and credentials to the remote logging service must be provided, and the logging is done against that endpoint. In most cases, this will be done via a client library that hides the details from the applications. As for system-level logging, the common method is to collect all the necessary logs by a dedicated logging agent and forward them to the remote logging service.

Dealing with sensitive log information

OK. We can collect the logs and send them to a central logging service. If the central logging service is remote, you might need to be selective about which information you log.

For example, **personally identifiable information** (**PII**) and **protected health information** (**PHI**) are two categories of information that you probably shouldn't log without making sure access to the logs is properly controlled.

It is common to redact or remove PII like user names and emails from log statements.

Using Fluentd for log collection

Fluentd (`https://www.fluentd.org`) is an open source CNCF-graduated project. It is considered best in class in Kubernetes, and it can integrate with pretty much every logging backend you want. If you set up your own centralized logging solution, I recommend using Fluentd. Fluentd operates as a node agent. The following diagram shows how Fluentd can be deployed as a DaemonSet in a Kubernetes cluster:

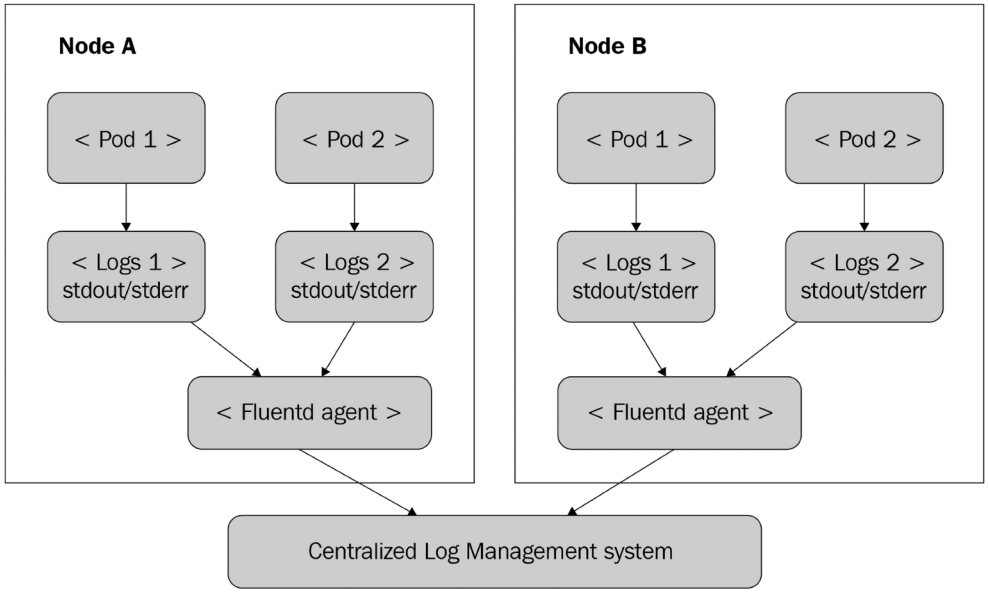


Figure 13.5: Deploying Fluentd as a DaemonSet in a Kubernetes cluster

One of the most popular DIY centralized logging solutions is ELK, where E stands for ElasticSearch, L stands for LogStash, and K stands for Kibana. On Kubernetes, EFK (where Fluentd replaces LogStash) is very common.

Fluentd has plugin-based architecture so don't feel limited to EFK. Fluentd doesn't require a lot of resources, but if you really need a high-performance solution, Fluentbit (`http://fluentbit.io/`) is a pure forwarder that uses barely 450 KB of memory.

We have covered a lot of ground about logging. Let's look at the next piece of the observability story, which is metrics.

Collecting metrics with Kubernetes

Kubernetes has a Metrics API. It supports node and pod metrics out of the box. You can also define your own custom metrics.

A metric contains a timestamp, a usage field, and the time range in which the metric was collected (many metrics are accumulated over a time period). Here is the API definition for node metrics:

```
type NodeMetrics struct {
    metav1.TypeMeta
    metav1.ObjectMeta

    Timestamp metav1.Time
    Window    metav1.Duration

    Usage corev1.ResourceList
}

// NodeMetricsList is a list of NodeMetrics.
type NodeMetricsList struct {
    metav1.TypeMeta
    // Standard list metadata.
    // More info: https://git.k8s.io/community/contributors/devel/sig-
architecture/api-conventions.md#types-kinds
    metav1.ListMeta

    // List of node metrics.
    Items []NodeMetrics
}
```

The usage field type is `ResourceList`, but it's actually a map of a resource name to a quantity:

```
// ResourceList is a set of (resource name, quantity) pairs.
type ResourceList map[ResourceName]resource.Quantity
```

A `Quantity` represents a fixed-point number. It allows easy marshaling/unmarshaling in JSON and YAML, and accessors such as `String()` and `Int64()`:

```
type Quantity struct {
    // i is the quantity in int64 scaled form, if d.Dec == nil
    i int64Amount

    // d is the quantity in inf.Dec form if d.Dec != nil
    d infDecAmount

    // s is the generated value of this quantity to avoid recalculation
    s string

    // Change Format at will. See the comment for Canonicalize for more
details.
    Format
}
```

Monitoring with the Metrics Server

The Kubernetes Metrics Server implements the Kubernetes Metrics API.

You can deploy it with Helm:

```
$ helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-server/
$ helm upgrade --install metrics-server metrics-server/metrics-server
Release "metrics-server" does not exist. Installing it now.
NAME: metrics-server
LAST DEPLOYED: Sun Oct  9 14:11:54 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
**********************************************************************
* Metrics Server                                                     *
**********************************************************************
  Chart version: 3.8.2
  App version:   0.6.1
  Image tag:     k8s.gcr.io/metrics-server/metrics-server:v0.6.1
**********************************************************************
```

On minikube, you can just enable it as an add-on:

```
$ minikube addons enable metrics-server
    ▪ Using image k8s.gcr.io/metrics-server/metrics-server:v0.4.2
🌟  The 'metrics-server' addon is enabled
```

Note that at the time of writing, there was an issue with the Metrics Server on minikube, which was fixed in Kubernetes 1.27 (see `https://github.com/kubernetes/minikube/issues/13969`).

We will use a kind cluster to deploy the `metrics-server`.

After a few minutes to let the metrics server collect some data, you can query it using these commands for node metrics:

```
$ k get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | jq .
{
  "kind": "NodeMetricsList",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {},
  "items": [
    {
      "metadata": {
        "name": "kind-control-plane",
        "creationTimestamp": "2022-10-09T21:24:12Z",
        "labels": {
          "beta.kubernetes.io/arch": "arm64",
          "beta.kubernetes.io/os": "linux",
          "kubernetes.io/arch": "arm64",
          "kubernetes.io/hostname": "kind-control-plane",
          "kubernetes.io/os": "linux",
          "node-role.kubernetes.io/control-plane": "",
          "node.kubernetes.io/exclude-from-external-load-balancers": ""
        }
      },
      "timestamp": "2022-10-09T21:24:05Z",
      "window": "20.022s",
      "usage": {
        "cpu": "115537281n",
        "memory": "47344Ki"
      }
    }
  ]
}
```

In addition, the `kubectl top` command gets its information from the metrics server:

```
$ k top nodes
NAME                 CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
kind-control-plane   125m         3%     46Mi            1%
```

We can also get metrics for pods:

```
$ k top pods -A
NAMESPACE            NAME                                         CPU(cores)
MEMORY(bytes)
default              metrics-server-554f79c654-hw2c7              4m
18Mi
kube-system          coredns-565d847f94-t8knf                     2m
12Mi
kube-system          coredns-565d847f94-wdqzx                     2m
14Mi
kube-system          etcd-kind-control-plane                      24m
28Mi
kube-system          kindnet-fvfs7                                1m
7Mi
kube-system          kube-apiserver-kind-control-plane            43m
339Mi
kube-system          kube-controller-manager-kind-control-plane   18m
48Mi
kube-system          kube-proxy-svdc6                             1m
11Mi
kube-system          kube-scheduler-kind-control-plane            4m
21Mi
local-path-storage   local-path-provisioner-684f458cdd-24w88      2m
6Mi
```

The metrics server is also the source of performance information in the Kubernetes dashboard.

The rise of Prometheus

Prometheus (`https://prometheus.io/`) is yet another graduated CNCF open source project. It focuses on metrics collection and alert management. It has a simple yet powerful data model for managing time-series data and a sophisticated query language. It is considered best in class in the Kubernetes world. Prometheus lets you define recording rules that are fired at regular intervals and collect data from targets. In addition, you can define alerting rules that evaluate a condition and trigger alerts if the condition is satisfied.

It has several unique features compared to other monitoring solutions:

- The collection system is pulled over HTTP. Nobody has to push metrics to Prometheus (but push is supported via a gateway).
- A multidimensional data model (each metric is a named time series with a set of key/value pairs attached to each data point).
- PromQL: A powerful and flexible query language to slice and dice your metrics.
- Prometheus server nodes are independent and don't rely on shared storage.
- Target discovery can be dynamic or via static configuration.
- Built-in time series storage, but supports other backends if necessary.
- Built-in alert manager and the ability to define alerting rules.

The following diagram illustrates the entire system:

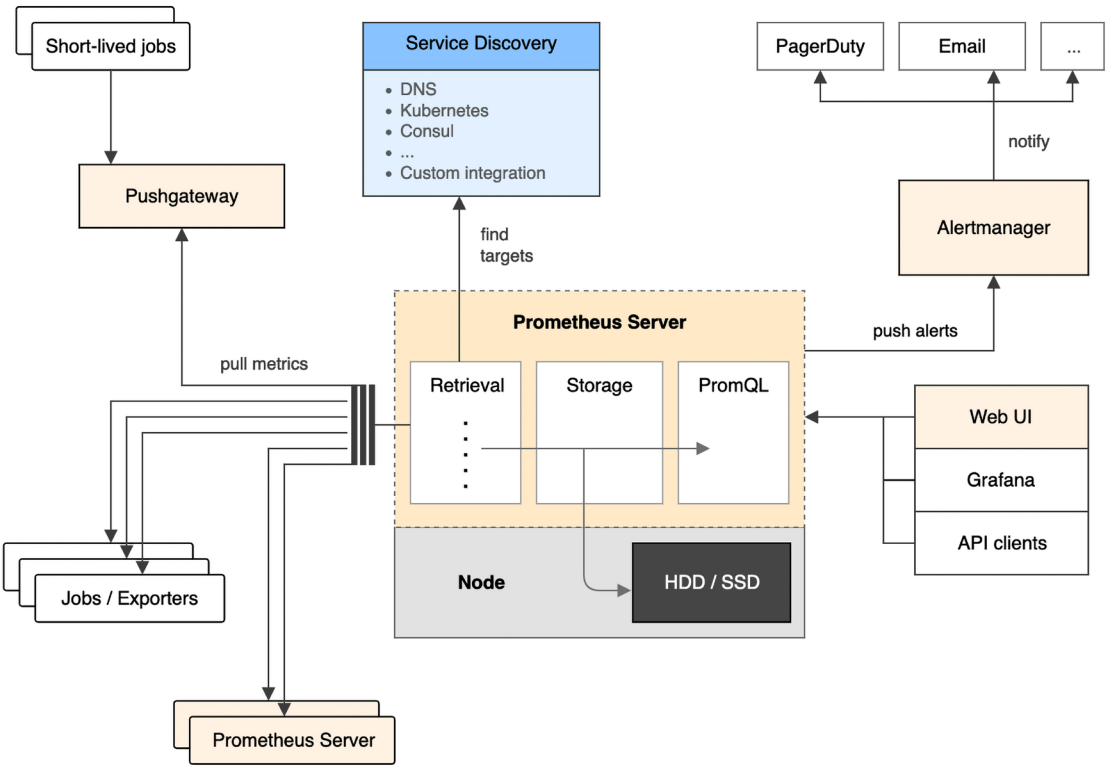


Figure 13.6: Prometheus architecture

Installing Prometheus

Prometheus is a complex beast, as you can see. The best way to install it is using the Prometheus operator (`https://github.com/prometheus-operator/`). The kube-prometheus (`https://github.com/prometheus-operator/kube-prometheus`) sub-project installs the operator itself, and a lot of additional components, and configures them in a robust manner.

The first step is cloning the git repo:

```
$ git clone https://github.com/prometheus-operator/kube-prometheus.git
Cloning into 'kube-prometheus'...
remote: Enumerating objects: 17062, done.
remote: Counting objects: 100% (185/185), done.
remote: Compressing objects: 100% (63/63), done.
remote: Total 17062 (delta 135), reused 155 (delta 116), pack-reused 16877
Receiving objects: 100% (17062/17062), 8.76 MiB | 11.63 MiB/s, done.
Resolving deltas: 100% (11135/11135), done.
```

Next, the setup manifests install several CRDs and creates a namespace called `monitoring`:

```
$ kubectl create -f manifests/setup
customresourcedefinition.apiextensions.k8s.io/alertmanagerconfigs.monitoring.coreos.
com created
customresourcedefinition.apiextensions.k8s.io/alertmanagers.monitoring.coreos.com
created
customresourcedefinition.apiextensions.k8s.io/podmonitors.monitoring.coreos.com
created
customresourcedefinition.apiextensions.k8s.io/probes.monitoring.coreos.com created
customresourcedefinition.apiextensions.k8s.io/prometheuses.monitoring.coreos.com
created
customresourcedefinition.apiextensions.k8s.io/prometheusrules.monitoring.coreos.com
created
customresourcedefinition.apiextensions.k8s.io/servicemonitors.monitoring.coreos.com
created
customresourcedefinition.apiextensions.k8s.io/thanosrulers.monitoring.coreos.com
created
namespace/monitoring created
```

Now, we can install the manifests:

```
$ kubectl create -f manifests
...
```

The output is too long to display, but let's examine what was actually installed. It turns out that it installed multiple deployments, StatefulSets, a DaemonSet, and many services:

```
$ k get deployments -n monitoring
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
blackbox-exporter     1/1     1            1           3m38s
grafana               1/1     1            1           3m37s
kube-state-metrics    1/1     1            1           3m37s
prometheus-adapter    2/2     2            2           3m37s
prometheus-operator   1/1     1            1           3m37s

$ k get statefulsets -n monitoring
NAME                READY   AGE
alertmanager-main   3/3     2m57s
prometheus-k8s      2/2     2m57s

$ k get daemonsets -n monitoring
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
AGE
```

```
node-exporter   1         1         1       1           1           kubernetes.io/
os=linux   4m4s

$ k get services -n monitoring
NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
alertmanager-main       ClusterIP   10.96.231.0     <none>        9093/TCP,8080/TCP
4m25s
alertmanager-operated   ClusterIP   None            <none>        9093/TCP,9094/
TCP,9094/UDP   3m35s
blackbox-exporter       ClusterIP   10.96.239.94    <none>        9115/TCP,19115/
TCP            4m25s
grafana                 ClusterIP   10.96.80.116    <none>        3000/TCP
4m24s
kube-state-metrics      ClusterIP   None            <none>        8443/TCP,9443/TCP
4m24s
node-exporter           ClusterIP   None            <none>        9100/TCP
4m24s
prometheus-adapter      ClusterIP   10.96.139.149   <none>        443/TCP
4m24s
prometheus-k8s          ClusterIP   10.96.51.85     <none>        9090/TCP,8080/TCP
4m24s
prometheus-operated     ClusterIP   None            <none>        9090/TCP
3m35s
prometheus-operator     ClusterIP   None            <none>        8443/TCP
4m24s
```

This is a high-availability setup. As you can see, Prometheus itself is deployed as a StatefulSet with two replicas, and the alert manager is deployed as a StatefulSet with three replicas.

The deployments include the `blackbox-exporter`, Grafana for visualizing the metrics, `kube-state-metrics` to collect Kubernetes-specific metrics, the Prometheus adapter (a compatible replacement to the standard Kubernetes Metrics Server), and finally, the Prometheus operator.

Interacting with Prometheus

Prometheus has a basic web UI that you can use to explore its metrics. Let's do port forwarding to `localhost`:

```
$ k port-forward -n monitoring statefulset/prometheus-k8s 9090
Forwarding from 127.0.0.1:9090 -> 9090
Forwarding from [::1]:9090 -> 9090
```

Then, you can browse to `http://localhost:9090`, where you can select different metrics and view raw data or graphs:

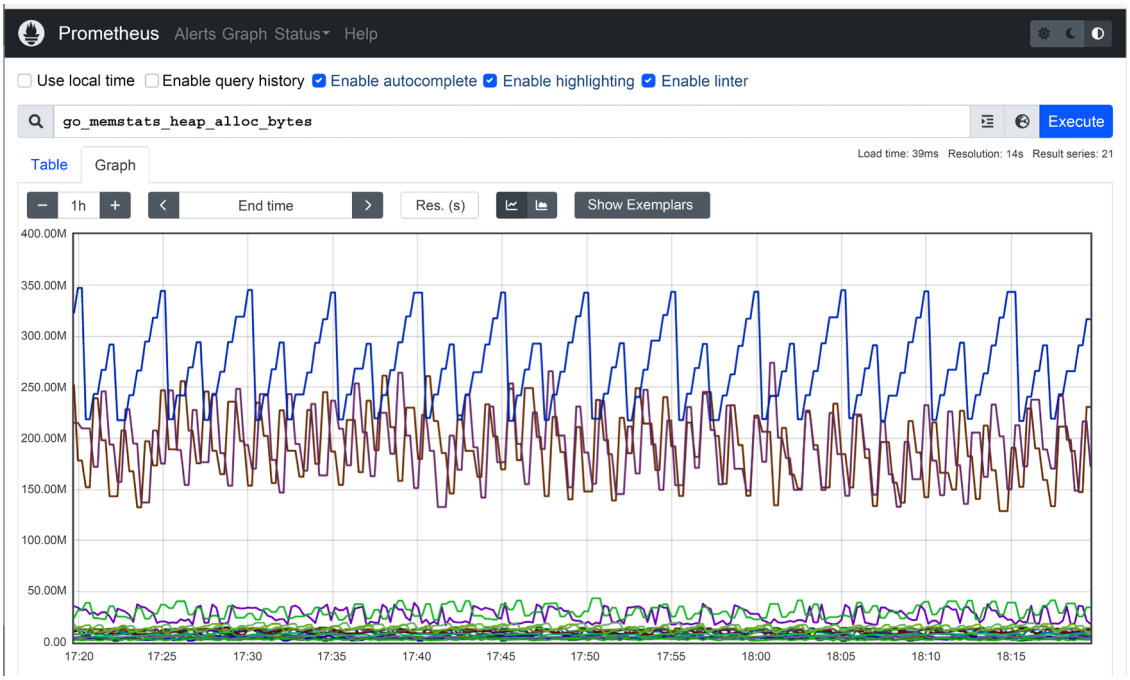


Figure 13.7: Prometheus UI

Prometheus records an outstanding number of metrics (`9090` in my current setup). The most relevant metrics on Kubernetes are the metrics exposed by the `kube-state-metrics` and node exporters.

Incorporating kube-state-metrics

The Prometheus operator already installs `kube-state-metrics`. It is a service that listens to Kubernetes events and exposes them through a `/metrics` HTTP endpoint in the format Prometheus expects. So, it is a Prometheus exporter.

This is very different from the Kubernetes metrics server, which is the standard way Kubernetes exposes metrics for nodes and pods and allows you to expose your own custom metrics too. The Kubernetes metrics server is a service that periodically queries Kubernetes for data and stores it in memory. It exposes its data through the Kubernetes Metrics API. The Prometheus adapter adapts the Kubernetes metrics server information and exposes it in Prometheus format.

The metrics exposed by `kube-state-metrics` are vast. Here is the list of the groups of metrics, which is pretty massive on its own. Each group corresponds to a Kubernetes API object and contains multiple metrics:

- `CertificateSigningRequest` Metrics
- `ConfigMap` Metrics
- `CronJob` Metrics

- `DaemonSet` Metrics
- `Deployment` Metrics
- `Endpoint` Metrics
- `HorizontalPodAutoscaler` Metrics
- `Ingress` Metrics
- `Job` Metrics
- `LimitRange` Metrics
- `MutatingWebhookConfiguration` Metrics
- `Namespace` Metrics
- `NetworkPolicy` Metrics
- `Node` Metrics
- `PersistentVolume` Metrics
- `PersistentVolumeClaim` Metrics
- `PodDisruptionBudget` Metrics
- `Pod` Metrics
- `ReplicaSet` Metrics
- `ReplicationController` Metrics
- `ResourceQuota` Metrics
- `Secret` Metrics
- `Service` Metrics
- `StatefulSet` Metrics
- `StorageClass` Metrics
- `ValidatingWebhookConfiguration` Metrics
- `VerticalPodAutoscaler` Metrics
- `VolumeAttachment` Metrics

For example, here are the metrics collected for Kubernetes services:

- `kube_service_info`
- `kube_service_labels`
- `kube_service_created`
- `kube_service_spec_type`

Utilizing the node exporter

`kube-state-metrics` collects node information from the Kubernetes API server, but this information is pretty limited. Prometheus comes with its own node exporter, which collects tons of low-level information about the nodes. Remember that Prometheus may be the de-facto standard metrics platform on Kubernetes, but it is not Kubernetes-specific. For other systems that use Prometheus, the node exporter is super important. On Kubernetes, if you manage your own nodes, this information can be invaluable too.

Here is a small subset of the metrics exposed by the node exporter:

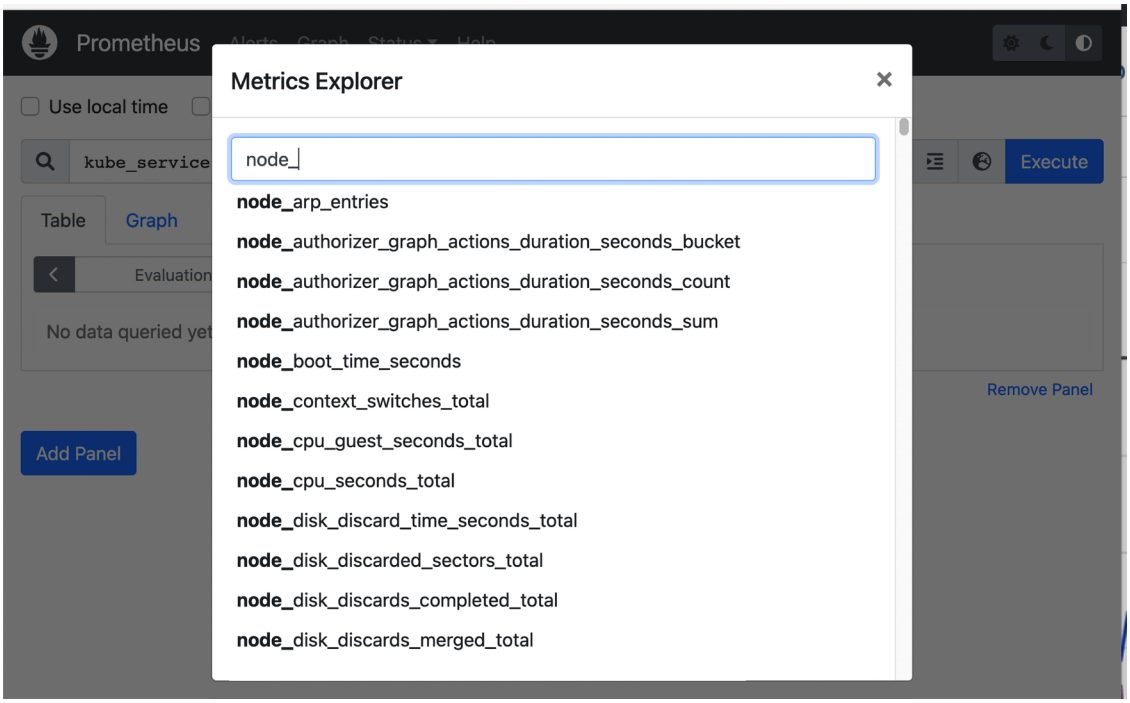


Figure 13.8: Node exporter metrics

Incorporating custom metrics

The built-in metrics, node metrics, and Kubernetes metrics are great, but very often, the most interesting metrics are domain-specific and need to be captured as custom metrics. There are two ways to do it:

- Write your own exporter and tell Prometheus to scrape it
- Use the Push gateway, which allows you to push metrics into Prometheus

In my book *Hands-On Microservices with Kubernetes* (`https://www.packtpub.com/product/hands-on-microservices-with-kubernetes/9781789805468`), I provide a full-fledged example of how to implement your own exporter from a Go service.

The Push gateway is more appropriate if you already have a push-based metrics collector in place, and you just want to have Prometheus record those metrics. It provides a convenient migration path from other metrics collection systems to Prometheus.

Alerting with Alertmanager

Collecting metrics is great, but when things go south (or ideally, BEFORE things go south), you want to get notified. In Prometheus, this is the job of the `Alertmanager`. You can define rules as expressions-based metrics, and when those expressions become true, they trigger an alert.

Alerts can serve multiple purposes. They can be handled automatically by a controller that is responsible for mitigating specific problems, they can wake up a poor on-call engineer at 3 am, they can result in an email or group chat message, or any combination of these.

The `Alertmanager` lets you group similar alerts into a single notification, inhibiting notifications if other alerts are already firing and silencing alerts. All those features are useful when a large-scale system is in trouble. The stakeholders are aware of the situation and don't need repeated alerts or multiple variations of the same alert to fire constantly while troubleshooting and trying to find the root cause.

One of the cool things about the Prometheus operator is that it manages everything in CRDs. That includes all the rules, including the alert rules:

```
$ k get prometheusrules -n monitoring
NAME                              AGE
alertmanager-main-rules           11h
grafana-rules                     11h
kube-prometheus-rules             11h
kube-state-metrics-rules          11h
kubernetes-monitoring-rules       11h
node-exporter-rules               11h
prometheus-k8s-prometheus-rules   11h
prometheus-operator-rules         11h
```

Here is the `NodeFilesystemAlmostOutOfSpace` alert that checks if available disk space for the file system on the node is less than a threshold for 30 minutes. If you notice, there are two almost identical alerts. When the available space drops below 5%, a warning alert is triggered. However, if the space drops below 3%, then a critical alert is triggered. Note the `runbook_url` field, which points to a page that explains more about the alert and how to mitigate the problem:

```
$ k get prometheusrules node-exporter-rules -n monitoring -o yaml | grep
NodeFilesystemAlmostOutOfSpace -A 14
    - alert: NodeFilesystemAlmostOutOfSpace
      annotations:
        description: Filesystem on {{ $labels.device }} at {{ $labels.instance }}
          has only {{ printf "%.2f" $value }}% available space left.
        runbook_url: https://runbooks.prometheus-operator.dev/runbooks/node/
nodefilesystemalmostoutofspace
      expr: |
        (
```

```
          node_filesystem_avail_bytes{job="node-exporter",fstype!=""} / node_
filesystem_size_bytes{job="node-exporter",fstype!=""} * 100 < 5
        and
          node_filesystem_readonly{job="node-exporter",fstype!=""} == 0
        )
      for: 30m
      labels:
        severity: warning
    - alert: NodeFilesystemAlmostOutOfSpace
      annotations:
        description: Filesystem on {{ $labels.device }} at {{ $labels.instance }}
          has only {{ printf "%.2f" $value }}% available space left.
        runbook_url: https://runbooks.prometheus-operator.dev/runbooks/node/
nodefilesystemalmostoutofspace
        summary: Filesystem has less than 3% space left.
      expr: |
        (
          node_filesystem_avail_bytes{job="node-exporter",fstype!=""} / node_
filesystem_size_bytes{job="node-exporter",fstype!=""} * 100 < 3
        and
          node_filesystem_readonly{job="node-exporter",fstype!=""} == 0
        )
      for: 30m
      labels:
        severity: critical
```

Alerts are very important, but there are cases where you want to visualize the overall state of your system or drill down into specific aspects. This is where visualization comes into play.

Visualizing your metrics with Grafana

You've already seen the Prometheus Expression browser, which can display your metrics as a graph or in table form. But we can do much better. Grafana (`https://grafana.com`) is an open source monitoring system that specializes in stunningly beautiful visualizations of metrics. It doesn't store the metrics itself but works with many data sources, and Prometheus is one of them. Grafana has alerting capabilities, too. When working with Prometheus, you may prefer to rely on its `Alertmanager`.

The Prometheus operator installs Grafana and configures a large number of useful Kubernetes dashboards. Check out this beautiful dashboard of Kubernetes capacity:



Figure 13.9: Grafana dashboard

To access Grafana, type the following commands:

```
$ k port-forward -n monitoring deploy/grafana 3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

Then you can browse to `http://localhost:3000` and have some fun with Grafana. Grafana requires a username and password. The default credentials are *admin* for the user and *admin* for the password.

Here are some of the default dashboards that are configured when deploying Grafana via kube-prometheus:

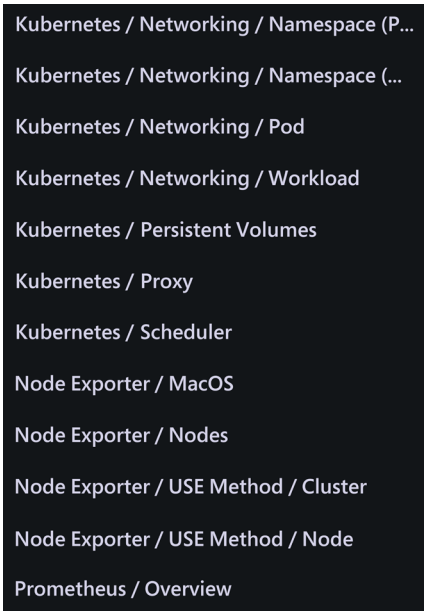


Figure 13.10: Default Grafana Dashboards

As you can see, the list is pretty extensive, but you can define your own dashboards if you want. There are a lot of fancy visualizations you can create with Grafana. I encourage you to explore it further. Grafana dashboards are stored as config maps. If you want to add a custom dashboard, just add a config map that contains your dashboard spec. There is a dedicated sidecar container watching new config maps being added and it will make sure to add your custom dashboard.

You can also add dashboards via the Grafana UI.

Considering Loki

If you like Prometheus and Grafana and you didn't settle on a centralized logging solution yet (or if you're unhappy with your current logging solution), then you should consider Grafana Loki (`https://grafana.com/oss/loki/`). Loki is an open source project for log aggregation inspired by Prometheus. Unlike most log aggregation systems, it doesn't index the log contents but rather a set of labels applied to the log. That makes it very efficient. It is still relatively new (started in 2018), so you should evaluate if it fits your needs before making the decision to adopt it. One thing is sure: Loki has excellent Grafana support.

There are several advantages for Loki compared to a solution like EFK when Prometheus is used as the metrics platform. In particular, the set of labels you use to tag your metrics will serve just as well to tag your logs. Also, the fact that Grafana is used as a uniform visualization platform for both logs and metrics is useful.

We have dedicated a lot of time to discussing metrics on Kubernetes. Let's talk about distributed tracing and the Jaeger project.

Distributed tracing with Kubernetes

In a microservice-based system, every request may travel between multiple microservices calling each other, wait in queues, and trigger serverless functions. To debug and troubleshoot such systems, you need to be able to keep track of requests and follow them along their path.

Distributed tracing provides several capabilities that allow the developers and operators to understand their distributed systems:

- Distributed transaction monitoring
- Performance and latency tracking
- Root cause analysis
- Service dependency analysis
- Distributed context propagation

Distributed tracing often requires the participation of the applications and services instrumenting endpoints. Since the microservices world is polyglot, multiple programming languages may be used. It makes sense to use a shared distributed tracing specification and framework that supports many programming languages. Enter OpenTelemetry.

What is OpenTelemetry?

OpenTelemetry (`https://opentelemetry.io`) is an API specification and a set of frameworks and libraries in different languages to instrument, collect, and export logs, metrics, and traces. It was born by merging the OpenCensus and OpenTracing projects in May 2019. It is also an incubating CNCF project. OpenTelemetry is supported by multiple products and became a de-facto standard. It can collect data from a variety of open source and commercial sources. Check out the full list here: `https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/receiver`.

By using a product that complies with OpenTelemetry, you are not locked in, and you will work with an API that may be familiar to your developers.

There are instrumentation libraries for pretty much all the mainstream programming languages:

- C++
- .NET
- Erlang/Elixir
- Go
- Java

- JavaScript
- PHP
- Python
- Ruby
- Rust
- Swift

OpenTelemetry tracing concepts

We will focus here on the tracing concept of OpenTelemetry and skip the logging and metrics concepts we covered earlier.

The two main concepts are **Span** and **Trace**.

A **Span** is the basic unit of work or operation. It has a name, start time, and duration. Spans can be nested if one operation starts another operation. Spans propagate with a unique ID and context. The **Trace** is an acyclic graph of Spans that originated from the same request and shares the same context. A **Trace** represents the execution path of a request throughout the system. The following diagram illustrates the relationship between a Trace and Spans:

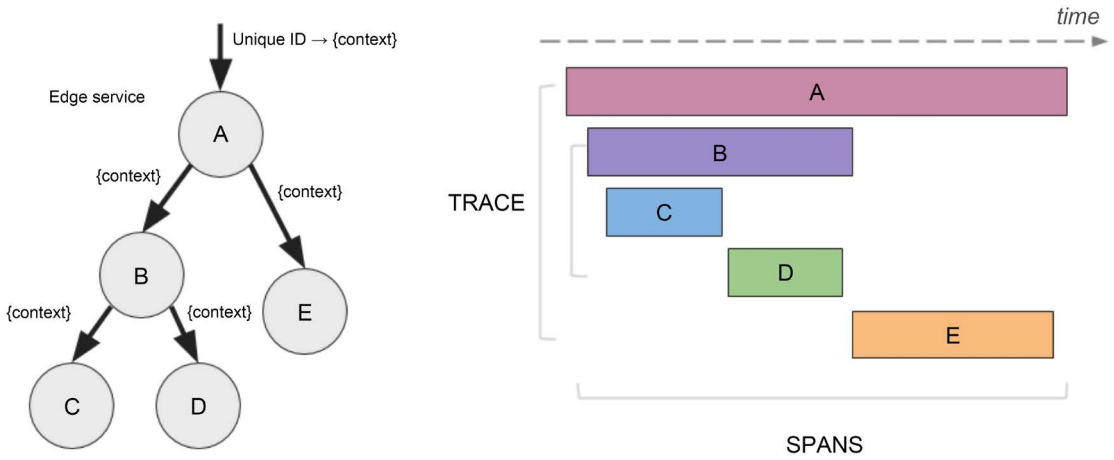


Figure 13.11: The relationship between Traces and Spans in OpenTelemetry

Now that we understand what OpenTelemetry is about, let's take a look at the Jaeger project.

Introducing Jaeger

Jaeger (`https://www.jaegertracing.io/`) is yet another CNCF-graduated project, just like Fluentd and Prometheus. It completes the trinity of CNCF-graduated observability projects for Kubernetes. Jaeger was developed originally by Uber and quickly became the forerunner distributed tracing solution for Kubernetes.

There are other open-source distributed tracing systems like Zipkin (`https://zipkin.io`) and SigNoz (`https://signoz.io`). The inspiration for most of these systems (as well as Jaeger) is Google's Dapper (`https://research.google.com/pubs/pub36356.html`). Cloud platforms provide their own tracers, like AWS X-Ray. There are also multiple commercial products in the space:

- Aspecto (`https://www.aspecto.io`)
- Honeycomb (`https://www.honeycomb.io`)
- Lightstep (`http://lightstep.com`)

Jaeger's strong points are:

- Scalable design
- Supports multiple protocols – OpenTelemetry, OpenTracing, and Zipkin
- Light memory footprint
- Agents collect metrics over UDP
- Advanced sampling control

Jaeger architecture

Jaeger is a scalable system. It can be deployed as a single binary with all its components and store the data in memory, but also as a distributed system where spans and traces are stored in persistent storage.

Jaeger has several components that collaborate to provide a world-class distributed tracing experience. The following diagram illustrates the architecture:

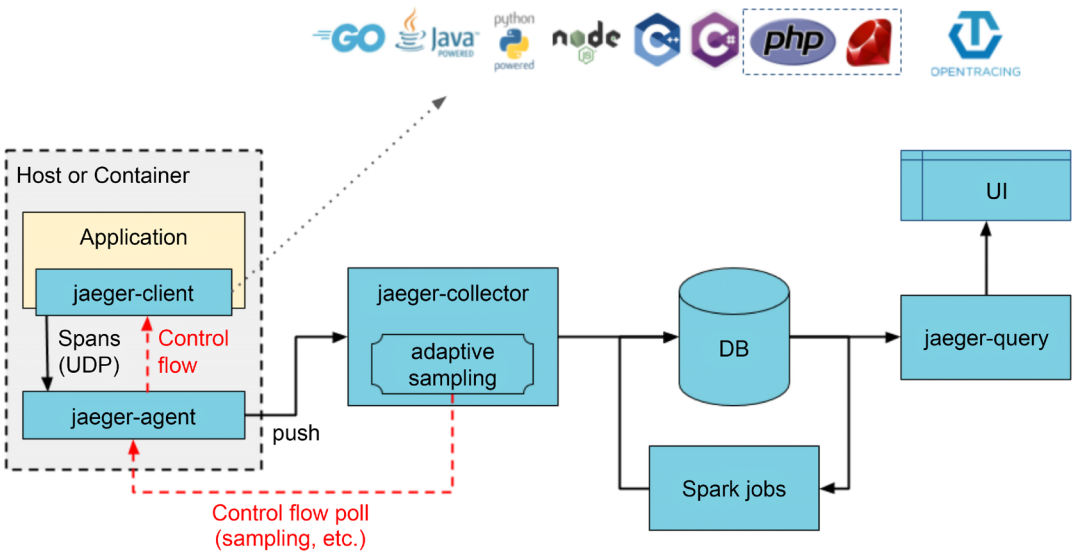


Figure 13.12: Jaeger architecture

Let's understand the purpose of each component.

Client libraries

Originally, Jaeger had its own client libraries that implemented the OpenTracing API in order to instrument a service or application for distributed tracing. Now, Jaeger recommends using the OpenTelemetry client libraries. The Jaeger client libraries have been retired.

Jaeger agent

The agent is deployed locally to each node. It listens to spans over UDP – which makes it pretty performant – batches them and sends them in bulk to the collector. This way, services don't need to discover the collector or worry about connecting to it. Instrumented services simply send their spans to the local agent. The agent can also inform the client about sampling strategies.

Jaeger collector

The collector receives traces from all the agents. It is responsible for validating and transforming the traces. It then sends the traces to a data store to a Kafka instance, which enables async processing of traces.

Jaeger ingester

The ingester indexes the traces for easy and performant queries later and stores them in the data store, which can be a Cassandra or Elasticsearch cluster.

Jaeger query

The Jaeger query service is responsible for presenting a UI to query the traces and the spans that the collector put in storage.

That covers Jaeger's architecture and its components. Let's see how to install and work with it.

Installing Jaeger

There are Helm charts to install Jaeger and the Jaeger operator:

```
$ helm repo add jaegertracing https://jaegertracing.github.io/helm-charts
"jaegertracing" has been added to your repositories


$ helm search repo jaegertracing
NAME                            CHART VERSION   APP VERSION DESCRIPTION
jaegertracing/jaeger            0.62.1          1.37.0      A Jaeger Helm chart for
Kubernetes
jaegertracing/jaeger-operator   2.36.0          1.38.0      jaeger-operator Helm
chart for Kubernetes
```

The Jaeger operator requires `cert-manager`, but doesn't install it automatically. Let's install it first:

```
$ helm repo add jetstack https://charts.jetstack.io
"jetstack" has been added to your repositories
```

```
$ helm install \
  cert-manager jetstack/cert-manager \
  --namespace cert-manager \
  --create-namespace \
  --version v1.9.1 \
  --set installCRDs=true
NAME: cert-manager
LAST DEPLOYED: Mon Oct 17 10:28:43 2022
NAMESPACE: cert-manager
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
cert-manager v1.9.1 has been deployed successfully!

In order to begin issuing certificates, you will need to set up a ClusterIssuer
or Issuer resource (for example, by creating a 'letsencrypt-staging' issuer).

More information on the different types of issuers and how to configure them
can be found in our documentation:

https://cert-manager.io/docs/configuration/

For information on how to configure cert-manager to automatically provision
Certificates for Ingress resources, take a look at the `ingress-shim`
documentation:

https://cert-manager.io/docs/usage/ingress/
```

Now, we can install the Jaeger operator into the observability namespace:

```
$ helm install jaeger jaegertracing/jaeger-operator \
       -n observability --create-namespace
NAME: jaeger
LAST DEPLOYED: Mon Oct 17 10:30:58 2022
NAMESPACE: observability
STATUS: deployed
REVISION: 1
```

```
TEST SUITE: None
NOTES:
jaeger-operator is installed.


Check the jaeger-operator logs
  export POD=$(kubectl get pods -l app.kubernetes.io/instance=jaeger -l app.
kubernetes.io/name=jaeger-operator --namespace observability --output name)
  kubectl logs $POD --namespace=observability
```

The deployment is called `jaeger-jaeger-operator`:

```
$ k get deploy -n observability
NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
jaeger-jaeger-operator   1/1     1            1           3m21s
```

Now, we can create a Jaeger instance using the Jaeger CRD. The operator watches for this custom resource and creates all the necessary resources. Here is the simplest possible Jaeger configuration. It uses the default `AllInOne` strategy, which deploys a single pod that contains all the components (agent, collector, query, ingester, and Jaeger UI) and uses in-memory storage. This is suitable for local development and testing purposes:

```
$ cat <<EOF | k apply -f -
apiVersion: jaegertracing.io/v1
kind: Jaeger
metadata:
  name: simplest
  namespace: observability
EOF

jaeger.jaegertracing.io/simplest created
$ k get jaegers -n observability
NAME       STATUS   VERSION   STRATEGY   STORAGE   AGE
simplest                                          5m54s
```

Let's bring up the Jaeger UI:

```
$ k port-forward deploy/simplest 8080:16686 -n observability
Forwarding from 127.0.0.1:8080 -> 16686
Forwarding from [::1]:8080 -> 16686
```

Now, we can browse to `http://localhost:8080` and see the Jaeger UI:

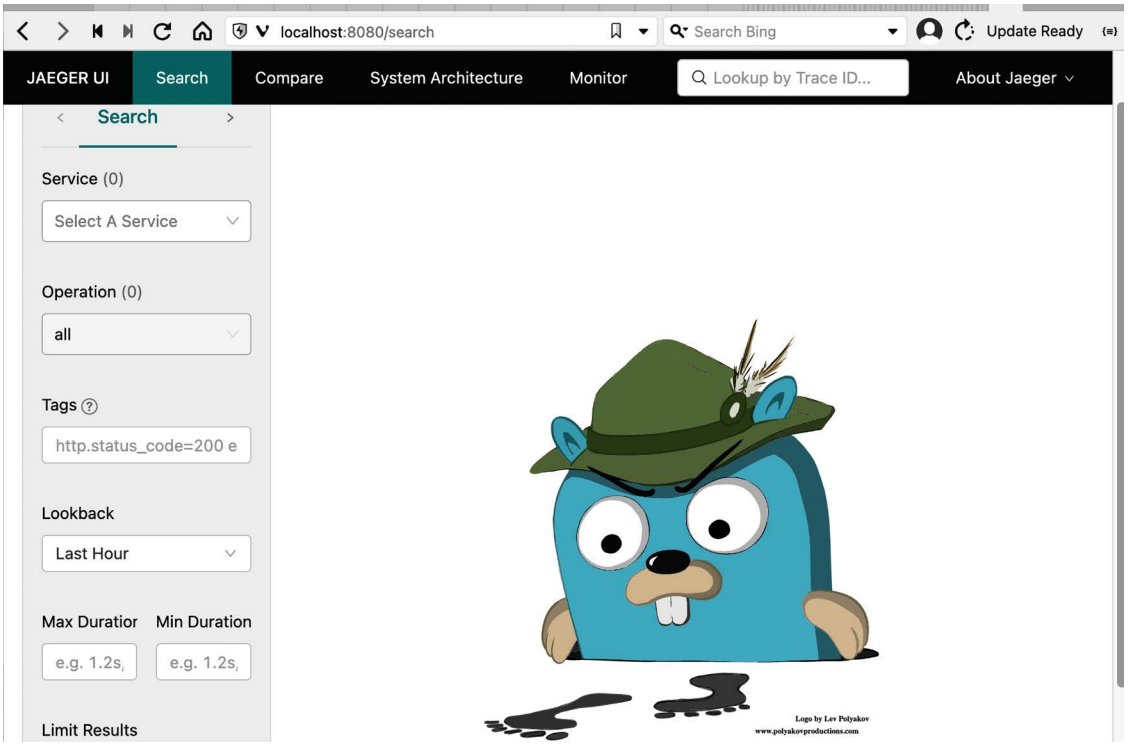


Figure 13.13: Jaeger UI

In the next chapter, *Chapter 14*, *Utilizing Service Meshes*, we will see more of Jaeger and how to use it specifically to trace requests going through the mesh. Now, let's turn our attention to troubleshooting using all the monitoring and observability mechanisms we discussed.

Troubleshooting problems

Troubleshooting a complex distributed system is no picnic. Abstractions, separation of concerns, information hiding, and encapsulation are great during development, testing, and when making changes to the system. But when things go wrong, you need to cross all those boundaries and layers of abstraction from the user action in their app, through the entire stack, all the way to the infrastructure, crossing all the business logic, asynchronous processes, legacy systems, and third-party integrations. This is a challenge even with large monolithic systems, but even more so with microservice-based distributed systems. Monitoring will assist you, but let's talk first about preparation, processes, and best practices.

Taking advantage of staging environments

When building a large system, developers work on their local machines (ignoring the cloud development environment here) and, eventually, the code is deployed to the production environment. But there are a few steps between those two extremes. Complex systems operate in an environment that is not easy to duplicate locally.

You should test changes to code or configuration in an environment that is similar to your production environment. This is your staging environment, where you should catch most problems that can't be caught by the developer running tests locally in their development environment.

The software delivery process should accommodate the detection of bad code and configuration as early as possible. But, sometimes, bad changes will be detected only in production and cause an incident. You should have an incident management process in place as well, which typically involves reverting to the previous version of whatever component caused the issue and then trying to find the root cause by looking at logs, metrics, and traces – sometimes, by debugging in the staging environment too.

But, sometimes, the problem is not with your code or configuration. In the end, your Kubernetes cluster runs on nodes (yes, even if it's managed), and those nodes can suffer many issues.

Detecting problems at the node level

In Kubernetes' conceptual model, the unit of work is the pod. However, pods are scheduled on nodes. When it comes to monitoring and the reliability of the infrastructure, the nodes are what require the most attention, as Kubernetes itself (the scheduler, replica sets, and horizontal pod autoscalers) takes care of the pods. The kubelet is aware of many issues on the nodes and it will update the API server. You can see the node status and if it's ready using this command:

```
$ k describe no kind-control-plane | grep Conditions -A 6
Conditions:
  Type             Status  LastHeartbeatTime                 LastTransitionTime
Reason                       Message
  ----             ------  -----------------                 ------------------
------                       -------
  MemoryPressure   False   Fri, 21 Oct 2022 01:09:33 -0700   Mon, 17 Oct 2022
10:27:24 -0700   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   Fri, 21 Oct 2022 01:09:33 -0700   Mon, 17 Oct 2022
10:27:24 -0700   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Fri, 21 Oct 2022 01:09:33 -0700   Mon, 17 Oct 2022
10:27:24 -0700   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Fri, 21 Oct 2022 01:09:33 -0700   Mon, 17 Oct 2022
10:27:52 -0700   KubeletReady                 kubelet is posting ready status
```

Note the last condition, `Ready`. This means that Kubernetes can schedule pending pods to this node.

But, there might be problems that the kubelet can't detect. Some of the problems are:

- Bad CPU
- Bad memory
- Bad disk
- Kernel deadlock
- Corrupt filesystem
- Problems with the container runtime (e.g., Docker daemon)

We need another solution. Enter the node problem detector.

The node problem detector is a pod that runs on every node. It needs to solve a difficult problem. It must be able to detect various low-level problems across different environments, different hardware, and different operating systems. It must be reliable enough not to be affected itself (otherwise, it can't report on problems), and it needs to have a relatively low overhead to avoid spamming the control plane. The source code is at `https://github.com/kubernetes/node-problem-detector`.

The most natural way is to deploy the node problem detector as a DaemonSet, so every node always has a node problem detector running on it. On Google's GKE clusters, it runs as an add-on.

Problem daemons

The problem with the node problem detector (pun intended) is that there are too many problems that it needs to handle. Trying to cram all of them into a single codebase can lead to a complex, bloated, and never-stabilizing codebase. The design of the node problem detector calls for the separation of the core functionality of reporting node problems to the master from the specific problem detection. The reporting API is based on generic conditions and events. The problem detection should be done by separate problem daemons (each in its own container). This way, it is possible to add and evolve new problem detectors without impacting the core node problem detector. In addition, the control plane may have a remedy controller that can resolve some node problems automatically, therefore implementing self-healing.

At this time, problem daemons are baked into the node problem detector binary, and they execute as Goroutines, so you don't get the benefits of the loosely-coupled design just yet. In the future, each problem daemon will run in its own container.

In addition to problems with nodes, the other area where things can break down is networking. The various monitoring tools we discussed earlier can help us identify problems across the infrastructure, in our code, or with third-party dependencies.

Let's talk about the various options in our toolbox, how they compare, and how to utilize them for maximal effect.

Dashboards vs. alerts

Dashboards are purely for humans. The idea of a good dashboard is to provide, at one glance, a lot of useful information about the state of the system or a particular component. There are many user experience elements to designing good dashboards, just like designing any user interface. Monitoring dashboards can cover a lot of data across many components, over long time periods, and may support drilling down into finer and finer levels of detail.

Alerts, on the other hand, are periodically checking certain conditions (often based on metrics) and, when triggered, can either result in automatic resolution of the cause of the alert or eventually notify a human, who will probably start the investigation by looking at some dashboards.

Self-healing systems can handle certain alerts automatically (or ideally, resolve the issue before an alert is even raised). Humans will typically be involved in troubleshooting. Even in cases where the system automatically recovered from a problem at some point, a human will review the actions the system took and verify that the current behavior, including the automatic recovery from problems, is adequate.

In many cases, severe problems (a.k.a. incidents) discovered by humans looking at dashboards (not scalable) or notified by alerts will require some investigation, remediation, and later, post-mortem. In all those stages, the next layer of monitoring comes into play.

Logs vs metrics vs. error reports

Let's understand where each of these tools excels and how to best combine their strengths to debug difficult problems. Let's assume we have good test coverage and our business/domain logic code is by and large correct. We run into problems in the production environment. There could be several types of problems that happen only in production:

- Misconfiguration (production configuration is incorrect or out of date)
- Infrastructure provisioning
- Insufficient permissions and access to data, services, or third-party integrations
- Environment-specific code
- Software bugs that are exposed by production inputs
- Scalability and performance issues

That's quite a list and it's probably not even complete. Typically, when something goes wrong, it is in response to some change. What kind of changes are we talking about? Here are a few:

- Deployment of a new version of the code
- Dynamic re-configuration of a deployed application
- New users or existing users changing the way they interact with the system
- Changes to the underlying infrastructure (e.g., by the cloud provider)
- A new path in the code is utilized for the first time (e.g., fallback to another region)

Since there is such a broad spectrum of problems and causes, it is difficult to suggest a linear path to resolution. For example, if the failure caused an error, then looking at an error report might be the best starting point. But, if the issue is that some action that was supposed to happen didn't happen, then there is no error to look at. In this case, it might make sense to look at the logs and compare them to the logs from a previous successful request. In case of infrastructure or scalability problems, metrics may give us the best initial insight.

The bottom line is that debugging distributed systems requires using multiple tools together in the pursuit of the ever-elusive root cause.

Of course, in distributed systems with lots of components and microservices, it is not even clear where to look. This is where distributed tracing shines and can help us narrow down and identify the culprit.

Detecting performance and root cause with distributed tracing

With distributed tracing in place, every request will generate a trace with a graph of spans. Jaeger uses sampling of 1/1000 by default so, once in a blue moon, issues might escape it, but for persistent problems, we will be able to follow the path of a request, see how long each span takes, and if the processing of a request bails out for some reason, it will be very easy to notice. At this point, you're back to the logs, metrics, and errors to hunt the root cause.

As you can see, troubleshooting problems in a complex system like Kubernetes is far from trivial. You need comprehensive observability in place including logging, metrics, and distributed tracing. You also need deep knowledge and understanding of your system to be able to configure, monitor, and mitigate issues quickly and reliably.

Summary

In this chapter, we covered the topics of monitoring, observability, and troubleshooting. We started with a review of the various aspects of monitoring: logs, metrics, error reporting, and distributed tracing. Then, we discussed how to incorporate monitoring capabilities into your Kubernetes cluster. We looked at several CNCF projects, like Fluentd for log aggregation, Prometheus for metrics collection and alert management, Grafana for visualization, and Jaeger for distributed tracing. Then, we explored troubleshooting large distributed systems. We realized how difficult it can be and why we need so many different tools to conquer the issues.

In the next chapter, we will take it to the next level and dive into service meshes. I'm super excited about service meshes because they take much of the complexity related to cloud-native microservice-based applications and externalize them outside of the microservices. That has a lot of real-world value.

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`



14

Utilizing Service Meshes

In the previous chapter, we looked at monitoring and observability. One of the obstacles to a comprehensive monitoring story is that it requires a lot of changes to the code that are orthogonal to the business logic.

In this chapter, we will learn how service meshes allow you to externalize many of those cross-cutting concerns from the application code. The service mesh is a true paradigm shift in the way you design, evolve, and operate distributed systems on Kubernetes. I like to think of it as aspect-oriented programming for cloud-native distributed systems. We will also take a deeper look into the Istio service mesh. The topics we will cover are:

- What is a service mesh?
- Choosing a service mesh
- Understanding Istio architecture
- Incorporating Istio into your Kubernetes cluster
- Working with Istio

Let's jump right in.

What is a service mesh?

Service mesh is an architectural pattern for large-scale cloud-native applications that are composed of many microservices. When your application is structured as a collection of microservices, there is a lot going on in the boundary between microservices inside your Kubernetes cluster. This is different from traditional monolithic applications where most of the work is done by a single OS process.

Here are some concerns that are relevant to each microservice or interaction between microservices:

- Advanced load balancing
- Service discovery
- Support for canary deployments
- Caching

- Tracing a request across multiple microservices
- Authentication between services
- Throttling the number of requests a service handles at a given time
- Automatically retrying failed requests
- Failing over to an alternative component when a component fails consistently
- Collecting metrics

All these concerns are completely orthogonal to the domain logic of the service, but they are all very important. A naive approach is to simply code all these concerns directly in each microservice. This obviously doesn't scale. So, a typical approach is to package all this functionality in a big library or set of libraries and use these libraries in each service.

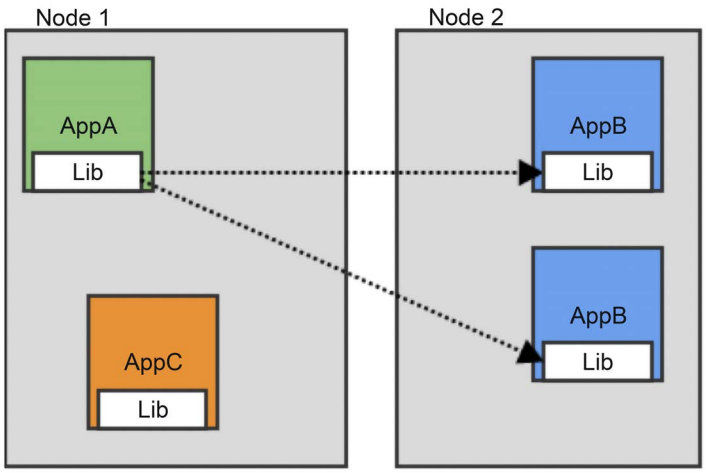


Figure 14.1: A typical library-based architecture

There are several problems with the big library approach:

- You need to implement the library in all the programming languages you use and make sure they are compatible
- If you want to update your library, you must bump the version of all your services
- It's difficult to upgrade services incrementally if a new version of the library is not backward-compatible

In comparison, the service mesh doesn't touch your application. It injects a sidecar proxy container into each pod and uses a service mesh controller. The proxies intercept all communication between the pods and, in collaboration with the mesh controller, can take care of all the cross-cutting concerns.

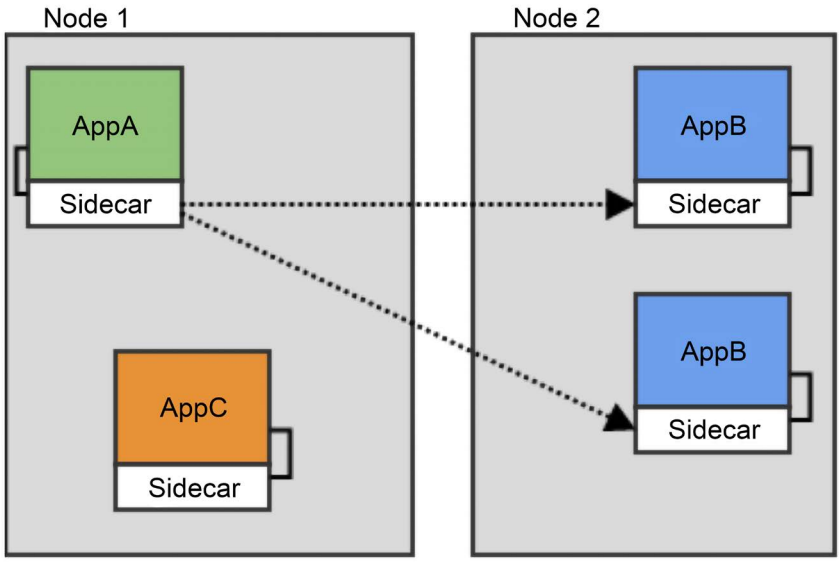


Figure 14.2: Sidecar service mesh architecture

Here are some attributes of the proxy injection approach:

- The application is unaware of the service mesh
- You can turn the mesh on or off per pod and update the mesh independently
- No need to deploy an agent on each node
- Different pods on the same node can have different sidecars (or versions)
- Each pod has its own copy of the proxy

On Kubernetes, it looks like this:

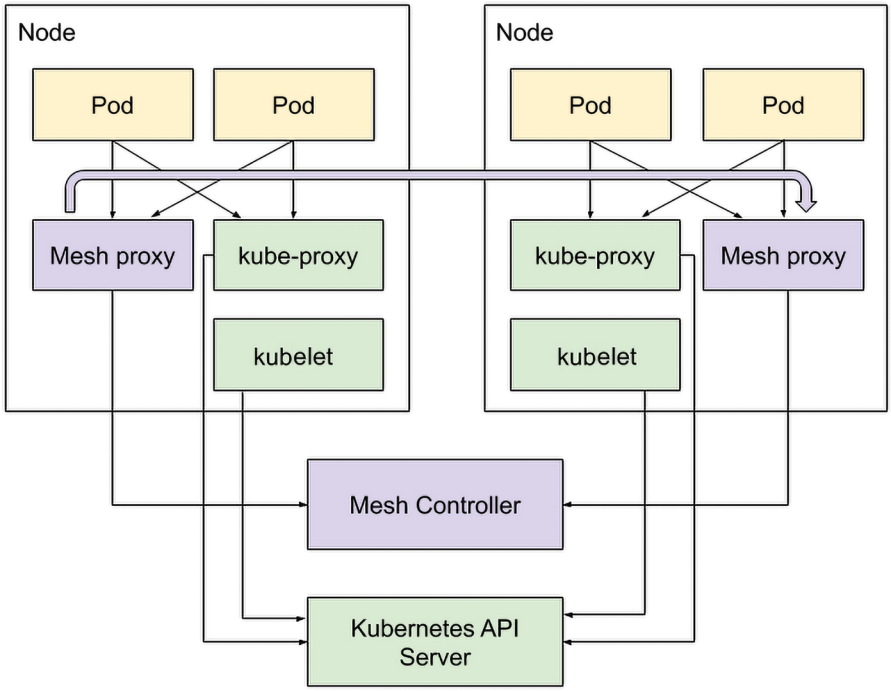


Figure 14.3: Service mesh architecture in Kubernetes

There is another way to implement the service mesh proxy as a node agent, where it is not injected into each pod. This approach is less common, but in some cases (especially in non-Kubernetes environments), it is useful. It can save resources on nodes that run a lot of small pods where the overhead of all the sidecar containers adds up.

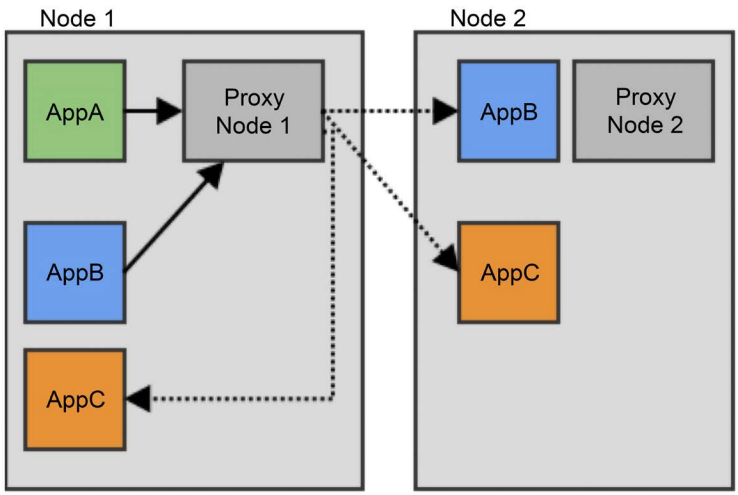


Figure 14.4: Node agent service mesh architecture

In the service mesh world, there is a control plane, which is typically a set of controllers on Kubernetes, and there is a data plane, which is made up of the proxies that connect all the services in the mesh. The data plane consists of all the sidecar containers (or node agents) that intercept the communication between services in the mesh. The control plane is responsible for managing the proxies and configuring what actually happens when any traffic between services or a service and the outside world is intercepted.

Now, that we have a good idea of what a service mesh is, how it works, and why it is so useful, let's review some of the service meshes out there.

Choosing a service mesh

The service mesh concept is relatively new, but there are already many choices out there. We will be using Istio later in the chapter. However, you may prefer a different service mesh for your use case. Here is a concise review of the current cohort of service meshes.

Envoy

Envoy (`https://www.envoyproxy.io`) is yet another CNCF graduated project. It is a very versatile and high-performance L7 proxy. It provides many service mesh capabilities; however, it is considered pretty low-level and difficult to configure. It is also not Kubernetes-specific. Some of the Kubernetes service meshes use Envoy as the underlying data plane and provide a Kubernetes-native control plane to configure and interact with it. If you want to use Envoy directly on Kubernetes, then the recommendation is to use other open source projects like Ambassador and Gloo as an ingress controller and/or API gateway.

Linkerd 2

Linkerd 2 (`https://linkerd.io`) is a Kubernetes-specific service as well as a CNCF incubating project. It is developed by Buoyant (`https://buoyant.io`). Buoyant coined the term service mesh and introduced it to the world a few years ago. They started with a Scala-based service mesh for multiple platforms including Kubernetes called Linkerd. But they decided to develop a better and more performant service mesh targeting Kubernetes only. That's where Linkerd 2 comes in, which is Kubernetes-specific. They implemented the data plane (proxy layer) in Rust and the control plane in Go.

Kuma

Kuma (`https://kuma.io/`) is an open source service mesh powered by Envoy. It was originally developed by Kong, which offers an enterprise product called Kong Mesh on top of Kuma. It works on Kubernetes as well as other environments. Its claims to fame is that it is super easy to configure and that it allows mixing Kubernetes and VM-based systems in a single mesh.

AWS App Mesh

AWS, of course, has its own proprietary service mesh – AWS App Mesh (`https://aws.amazon.com/app-mesh`). App Mesh also uses Envoy as its data plane. It can run on EC2, Fargate, ECS and EKS, and plain Kubernetes. App Mesh is a bit late to the service mesh scene, so it's not as mature as some other service meshes, but it is catching up. It is based on solid Envoy, and may be the best choice due to its tight integration with AWS services.

Mæsh

Mæsh (`https://mae.sh`) is developed by the makers of Træfik (`https://containo.us/traefik`). It is interesting because it uses the node agent approach as opposed to sidecar containers. It is based heavily on Traefik middleware to implement the service mesh functionality. You can configure it by using annotations on your services. It may be an interesting and lightweight approach to try service meshes if you utilize Traefik at the edge of your cluster.

Istio

Istio (`https://istio.io/`) is the most well-known service mesh on Kubernetes. It is built on top of Envoy and allows you to configure it in a Kubernetes-native way via YAML manifests. Istio was started by Google, IBM, and Lyft (the Envoy developers). It's a one-click install on Google GKE, but it is widely used in the Kubernetes community in any environment. It is also the default ingress/API gateway solution for Knative, which promotes its adoption even further.

OSM (Open Service Mesh)

OSM (`https://openservicemesh.io`) is yet another service mesh based on Envoy. It is configurable via **SMI** (**Service Mesh Interface**), which is a spec that attempts to provide a provider-agnostic set of APIs to configure service meshes. See `https://smi-spec.io` for more details. Both OSM and SMI are CNCF sandbox projects.

OSM was developed by Microsoft and contributed to CNCF.

Cilium Service Mesh

Cilium Service Mesh (`https://isovalent.com/blog/post/cilium-service-mesh`) is a newcomer to the service mesh arena. It is developed by Isovalent (`https://isovalent.com`). It is notable for attempting to bring the benefits of eBPF to the service mesh and utilize a sidecar-free approach. It is still early days, and Cilium Service Mesh is not as mature as other service meshes. However, it has the interesting concept of allowing you to bring your own control plane. It can integrate with Istio and interoperate with sidecars as well. It's worth keeping an eye on.

After discussing the various service mesh choices, let's take Istio for a ride. The reason we picked Istio is that it is one of the most mature service meshes, with a large community, a lot of users, and the backing of industry leaders.

Understanding the Istio architecture

In this section, we will get to know Istio a little better.

First, let's meet the main components of Istio and understand what they do and how they relate.

Istio is a large framework that provides a lot of capabilities, and it has multiple parts that interact with each other and with Kubernetes components (mostly indirectly and unobtrusively). It is divided into a control plane and a data plane. The data plane is a set of proxies (one per pod). Their control plane is a set of components that are responsible for configuring the proxies and collecting telemetry data.

The following diagram illustrates the different parts of Istio, how they are related to each other, and what information is exchanged between them.

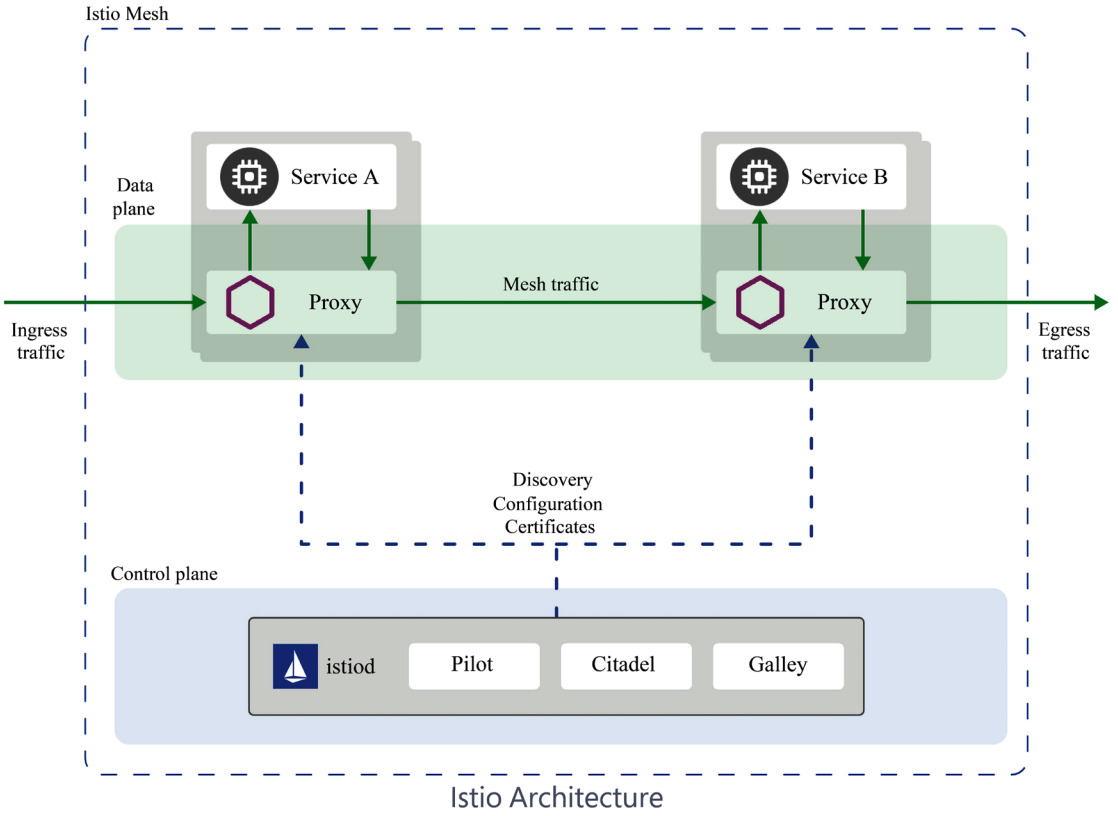


Figure 14.5: Istio architecture

As you can see, there are two primary components: the Envoy proxy, which is the sidecar container attached to every service instance (every pod), and istiod, which is responsible for discovery, configuration, and certificates. Istiod is a single binary that actually contains multiple components: Pilot, Citadel, and Galley. These components used to run as separate binaries. They were combined into a single binary in Istio 1.5 to simplify the experience of installing, running, and upgrading Istio.

Let's go a little deeper into each component, starting with the Envoy proxy.

Envoy

We discussed Envoy briefly when we reviewed service meshes for Kubernetes. Here, it serves as the data plane of Istio. Envoy is implemented in C++ and is a high-performance proxy. For each pod in the service mesh, Istio injects (either automatically or through the istioctl CLI) an Envoy side container that does the heavy lifting.

Here are some of the tasks Envoy performs:

- Proxy HTTP, HTTP/2, and gRPC traffic between pods
- Sophisticated load balancing
- mTLS termination
- HTTP/2 and gRPC proxies
- Providing service health
- Circuit breaking for unhealthy services
- Percentage-based traffic shaping
- Injecting faults for testing
- Detailed metrics

The Envoy proxy controls all the incoming and outgoing communication to its pod. It is, by far, the most important component of Istio. The configuration of Envoy is not trivial, and this is a large part of what the Istio control plane deals with.

The next component is Pilot.

Pilot

Pilot is responsible for platform-agnostic service discovery, dynamic load balancing, and routing. It translates high-level routing rules into an Envoy configuration. This abstraction layer allows Istio to run on multiple orchestration platforms. Pilot takes all the platform-specific information, converts it into the Envoy data plane configuration format, and propagates it to each Envoy proxy with the Envoy data plane API. Pilot is stateless; in Kubernetes, all the configuration is stored as **custom resource definitions** (**CRDs**) in etcd.

The next component is Citadel.

Citadel

Citadel is responsible for certificate and key management. It is a key part of Istio security. Citadel integrates with various platforms and aligns with their identity mechanisms. For example, in Kubernetes, it uses service accounts; on AWS, it uses AWS IAM; on Azure, it uses AAD, and on GCP/GKE, it can use GCP IAM. The Istio PKI is based on Citadel. It uses X.509 certificates in SPIFEE format as a vehicle for service identity.

Here is the workflow for a strong identity to envoy proxies in Kubernetes:

1. Citadel creates certificates and key pairs for existing service accounts.
2. Citadel watches the Kubernetes API server for new service accounts to provision with a certificate and a key pair.
3. Citadel stores the certificates and keys as Kubernetes secrets.
4. Kubernetes mounts the secrets into each new pod that is associated with the service account (this is standard Kubernetes practice).

5. Citadel automatically rotates the Kubernetes secrets when the certificates expire.
6. Pilot generates secure naming information that associates a service account with an Istio service. Pilot then passes the secure naming information to the Envoy proxy.

The final major component that we will cover is Galley.

Galley

Galley is responsible for abstracting the user configuration on different platforms. It provides the ingested configuration to Pilot. It is a pretty simple component.

Now that we have broken down Istio into its major components, let's get hands-on with Istio and incorporate it into a Kubernetes cluster.

Incorporating Istio into your Kubernetes cluster

In this section, we will install Istio in a fresh cluster and explore all the service goodness it provides.

Preparing a minikube cluster for Istio

We will use a minikube cluster for checking out Istio. Before installing Istio, we should make sure our cluster has enough capacity to handle Istio as well as its demo application, BookInfo. We will start minikube with 16 GB of memory and four CPUs, which should be adequate. Make sure the Docker VM you're using (e.g., Rancher Desktop) has sufficient CPU and memory:

```
$ minikube start --memory=16384 --cpus=4
```

Minikube can provide a load balancer for Istio. Let's run this command in a separate terminal as it will block (do not stop the tunnel until you are done):

```
$ minikube tunnel
✅  Tunnel successfully started

📌  NOTE: Please do not close this terminal as this process must stay alive for the
tunnel to be accessible ...
```

Minikube sometimes doesn't clean up the tunnel network, so it's recommended to run the following command after you stop the cluster:

```
minikube tunnel --cleanup
```

Installing Istio

With minikube up and running, we can install Istio itself. There are multiple ways to install Istio:

- Customized installation with istioctl (the Istio CLI)
- Customized installation with Helm using the Istio operator (supported, but discouraged)
- Multi-cluster installation
- External control plane
- Virtual machine installation

We will go with the recommended istioctl option. The Istio version may be higher than 1.15:

```
$ curl -L https://istio.io/downloadIstio | sh -
```

The istioctl tool is located in `istio-1.15.2/bin` (the version may be different when you download it). Make sure it's in your path. The Kubernetes installation manifests are in `istio-1.15.2/install/kubernetes` and the examples are in `istio-1.15.2/samples`.

Let's run some preinstall checks:

```
$ istioctl x precheck
✓ No issues found when checking the cluster. Istio is safe to install or upgrade!
  To get started, check out https://istio.io/latest/docs/setup/getting-started/
```

We will install the built-in demo profile, which is great for evaluating Istio:

```
$ istioctl install --set profile=demo -y
✓ Istio core installed
✓ Istiod installed
✓ Egress gateways installed
✓ Ingress gateways installed
✓ Installation complete
Making this installation the default for injection and validation.

Thank you for installing Istio 1.15
```

Let's also install some observability add-ons such as `prometheus`, `grafana`, `jaeger`, and `kiali`:

```
$ k apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
addons/prometheus.yaml
serviceaccount/prometheus created
configmap/prometheus created
clusterrole.rbac.authorization.k8s.io/prometheus created
clusterrolebinding.rbac.authorization.k8s.io/prometheus created
service/prometheus created
deployment.apps/prometheus created

$ kapply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
addons/grafana.yaml
serviceaccount/grafana created
configmap/grafana created
service/grafana created
deployment.apps/grafana created
configmap/istio-grafana-dashboards created
configmap/istio-services-grafana-dashboards created
```

```
$ k apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
addons/jaeger.yaml
deployment.apps/jaeger created
service/tracing created
service/zipkin created
service/jaeger-collector created

$ k apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
addons/kiali.yaml
serviceaccount/kiali created
configmap/kiali created
clusterrole.rbac.authorization.k8s.io/kiali-viewer created
clusterrole.rbac.authorization.k8s.io/kiali created
clusterrolebinding.rbac.authorization.k8s.io/kiali created
role.rbac.authorization.k8s.io/kiali-controlplane created
rolebinding.rbac.authorization.k8s.io/kiali-controlplane created
service/kiali created
deployment.apps/kiali created
```

Let's review our cluster and see what is actually installed. Istio installs itself in the `istio-system` namespace, which is very convenient since it installs a lot of stuff. Let's check out what services Istio installed:

```
$ k get svc -n istio-system -o name
service/grafana
service/istio-egressgateway
service/istio-ingressgateway
service/istiod
service/jaeger-collector
service/kiali
service/prometheus
service/tracing
service/zipkin
```

There are quite a few services with an `istio-` prefix and then a bunch of other services:

- Prometheus
- Grafana
- Jaeger
- Zipkin
- Tracing
- Kiali

OK. We installed Istio and a variety of integrations successfully. Let's install the BookInfo application, which is Istio's sample application, in our cluster.

Installing BookInfo

BookInfo is a simple microservice-based application that displays, as the name suggests, information on a single book such as name, description, ISBN, and even reviews. The BookInfo developers really embraced the polyglot lifestyle and each microservice is implemented in a different programming language:

- ProductPage service in Python
- Reviews service in Java
- Details service in Ruby
- Ratings service in JavaScript (Node.js)

The following diagram describes the relationships and flow of information between the BookInfo services:

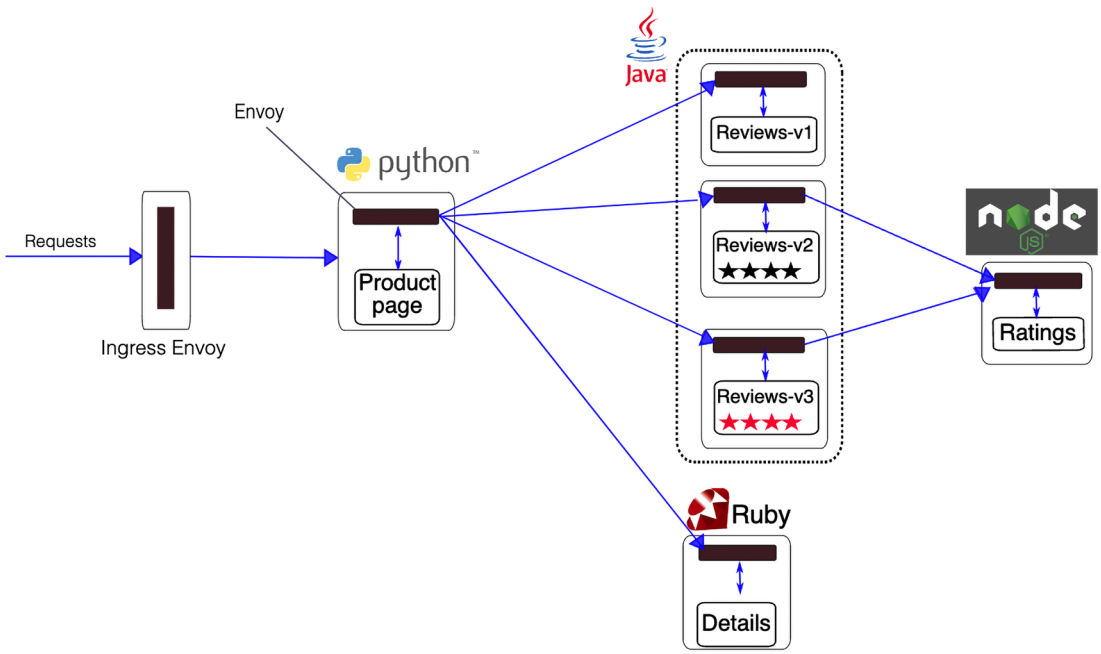


Figure 14.6: The flow of information between BookInfo services

We're going to install it in its own `bookinfo` namespace. Let's create the namespace and then enable the Istio auto-injection of the sidecar proxies by adding a label to the namespace:

```
$ k create ns bookinfo
namespace/bookinfo created
```

```
$ k label namespace bookinfo istio-injection=enabled
namespace/bookinfo labeled
```

Installing the application itself is a simple one-liner:

```
$ k apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
bookinfo/platform/kube/bookinfo.yaml -n bookinfo
service/details created
serviceaccount/bookinfo-details created
deployment.apps/details-v1 created
service/ratings created
serviceaccount/bookinfo-ratings created
deployment.apps/ratings-v1 created
service/reviews created
serviceaccount/bookinfo-reviews created
deployment.apps/reviews-v1 created
deployment.apps/reviews-v2 created
deployment.apps/reviews-v3 created
service/productpage created
serviceaccount/bookinfo-productpage created
deployment.apps/productpage-v1 created
```

Alright, the application was deployed successfully, including separate service accounts for each service. As you can see, three versions of the reviews service were deployed. This will come in handy later when we play with upgrades and advanced routing and deployment patterns.

Before going on, we still need to wait for all the pods to initialize and then Istio will inject its sidecar proxy container. When the dust settles, you should see something like this:

```
$ k get po -n bookinfo
NAME                              READY   STATUS    RESTARTS   AGE
details-v1-5ffd6b64f7-c62l6       2/2     Running   0          3m48s
productpage-v1-979d4d9fc-7hzkj    2/2     Running   0          3m48s
ratings-v1-5f9699cfdf-mns6n       2/2     Running   0          3m48s
reviews-v1-569db879f5-jmfrj       2/2     Running   0          3m48s
reviews-v2-65c4dc6fdc-cc8nn       2/2     Running   0          3m48s
reviews-v3-c9c4fb987-bpk9f        2/2     Running   0          3m48s
```

Note that under the `READY` column, each pod shows 2/2, which means 2 containers per pod. One is the application container and the other is the injected proxy.

Since we're going to operate in the `bookinfo` namespace, let's define a little alias that will make our life simpler:

```
$ alias kb='kubectl -n bookinfo'
```

Now, armed with our little `kb` alias, we can verify that we can get the product page from the ratings service:

```
$ kb exec -it $(kb get pod -l app=ratings -o jsonpath='{.items[0].metadata.name}') -c
ratings -- curl productpage:9080/productpage | grep -o "<title>.*</title>"

<title>Simple Bookstore App</title>
```

But the application is not accessible to the outside world yet. This is where the Istio gateway comes in. Let's deploy it:

```
$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
bookinfo/networking/bookinfo-gateway.yaml
gateway.networking.istio.io/bookinfo-gateway created
virtualservice.networking.istio.io/bookinfo created
```

Let's get the URLs to access the application from the outside:

```
$ export INGRESS_HOST=$(kubectl -n istio-system get service istio-ingressgateway -o
jsonpath='{.status.loadBalancer.ingress[0].ip}')
$ export INGRESS_PORT=$(kubectl -n istio-system get service istio-ingressgateway -o
jsonpath='{.spec.ports[?(@.name=="http2")].port}')
$ export SECURE_INGRESS_PORT=$(kubectl -n istio-system get service istio-
ingressgateway -o jsonpath='{.spec.ports[?(@.name=="https")].port}')
$ export GATEWAY_URL=${INGRESS_HOST}:${INGRESS_PORT}
```

Now we can try it from the outside:

```
$ http http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"
<title>Simple Bookstore App</title>
```

You can also open the URL in your browser and see some information about Shakespeare's "The Comedy of Errors":

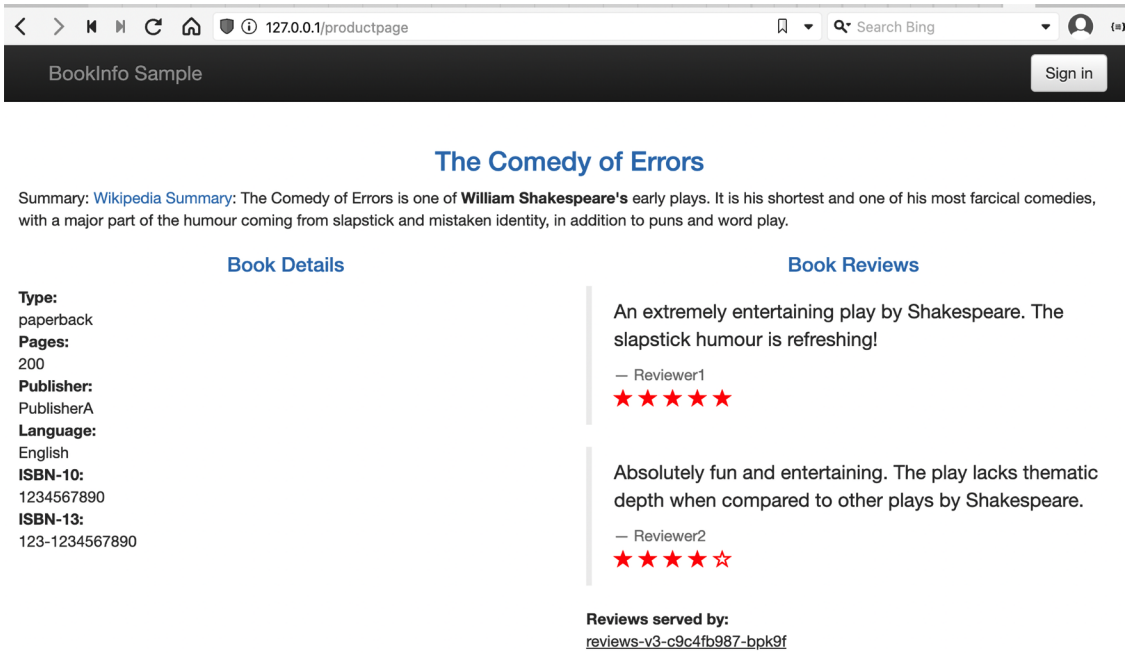


Figure 14.7: A sample BookInfo review

Alright. We're all set to start exploring the capabilities that Istio brings to the table.

Working with Istio

In this section, we will work with Istio resources and policies and utilize them to improve the operation of the BookInfo application.

Let's start with traffic management.

Traffic management

Istio traffic management is about routing traffic to your services according to the destination rules you define. Istio keeps a service registry for all your services and their endpoints. Basic traffic management allows traffic between each pair of services and does simple round-robin load balancing between each service instance. But Istio can do much more. The traffic management API of Istio consists of five resources:

- Virtual services
- Destination rules
- Gateways

- Service entries
- Sidecars

Let's start by applying the default destination rules for BookInfo:

```
$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
bookinfo/networking/destination-rule-all.yaml
destinationrule.networking.istio.io/productpage created
destinationrule.networking.istio.io/reviews created
destinationrule.networking.istio.io/ratings created
destinationrule.networking.istio.io/details created
```

Then, let's create the Istio virtual services that represent the services in the mesh:

```
$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
bookinfo/networking/virtual-service-all-v1.yaml
virtualservice.networking.istio.io/productpage created
virtualservice.networking.istio.io/reviews created
virtualservice.networking.istio.io/ratings created
virtualservice.networking.istio.io/details created
```

We need to wait a little for the virtual service configuration to propagate. Let's then check out the product page virtual service using the neat kubectl plugin. If you don't have it installed already follow the instructions at `https://github.com/itaysk/kubectl-neat`.

```
$ kb get virtualservices productpage -o yaml | k neat
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: productpage
  namespace: default
spec:
  hosts:
  - productpage
  http:
  - route:
    - destination:
        host: productpage
        subset: v1
```

It is pretty straightforward, specifying the HTTP route and the version. The v1 subset is important for the reviews service, which has multiple versions. The product page service will hit its v1 version because that is the subset that's configured.

Let's make it a little more interesting and do routing based on the logged-in user. Istio itself doesn't have a concept of user identity, but it routes traffic based on headers. The BookInfo application adds an end-user header to all requests.

The following command will update the routing rules:

```
$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
bookinfo/networking/virtual-service-reviews-test-v2.yaml
virtualservice.networking.istio.io/reviews configured
```

Let's check the new rules:

```
$ kb get virtualservice reviews -o yaml | k neat
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: reviews
  namespace: default
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: jason
    route:
    - destination:
        host: reviews
        subset: v2
  - route:
    - destination:
        host: reviews
        subset: v1
```

As you can see, if the HTTP header `end-user` matches `jason`, then the request will be routed to subset 2 of the reviews service, otherwise to subset 1. Version 2 of the reviews service adds a star rating to the reviews part of the page. To test it, we can sign in as user `jason` (any password will do), refresh the browser, and see that the reviews have stars next to them:

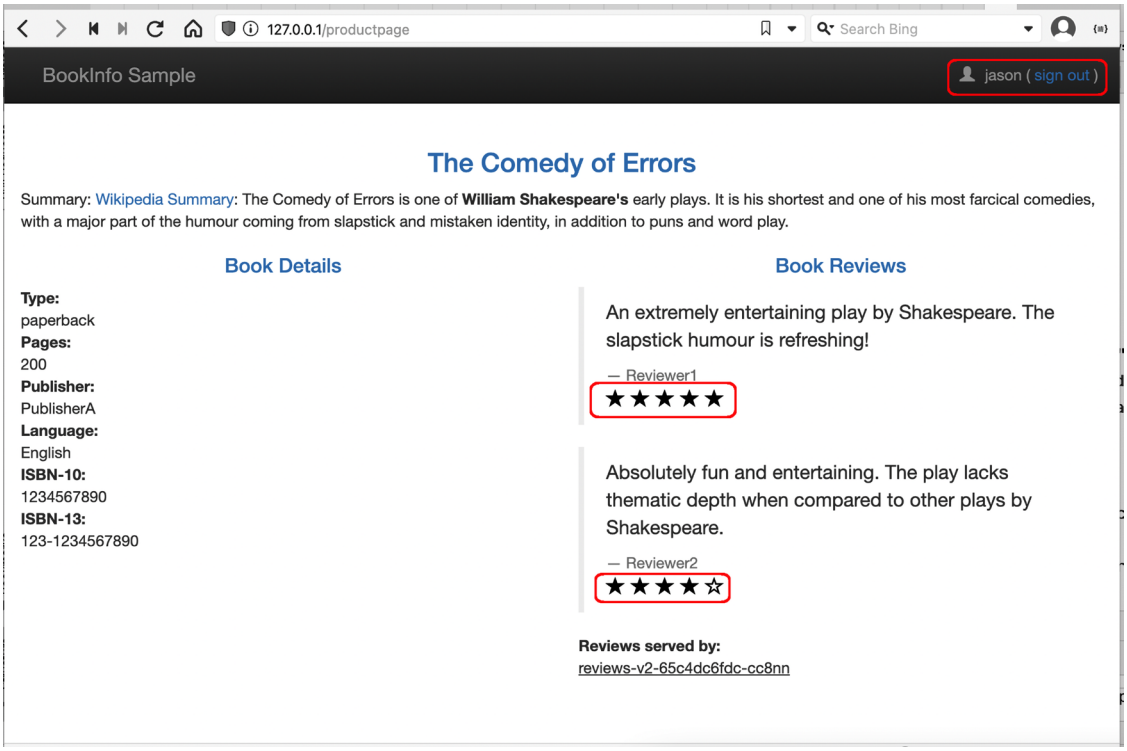


Figure 14.8: A sample BookInfo review with star ratings

There is much more Istio can do in the arena of traffic management:

- Fault injection for test purposes
- HTTP and TCP traffic shifting (gradually shifting traffic from one version to the next)
- Request timeouts
- Circuit breaking
- Mirroring

In addition to internal traffic management, Istio supports configuring ingress into the cluster and egress from the cluster including secure options with TLS and mutual TLS.

Security

Security is a core fixture of Istio. It provides identity management, authentication and authorization, security policies, and encryption. The security support is spread across many layers using multiple industry-standard protocols and best-practice security principles like defense in depth, security by default, and zero trust.

Here is the big picture of the Istio security architecture:

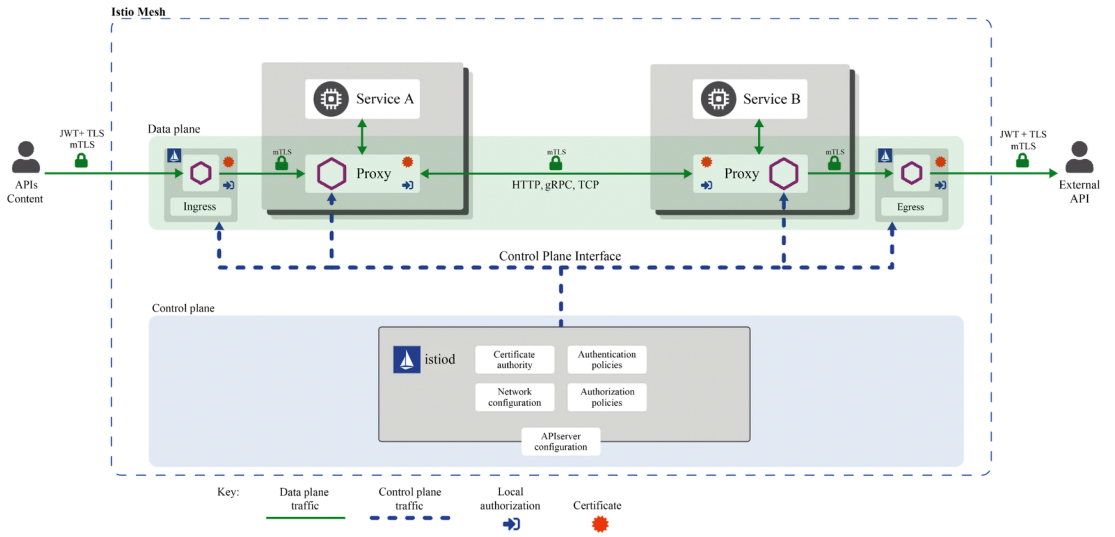


Figure 14.9: Istio security architecture

Istio enables a strong security posture via the following capabilities:

- Sidecar and perimeter proxies implement authenticated and authorized communication between clients and servers
- Control plane manages keys and certificates
- Control plane distributes security policies and secure naming information to the proxies
- Control plane manages auditing

Let's break it down piece by piece.

Istio identity

Istio utilizes secure naming where service names as defined by the service discovery mechanism (e.g., DNS) are mapped to server identities based on certificates. The clients verify the server identities. The server may be configured to verify the client's identity. All the security policies apply to given identities. The servers decide what access a client has based on their identity.

The Istio identity model can utilize existing identity infrastructure on the platform it is running on. On Kubernetes, it utilizes Kubernetes service accounts, of course.

Istio securely assigns an x.509 certificate to each workload via an agent running together with the Envoy proxy. The agent works with istiod to automatically provision and rotate certificates and private keys.

Istio certificate management

Here is the workflow for provisioning certificates and keys:

1. istiod exposes a gRPC service that listens to **certificate signing requests** (**CSRs**).
2. The process begins with the Istio agent which, upon startup, generates a private key and a CSR. It then transmits the CSR, along with its own credentials, to the CSR service of istiod.
3. At this point, the istiod **Certificate Authority** (**CA**) examines the agent credentials contained within the CSR. If they are deemed valid, the istiod CA proceeds to sign the CSR, resulting in the creation of a certificate.
4. When a workload is launched, the Envoy proxy, residing within the same container, utilizes the Envoy **SDS** (**Secret Discovery Service**) API to request the certificate and corresponding key from the Istio agent.
5. The Istio agent actively monitors the expiration of the workload certificate, initiating a periodic process to refresh the certificate and key to ensure they remain up to date.

Istio authentication

The secure identity model underlies the authentication framework of Istio. Istio supports two modes of authentication: peer authentication and request authentication.

Peer authentication

Peer authentication is used for service-to-service authentication. The cool thing about it is that Istio provides it with no code changes. It ensures that service-to-service communication will take place only between services you configure with authentication policies.

Here is an authentication policy for the reviews service, which requires mutual TLS:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: "example-peer-policy"
  namespace: "foo"
spec:
  selector:
    matchLabels:
      app: reviews
  mtls:
    mode: STRICT
```

Request authentication

Request authentication is used for end-user authentication. Istio will verify that the end user making the request is allowed to make that request. This request-level authentication utilizes **JWT** (**JSON Web Token**) and supports many OpenID Connect backends.

Once the identity of the caller has been established, the authentication framework passes it along with other claims to the next link in the chain – the authorization framework.

Istio authorization

Istio can authorize requests at many levels:

- Entire mesh
- Entire namespace
- Workload-level

Here is the authorization architecture of Istio:

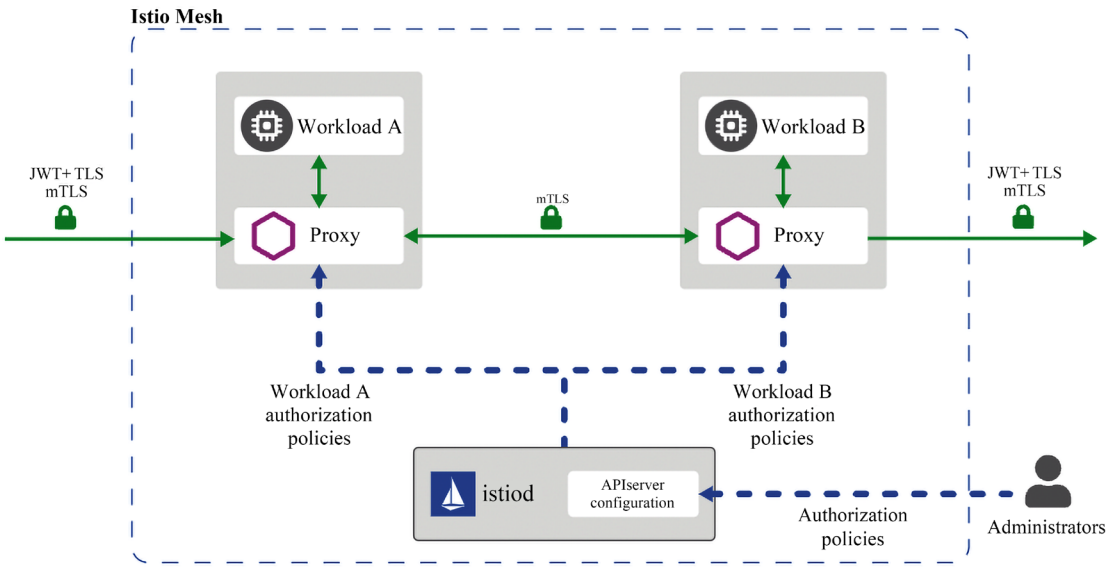


Figure 14.10: Istio authorization architecture

Authorization is based on authorization policies. Each policy has a selector (what workloads it applies to) and rules (who is allowed to access a resource and under what conditions).

If no policy is defined on a workload, all requests are allowed. However, if a policy is defined for a workload, only requests that are allowed by a rule in the policy are allowed. You can also define exclusion rules.

Here is an authorization policy that allows two sources (service account `cluster.local/ns/default/sa/sleep` and namespace `dev`) to access the workloads with the labels `app: httpbin` and `version: v1` in namespace and `foo` when the request is sent with a valid JWT token.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
 name: httpbin
 namespace: foo
spec:
```

```
  selector:
    matchLabels:
      app: httpbin
      version: v1
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/sleep"]
    - source:
        namespaces: ["dev"]
    to:
    - operation:
        methods: ["GET"]
    when:
    - key: request.auth.claims[iss]
      values: ["https://accounts.google.com"]
```

The granularity doesn't have to be at the workload level. We can limit access to specific endpoints and methods too. We can specify the operation using prefix match, suffix match, or presence match, in addition to exact match. For example, the following policy allows access to all paths that start with `/test/` and all paths that end in `/info`. The allowed methods are `GET` and `HEAD` only:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: tester
  namespace: default
spec:
  selector:
    matchLabels:
      app: products
  rules:
  - to:
    - operation:
        paths: ["/test/*", "*/info"]
        methods: ["GET", "HEAD"]
```

If we want to get even more fancy, we can specify conditions. For example, we can allow only requests with a specific header. Here is a policy that requires a version header with a value of v1 or v2:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
```

```
metadata:
 name: httpbin
 namespace: foo
spec:
 selector:
   matchLabels:
     app: httpbin
     version: v1
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/default/sa/sleep"]
   to:
   - operation:
       methods: ["GET"]
   when:
   - key: request.headers[version]
     values: ["v1", "v2"]
```

For TCP services, the `paths` and `methods` fields of the operation don't apply. Istio will simply ignore them. But, we can specify policies for specific ports:

```
apiVersion: "security.istio.io/v1beta1"
kind: AuthorizationPolicy
metadata:
  name: mongodb-policy
  namespace: default
spec:
 selector:
   matchLabels:
     app: mongodb
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/default/sa/bookinfo-ratings-v2"]
   to:
   - operation:
       ports: ["27017"]
```

Let's look at one of the areas where Istio provides tremendous value – telemetry.

Monitoring and observability

Instrumenting your applications for telemetry is a thankless job. The surface area is huge. You need to log, collect metrics, and create spans for tracing. Comprehensive observability is crucial for troubleshooting and mitigating incidents, but it's far from trivial:

- It takes time and effort to do it in the first place
- It takes more time and effort to ensure it is consistent across all the services in your cluster
- You can easily miss an important instrumentation point or configure it incorrectly
- If you want to change your log provider or distributed tracing solution, you might need to modify all your services
- It litters your code with lots of stuff that obscures the business logic
- You might need to explicitly turn it off for testing

What if all of it was taken care of automatically and never required any code changes? That's the promise of service mesh telemetry. Of course, you may need to do some work at the application/service level, especially if you want to capture custom metrics or do some specific logging. If your system is divided into coherent microservices along boundaries that really represent your domain and workflows, then Istio can help you get decent instrumentation right out of the box. The idea is that Istio can keep track of what's going on in the seams between your services.

Istio access logs

We can capture the access logs of Envoy proxies to give a picture of the network traffic from the perspective of each workload.

We will use two new workloads in this section: `sleep` and `httpbin`. Let's deploy them:

```
$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
sleep/sleep.yaml
serviceaccount/sleep created
service/sleep created
deployment.apps/sleep created

$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
httpbin/httpbin.yaml
serviceaccount/httpbin created
service/httpbin created
deployment.apps/httpbin created
```

In addition, let's deploy an `OpenTelemetry` collector to the `istio-system` namespace:

```
$ k apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/open-
telemetry/otel.yaml -n istio-system
configmap/opentelemetry-collector-conf created
service/opentelemetry-collector created
deployment.apps/opentelemetry-collector created
```

Istio configures providers and much more in the Istio ConfigMap, which already contains a provider entry for the `opentelemetry-collector` service. Let's use `yq` (`https://github.com/mikefarah/yq`) to review just the data field of the config map:

```
$ k get cm istio -n istio-system -o yaml | yq .data
mesh: |-
  accessLogFile: /dev/stdout
  defaultConfig:
    discoveryAddress: istiod.istio-system.svc:15012
    proxyMetadata: {}
    tracing:
      zipkin:
        address: zipkin.istio-system:9411
  enablePrometheusMerge: true
  extensionProviders:
  - envoyOtelAls:
      port: 4317
      service: opentelemetry-collector.istio-system.svc.cluster.local
    name: otel
  rootNamespace: istio-system
  trustDomain: cluster.local
meshNetworks: 'networks: {}'
```

To enable logging from the `sleep` workload to the `otel` collector, we need to configure a `Telemetry` resource:

```
$ cat <<EOF | kb apply -f -
apiVersion: telemetry.istio.io/v1alpha1
kind: Telemetry
metadata:
  name: sleep-logging
spec:
  selector:
    matchLabels:
      app: sleep
  accessLogging:
    - providers:
      - name: otel
EOF
telemetry.telemetry.istio.io/sleep-logging created
```

The default access log format is:

```
[%START_TIME%] \"%REQ(:METHOD)% %REQ(X-ENVOY-ORIGINAL-PATH?:PATH)% %PROTOCOL%\"
%RESPONSE_CODE% %RESPONSE_FLAGS% %RESPONSE_CODE_DETAILS% %CONNECTION_TERMINATION_
DETAILS%
\"%UPSTREAM_TRANSPORT_FAILURE_REASON%\" %BYTES_RECEIVED% %BYTES_SENT% %DURATION%
%RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)% \"%REQ(X-FORWARDED-FOR)%\" \"%REQ(USER-
AGENT)%\" \"%REQ(X-REQUEST-ID)%\"
\"%REQ(:AUTHORITY)%\" \"%UPSTREAM_HOST%\" %UPSTREAM_CLUSTER% %UPSTREAM_LOCAL_ADDRESS%
%DOWNSTREAM_LOCAL_ADDRESS% %DOWNSTREAM_REMOTE_ADDRESS% %REQUESTED_SERVER_NAME%
%ROUTE_NAME%\n
```

That's pretty verbose, but when debugging or troubleshooting, you want as much information as possible. The log format is configurable if you want to change it.

Alright. Let's try it out. The `sleep` workload is really just a pod from which we can make network requests to the httpbin application. The httpbin service is running on port `8000` and is known as simply `httpbin` inside the cluster. We will query `httpbin` from the `sleep` pod about the infamous 418 HTTP status (`https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418`):

```
$ kb exec deploy/sleep -c sleep -- curl -sS -v httpbin:8000/status/418
*   Trying 10.101.189.162:8000...
* Connected to httpbin (10.101.189.162) port 8000 (#0)
> GET /status/418 HTTP/1.1
> Host: httpbin:8000
> User-Agent: curl/7.86.0-DEV
> Accept: */*
>


    -=[ teapot ]=-

       _...._
     .'  _ _ `.
    | ."` ^ `". _,
    \_;`"---"`|//
      |       ;/
      \_     _/
        `"""`
* Mark bundle as not supporting multiuse
< HTTP/1.1 418 Unknown
< server: envoy
< date: Sat, 29 Oct 2022 04:35:07 GMT
< x-more-info: http://tools.ietf.org/html/rfc2324
< access-control-allow-origin: *
```

```
< access-control-allow-credentials: true
< content-length: 135
< x-envoy-upstream-service-time: 61
<
{ [135 bytes data]
* Connection #0 to host httpbin left intact
```

Yay, we got our expected teapot response. Now, let's check the access logs:

```
$ k logs -l app=opentelemetry-collector -n istio-system
LogRecord #0
ObservedTimestamp: 1970-01-01 00:00:00 +0000 UTC
Timestamp: 2022-10-29 04:35:07.599108 +0000 UTC
Severity:
Body: [2022-10-29T04:35:07.599Z] "GET /status/418 HTTP/1.1" 418 - via_upstream
- "-" 0 135 63 61 "-" "curl/7.86.0-DEV" "d36495d6-642a-9790-9b9a-d10b2af096f5"
"httpbin:8000" "172.17.0.17:80" outbound|8000||httpbin.bookinfo.svc.cluster.local
172.17.0.16:33876 10.101.189.162:8000 172.17.0.16:45986 - default

Trace ID:
Span ID:
Flags: 0
```

As you can see, we got a lot of information according to the default access log format, including the timestamp, request URL, the response status, the user agent, and the IP addresses of the source and destination.

In a production system, you may want to forward the collector's logs to a centralized logging system. Let's see what Istio offers for metrics.

Metrics

Istio collects three types of metrics:

- Proxy metrics
- Control plane metrics
- Service metrics

The collected metrics cover all traffic into, from, and inside the service mesh. As operators, we need to configure Istio for metric collection. We installed Prometheus and Grafana earlier for metric collection and the visualization backend. Istio follows the four golden signals doctrine and records the latency, traffic, errors, and saturation.

Let's look at an example of proxy-level (Envoy) metrics:

```
envoy_cluster_internal_upstream_rq{response_code_class="2xx",cluster_name="xds-
grpc"} 7163
envoy_cluster_upstream_rq_completed{cluster_name="xds-grpc"} 7164
```

```
envoy_cluster_ssl_connection_error{cluster_name="xds-grpc"} 0
envoy_cluster_lb_subsets_removed{cluster_name="xds-grpc"} 0
envoy_cluster_internal_upstream_rq{response_code="503",cluster_name="xds-grpc"}
1
```

And here is an example of service-level metrics:

```
istio_requests_total{
  connection_security_policy="mutual_tls",
  destination_app="details",
  destination_principal="cluster.local/ns/default/sa/default",
  destination_service="details.default.svc.cluster.local",
  destination_service_name="details",
  destination_service_namespace="default",
  destination_version="v1",
  destination_workload="details-v1",
  destination_workload_namespace="default",
  reporter="destination",
  request_protocol="http",
  response_code="200",
  response_flags="-",
  source_app="productpage",
  source_principal="cluster.local/ns/default/sa/default",
  source_version="v1",
  source_workload="productpage-v1",
  source_workload_namespace="default"
} 214
```

We can also collect metrics for TCP services. Let's install v2 of the ratings service, which uses MongoDB (a TCP service):

```
$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
bookinfo/platform/kube/bookinfo-ratings-v2.yaml
serviceaccount/bookinfo-ratings-v2 created
deployment.apps/ratings-v2 created
```

Next, we install MongoDB itself:

```
$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
bookinfo/platform/kube/bookinfo-db.yaml
service/mongodb created
deployment.apps/mongodb-v1 created
```

Finally, we need to create virtual services for the reviews and ratings services:

```
$ kb apply -f https://raw.githubusercontent.com/istio/istio/release-1.15/samples/
bookinfo/networking/virtual-service-ratings-db.yaml
virtualservice.networking.istio.io/reviews configured
virtualservice.networking.istio.io/ratings configured
```

Let's hit our product page to generate traffic:

```
$ http http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"
<title>Simple Bookstore App</title>
```

At this point, we can expose Prometheus directly:

```
$ k -n istio-system port-forward deploy/prometheus 9090:9090
Forwarding from 127.0.0.1:9090 -> 9090
Forwarding from [::1]:9090 -> 9090
```

Or, alternatively, using `istioctl dashboard prometheus`, which will do the port-forwarding as well as launching the browser for you at the forwarded URL of `http://localhost:9090/`.

We can view the slew of new metrics available from both Istio services, Istio control plane and especially Envoy. Here is a very small subset of the available metrics:

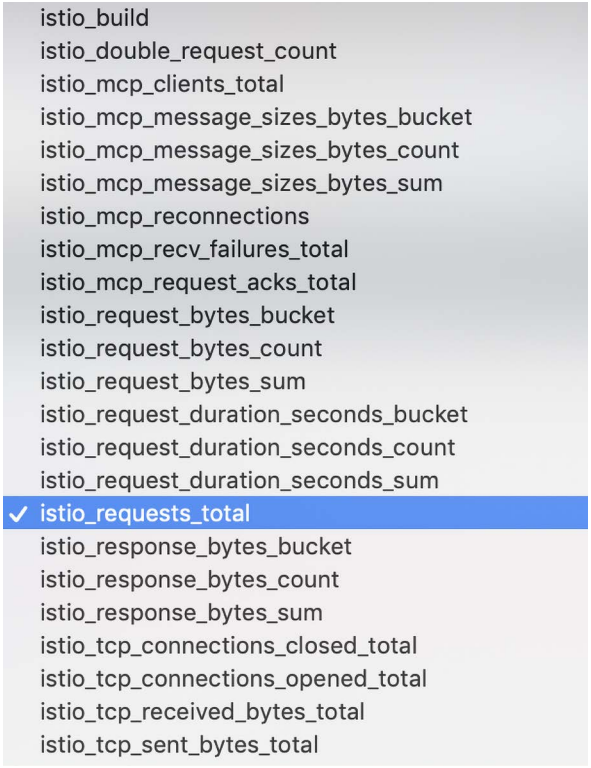


Figure 14.11: Available Istio metrics

The last pillar of observability is distributed tracing.

Distributed tracing

Istio configures the Envoy proxies to generate trace spans for their associated services. The services themselves are responsible for forwarding the request context. Istio can work with multiple tracing backends, such as:

- Jaeger
- Zipkin
- LightStep
- DataDog

Here are the request headers that services should propagate (only some may be present for each request depending on the tracing backend):

```
x-request-id
x-b3-traceid
x-b3-spanid
x-b3-parentspanid
x-b3-sampled
x-b3-flags
x-ot-span-context
x-cloud-trace-context
traceparent
grpc-trace-bin
```

The sampling rate for tracing is controlled by the mesh config. The default is 1%. Let's change it to 100% for demonstration purposes:

```
$ cat <<'EOF' > ./tracing.yaml
apiVersion: install.istio.io/v1alpha1
kind: IstioOperator
spec:
  meshConfig:
    enableTracing: true
    defaultConfig:
      tracing:
        sampling: 100
EOF

$ istioctl install -f ./tracing.yaml
```

Let's verify the sampling rate was updated to `100`:

```
$ k get cm -n istio-system istio -o yaml | yq .data
mesh: |-
  defaultConfig:
    discoveryAddress: istiod.istio-system.svc:15012
    proxyMetadata: {}
    tracing:
      sampling: 100
      zipkin:
        address: zipkin.istio-system:9411
  enablePrometheusMerge: true
  enableTracing: true
  rootNamespace: istio-system
  trustDomain: cluster.local
meshNetworks: 'networks: {}'
```

Let's hit the product page a couple of times:

```
$ http http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"
<title>Simple Bookstore App</title>

$ http http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"
<title>Simple Bookstore App</title>

$ http http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"
<title>Simple Bookstore App</title>
```

Now, we can start the Jaeger UI and explore the traces:

```
$ istioctl dashboard jaeger
http://localhost:52466
Handling connection for 9090
```

Your browser will automatically open and you should see the familiar Jaeger dashboard where you can select a service and search for traces:

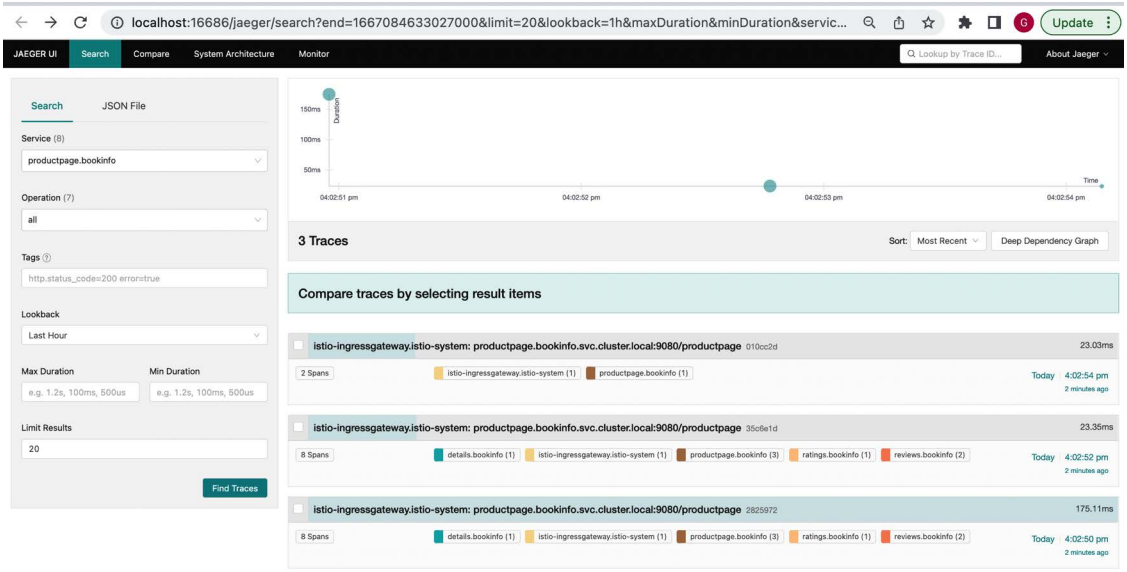


Figure 14.12: Jaeger dashboard

You can click on a trace to see a detailed view of the flow of the request through the system:

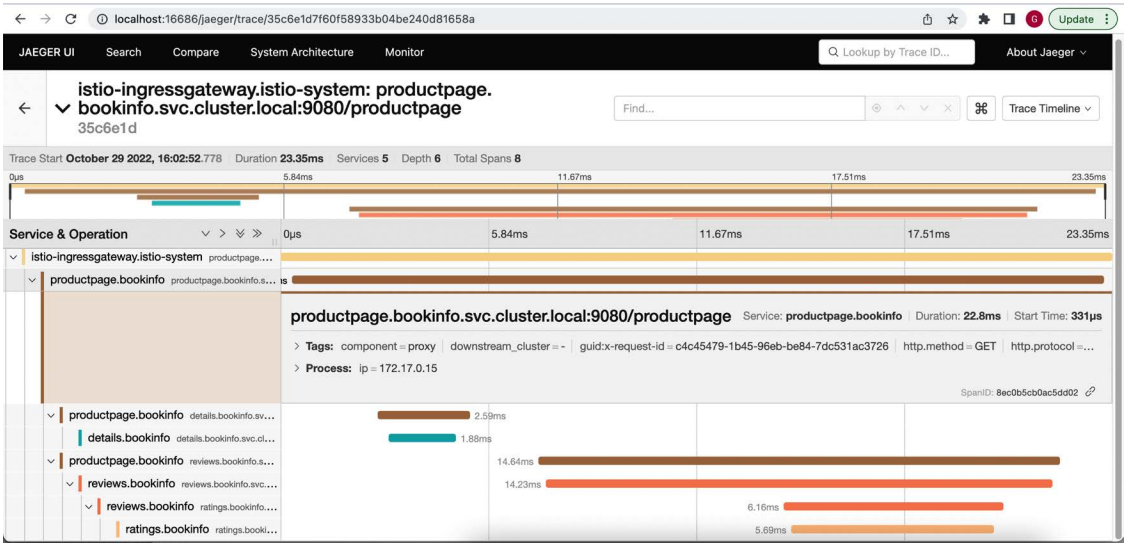


Figure 14.13: Flow of a request through the system

Let's look at a dedicated service mesh visualization tool.

Visualizing your service mesh with Kiali

Kiali is an open source project that ties together Prometheus, Grafana, and Jaeger to provide an observability console to your Istio service mesh. It can answer questions like:

- What microservices participate in the Istio service mesh?
- How are these microservices connected?
- How are these microservices performing?

It has various views, and it really allows you to slice and dice your service mesh by zooming in and out, filtering, and selecting various properties to display. It's got several views that you can switch between.

You can start it like so:

```
$ istioctl dashboard kiali
```

Here is the **Overview** page:

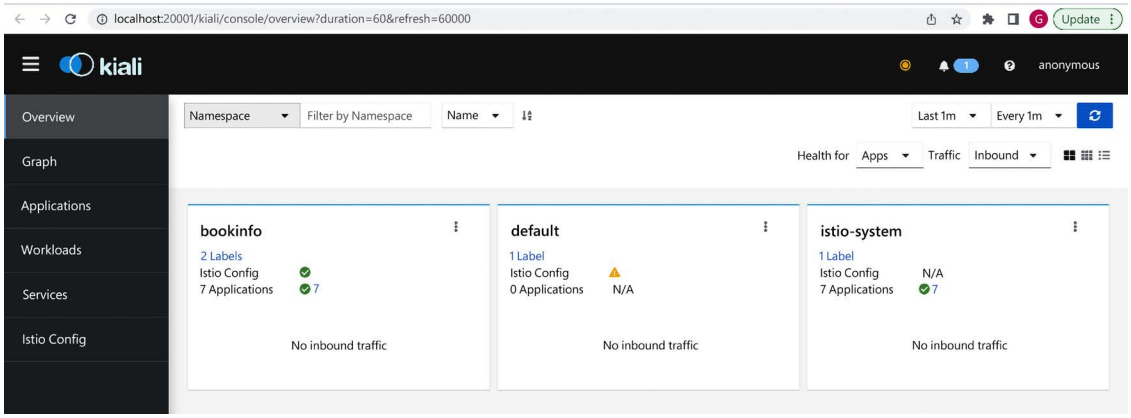


Figure 14.14: Kiali overview page

But, the most interesting view is the graph view, which can show your services and how they relate to each other and is fully aware of versions and requests flowing between different workloads, including the percentage of requests and latency. It can show both HTTP and TCP services and really provides a great picture of how your service mesh behaves.

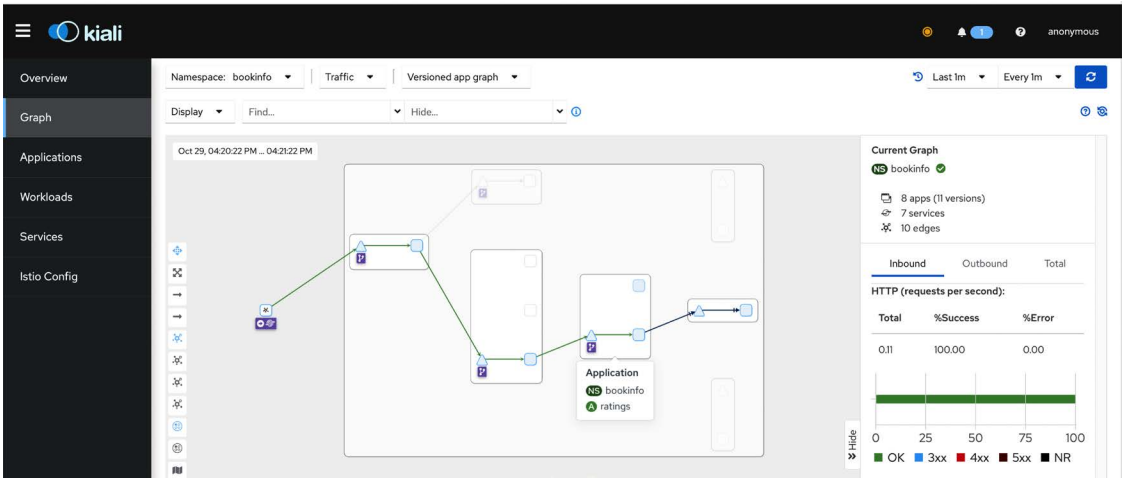


Figure 14.15: Kiali graph view

We have covered the monitoring and observability of Istio, including logs, metrics, and distributed tracing, and have shown how to use Kiali to visualize your service mesh.

Summary

In this chapter, we did a very comprehensive study of service meshes on Kubernetes. Service meshes are here to stay. They are simply the right way to operate a complex distributed system. Separating all operational concerns from the proxies and having the service mesh control them is a paradigm shift. Kubernetes, of course, is designed primarily for complex distributed systems, so the value of the service mesh becomes clear right away. It is also great to see that there are many options for service meshes on Kubernetes. While most service meshes are not specific to Kubernetes, it is one of the most important deployment platforms. In addition, we did a thorough review of Istio – arguably the service mesh with the most momentum – and took it through its paces. We demonstrated many of the benefits of service meshes and how they integrate with various other systems. You should be able to evaluate how useful a service mesh could be for your system and whether you should deploy one and start enjoying the benefits.

In the next chapter, we look at the myriad ways that we can extend Kubernetes and take advantage of its modular and flexible design. This is one of the hallmarks of Kubernetes and why it was adopted so quickly by so many communities.

15

Extending Kubernetes

In this chapter, we will dig deep into the guts of Kubernetes. We will start with the Kubernetes API and learn how to work with Kubernetes programmatically via direct access to the API, the controller-runtime Go library, and automating kubectl. Then, we'll look into extending the Kubernetes API with custom resources. The last part is all about the various plugins Kubernetes supports. Many aspects of Kubernetes operation are modular and designed for extension. We will examine the API aggregation layer and several types of plugins, such as custom schedulers, authorization, admission control, custom metrics, and volumes. Finally, we'll look into extending kubectl and adding your own commands.

The covered topics are as follows:

- Working with the Kubernetes API
- Extending the Kubernetes API
- Writing Kubernetes and kubectl plugins
- Writing webhooks

Working with the Kubernetes API

The Kubernetes API is comprehensive and encompasses the entire functionality of Kubernetes. As you may expect, it is huge. But it is designed very well using best practices, and it is consistent. If you understand the basic principles, you can discover everything you need to know. We covered the Kubernetes API itself in *Chapter 1*, *Understanding Kubernetes Architecture*. If you need a refresher, go and take a look. In this section, we're going to dive deeper and learn how to access and work with the Kubernetes API. But, first let's look at OpenAPI, which is the formal foundation that gives structure to the entire Kubernetes API.

Understanding OpenAPI

OpenAPI (formerly Swagger) is an open standard that defines a language- and framework-agnostic way to describe RESTful APIs. It provides a standardized, machine-readable format for describing APIs, including their endpoints, parameters, request and response bodies, authentication, and other metadata.

In the context of Kubernetes, OpenAPI is used to define and document the API surface of a Kubernetes cluster. OpenAPI is used in Kubernetes to provide a standardized way to document and define the API objects that can be used to configure and manage the cluster. The Kubernetes API is based on a declarative model, where users define the desired state of their resources using YAML or JSON manifests. These manifests follow the OpenAPI schema, which defines the structure and properties of each resource. Kubernetes uses the OpenAPI schema to validate manifests, provide auto-completion and documentation in API clients, and generate API reference documentation.

One of the key benefits of using OpenAPI in Kubernetes is that it enables code generation for client libraries. This allows developers to interact with the Kubernetes API using their programming language of choice and generated client libraries, which provide a native and type-safe way to interact with the API.

Additionally, OpenAPI allows tools like kubectl to provide autocompletion and validation for Kubernetes resources.

OpenAPI also enables automated documentation generation for the Kubernetes API. With the OpenAPI schema, Kubernetes can automatically generate API reference documentation, which serves as a comprehensive and up-to-date resource for understanding the Kubernetes API and its capabilities.

Kubernetes has had stable support for OpenAPI v3 since Kubernetes 1.27.

Check out `https://www.openapis.org` for more details.

In order to work with the Kubernetes API locally, we need to set up a proxy.

Setting up a proxy

To simplify access, you can use kubectl to set up a proxy:

```
$ k proxy --port 8080
```

Now, you can access the API server on `http://localhost:8080` and it will reach the same Kubernetes API server that kubectl is configured for.

Exploring the Kubernetes API directly

The Kubernetes API is highly discoverable. You can just browse to the URL of the API server at `http://localhost:8080` and get a nice JSON document that describes all the available operations under the `paths` key.

Here is a partial list due to space constraints:

```
{
    "paths": [
        "/api",
        "/api/v1",
        "/apis",
        "/apis/",
        "/apis/admissionregistration.k8s.io",
```

```
        "/apis/admissionregistration.k8s.io/v1",
        "/apis/apiextensions.k8s.io",
        "/livez/poststarthook/storage-object-count-tracker-hook",
        "/logs",
        "/metrics",
        "/openapi/v2",
        "/openapi/v3",
        "/openapi/v3/",
        "/openid/v1/jwks",
        "/readyz/shutdown",
        "/version"
    ]
}
```

You can drill down any one of the paths. For example, to discover the endpoint for the `default` namespace, I first called the `/api` endpoint, then discovered `/api/v1`, which told me there was `/api/v1/namespaces`, which pointed me to `/api/v1/namespaces/default`. Here is the response from the `/api/v1/namespaces/default` endpoint:

```
{
  "kind": "Namespace",
  "apiVersion": "v1",
  "metadata": {
    "name": "default",
    "uid": "7e39c279-949a-4fb6-ae47-796bb797082d",
    "resourceVersion": "192",
    "creationTimestamp": "2022-11-13T04:33:00Z",
    "labels": {
      "kubernetes.io/metadata.name": "default"
    },
    "managedFields": [
      {
        "manager": "kube-apiserver",
        "operation": "Update",
        "apiVersion": "v1",
        "time": "2022-11-13T04:33:00Z",
        "fieldsType": "FieldsV1",
        "fieldsV1": {
          "f:metadata": {
            "f:labels": {
              ".": {},
              "f:kubernetes.io/metadata.name": {}
            }
          }
        }
```

```
      }
    ]
  },
  "spec": {
    "finalizers": [
      "kubernetes"
    ]
  },
  "status": {
    "phase": "Active"
  }
}
```

You can explore the Kubernetes API from the command line using tools like cURL or even kubectl itself, but sometimes using a GUI application is more convenient.

Using Postman to explore the Kubernetes API

Postman (`https://www.getpostman.com`) is a very polished application for working with RESTful APIs. If you lean more to the GUI side, you may find it extremely useful.

The following screenshot shows the available endpoints under the batch v1 API group:

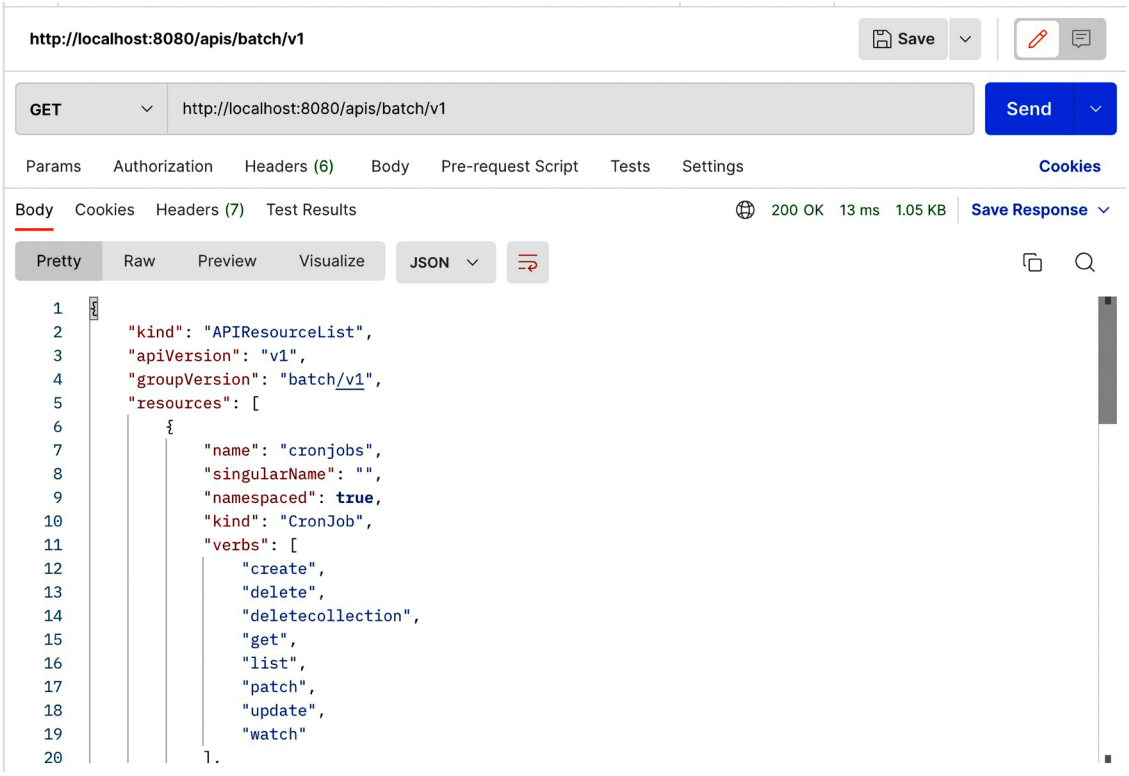


Figure 15.1: The available endpoints under the batch v1 API group

Postman has a lot of options, and it organizes the information in a very pleasing way. Give it a try.

Filtering the output with HTTPie and jq

The output from the API can be too verbose sometimes. Often, you're interested just in one value out of a huge chunk of a JSON response. For example, if you want to get the names of all running services, you can hit the `/api/v1/services` endpoint. The response, however, includes a lot of additional information that is irrelevant. Here is a very partial subset of the output:

```
$ http http://localhost:8080/api/v1/services
{
  "kind": "ServiceList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "3237"
  },
  "items": [
    ...
    {
      "metadata": {
        "name": "kube-dns",
        "namespace": "kube-system",
        ...
      },
      "spec": {
        ...
        "selector": {
          "k8s-app": "kube-dns"
        },
        "clusterIP": "10.96.0.10",
        "type": "ClusterIP",
        "sessionAffinity": "None",
      },
      "status": {
        "loadBalancer": {}
      }
    }
  ]
}
```

The complete output is 193 lines long! Let's see how to use HTTPie and jq to gain full control over the output and show only the names of the services. I prefer HTTPie(`https://httpie.org/`) over cURL for interacting with REST APIs on the command line. The jq (`https://stedolan.github.io/jq/`) command-line JSON processor is great for slicing and dicing JSON.

Examining the full output, you can see that the service name is in the `metadata` section of each item in the `items` array. The jq expression that will select just the name is as follows:

```
.items[].metadata.name
```

Here is the full command and output on a fresh kind cluster:

```
$ http http://localhost:8080/api/v1/services | jq '.items[].metadata.name'
"kubernetes"
"kube-dns"
```

Accessing the Kubernetes API via the Python client

Exploring the API interactively using HTTPie and jq is great, but the real power of APIs comes when you consume and integrate them with other software. The Kubernetes Incubator project provides a full-fledged and very well-documented Python client library. It is available at `https://github.com/kubernetes-incubator/client-python`.

First, make sure you have Python installed (`https://wiki.python.org/moin/BeginnersGuide/Download`). Then install the Kubernetes package:

```
$ pip install kubernetes
```

To start talking to a Kubernetes cluster, you need to connect to it. Start an interactive Python session:

```
$ python
Python 3.9.12 (main, Aug 25 2022, 11:03:34)
[Clang 13.1.6 (clang-1316.0.21.2.3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The Python client can read your kubectl config:

```
>>> from kubernetes import client, config
>>> config.load_kube_config()

>>> v1 = client.CoreV1Api()
```

Or it can connect directly to an already running proxy:

```
>>> from kubernetes import client, config
>>> client.Configuration().host = 'http://localhost:8080'
>>> v1 = client.CoreV1Api()
```

Note that the `client` module provides methods to get access to different group versions, such as `CoreV1Api`.

Dissecting the CoreV1Api group

Let's dive in and understand the `CoreV1Api` group. The Python object has 407 public attributes!

```
>>> attributes = [x for x in dir(v1) if not x.startswith('__')]
>>> len(attributes)
407
```

We ignore the attributes that start with double underscores because those are special class/instance methods unrelated to Kubernetes.

Let's pick ten random methods and see what they look like:

```
>>> import random
>>> from pprint import pprint as pp
>>> pp(random.sample(attributes, 10))
['replace_namespaced_persistent_volume_claim',
 'list_config_map_for_all_namespaces_with_http_info',
 'connect_get_namespaced_pod_attach_with_http_info',
 'create_namespaced_event',
 'connect_head_node_proxy_with_path',
 'create_namespaced_secret_with_http_info',
 'list_namespaced_service_account',
 'connect_post_namespaced_pod_portforward_with_http_info',
 'create_namespaced_service_account_token',
 'create_namespace_with_http_info']
```

Very interesting. The attributes begin with a verb such as `replace`, `list`, or `create`. Many of them have a notion of a namespace and many have a `with_http_info` suffix. To understand this better, let's count how many verbs exist and how many attributes use each verb (where the verb is the first token before the underscore):

```
>>> from collections import Counter
>>> verbs = [x.split('_')[0] for x in attributes]
>>> pp(dict(Counter(verbs)))
{'api': 1,
 'connect': 96,
 'create': 38,
 'delete': 58,
 'get': 2,
 'list': 56,
 'patch': 50,
 'read': 54,
 'replace': 52}
```

We can drill further and look at the interactive help for a specific attribute:

```
>>> help(v1.create_node)
```

```
Help on method create_node in module kubernetes.client.apis.core_v1_api:

create_node(body, **kwargs) method of kubernetes.client.api.core_v1_api.CoreV1Api
instance
    create_node  # noqa: E501

    create a Node  # noqa: E501
    This method makes a synchronous HTTP request by default. To make an
    asynchronous HTTP request, please pass async_req=True
    >>> thread = api.create_node(body, async_req=True)
    >>> result = thread.get()

    :param async_req bool: execute request asynchronously
    :param V1Node body: (required)
    :param str pretty: If 'true', then the output is pretty printed.
    :param str dry_run: When present, indicates that modifications should not be
persisted. An invalid or unrecognized dryRun directive will result in an error
response and no further processing of the request. Valid values are: - All: all dry
run stages will be processed
    :param str field_manager: fieldManager is a name associated with the actor or
entity that is making these changes. The value must be less than or 128 characters
long, and only contain printable characters, as defined by https://golang.org/pkg/
unicode/#IsPrint.
    :param str field_validation: fieldValidation instructs the server on how to
handle objects in the request (POST/PUT/PATCH) containing unknown or duplicate
fields, provided that the `ServerSideFieldValidation` feature gate is also enabled.
Valid values are: - Ignore: This will ignore any unknown fields that are silently
dropped from the object, and will ignore all but the last duplicate field that
the decoder encounters. This is the default behavior prior to v1.23 and is the
default behavior when the `ServerSideFieldValidation` feature gate is disabled. -
Warn: This will send a warning via the standard warning response header for each
unknown field that is dropped from the object, and for each duplicate field that
is encountered. The request will still succeed if there are no other errors, and
will only persist the last of any duplicate fields. This is the default when the
`ServerSideFieldValidation` feature gate is enabled. - Strict: This will fail the
request with a BadRequest error if any unknown fields would be dropped from the
object, or if any duplicate fields are present. The error returned from the server
will contain all unknown and duplicate fields encountered.
    :param _preload_content: if False, the urllib3.HTTPResponse object will
                             be returned without reading/decoding response
                             data. Default is True.
    :param _request_timeout: timeout setting for this request. If one
                             number provided, it will be total request
```

```
                          timeout. It can also be a pair (tuple) of
                          (connection, read) timeouts.
    :return: V1Node
             If the method is called
             returns the request thread.
```

We see that the API is vast, which makes sense because it represents the entire Kubernetes API. We also learned how to discover groups of related methods and how to get detailed information on specific methods.

You can poke around yourself and learn more about the API. Let's look at some common operations, such as listing, creating, and watching objects.

Listing objects

You can list different kinds of objects. The method names start with `list_`. Here is an example listing all namespaces:

```
>>> for ns in v1.list_namespace().items:
...     print(ns.metadata.name)
...
default
kube-node-lease
kube-public
kube-system
local-path-storage
```

Creating objects

To create an object, you need to pass a `body` parameter to the `create` method. The `body` must be a Python dictionary that is equivalent to a YAML configuration manifest you would use with kubectl. The easiest way to do it is to actually use a YAML manifest and then use the Python YAML module (not part of the standard library and must be installed separately) to read the YAML file and load it into a dictionary. For example, to create an `nginx-deployment` with 3 replicas, we can use this YAML manifest (`nginx-deployment.yaml`):

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
```

```
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

To install the `yaml` Python module, type this command:

```
$ pip install yaml
```

Then the following Python program (`create_nginx_deployment.py`) will create the deployment:

```
from os import path

import yaml
from kubernetes import client, config


def main():
    # Configs can be set in Configuration class directly or using
    # helper utility. If no argument provided, the config will be
    # loaded from default location.
    config.load_kube_config()

    with open(path.join(path.dirname(__file__),
                        'nginx-deployment.yaml')) as f:
        dep = yaml.safe_load(f)
        k8s = client.AppsV1Api()
        dep = k8s.create_namespaced_deployment(body=dep,
                                               namespace="default")
        print(f"Deployment created. status='{dep.status}'")


if __name__ == '__main__':
    main()
```

Let's run it and check the deployment was actually created using kubectl:

```
$ python create_nginx_deployment.py
Deployment created. status='{'available_replicas': None,
```

```
 'collision_count': None,
 'conditions': None,
 'observed_generation': None,
 'ready_replicas': None,
 'replicas': None,
 'unavailable_replicas': None,
 'updated_replicas': None}'

$ k get deploy
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           56s
```

Watching objects

Watching objects is an advanced capability. It is implemented using a separate `watch` module. Here is an example of watching 10 namespace events and printing them to the screen (`watch_demo.py`):

```
from kubernetes import client, config, watch

# Configs can be set in Configuration class directly or using helper utility
config.load_kube_config()
v1 = client.CoreV1Api()

count = 10
w = watch.Watch()
for event in w.stream(v1.list_namespace, _request_timeout=60):
    print(f"Event: {event['type']} {event['object'].metadata.name}")
    count -= 1
    if count == 0:
        w.stop()
print('Done.')
```

Here is the output:

```
$ python watch_demo.py
Event: ADDED kube-node-lease
Event: ADDED default
Event: ADDED local-path-storage
Event: ADDED kube-system
Event: ADDED kube-public
```

Note that only 5 events were printed (one for each namespace) and the program continues to watch for more events.

Let's create and delete some namespaces in a separate terminal window, so the program can end:

```
$ k create ns ns-1
namespace/ns-1 created

$ k delete ns ns-1
namespace "ns-1" deleted

$ k create ns ns-2
namespace/ns-2 created
The final output is:
$ python watch_demo.py
Event: ADDED default
Event: ADDED local-path-storage
Event: ADDED kube-system
Event: ADDED kube-public
Event: ADDED kube-node-lease
Event: ADDED ns-1
Event: MODIFIED ns-1
Event: MODIFIED ns-1
Event: DELETED ns-1
Event: ADDED ns-2
Done.
```

You can of course react to events and perform a useful action when an event happens (e.g., automatically deploy a workload in each new namespace).

Creating a pod via the Kubernetes API

The API can be used for creating, updating, and deleting resources too. Unlike working with kubectl, the API requires specifying the manifests in JSON and not YAML syntax (although every JSON document is also valid YAML). Here is a JSON pod definition (`nginx-pod.json`):

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata":{
    "name": "nginx",
    "namespace": "default",
    "labels": {
      "name": "nginx"
    }
  },
  "spec": {
```

```
    "containers": [{
                     "name": "nginx",
                     "image": "nginx",
                     "ports": [{"containerPort": 80}]
                   }]
  }
}
```

The following command will create the pod via the API:

```
$ http POST http://localhost:8080/api/v1/namespaces/default/pods @nginx-pod.json
```

To verify it worked, let's extract the name and status of the current pods. The endpoint is `/api/v1/namespaces/default/pods`.

The jq expression is `items[].metadata.name,.items[].status.phase`.

Here is the full command and output:

```
$ FILTER='.items[].metadata.name,.items[].status.phase'
$ http http://localhost:8080/api/v1/namespaces/default/pods | jq $FILTER
"nginx"
"Running"
```

Controlling Kubernetes using Go and controller-runtime

Python is cool and easy to work with, but for production-level tools, controllers, and operators, I prefer to use Go, and in particular the controller-runtime project. The controller-runtime is the standard Go client to use to access the Kubernetes API.

Using controller-runtime via go-k8s

The controller-runtime project is a set of Go libraries that can fully query and manipulate Kubernetes in a very efficient manner (e.g., advanced caching to avoid overwhelming the API server).

Working directly with controller-runtime is not easy. There are many interlocking pieces and different ways to accomplish things.

See `https://pkg.go.dev/sigs.k8s.io/controller-runtime`.

I created a little open-source project called `go-k8s` that encapsulates some of the complexity and helps with using a subset of the controller-runtime functionality with less hassle.

Check it out here: `https://github.com/the-gigi/go-k8s/tree/main/pkg/client`.

Note that the go-k8s project has other libraries, but we will focus on the client library.

The go-k8s client package supports two types of clients: `Clientset` and `DynamicClient`. The `Clientset` client supports working with well-known kinds, but explicitly specifying the API version, kind, and operation as method names. For example, listing all pods using `Clientset` looks like this:

```
podList, err := clientset.CoreV1().Pods("ns-1").List(context.Background(),
metav1.ListOptions{})
```

It returns a pod list and an error. The error is `nil` if everything is OK. The pod list is of struct type `PodList`, which is defined here: `https://github.com/kubernetes/kubernetes/blob/master/pkg/apis/core/types.go#L2514`.

Conveniently, you can find all the Kubernetes API types in the same file. The API is very nested, for example, a `PodList`, as you may expect, is a list of `Pod` objects. Each `Pod` object has `TypeMeta`, `ObjectMeta`, `PodSpec`, and `PodStatus`:

```
type Pod struct {
    metav1.TypeMeta
    metav1.ObjectMeta
    Spec PodSpec
    Status PodStatus
}
```

In practice, this means that when you make a call through the `Clientset`, you get back a strongly typed nested object that is very easy to work with. For example, if we want to check if a pod has a label called `app` and its value, we can do it in one line:

```
app, ok := pods[0].ObjectMeta.Labels["app"]
```

If the label doesn't exist, `ok` will be false. If it does exist, then its value will be available in the `app` variable.

Now, let's look at `DynamicClient`. Here, you get the ultimate flexibility and the ability to work with well-known types as well as custom types. In particular, if you want to create arbitrary resources, the dynamic client can operate in a generic way on any Kubernetes type.

However, with the dynamic client, you always get back a generic object of type `Unstructured`, defined here: `https://github.com/kubernetes/apimachinery/blob/master/pkg/apis/meta/v1/unstructured/unstructured.go#L41`.

It is really a very thin wrapper around the generic Golang type `map[string]interface{}`. It has a single field called `Object` of type `map[string]interface{}`. This means that the object you get back is a map of field names to arbitrary other objects (represented as `interface{}`). To drill down the hierarchy, we have to perform typecasting, which means taking an `interface{}` value and casting it explicitly to its actual type. Here is a simple example:

```
var i interface{} = 5
x, ok := i.(int)
```

Now, x is a variable of type `int` with a value of 5 that can be used as an integer. The original `i` variable can't be used as an integer because its type is the generic `interface{}` even if it contains an integer value.

In the case of the objects returned from the dynamic client, we have to keep typecasting an `interface{}` to a `map[string]interface{}` until we get to the field we are interested in. To get to the `app` label of our pod, we need to follow this path:

```
pod := pods[0].Object
metadata := pod["metadata"].(map[string]interface{})
labels := metadata["labels"].(map[string]interface{})
app, ok := labels["app"].(string)
```

This is extremely tiresome and error-prone. Luckily, there is a better way. The Kubernetes `apimachinery/runtime` package provides a conversion function that can take an unstructured object and convert it into a known type:

```
pod := pods[0].Object
var p corev1.Pod
err = runtime.DefaultUnstructuredConverter.FromUnstructured(pod, &p)
if err != nil {
  return err
}
app, ok = p.ObjectMeta.Labels["app"]
```

The controller-runtime is very powerful, but it can be tedious to deal with all the types. One way to "cheat" is to use kubectl, which actually uses the controller-runtime under the covers. This is especially easy using Python and its dynamic typing.

Invoking kubectl programmatically from Python and Go

If you don't want to deal with the REST API directly or client libraries, you have another option. Kubectl is used mostly as an interactive command-line tool, but nothing is stopping you from automating it and invoking it through scripts and programs. There are some benefits to using kubectl as your Kubernetes API client:

- Easy to find examples for any usage
- Easy to experiment on the command line to find the right combination of commands and arguments
- kubectl supports output in JSON or YAML for quick parsing
- Authentication is built in via kubectl configuration

Using Python subprocess to run kubectl

Let's use Python first, so you can compare using the official Python client to rolling your own. Python has a module called `subprocess` that can run external processes such as kubectl and capture the output.

Here is a Python 3 example running kubectl on its own and displaying the beginning of the usage output:

```
>>> import subprocess
>>> out = subprocess.check_output('kubectl').decode('utf-8')
>>> print(out[:276])
```

Kubectl controls the Kubernetes cluster manager.

Find more information at `https://kubernetes.io/docs/reference/kubectl/overview/`.

The `check_output()` function captures the output as a bytes array, which needs to be decoded into `utf-8` to be displayed properly. We can generalize it a little bit and create a convenience function called `k()` in the `k.py` file. It accepts any number of arguments it feeds to kubectl, and then decodes the output and returns it:

```
from subprocess import check_output

def k(*args):
    out = check_output(['kubectl'] + list(args))
    return out.decode('utf-8')
```

Let's use it to list all the running pods in the default namespace:

```
>>> from k import k
>>> print(k('get', 'po'))
NAME                                     READY   STATUS
RESTARTS   AGE
nginx                                    1/1     Running
0          4h48m
nginx-deployment-679f9c75b-c79mv         1/1     Running
0          132m
nginx-deployment-679f9c75b-cnmvk         1/1     Running
0          132m
nginx-deployment-679f9c75b-gzfgk         1/1     Running
0          132m
```

This is nice for display, but kubectl already does that. The real power comes when you use the structured output options with the `-o` flag. Then the result can be converted automatically into a Python object. Here is a modified version of the `k()` function that accepts a Boolean `use_json` keyword argument (defaults to `False`), and if `True`, adds `-o json` and then parses the JSON output to a Python object (dictionary):

```
from subprocess import check_output
import json


def k(*args, use_json=False):
    cmd = ['kubectl'] + list(args)
```

```
    if use_json:
        cmd += ['-o', 'json']
    out = check_output(cmd).decode('utf-8')
    if use_json:
        out = json.loads(out)

    return out
```

That returns a full-fledged API object, which can be navigated and drilled down just like when accessing the REST API directly or using the official Python client:

```
result = k('get', 'po', use_json=True)
>>> for r in result['items']:
...     print(r['metadata']['name'])
...
nginx-deployment-679f9c75b-c79mv
nginx-deployment-679f9c75b-cnmvk
nginx-deployment-679f9c75b-gzfgk
```

Let's see how to delete the deployment and wait until all the pods are gone. The kubectl `delete` command doesn't accept the `-o json` option (although it has `-o name`), so let's leave out `use_json`:

```
>>> k('delete', 'deployment', 'nginx-deployment')
while len(k('get', 'po', use_json=True)['items']) > 0:
    print('.')
print('Done.')
.
.
.
.
Done.
```

Python is great, but what if you prefer Go for automating kubectl? No worries, I have just the package for you. The `kugo` package provides a simple Go API to automate kubectl. You can find the code here: `https://github.com/the-gigi/kugo`.

It provides 3 functions: `Run()`, `Get()`, and `Exec()`.

The `Run()` function is your Swiss Army knife. It can run any kubectl command as is. Here is an example:

```
    cmd := fmt.Sprintf("create deployment test-deployment --image nginx
--replicas 3 -n ns-1")
    _, err := kugo.Run(cmd)
```

This is super convenient because you can interactively compose the exact command and parameters you need using kubectl and then, once you've figured out everything, you can literally take the same command and pass it to `kuge.Run()` in your Go program.

The `Get()` function is a smart wrapper around `kubectl get`. It accepts a `GetRequest` parameter and provides several amenities: it supports field selectors, fetching by label, and different output types. Here is an example of fetching all namespaces by name using a custom kube config file and custom kube context:

```
    output, err := kugo.Get(kugo.GetRequest{
        BaseRequest: kugo.BaseRequest{
            KubeConfigFile: c.kubeConfigFile,
            KubeContext:    c.GetKubeContext(),
        },
        Kind:   "ns",
        Output: "name",
    })
```

Finally, the `Exec()` function is a wrapper around `kubectl exec` and lets you execute commands on a running pod/container. It accepts an `ExecRequest` that looks like this:

```
type GetRequest struct {
    BaseRequest

    Kind           string
    FieldSelectors []string
    Label          string
    Output         string
}
```

Let's look at the code of the Exec() function. It is pretty straightforward. It does basic validation that required fields like `Command` and `Target` were provided and then it builds a kubectl argument list starting with the `exec` command and finally calls the `Run()` function:

```
// Exec executes a command in a pod
//
// The target pod can specified by name or an arbitrary pod
// from a deployment or service.
//
// If the pod has multiple containers you can choose which
// container to run the command in
func Exec(r ExecRequest) (result string, err error) {
    if r.Command == "" {
        err = errors.New("Must specify Command field")
        return
    }

    if r.Target == "" {
```

```
        err = errors.New("Must specify Target field")
        return
    }

    args := []string{"exec", r.Target}
    if r.Container != "" {
        args = append(args, "-c", r.Container)
    }

    args = handleCommonArgs(args, r.BaseRequest)
    args = append(args, "--", r.Command)

    return Run(args...)
}
```

Now, that we have accessed Kubernetes programmatically via its REST API, client libraries, and by controlling kubectl, it's time to learn how to extend Kubernetes.

Extending the Kubernetes API

Kubernetes is an extremely flexible platform. It was designed from the get-go for extensibility and as it evolved, more parts of Kubernetes were opened up, exposed through robust interfaces, and could be replaced by alternative implementations. I would venture to say that the exponential adoption of Kubernetes across the board by start-ups, large companies, infrastructure providers, and cloud providers is a direct result of Kubernetes providing a lot of capabilities out of the box, but allowing easy integration with other actors. In this section, we will cover many of the available extension points, such as:

- User-defined types (custom resources)
- API access extensions
- Infrastructure extensions
- Operators
- Scheduler extensions

Let's understand the various ways you can extend Kubernetes.

Understanding Kubernetes extension points and patterns

Kubernetes is made of multiple components: the API server, etcd state store, controller manager, kube-proxy, kubelet, and container runtime. You can deeply extend and customize each and every one of these components, as well as adding your own custom components that watch and react to events, handle new requests, and modify everything about incoming requests.

The following diagram shows some of the available extension points and how they are connected to various Kubernetes components:

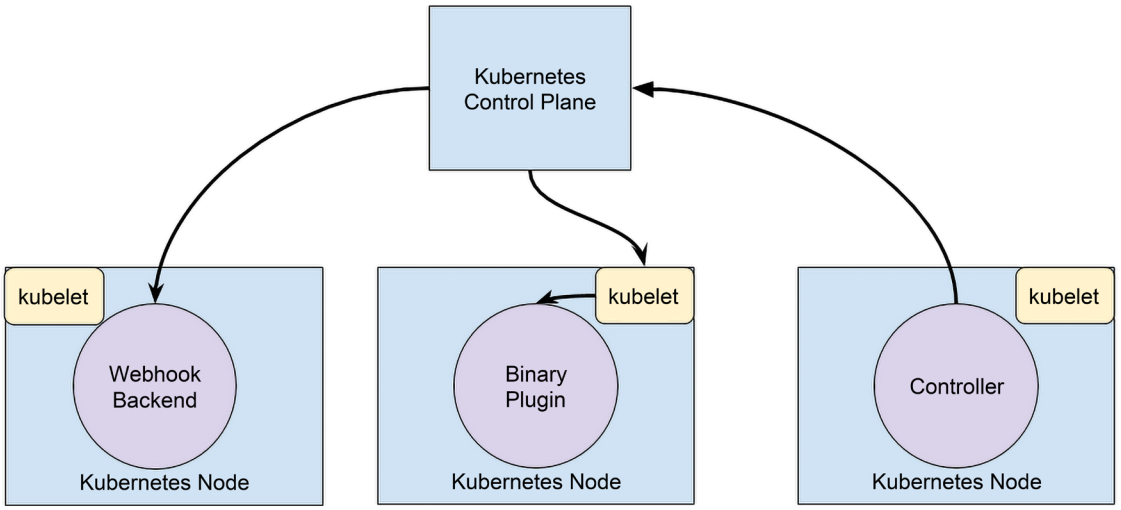


Figure 15.2: Available extension points

Let's see how to extend Kubernetes with plugins.

Extending Kubernetes with plugins

Kubernetes defines several interfaces that allow it to interact with a wide variety of plugins from infrastructure providers. We discussed some of these interfaces and plugins in detail in previous chapters. We will just list them here for completeness:

- **Container networking interface** (**CNI**) – the CNI supports a large number of networking solutions for connecting nodes and containers
- **Container storage interface** (**CSI**) – the CSI supports a large number of storage options for Kubernetes
- Device plugins – allows nodes to discover new node resources beyond CPU and memory (e.g., a GPU)

Extending Kubernetes with the cloud controller manager

Kubernetes needs to be deployed eventually on some nodes and use some storage and networking resources. Initially, Kubernetes supported only Google Cloud Platform and AWS. Other cloud providers had to customize multiple Kubernetes core components (Kubelet, the Kubernetes controller manager, and the Kubernetes API server) in order to integrate with Kubernetes. The Kubernetes developers identified it as a problem for adoption and created the **cloud controller manager** (**CCM**). The CCM cleanly defines the interaction between Kubernetes and the infrastructure layer it is deployed on. Now, cloud providers just provide an implementation of the CCM tailored to their infrastructure, and they can utilize upstream Kubernetes without costly and error-prone modification to the Kubernetes code. All the Kubernetes components interact with the CCM via the predefined interfaces and Kubernetes is blissfully unaware of which cloud (or no cloud) it is running on.

The following diagram demonstrates the interaction between Kubernetes and a cloud provider via the CCM:

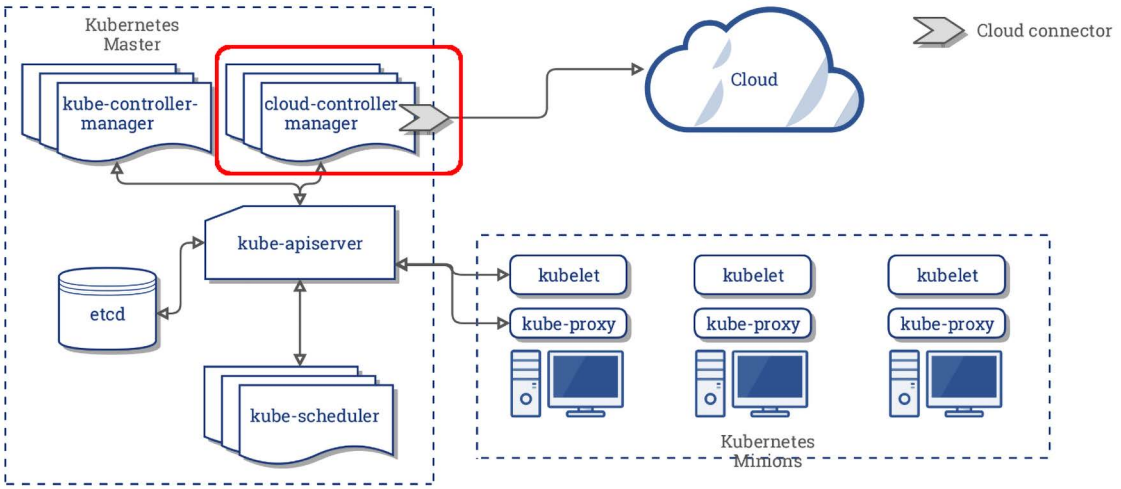


Figure 15.3: Interaction between Kubernetes and a cloud provider via the CCM

If you want to learn more about the CCM, check out this concise article I wrote a few years ago: `https://medium.com/@the.gigi/kubernetes-and-cloud-providers-b7a6227d3198`.

Extending Kubernetes with webhooks

Plugins run in the cluster, but in some cases, a better extensibility pattern is to delegate some functions to an out-of-cluster service. This is very common in the area of access control where companies and organizations may already have a centralized solution for identity and access control. In those cases, the webhook extensibility pattern is useful. The idea is that you can configure Kubernetes with an endpoint (webhook). Kubernetes will call the endpoint where you can implement your own custom functionality and Kubernetes will take action based on the response. We saw this pattern when we discussed authentication, authorization, and dynamic admission control in *Chapter 4*, *Securing Kubernetes*.

Kubernetes defines the expected payloads for each webhook. The webhook implementation must adhere to them in order to successfully interact with Kubernetes.

Extending Kubernetes with controllers and operators

The controller pattern is where you write a program that can run inside the cluster or outside the cluster, watch for events, and respond to them. The conceptual model for a controller is to reconcile the current state of the cluster (the parts the controller is interested in) with the desired state. A common practice for controllers is to read the `Spec` of an object, take some actions, and update its `Status`. A lot of the core logic of Kubernetes is implemented by a large set of controllers managed by the controller manager, but there is nothing stopping us from deploying our own controllers to the cluster or running controllers that access the API server remotely.

The operator pattern is another flavor of the controller pattern. Think of an operator as a controller that also has its own set of custom resources, which represents an application it manages. The goal of operators is to manage the lifecycle of an application that is deployed in the cluster or some out-of-cluster infrastructure. Check out `https://operatorhub.io` for examples of existing operators.

If you plan to build your own controllers, I recommend starting with Kubebuilder (`https://github.com/kubernetes-sigs/kubebuilder`). It is an open project maintained by the Kubernetes API Machinery SIG and has support for defining multiple custom APIs using CRDs, and scaffolds out the controller code to watch these resources. You will implement your controller in Go.

However, there are multiple other frameworks for writing controllers and operators with different approaches and using other programming languages:

- The Operator Framework
- Kopf
- kube-rs
- KubeOps
- KUDO
- Metacontroller

Check them out before you make your decision.

Extending Kubernetes scheduling

Kubernetes' primary job, in one sentence, is to schedule pods on nodes. Scheduling is at the heart of what Kubernetes does, and it does it very well. The Kubernetes scheduler can be configured in very advanced ways (daemon sets, taints, tolerations, etc.). But still, the Kubernetes developers recognize that there may be extraordinary circumstances where you may want to control the core scheduling algorithm. It is possible to replace the core Kubernetes scheduler with your own scheduler or run another scheduler side by side with the built-in scheduler to control the scheduling of a subset of the pods. We will see how to do that later in the chapter.

Extending Kubernetes with custom container runtimes

Kubernetes originally supported only Docker as a container runtime. The Docker support was embedded into the core Kubernetes codebase. Later, dedicated support for rkt was added. The Kubernetes developers saw the light and introduced the **container runtime interface** (**CRI**), a gRPC interface that enables any container runtime that implements it to communicate with the kubelet. Eventually, the hard-coded support for Docker and rkt was phased out and now the kubelet talks to the container runtime only through the CRI:

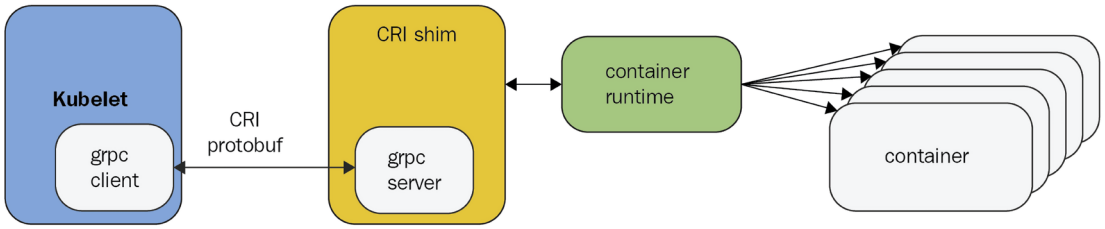


Figure 15.4: Kubelet talking to the container runtime through the CRI

Since the introduction of the CRI, the number of container runtimes that work with Kubernetes has exploded.

We've covered multiple ways to extend different aspects of Kubernetes. Let's turn our attention to the major concept of custom resources, which allow you to extend the Kubernetes API itself.

Introducing custom resources

One of the primary ways to extend Kubernetes is to define new types of resources called custom resources. What can you do with custom resources? Plenty. You can use them to manage, through the Kubernetes API, resources that live outside the Kubernetes cluster but that your pods communicate with. By adding those external resources as custom resources, you get a full picture of your system, and you benefit from many Kubernetes API features such as:

- Custom CRUD REST endpoints
- Versioning
- Watches
- Automatic integration with generic Kubernetes tooling

Other use cases for custom resources are metadata for custom controllers and automation programs.

Let's dive in and see what custom resources are all about.

In order to play nice with the Kubernetes API server, custom resources must conform to some basic requirements. Similar to built-in API objects, they must have the following fields:

- `apiVersion`: `apiextensions.k8s.io/v1`
- `metadata`: Standard Kubernetes object metadata
- `kind`: `CustomResourceDefinition`
- `spec`: Describes how the resource appears in the API and tools
- `status`: Indicates the current status of the CRD

The `spec` has an internal structure that includes fields like `group`, `names`, `scope`, `validation`, and `version`. The `status` includes the fields `acceptedNames` and `Conditions`. In the next section, I'll show you an example that clarifies the meaning of these fields.

Developing custom resource definitions

You develop your custom resources using custom resource definitions, AKA CRDs. The intention is for CRDs to integrate smoothly with Kubernetes, its API, and tooling. That means you need to provide a lot of information. Here is an example for a custom resource called Candy:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: candies.awesome.corp.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: awesome.corp.com
  # version name to use for REST API: /apis/<group>/<version>
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version.
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                flavor:
                  type: string
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: candies
    # singular name to be used as an alias on the CLI and for display
    singular: candy
    # kind is normally the CamelCased singular type. Your resource manifests
use this.
    kind: Candy
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
      - cn
```

The Candy CRD has several interesting parts. The metadata has a fully qualified name, which should be unique since CRDs are cluster-scoped. The spec has a `versions` section, which can contain multiple versions with a schema for each version that specifies the field of the custom resource. The schema follows the OpenAPI v3 specification (`https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#schemaObject`). The `scope` field could be either `Namespaced` or `Cluster`. If the scope is `Namespaced`, then the custom resources you create from the CRD will exist only in the namespace they were created in, whereas cluster-scoped custom resources are available in any namespace. Finally, the `names` section refers to the names of the custom resource (not the name of the CRD from the `metadata` section). The `names` section has `plural`, `singular`, `kind`, and `shortNames` options.

Let's create the CRD:

```
$ k create -f candy-crd.yaml
customresourcedefinition.apiextensions.k8s.io/candies.awesome.corp.com created
```

Note, that the metadata name is returned. It is common to use a plural name. Now, let's verify we can access it:

```
$ k get crd
NAME                        CREATED AT
candies.awesome.corp.com    2022-11-24T22:56:27Z
```

There is also an API endpoint for managing this new resource:

```
/apis/awesome.corp.com/v1/namespaces/<namespace>/candies/
```

Integrating custom resources

Once the `CustomResourceDefinition` object has been created, you can create custom resources of that resource kind – `Candy` in this case (candy becomes CamelCase Candy). Custom resources must respect the schema of the CRD. In the following example, the `flavor` field is set on the `Candy` object with the name `chocolate`. The `apiVersion` field is derived from the CRD `spec` group and `versions` fields:

```yaml
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  name: chocolate
spec:
  flavor: sweeeeeeet
```

Let's create it:

```
$ k create -f chocolate.yaml
candy.awesome.corp.com/chocolate created
```

Note that the spec must contain the `flavor` field from the schema.

At this point, kubectl can operate on Candy objects just like it works on built-in objects. Resource names are case-insensitive when using kubectl:

```
$ k get candies
NAME        AGE
chocolate   34s
```

We can also view the raw JSON data using the standard `-o json` flag. Let's use the short name `cn` this time:

```
$ k get cn -o json
{
    "apiVersion": "v1",
    "items": [
        {
            "apiVersion": "awesome.corp.com/v1",
            "kind": "Candy",
            "metadata": {
                "creationTimestamp": "2022-11-24T23:11:01Z",
                "generation": 1,
                "name": "chocolate",
                "namespace": "default",
                "resourceVersion": "750357",
                "uid": "49f68d80-e9c0-4c20-a87d-0597a60c4ed8"
            },
            "spec": {
                "flavor": "sweeeeeeet"
            }
        }
    ],
    "kind": "List",
    "metadata": {
        "resourceVersion": ""
    }
}
```

Dealing with unknown fields

The schema in the spec was introduced with the `apiextensions.k8s.io/v1` version of CRDs that became stable in Kubernetes 1.17. With `apiextensions.k8s.io/v1beta`, a schema wasn't required so arbitrary fields were the way to go. If you just try to change the version of your CRD from v1beta to v1, you're in for a rude awakening. Kubernetes will let you update the CRD, but when you try to create a custom resource later with unknown fields, it will fail.

You must define a schema for all your CRDs. If you must deal with custom resources that may have additional unknown fields, you can turn validation off, but the additional fields will be stripped off.

Here is a `Candy` resource that has an extra field, `texture`, not specified in the schema:

```
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  name: gummy-bear
spec:
  flavor: delicious
  texture: rubbery
```

If we try to create it with validation, it will fail:

```
$ k create -f gummy-bear.yaml
Error from server (BadRequest): error when creating "gummy-bear.yaml": Candy in
version "v1" cannot be handled as a Candy: strict decoding error: unknown field
"spec.texture"
```

But, if we turn validation off, then all is well, except that only the `flavor` field will be present and the `texture` field will not:

```
$ k create -f gummy-bear.yaml --validate=false
candy.awesome.corp.com/gummy-bear created

$ k get cn gummy-bear -o yaml
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  creationTimestamp: "2022-11-24T23:13:33Z"
  generation: 1
  name: gummy-bear
  namespace: default
  resourceVersion: "750534"
  uid: d77d9bdc-5a53-4f8e-8468-c29e2d46f919
spec:
  flavor: delicious
```

Sometimes, it can be useful to keep unknown fields. CRDs can support unknown fields by adding a special field to the schema.

Let's delete the current Candy CRD and replace it with a CRD that supports unknown fields:

```
$ k delete -f candy-crd.yaml
customresourcedefinition.apiextensions.k8s.io "candies.awesome.corp.com" deleted
```

```
$ k create -f candy-with-unknown-fields-crd.yaml
customresourcedefinition.apiextensions.k8s.io/candies.awesome.corp.com created
```

The new CRD has the `x-kubernetes-preserve-unknown-fields` field set to `true` in the `spec` property:

```
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              x-kubernetes-preserve-unknown-fields: true
              properties:
                flavor:
                  type: string
```

Let's create our gummy bear again WITH validation and check that the unknown `texture` field is present:

```
$ k create -f gummy-bear.yaml
candy.awesome.corp.com/gummy-bear created

$ k get cn gummy-bear -o yaml
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  creationTimestamp: "2022-11-24T23:38:01Z"
  generation: 1
  name: gummy-bear
  namespace: default
  resourceVersion: "752234"
  uid: 6863f767-5dc0-43f7-91f3-1c734931b979
spec:
  flavor: delicious
  texture: rubbery
```

Finalizing custom resources

Custom resources support finalizers just like standard API objects. A finalizer is a mechanism where objects are not deleted immediately but have to wait for special controllers that run in the background and watch for deletion requests. The controller may perform any necessary cleanup options and then remove its finalizer from the target object. There may be multiple finalizers on an object. Kubernetes will wait until all finalizers have been removed and only then delete the object. The finalizers in the metadata are just arbitrary strings that their corresponding controller can identify. Kubernetes doesn't know what they mean.

It just waits patiently for all the finalizers to be removed before deleting the object. Here is an example with a `Candy` object that has two finalizers: `eat-me` and `drink-me`:

```
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  name: chocolate
  finalizers:
  - eat-me
  - drink-me
spec:
  flavor: sweeeeeeet
```

Adding custom printer columns

By default, when you list custom resources with kubectl, you get only the name and the age of the resource:

```
$ k get cn
NAME         AGE
chocolate    11h
gummy-bear   16m
```

But the CRD schema allows you to add your own columns. Let's add the flavor and the age as printable columns to our `Candy` objects:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: candies.awesome.corp.com
spec:
  group: awesome.corp.com
  versions:
    - name: v1
        ...
      additionalPrinterColumns:
        - name: Flavor
          type: string
          description: The flavor of the candy
          jsonPath: .spec.flavor
        - name: Age
          type: date
          jsonPath: .metadata.creationTimestamp
   ...
```

Then we can apply it, add our candies again, and list them:

```
$ k apply -f candy-with-flavor-crd.yaml
customresourcedefinition.apiextensions.k8s.io/candies.awesome.corp.com configured

$ k get cn
NAME          FLAVOR       AGE
chocolate     sweeeeeeet   13m
gummy-bear    delicious    18m
```

Understanding API server aggregation

CRDs are great when all you need is some CRUD operations on your own types. You can just piggyback on the Kubernetes API server, which will store your objects and provide API support and integration with tooling like kubectl. If you need more power, you can run controllers that watch for your custom resources and perform some operations when they are created, updated, or deleted. The Kubebuilder (`https://github.com/kubernetes-sigs/kubebuilder`) project is a great framework for building Kubernetes APIs on top of CRDs with your own controllers.

But CRDs have limitations. If you need more advanced features and customization, you can use API server aggregation and write your own API server, which the Kubernetes API server will delegate to. Your API server will use the same API machinery as the Kubernetes API server itself. Some advanced capabilities are available only through the aggregation layer:

- Make your API server adopt different storage APIs rather than etcd
- Extend long-running subresources/endpoints like WebSocket for your own resources
- Integrate your API server with any other external systems
- Control the storage of your objects (custom resources are always stored in etcd)
- Custom operations beyond CRUD (e.g., exec or scale)
- Use protocol buffer payloads

Writing an extension API server is a non-trivial effort. If you decide you need all that power, there are a couple of good starting points. You can look at the sample API server for inspiration (`https://github.com/kubernetes/sample-apiserver`). You may want to check out the `apiserver-builder-alpha` project (`https://github.com/kubernetes-sigs/apiserver-builder-alpha`). It takes care of a lot of the necessary boilerplate code. The API builder provides the following capabilities:

- Bootstrap complete type definitions, controllers, and tests as well as documentation
- An extension control plane you can run on a local cluster or on an actual remote cluster
- Your generated controllers will be able to watch and update API objects
- Add resources (including sub-resources)
- Default values you can override if needed

There is also a walkthrough here: `https://kubernetes.io/docs/tasks/extend-kubernetes/setup-extension-api-server/`.

Building Kubernetes-like control planes

What if you want to use the Kubernetes model to manage other things and not just pods? It turns out that this is a very desirable capability. There is a project with a lot of momentum that provides it: `https://github.com/kcp-dev/kcp`.

kcp also ventures into multi-cluster management.

What does kcp bring to the table?

- It is a control plane for multiple conceptual clusters called workspaces
- It enables external API service providers to integrate with the central control plane using multi-tenant operators
- Users can consume APIs easily in their workspaces
- Scheduling workloads flexibly to physical clusters
- Move workloads transparently between compatible physical clusters
- Users can deploy their workloads while taking advantage of capabilities such as geographic replication and cross-cloud replication.

We have covered different ways to extend Kubernetes by adding controllers and aggregated API servers. Let's take a look at another mode of extending Kubernetes, by writing plugins.

Writing Kubernetes plugins

In this section, we will dive into the guts of Kubernetes and learn how to take advantage of its famous flexibility and extensibility. We will learn about different aspects that can be customized via plugins and how to implement such plugins and integrate them with Kubernetes.

Writing a custom scheduler

Kubernetes is all about orchestrating containerized workloads. The most fundamental responsibility is to schedule pods to run on cluster nodes. Before we can write our own scheduler, we need to understand how scheduling works in Kubernetes

Understanding the design of the Kubernetes scheduler

The Kubernetes scheduler has a very simple role – when a new pod needs to be created, assign it to a target node. That's it. The Kubelet on the target node will take it from there and instruct the container runtime on the node to run the pod's containers.

The Kubernetes scheduler implements the Controller pattern:

- Watch for pending pods
- Select the proper node for the pod
- Update the node's spec by setting the `nodeName` field

The only complicated part is selecting the target node. This process involves multiple steps split into two cycles:

1. The scheduling cycle
2. The binding cycle

While scheduling cycles are executed sequentially, binding cycles can be executed in parallel. If the target pod is considered unschedulable or an internal error occurs, the cycle will be terminated, and the pod will be placed back in the queue to be retried at a later time.

The scheduler is implemented using an extensible scheduler framework. The framework defines multiple extension points that you can plug into to affect the scheduling process. The following diagram shows the overall process and the extension points:

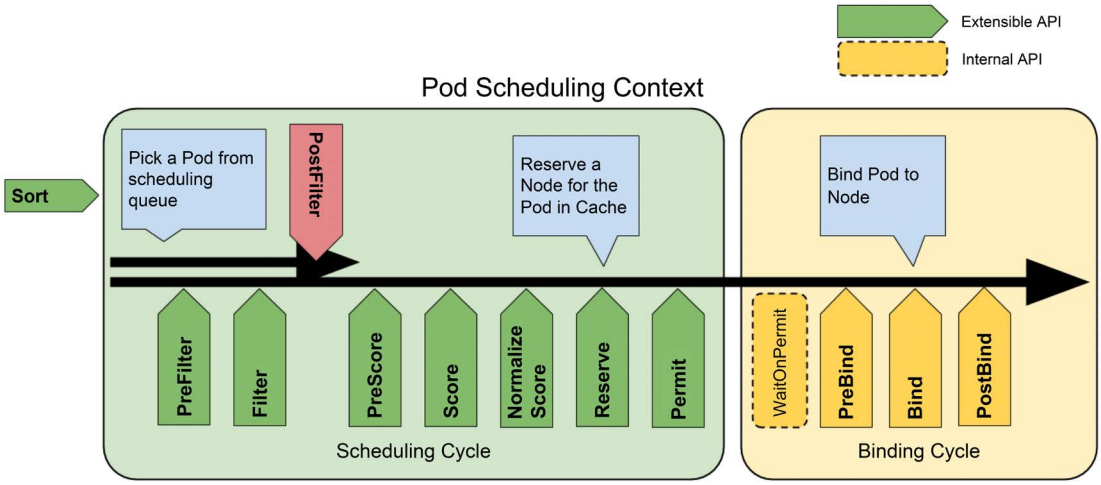


Figure 15.5: The workflow of the Kubernetes scheduler

The scheduler takes a tremendous amount of information and configuration into account. Filtering removes nodes that don't satisfy one of the hard constraints from the candidate list. Ranking nodes assigns a score to each of the remaining nodes and chooses the best node.

Here are the factors the scheduler evaluates when filtering nodes:

- Verify that the ports requested by the pod are available on the node, ensuring the required network connectivity.
- Ensure that the pod is scheduled on a node whose hostname matches the specified node preference.
- Validate the availability of requested resources (CPU and memory) on the node to meet the pod's requirements.
- Match the node's labels with the pod's node selector or node affinity to ensure proper scheduling.
- Confirm that the node supports the requested volume types, considering any failure zone restrictions for storage.

- Evaluate the node's capacity to accommodate the pod's volume requests, accounting for existing mounted volumes.
- Ensure the node's health by checking for indicators such as memory pressure or PID pressure.
- Evaluate the pod's tolerations to determine compatibility with the node's taints, enabling or restricting scheduling accordingly.

Once the nodes have been filtered, the scheduler will score the modes based on the following policies (which you can configure):

- Distribute pods across hosts while considering pods belonging to the same Service, StatefulSet, or ReplicaSet.
- Give priority to inter-pod affinity, which means favoring pods that have an affinity or preference for running on the same node.
- Apply the "Least requested" priority, which favors nodes with fewer requested resources. This policy aims to distribute pods across all nodes in the cluster.
- Apply the "Most requested" priority, which favors nodes with the highest requested resources. This policy tends to pack pods into a smaller set of nodes.
- Use the "Requested to capacity ratio" priority, which calculates a priority based on the ratio of requested resources to the node's capacity. It uses a default resource scoring function shape.
- Prioritize nodes with balanced resource allocation, favoring nodes with balanced resource usage.
- Utilize the "Node prefer avoid pods" priority, which prioritizes nodes based on the node annotation `scheduler.alpha.kubernetes.io/preferAvoidPods`. This annotation is used to indicate that two different pods should not run on the same node.
- Apply node affinity priority, giving preference to nodes based on the node affinity scheduling preferences specified in `PreferredDuringSchedulingIgnoredDuringExecution`.
- Consider taint toleration priority, preparing a priority list for all nodes based on the number of intolerable taints on each node. This policy adjusts a node's rank, taking taints into account.
- Give priority to nodes that already have the container images required by the pod using the "Image locality" priority.
- Prioritize spreading pods backing a service across different nodes with the "Service spreading" priority.
- Apply pod anti-affinity, which means avoiding running pods on nodes that already have similar pods based on anti-affinity rules.
- Use the "Equal priority map," where all nodes have the same weight and there are no favorites or biases.

Check out `https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/` for more details.

As you can see, the default scheduler is very sophisticated and can be configured in a very fine-grained way to accommodate most of your needs. But, under some circumstances, it might not be the best choice.

In particular, in large clusters with many nodes (hundreds or thousands), every time a pod is scheduled, all the nodes need to go through this rigorous and heavyweight procedure of filtering and scoring. Now, consider a situation where you need to schedule a large number of pods at once (e.g., training machine learning models). This can put a lot of pressure on your cluster and lead to performance issues.

Kubernetes can make the filtering and scoring process more lightweight by allowing you to filter and score only some of the pods, but still, you may want better control.

Fortunately, Kubernetes allows you to influence the scheduling process in several ways. Those ways include:

- Direct scheduling of pods to nodes
- Replacing the scheduler with your own scheduler
- Extending the scheduler with additional filters
- Adding another scheduler that runs side by side with the default scheduler

Let's review various methods you can use to influence pod scheduling.

Scheduling pods manually

Guess what? We can just tell Kubernetes where to place our pod when we create the pod. All it takes is to specify a node name in the pod's spec and the scheduler will ignore it. If you think about the loosely coupled nature of the controller pattern, it all makes sense. The scheduler is watching for pending pods that DON'T have a node name assigned yet. If you are passing the node name yourself, then the Kubelet on the target node, which watches for pending pods that DO have a node name, will just go ahead and make sure to create a new pod.

Let's look at the nodes of our k3d cluster:

```
$ k get no
NAME                        STATUS   ROLES                  AGE    VERSION
k3d-k3s-default-agent-1     Ready    <none>                 155d   v1.23.6+k3s1
k3d-k3s-default-server-0     Ready    control-plane,master   155d   v1.23.6+k3s1
k3d-k3s-default-agent-0     Ready    <none>                 155d   v1.23.6+k3s1```
```

Here is a pod with a pre-defined node name, `k3d-k3s-default-agent-1`:

```
apiVersion: v1
kind: Pod
metadata:
    name: some-pod-manual-scheduling
spec:
  containers:
    - name: some-container
      image: registry.k8s.io/pause:3.8
  nodeName: k3d-k3s-default-agent-1
  schedulerName: no-such-scheduler
```

Let's create the pod and see that it was indeed scheduled to the `k3d-k3s-default-agent-1` node as requested:

```
$ k create -f some-pod-manual-scheduling.yaml
pod/some-pod-manual-scheduling created

$ k get po some-pod-manual-scheduling -o wide
NAME                          READY   STATUS    RESTARTS   AGE   IP            NODE
NOMINATED NODE   READINESS GATES
some-pod-manual-scheduling   1/1     Running   0          26s   10.42.2.213   k3d-
k3s-default-agent-1   <none>           <none>
```

Direct scheduling is also useful for troubleshooting when you want to schedule a temporary pod to a tainted node without mucking around with adding tolerations.

Let's create our own custom scheduler now.

Preparing our own scheduler

Our scheduler will be super simple. It will just schedule all pending pods that request to be scheduled by the `custom-scheduler` to the node `k3d-k3s-default-agent-0`. Here is a Python implementation that uses the `kubernetes` client package:

```
from kubernetes import client, config, watch


def schedule_pod(cli, name):
    target = client.V1ObjectReference()
    target.kind = 'Node'
    target.apiVersion = 'v1'
    target.name = 'k3d-k3s-default-agent-0'
    meta = client.V1ObjectMeta()
    meta.name = name
    body = client.V1Binding(metadata=meta, target=target)
    return cli.create_namespaced_binding('default', body)


def main():
    config.load_kube_config()
    cli = client.CoreV1Api()
    w = watch.Watch()
    for event in w.stream(cli.list_namespaced_pod, 'default'):
        o = event['object']
        if o.status.phase != 'Pending' or o.spec.scheduler_name != 'custom-
scheduler':
```

```
            continue

        schedule_pod(cli, o.metadata.name)


if __name__ == '__main__':
    main()
```

If you want to run a custom scheduler long term, then you should deploy it into the cluster just like any other workload as a deployment. But, if you just want to play around with it, or you're still developing your custom scheduler logic, you can run it locally as long as it has the correct credentials to access the cluster and has permissions to watch for pending pods and update their node name.

Note that I strongly recommend building production custom schedulers on top of the scheduling framework (`https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/`).

Assigning pods to the custom scheduler

OK. We have a custom scheduler that we can run alongside the default scheduler. But how does Kubernetes choose which scheduler to use to schedule a pod when there are multiple schedulers?

The answer is that Kubernetes doesn't care. The pod can specify which scheduler it wants to schedule it. The default scheduler will schedule any pod that doesn't specify the schedule or that specifies explicitly `default-scheduler`. Other custom schedulers should be responsible and only schedule pods that request them. If multiple schedulers try to schedule the same pod, we will probably end up with multiple copies or naming conflicts.

For example, our simple custom scheduler is looking for pending pods that specify a scheduler name of `custom-scheduler`. All other pods will be ignored by it:

```
if o.status.phase != 'Pending' or o.spec.scheduler_name != 'custom-scheduler':
    continue
```

Here is a pod spec that specifies `custom-scheduler`:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod-with-custom-scheduler
spec:
  containers:
    - name: some-container
      image: registry.k8s.io/pause:3.8
  schedulerName: custom-scheduler
```

What happens if our custom scheduler is not running and we try to create this pod?

```
$ k create -f some-pod-with-custom-scheduler.yaml
```

```
pod/some-pod-with-custom-scheduler created

$ k get po
NAME                              READY   STATUS    RESTARTS   AGE
some-pod-manual-scheduling        1/1     Running   0          9m33s
some-pod-with-custom-scheduler    0/1     Pending   0          14s
```

The pod is created just fine (meaning the Kubernetes API server stored it in etcd), but it is pending, which means it wasn't scheduled yet. Since it specified an explicit scheduler, the default scheduler ignores it.

But, if we run our scheduler... it will immediately get scheduled:

```
python custom_scheduler.py
Waiting for pending pods...
Scheduling pod: some-pod-with-custom-scheduler
```

Now, we can see that the pod was assigned to a node, and it is in a running state:

```
$ k get po -o wide
NAME                              READY   STATUS    RESTARTS   AGE    IP
NODE                        NOMINATED NODE   READINESS GATES
some-pod-manual-scheduling        1/1     Running   0          4h5m   10.42.2.213
k3d-k3s-default-agent-1   <none>           <none>
some-pod-with-custom-scheduler    1/1     Running   0          87s    10.42.0.125
k3d-k3s-default-agent-0   <none>           <none>
```

That was a deep dive into scheduling and custom schedulers. Let's check out kubectl plugins.

Writing kubectl plugins

Kubectl is the workhorse of the aspiring Kubernetes developer and admin. There are now very good visual tools like k9s (`https://github.com/derailed/k9s`), octant (`https://github.com/vmware-tanzu/octant`), and Lens Desktop (`https://k8slens.dev`). But, for many engineers, kubectl is the most complete way to work interactively with your cluster, as well to participate in automation workflows.

Kubectl encompasses an impressive list of capabilities, but you will often need to string together multiple commands or a long chain of parameters to accomplish some tasks. You may also want to run some additional tools installed in your cluster.

You can package such functionality as scripts or containers, or any other way, but then you'll run into the issue of where to place them, how to discover them, and how to manage them. Kubectl plugins give you a one-stop shop for those extended capabilities. For example, recently I needed to periodically list and move around files on an SFTP server managed by a containerized application running on a Kubernetes cluster. I quickly wrote a few kubectl plugins that took advantage of my KUBECONFIG credentials to get access to secrets in the cluster that contained the credentials to access the SFTP server and then implemented a lot of application-specific logic for accessing and managing those SFTP directories and files.

Understanding kubectl plugins

Until Kubernetes 1.12, kubectl plugins required a dedicated YAML file where you specified various metadata and other files that implemented the functionality. In Kubernetes 1.12, kubectl started using the Git extension model where any executable on your path with the prefix `kubectl-` is treated as a plugin.

Kubectl provides the `kubectl plugins list` command to list all your current plugins. This model was very successful with Git and it is extremely simple now to add your own kubectl plugins.

If you add an executable called `kubectl-foo`, then you can run it via `kubectl foo`. You can have nested commands too. Add `kubectl-foo-bar` to your path and run it via `kubectl foo bar`. If you want to use dashes in your commands, then in your executable, use underscores. For example, the executable `kubectl-do_stuff` can be run using `kubectl do-stuff`.

The executable itself can be implemented in any language, have its own command-line arguments and flags, and display its own usage and help information.

Managing kubectl plugins with Krew

The lightweight plugin model is great for writing your own plugins, but what if you want to share your plugins with the community? Krew (`https://github.com/kubernetes-sigs/krew`) is a package manager for kubectl plugins that lets you discover, install, and manage curated plugins.

You can install Krew with Brew on Mac or follow the installation instructions for other platforms. Krew is itself a kubectl plugin as its executable is `kubectl-krew`. This means you can either run it directly with `kubectl-krew` or through kubectl `kubectl krew`. If you have a `k` alias for `kubectl`, you would probably prefer the latter:

```
$ k krew
krew is the kubectl plugin manager.
You can invoke krew through kubectl: "kubectl krew [command]..."

Usage:
  kubectl krew [command]

Available Commands:
  completion  generate the autocompletion script for the specified shell
  help        Help about any command
  index       Manage custom plugin indexes
  info        Show information about an available plugin
  install     Install kubectl plugins
  list        List installed kubectl plugins
  search      Discover kubectl plugins
  uninstall   Uninstall plugins
  update      Update the local copy of the plugin index
  upgrade     Upgrade installed plugins to newer versions
```

```
  version     Show krew version and diagnostics

Flags:
  -h, --help      help for krew
  -v, --v Level   number for the log level verbosity

Use "kubectl krew [command] --help" for more information about a command.
```

Note that the `krew list` command shows only Krew-managed plugins and not all kubectl plugins. It doesn't even show itself.

I recommend that you check out the available plugins. Some of them are very useful, and they may inspire you to write your own plugins. Let's see how easy it is to write our own plugin.

Creating your own kubectl plugin

Kubectl plugins can range from super simple to very complicated. I work a lot these days with AKS node pools created using the Cluster API and CAPZ (the Cluster API provider for Azure). I'm often interested in viewing all the node pools on a specific cloud provider. All the node pools are defined as custom resources in a namespace called `cluster-registry`. The following kubectl command lists all the node pools:

```
$ k get -n cluster-registry azuremanagedmachinepools.infrastructure.cluster.x-k8s.io
aks-centralus-cluster-001-nodepool001         116d
aks-centralus-cluster-001-nodepool002         116d
aks-centralus-cluster-002-nodepool001         139d
aks-centralus-cluster-002-nodepool002         139d
aks-centralus-cluster-002-nodepool003         139d
...
```

This is not a lot of information. I'm interested in information like the SKU (VM type and size) of each node pool, its Kubernetes version, and the number of nodes in each node pool. The following kubectl command can provide this information:

```
$ k get -n cluster-registry azuremanagedmachinepools.infrastructure.cluster.x-k8s.io
-o custom-columns=NAME:.metadata.name,SKU:.spec.sku,VERSION:.status.version,NODES:.
status.replicas

NAME                                    SKU                VERSION   NODES
aks-centralus-cluster-001-nodepool001   Standard_D4s_v4    1.23.8    10
aks-centralus-cluster-001-nodepool002   Standard_D8s_v4    1.23.8    20
aks-centralus-cluster-002-nodepool001   Standard_D16s_v4   1.23.8    30
aks-centralus-cluster-002-nodepool002   Standard_D8ads_v5  1.23.8    40
aks-centralus-cluster-002-nodepool003   Standard_D8ads_v5  1.23.8    50
```

However, this is a lot to type. I simply put this command in a file called `kubectl-npa-get` and stored it in `/usr/local/bin`. Now, I can invoke it just by calling `k npa get`. I could define a little alias or shell function, but a kubectl plugin is more appropriate as it is a central place for all kubectl-related enhancements. It enforces a uniform convention and it is discoverable via `kubectl list plugins`.

This was an example of an almost trivial kubectl plugin. Let's look at a more complicated example – deleting namespaces. It turns out that reliably deleting namespaces in Kubernetes is far from trivial. Under certain conditions, a namespace can be stuck forever in a terminating state after you try to delete it. I created a little Go program to reliably delete namespaces. You can check it out here: `https://github.com/the-gigi/k8s-namespace-deleter`.

This is a perfect use case for a kubectl plugin. The instructions in the README recommend building the executable and then saving it in your path as `kubectl-ns-delete`. Now, when you want to delete a namespace, you can just use `k ns delete <namespace>` to invoke `k8s-namespace-deleter` and reliably get rid of your namespace.

If you want to develop plugins and share them on Krew, there is a more rigorous process there. I highly recommend developing the plugin in Go and taking advantage of projects like `cli-runtime` (`https://github.com/kubernetes/cli-runtime/`) and `krew-plugin-template` (`https://github.com/replicatedhq/krew-plugin-template`).

Kubectl plugins are awesome, but there are some gotchas you should be aware of. I ran into some of these issues when working with kubectl plugins.

Don't forget your shebangs!

If you don't specify a shebang for your shell-based executables, you will get an obscure error message:

```
$ k npa get
Error: exec format error
```

Naming your plugin

Choosing a name for your plugin is not easy. Luckily, there are some good guidelines: `https://krew.sigs.k8s.io/docs/developer-guide/develop/naming-guide`.

Those naming guidelines are not just for Krew plugins, but make sense for any kubectl plugin.

Overriding existing kubectl commands

I originally named the plugin `kubectl-get-npa`. In theory, kubectl should try to match the longest plugin name to resolve ambiguities. But, apparently, it doesn't work with built-in commands like `kubectl get`. This is the error I got:

```
$ k get npa
error: the server doesn't have a resource type "npa"
```

Renaming the plugin to `kubectl-npa-get` solved the problem.

Flat namespace for Krew plugins

The space of kubectl plugins is flat. If you choose a generic plugin name like `kubectl-login`, you'll have a lot of problems. Even if you qualify it with something like `kubectl-gcp-login`, you might conflict with some other plugin. This is a scalability problem. I think the solution should involve a strong naming scheme for plugins like DNS and the ability to define short names and aliases for convenience.

We have covered kubectl plugins, how to write them, and how to use them. Let's take a look at extending access control with webhooks.

Employing access control webhooks

Kubernetes provides several ways for you to customize access control. In Kubernetes, access control can be denoted with triple-A: Authentication, Authorization, and Admission control. In early versions, access control happened through plugins that required Go programming, installing them into your cluster, registration, and other invasive procedures. Now, Kubernetes lets you customize authentication, authorization, and admission control via web hooks. Here is the access control workflow:

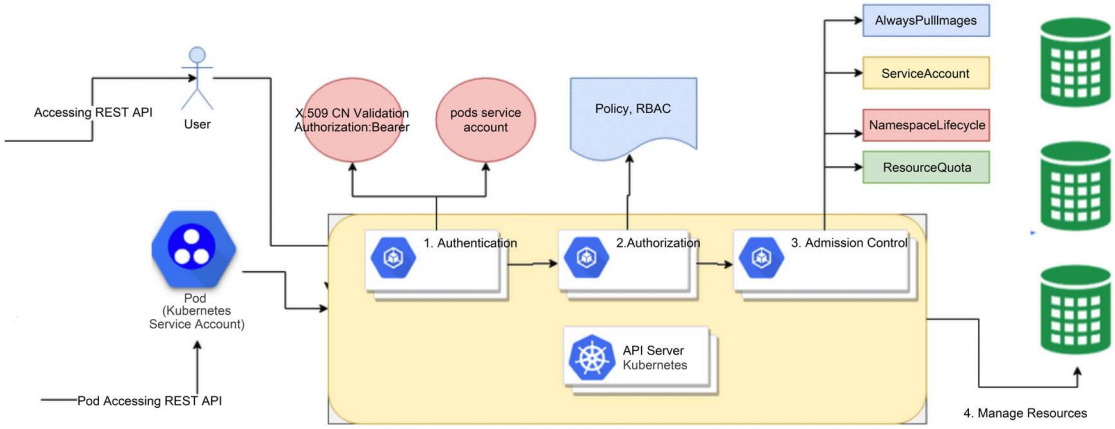


Figure 15.6: Access control workflow

Using an authentication webhook

Kubernetes lets you extend the authentication process by injecting a webhook for bearer tokens. It requires two pieces of information: how to access the remote authentication service and the duration of the authentication decision (it defaults to two minutes).

To provide this information and enable authentication webhooks, start the API server with the following command-line arguments:

```
- --authentication-token-webhook-config-file=<authentication config file>
- --authentication-token-webhook-cache-ttl (how long to cache auth decisions, default to 2 minutes)
```

The configuration file uses the kubeconfig file format. Here is an example:

```
# Kubernetes API version
apiVersion: v1
# kind of the API object
kind: Config
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-authn-service
    cluster:
      certificate-authority: /path/to/ca.pem         # CA for verifying the
remote service.
      server: https://authn.example.com/authenticate # URL of remote service to
query. Must use 'https'.

# users refers to the API server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: /path/to/cert.pem # cert for the webhook plugin to
use
      client-key: /path/to/key.pem          # key matching the cert

# kubeconfig files require a context. Provide one for the API server.
current-context: webhook
contexts:
- context:
    cluster: name-of-remote-authn-service
    user: name-of-api-sever
  name: webhook
```

Note that a client certificate and key must be provided to Kubernetes for mutual authentication against the remote authentication service.

The cache TTL is useful because often users will make multiple consecutive requests to Kubernetes. Having the authentication decision cached can save a lot of round trips to the remote authentication service.

When an API HTTP request comes in, Kubernetes extracts the bearer token from its headers and posts a `TokenReview` JSON request to the remote authentication service via the webhook:

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "TokenReview",
  "spec": {
```

```
    "token": "<bearer token from original request headers>"
  }
}
```

The remote authentication service will respond with a decision. The status authentication will either be `true` or `false`. Here is an example of a successful authentication:

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "TokenReview",
  "status": {
    "authenticated": true,
    "user": {
      "username": "gigi@gg.com",
      "uid": "42",
      "groups": [
        "developers",
      ],
      "extra": {
        "extrafield1": [
          "extravalue1",
          "extravalue2"
        ]
      }
    }
  }
}
```

A rejected response is much more concise:

```
{
  "apiVersion": "authentication.k8s.io/v1",
  "kind": "TokenReview",
  "status": {
    "authenticated": false
  }
}
```

Using an authorization webhook

The authorization webhook is very similar to the authentication webhook. It requires just a configuration file, which is in the same format as the authentication webhook configuration file. There is no authorization caching because, unlike authentication, the same user may make lots of requests to different API endpoints with different parameters, and authorization decisions may be different, so caching is not a viable option.

You configure the webhook by passing the following command-line argument to the API server:

```
--authorization-webhook-config-file=<configuration filename>
```

When a request passes authentication, Kubernetes will send a `SubjectAccessReview` JSON object to the remote authorization service. It will contain the requesting user (and any user groups it belongs to) and other attributes such as the requested API group, `namespace`, `resource`, and `verb`:

```
{
  "apiVersion": "authorization.k8s.io/v1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "awesome-namespace",
      "verb": "get",
      "group": "awesome.example.org",
      "resource": "pods"
    },
    "user": "gigi@gg.com",
    "group": [
      "group1",
      "group2"
    ]
  }
}
```

The request will either be allowed:

```
{
  "apiVersion": "authorization.k8s.io/v1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": true
  }
}
```

Or denied with a reason:

```
{

  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": false,
    "reason": "user does not have read access to the namespace"
```

```
    }
  }
```

A user may be authorized to access a resource, but not some non-resource attributes, such as `/api`, `/apis`, `/metrics`, `/resetMetrics`, `/logs`, `/debug`, `/healthz`, `/swagger-ui/`, `/swaggerapi/`, `/ui`, and `/version`.

Here is how to request access to the logs:

```
{
  "apiVersion": "authorization.k8s.io/v1",
  "kind": "SubjectAccessReview",
  "spec": {
    "nonResourceAttributes": {
      "path": "/logs",
      "verb": "get"
    },
    "user": "gigi@gg.com",
    "group": [
      "group1",
      "group2"
    ]
  }
}
```

We can check, using kubectl, if we are authorized to perform an operation using the `can-i` command. For example, let's see if we can create deployments:

```
$ k auth can-i create deployments
yes
```

We can also check if other users or service accounts are authorized to do something. The default service account is NOT allowed to create deployments:

```
$ k auth can-i create deployments --as default
no
```

Using an admission control webhook

Dynamic admission control supports webhooks too. It has been generally available since Kubernetes 1.16. Depending on your Kubernetes version, you may need to enable the `MutatingAdmissionWebhook` and `ValidatingAdmissionWebhook` admission controllers using `--enable-admission-plugins=Mutating,ValidatingAdmissionWebhook` flags to `kube-apiserver`.

There are several other admission controllers that the Kubernetes developers recommend running (the order matters):

`--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,DefaultTolerationSeconds,MutatingAdmissionWebhook,ValidatingAdmissionWebhook,ResourceQuota`

In Kubernetes 1.25, these plugins are enabled by default.

Configuring a webhook admission controller on the fly

Authentication and authorization webhooks must be configured when you start the API server. Admission control webhooks can be configured dynamically by creating `MutatingWebhookConfiguration` or `ValidatingWebhookConfiguration` API objects. Here is an example:

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
...
webhooks:
- name: admission-webhook.example.com
  rules:
  - operations: ["CREATE", "UPDATE"]
    apiGroups: ["apps"]
    apiVersions: ["v1", "v1beta1"]
    resources: ["deployments", "replicasets"]
    scope: "Namespaced"
  ...
```

An admission server accesses `AdmissionReview` requests such as:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "request": {
    "uid": "705ab4f5-6393-11e8-b7cc-42010a800002",
    "kind": {"group":"autoscaling","version":"v1","kind":"Scale"},
    "resource": {"group":"apps","version":"v1","resource":"deployments"},
    "subResource": "scale",
    "requestKind": {"group":"autoscaling","version":"v1","kind":"Scale"},
    "requestResource": {"group":"apps","version":"v1","resource":"deployments"},
    "requestSubResource": "scale",
    "name": "cool-deployment",
    "namespace": "cool-namespace",
    "operation": "UPDATE",
    "userInfo": {
```

```
      "username": "admin",
      "uid": "014fbff9a07c",
      "groups": ["system:authenticated","my-admin-group"],
      "extra": {
        "some-key":["some-value1", "some-value2"]
      }
    },

    "object": {"apiVersion":"autoscaling/v1","kind":"Scale",...},
    "oldObject": {"apiVersion":"autoscaling/v1","kind":"Scale",...},
    "options": {"apiVersion":"meta.k8s.io/v1","kind":"UpdateOptions",...},
    "dryRun": false
  }
}
```

If the request is admitted, the response will be:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": true
  }
}
```

If the request is not admitted, then `allowed` will be `False`. The admission server may provide a `status` section too with an HTTP status code and message:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": false,
    "status": {
      "code": 403,
      "message": "You cannot do this because I say so!!!!"
    }
  }
}
```

That concludes our discussion of dynamic admission control. Let's look at some more extension points.

Additional extension points

There are some additional extension points that don't fit into the categories we have discussed so far.

Providing custom metrics for horizontal pod autoscaling

Prior to Kubernetes 1.6, custom metrics were implemented as a Heapster model. In Kubernetes 1.6, new custom metrics APIs landed and matured gradually. As of Kubernetes 1.9, they are enabled by default. As you may recall, Keda (`https://keda.sh`) is a project that focuses on custom metrics for autoscaling. However, if for some reason Keda doesn't meet your needs, you can implement your own custom metrics. Custom metrics rely on API aggregation. The recommended path is to start with the custom metrics API server boilerplate, available here: `https://github.com/kubernetes-sigs/custom-metrics-apiserver`.

Then, you can implement the `CustomMetricsProvider` interface:

```
type CustomMetricsProvider interface {
    // GetRootScopedMetricByName fetches a particular metric for a particular
root-scoped object.
    GetRootScopedMetricByName(groupResource schema.GroupResource, name string,
metricName string) (*custom_metrics.MetricValue, error)

    // GetRootScopedMetricByName fetches a particular metric for a set of root-
scoped objects
    // matching the given label selector.
    GetRootScopedMetricBySelector(groupResource schema.GroupResource, selector
labels.Selector, metricName string) (*custom_metrics.MetricValueList, error)

    // GetNamespacedMetricByName fetches a particular metric for a particular
namespaced object.
    GetNamespacedMetricByName(groupResource schema.GroupResource, namespace
string, name string, metricName string) (*custom_metrics.MetricValue, error)

    // GetNamespacedMetricByName fetches a particular metric for a set of
namespaced objects
    // matching the given label selector.
    GetNamespacedMetricBySelector(groupResource schema.GroupResource, namespace
string, selector labels.Selector, metricName string) (*custom_metrics.
MetricValueList, error)

    // ListAllMetrics provides a list of all available metrics at
    // the current time. Note that this is not allowed to return
    // an error, so it is recommended that implementors cache and
    // periodically update this list, instead of querying every time.
    ListAllMetrics() []CustomMetricInfo
}
```

Extending Kubernetes with custom storage

Volume plugins are yet another type of plugin. Prior to Kubernetes 1.8, you had to write a kubelet plugin, which required registration with Kubernetes and linking with the kubelet. Kubernetes 1.8 introduced the FlexVolume, which is much more versatile. Kubernetes 1.9 took it to the next level with the CSI, which we covered in *Chapter 6*, *Managing Storage*. At this point, if you need to write storage plugins, the CSI is the way to go. Since the CSI uses the gRPC protocol, the CSI plugin must implement the following gRPC interface:

```
service Controller {
  rpc CreateVolume (CreateVolumeRequest)
    returns (CreateVolumeResponse) {}

  rpc DeleteVolume (DeleteVolumeRequest)
    returns (DeleteVolumeResponse) {}

  rpc ControllerPublishVolume (ControllerPublishVolumeRequest)
    returns (ControllerPublishVolumeResponse) {}

  rpc ControllerUnpublishVolume (ControllerUnpublishVolumeRequest)
    returns (ControllerUnpublishVolumeResponse) {}

  rpc ValidateVolumeCapabilities (ValidateVolumeCapabilitiesRequest)
    returns (ValidateVolumeCapabilitiesResponse) {}

  rpc ListVolumes (ListVolumesRequest)
    returns (ListVolumesResponse) {}

  rpc GetCapacity (GetCapacityRequest)
    returns (GetCapacityResponse) {}

  rpc ControllerGetCapabilities (ControllerGetCapabilitiesRequest)
    returns (ControllerGetCapabilitiesResponse) {}
}
```

This is not a trivial undertaking and typically only storage solution providers should implement CSI plugins.

The additional extension points of custom metrics and custom storage solutions demonstrate the commitment of Kubernetes to being truly extensible and allowing its users to customize almost every aspect of its operation.

Summary

In this chapter, we covered three major topics: working with the Kubernetes API, extending the Kubernetes API, and writing Kubernetes plugins. The Kubernetes API supports the OpenAPI spec and is a great example of REST API design that follows all current best practices. It is very consistent, well organized, and well documented. Yet it is a big API and not easy to understand. You can access the API directly via REST over HTTP, using client libraries including the official Python client, and even by invoking kubectl programmatically.

Extending the Kubernetes API may involve defining your own custom resources, writing controllers/operators, and optionally extending the API server itself via API aggregation.

Plugins and webhooks are a foundation of Kubernetes design. Kubernetes was always meant to be extended by users to accommodate any needs. We looked at various plugins, such as custom schedulers, kubectl plugins, and access control webhooks. It is very cool that Kubernetes provides such a seamless experience for writing, registering, and integrating all those plugins.

We also looked at custom metrics and even how to extend Kubernetes with custom storage options.

At this point, you should be well aware of all the major mechanisms to extend, customize, and control Kubernetes via API access, custom resources, controllers, operators, and custom plugins. You are in a great position to take advantage of these capabilities to augment the existing functionality of Kubernetes and adapt it to your needs and your systems.

In the next chapter, we will look at governing Kubernetes via policy engines. This will continue the theme of extending Kubernetes as policy engines are dynamic admission controllers on steroids. We will cover what governance is all about, review existing policy engines, and dive deep into Kyverno, which is the best policy engine for Kubernetes in my opinion.

16

Governing Kubernetes

In the previous chapter, we discussed at length different ways to extend Kubernetes, including validating and mutating requests during the admission control phase.

In this chapter, we will learn about the growing role of Kubernetes in large enterprise organizations, what governance is, and how it is applied in Kubernetes. We will look at policy engines, review some popular ones, and then dive deep into Kyverno.

This ties in nicely with the previous chapter because policy engines are built on top of the Kubernetes admission control mechanism.

More and more enterprise organizations put more and more of their proverbial eggs in the Kubernetes basket. These large organizations have severe security, compliance, and governance needs. Kubernetes policy engines are here to address these concerns and make sure that enterprise organizations can fully embrace Kubernetes.

Here are the topics we will cover:

- Kubernetes in the enterprise
- What is Kubernetes governance?
- Policy engines
- A Kyverno deep dive

Let's jump right in and understand the growing role and importance of Kubernetes in the enterprise.

Kubernetes in the enterprise

The journey and adoption rate of the Kubernetes platform are unprecedented. It launched officially in 2016, and in a few years it has conquered the world of infrastructure. 96% of organizations that participated in the most recent CNCF survey are using or evaluating Kubernetes. The penetration of Kubernetes is across multiple dimensions: organization size, geographical location, and production and no-production environments. What is even more impressive is that Kubernetes can go under the hood and become the foundation that other technologies and platforms are built on.

You can see this in its widespread adoption by all the cloud providers that offer various flavors of managed Kubernetes, as well as with the hosted platform-as-a-service offerings from many vendors. Check out the CNCF-certified Kubernetes software conformance list: `https://www.cncf.io/certification/software-conformance`.

Having a variety of certified vendors and value-added resellers, an ecosystem of multiple companies, etc. is extremely important for enterprise organizations. Enterprise organizations need much more than the latest shiny technology. The stakes are high, the failure rate of large infrastructure projects is high, and the consequences of failure are harsh. Combine all these factors, and the result is that enterprise organizations are very change-resistent and risk-averse when it comes to their technology. A lot of critical software systems in diverse fields like traffic control, insurance, healthcare, communication systems, and airlines are still running on software that was written 40–50 years ago, using languages like COBOL and Fortran.

Requirements of enterprise software

Let's look at some requirements of enterprise software:

- Handling large amounts of data
- Integrating with other systems and applications
- Providing robust security features
- Being scalable and available
- Being flexible and customizable
- Being compliant
- Having support from trusted vendors
- Having strong governance (much more on that later)

How does Kubernetes fit the bill?

Kubernetes and enterprise software

The reason Kubernetes usage has grown so much in the enterprise software area is that it actually ticks all the boxes and keeps improving.

As the de facto standard for container orchestration platforms, it can serve as the foundation for all container-based deployment. Its ecosystem satisfies any integration needs, as every vendor must be able to run on Kubernetes. The long-term prospects for Kubernetes are extremely high, as it is a true team effort from many companies and organizations, and it is steered by an open and successful process that keeps delivering. Kubernetes spearheads the shift toward multi-cloud and hybrid-cloud deployments following industry-wide standards

The extensibility and flexibility of Kubernetes mean it can cater to any type of customization a particular enterprise will need.

It is truly a remarkable project that is designed on solid conceptual architecture and is able to deliver results consistently in the real world.

At this point it's clear that Kubernetes is great for enterprise organizations, but how does it address the need for governance?

What is Kubernetes governance?

Governance is one of the important requirements for enterprise organizations. In a nutshell, it means controlling the way an organization operates. Some elements of governance are:

- Policies
- Ethics
- Processes
- Risk management
- Administration

Governance includes a way to specify policies and mechanisms to enforce the policies, as well as reporting and auditing. Let's look at various areas and practices of governance in Kubernetes.

Image management

Containers run software baked into images. Managing these images is a critical activity in operating a Kubernetes-based system. There are several dimensions to consider: how do you bake your images? How do you vet third-party images? Where do you store your images? Making poor choices here can impact the performance of your system (for example, if you use large bloated base images) and crucially the security of your system (for example, if you use compromised or vulnerable base images). Image management policies can force image scanning or ensure that you can only use vetted images from specific image registries.

Pod security

The unit of work of Kubernetes is the pod. There are many security settings you can set for a pod and its containers. The default security settings are unfortunately very lax. Validating and enforcing pod security policies can remediate this. Kubernetes has strong support and guidance for pod security standards as well as several built-in profiles. Each pod has a security context as we discussed in *Chapter 4, Securing Kubernetes*.

See `https://kubernetes.io/docs/concepts/security/pod-security-standards/` for more details.

Network policy

Kubernetes network policies control traffic flow between pods and other network entities at layers 3 and 4 of the OSI network model (IP addresses and ports). The network entities may be pods that have a specific set of labels or all pods in a namespace with a specific set of labels. Finally, a network policy can also block pod access to/from a specific IP block.

In the context of governance, network policies can be used to enforce compliance with security and regulatory requirements by controlling network access and communication between pods and other resources.

For example, network policies can be used to prevent pods from communicating with certain external networks. Network policies can also be used to enforce the separation of duties and prevent unauthorized access to sensitive resources within a cluster.

See `https://kubernetes.io/docs/concepts/services-networking/network-policies/` for more details.

Configuration constraints

Kubernetes is very flexible and provides a lot of controls for many aspects of its operation. The DevOps practices often used in a Kubernetes-based system allow teams a lot of control over how their workloads are deployed, how they scale, and what resources they use. Kubernetes provides configuration constraints like quotas and limits. With more advanced admission controllers you can validate and enforce policies that control any aspect of resource creation, such as the maximum size of an auto-scaling deployment, the total amount of persistent volume claims, and requiring that memory requests always equal memory limits (not necessarily a good idea).

RBAC and admission control

Kubernetes **RBAC** (**Role-Based Access Control**) operates at the resource and verb level. Every Kubernetes resource has operations (verbs) that can be performed against it. With RBAC you define roles that are sets of permissions over resources, which you can apply at the namespace level or cluster level. It is a bit of a coarse-grained tool, but it is very convenient, especially if you segregate your resources at the namespace level and use cluster-level permissions only to manage workloads that operate across the entire cluster.

If you need something more granular that depends on specific attributes of resources, then admission controllers can handle it. We will explore this option later in the chapter when discussing policy engines.

Policy management

Governance is built around policies. Managing all these policies, organizing them, and ensuring they address the governance needs of an organization takes a lot of effort and is an ongoing task. Be prepared to devote resources to evolving and maintaining your policies.

Policy validation and enforcement

Once a set of policies are in place, you need to validate requests to the Kubernetes API server against those policies and reject requests that violate these policies. There is another approach to enforcing policies that involves mutating incoming requests to comply with a policy. For example, if a policy requires that each pod must have a memory request of at most 2 GiB, then a mutating policy can trim down the memory request of pods with larger memory requests to 2 GiB.

Polices don't have to be rigid. Exceptions and exclusions can be made for special cases.

Reporting

When you manage a large number of policies and vet all requests it's important to have visibility into how your policies can help you govern your system, prevent issues, and learn from usage patterns. Reports can provide insights by capturing and consolidating the results of policy decisions. As a human user you may view reports about policy violations and rejected and mutated requests, and detect trends or anomalies. At a higher level you can employ automated analysis, including an ML-based model to extract meaning from a large number of detailed reports.

Audit

Kubernetes audit logs provide a timestamped play-by-play of every event in the system. When you couple audit data with governance reports you can piece together the timeline of incidents, especially security incidents, where the culprit can be identified by combining data from multiple sources, starting with a policy violation and ending with a root cause.

So far, we have covered the terrain of what governance is and how it specifically relates to Kubernetes. We emphasized the importance of policies to govern your system. Let's look at policy engines and how they implement these concepts.

Policy engines

Policy engines in Kubernetes provide comprehensive coverage of governance needs and complement built-in mechanisms, like network policies and RBAC. Policy engines can verify and ensure that your system utilizes best practices, follows security guidelines, and complies with external policies. In this section, we will look at admission control as the primary mechanism where policy engines hook into the system, the responsibilities of a policy engine, and a review of existing policy engines. After this, we will then dive deep into one of the best policy engines out there – Kyverno.

Admission control as the foundation of policy engines

Admission control is part of the life cycle of requests hitting the Kubernetes API server. We discussed it in depth in *Chapter 15*, *Extending Kubernetes*. As you recall, dynamic admission controllers are webhook servers that listen for admission review requests and accept, deny, or mutate them. Policy engines are first and foremost sophisticated admission controllers that register to listen for all requests that are relevant to their policies.

When a request comes in, the policy engine will apply all relevant policies to decide the fate of the request. For example, if a policy determines that Kubernetes services of the `LoadBalancer` type may be created only in a namespace called `load_balancer`, then the policy engine will register to listen for all Kubernetes service creation and update requests. When a service creation or update request arrives, the policy engine will check the type of the service and its namespace. If the service type is `LoadBalancer` and the namespace is not `load_balancer`, then the policy engine will reject the request. Note that this is something that can't be done using RBAC. This is because RBAC can't look at the type of service to determine if the request is valid or not.

Now that we understand how the policy engine utilizes the dynamic admission control process of Kubernetes, let's look at the responsibilities of a policy engine.

Responsibilities of a policy engine

The policy engine is the primary tool for applying governance to a Kubernetes-based system. The policy engine should allow the administrators to define policies that go above and beyond the built-in Kubernetes policies, like RBAC and network policies. That often means coming up with a policy declaration language. The policy declaration language needs to be rich enough to cover all the nuances of Kubernetes, including fine-grained application to different resources and access to all the relevant information to base accept or reject decisions on for each resource.

The policy engine should also provide a way to organize, view, and manage policies. Ideally, the policy engine provides a good way to test policies before applying them to a live cluster.

The policy engine has to provide a way to deploy policies to the cluster, and of course, it needs to apply the policies that are relevant for each request and decide if the request should be accepted as is, rejected, or modified (mutated). A policy engine may provide a way to generate additional resources when a request comes in. For example, when a new Kubernetes deployment is created, a policy engine may automatically generate a Horizontal Pod Autoscaler for the deployment. A policy engine may also listen to events that occur in the cluster and take action. Note that this capability goes beyond dynamic admission control, but it still enforces policies on the cluster.

Let's review some Kubernetes policy engines and how they fulfill these responsibilities.

Quick review of open source policy engines

When evaluating solutions, it's very helpful to come up with evaluation criteria up front, since policy engines can deeply impact the operation of your Kubernetes cluster and its workloads' maturity is a key element. Excellent documentation is crucial too, since the surface area of a policy engine is very large and you need to understand how to work with it. The capabilities of a policy engine determine what use cases it can handle. Writing policies is how administrators convey their governance intentions to the policy engine. It's important to evaluate the user experience of writing and testing policies and what tooling is available to support these activities. Deploying the policies to the cluster is another must-have element. Finally, viewing reports and understanding the state of governance can be neglected.

We will review five policy engines along these dimensions.

OPA/Gatekeeper

Open Policy Agent (**OPA**) is a general-purpose policy engine that goes beyond Kubernetes (`https://www.openpolicyagent.org`). Its scope is very broad and it operates on any JSON value.

Gatekeeper (`https://open-policy-agent.github.io/gatekeeper`) brings the OPA policy engine to Kubernetes by packaging it as an admission control webhook.

OPA/Gatekeeper is definitely the most mature policy engine. It was created in 2017. It is a graduated CNCF project, and it has 2.9k stars on GitHub at the time of writing. It is even used as a foundation for Azure policy on AKS. See `https://learn.microsoft.com/en-us/azure/governance/policy/concepts/policy-for-kubernetes`.

OPA has its own special language called Rego (`https://www.openpolicyagent.org/docs/latest/policy-language/`) for defining policies. Rego has a strong theoretical basis inspired by Datalog, but it may not be very intuitive and easy to grasp.

The following diagram shows the architecture of OPA/Gatekeeper:

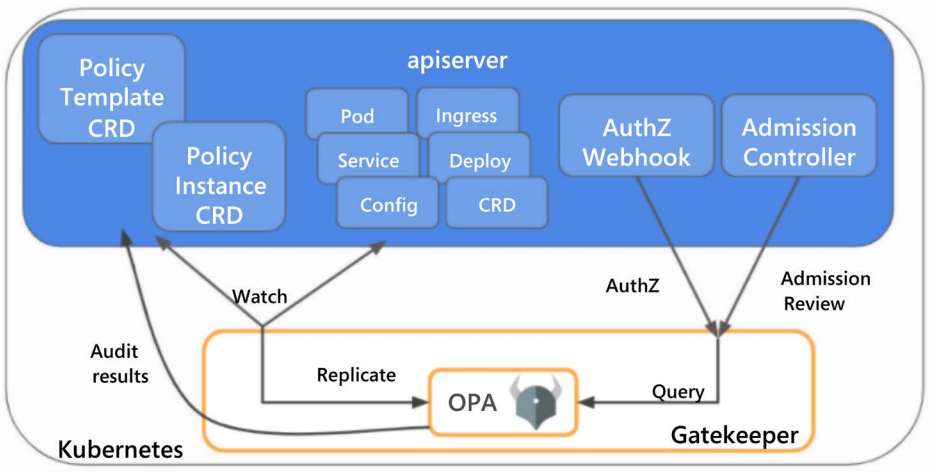


Figure 16.1: OPA/Gatekeeper architecture

Overall, OPA/Gatekeeper is very powerful but seems a little clunky compared to other Kubernetes policy engines, as the OPA policy engine is bolted on top of Kubernetes via Gatekeeper.

OPA/Gatekeeper has mediocre documentation that is not very easy to navigate. However, it does have a policy library you can use as a starting point.

However, if you appreciate the maturity, and you're not too concerned about using Rego and some friction, it may be a good choice for you.

Kyverno

Kyverno (`https://kyverno.io`) is a mature and robust policy engine that was designed especially for Kubernetes from the get-go. It was created in 2019 and has made huge strides since then. It is a CNCF incubating project and has surpassed OPA/Gatekeeper in popularity on GitHub with 3.3k stars at the time of writing. Kyverno uses YAML JMESPath (`https://jmespath.org`) to define policies, which are really just Kubernetes custom resources. It has excellent documentation and a lot of examples to get you started writing your own policy.

Overall, Kyverno is both powerful and easy to use. It has huge momentum behind it, and it keeps getting better and improving its performance and operation at scale. It is the best Kubernetes policy engine at the moment in my opinion. We will dive deep into Kyverno later in this chapter.

jsPolicy

jsPolicy (`https://www.jspolicy.com`) is an interesting project from Loft that has brought virtual clusters to the Kubernetes community. Its claim to fame is that it runs policies in a secure and performant browser-like sandbox, and you define the policies in JavaScript or TypeScript. The approach is refreshing, and the project is very slick and streamlined with good documentation. Unfortunately, it seems like Loft is focused on other projects and jsPolicy doesn't get a lot of attention. It has only 242 GitHub stars (`https://github.com/loft-sh/jspolicy`) at the time of writing, and the last commit was 6 months ago.

The idea of utilizing the JavaScript ecosystem to package and share policies, as well as use its robust tooling to test and debug policies, has a lot of merit.

jsPolicy provides validating, mutating, and controller policies. Controller policies allow you to react to events occurring in the cluster outside the scope of admission control.

The following diagram shows the architecture of jsPolicy:

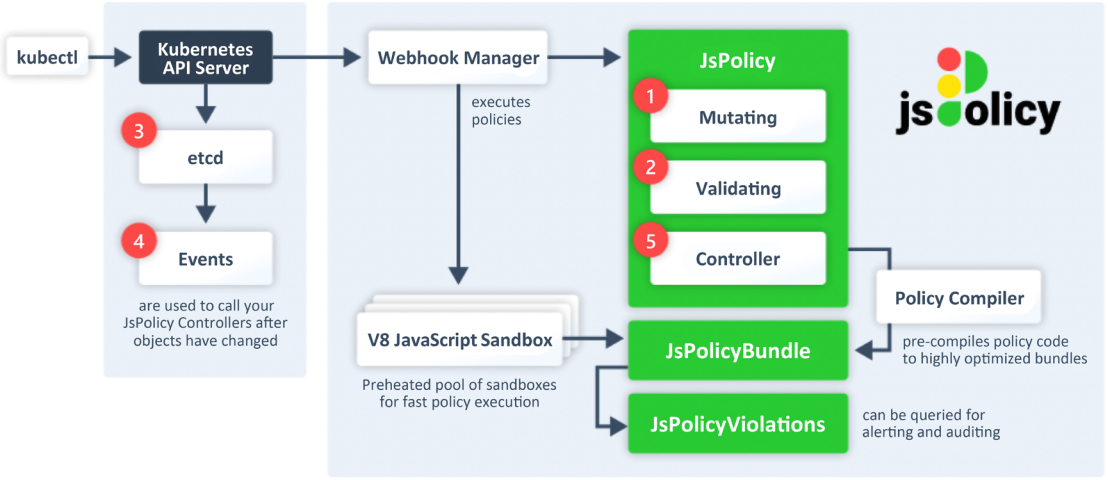


Figure 16.2: jsPolicy architecture

At this point, I wouldn't commit to jsPolicy since it may have been abandoned. However, if Loft or someone else decides to invest in it, it may be a contender in the field of Kubernetes policy engines.

Kubewarden

Kubewarden (`https://www.kubewarden.io`) is another innovative policy engine. It is a CNCF sandbox project. Kubewarden focuses on being language-agnostic and allows you to write your policies in a variety of languages. The policies are then packaged into WebAssembly modules that are stored in any OCI registry.

In theory, you can use any language that can be compiled into WebAssembly. In practice, the following languages are supported, but there are limitations:

- Rust (of course, the most mature)
- Go (you need to use a special compiler, TinyGo, which doesn't support all of Go's features)
- Rego (using OPA directly or Gatekeeper – missing mutating policies)
- Swift (using SwiftWasm, which requires some post-build optimizations)
- TypeScript (or rather a subset called AssemblyScript)

Kubewarden supports validating, mutating, and context-aware policies. Context-aware policies are policies that use additional information to form an opinion of whether a request should be admitted or rejected. The additional information may include, for example, lists of namespaces, services, and ingresses that exist in the cluster.

Kubewarden has a CLI called kwctl (`https://github.com/kubewarden/kwctl`) for managing your policies.

Here is a diagram of Kubewarden's architecture:

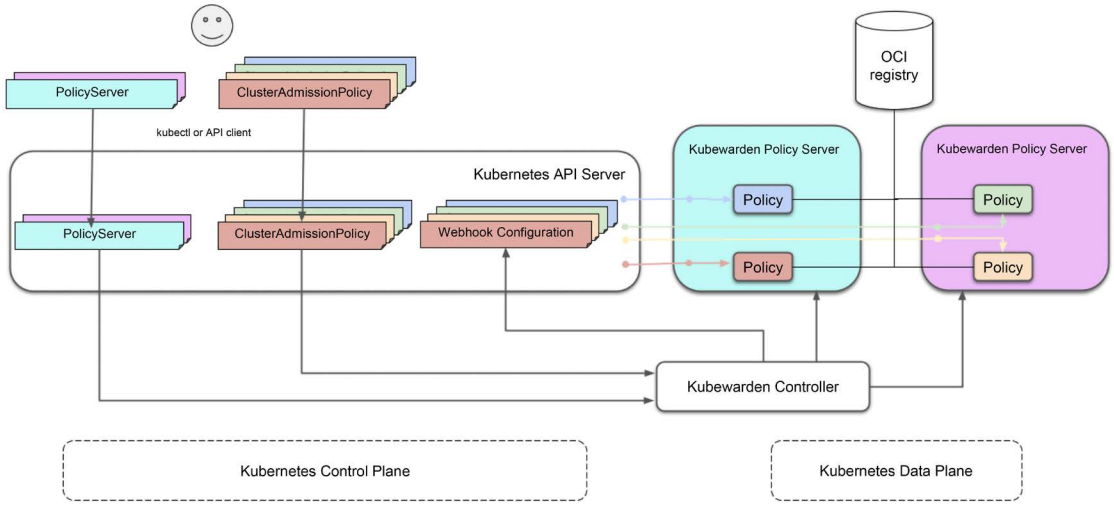


Figure 16.3: Kubewarden architecture

Kubewarden is still evolving and growing. It has some nice ideas and motivations, but at this stage, it may appeal to you most if you are on the Rust wagon and prefer to write your policies in Rust.

Now that we have looked at the landscape of Kubernetes open source policy engines, let's dive in and take a closer look at Kyverno.

Kyverno deep dive

Kyverno is a rising star in the Kubernetes policy engine arena. Let's get hands-on with it, and see how it works and why it is so popular. In this section, we will introduce Kyverno, install it, and learn how to write, apply, and test policies.

Quick intro to Kyverno

Kyverno is a policy engine that was designed especially for Kubernetes. If you have some experience working with kubectl, Kubernetes manifests, or YAML, then Kyverno will feel very familiar. You define policies and configuration using YAML manifests and the JMESPath language, which is very close to the JSONPATH format of kubectl.

The following diagram shows the Kyverno architecture:

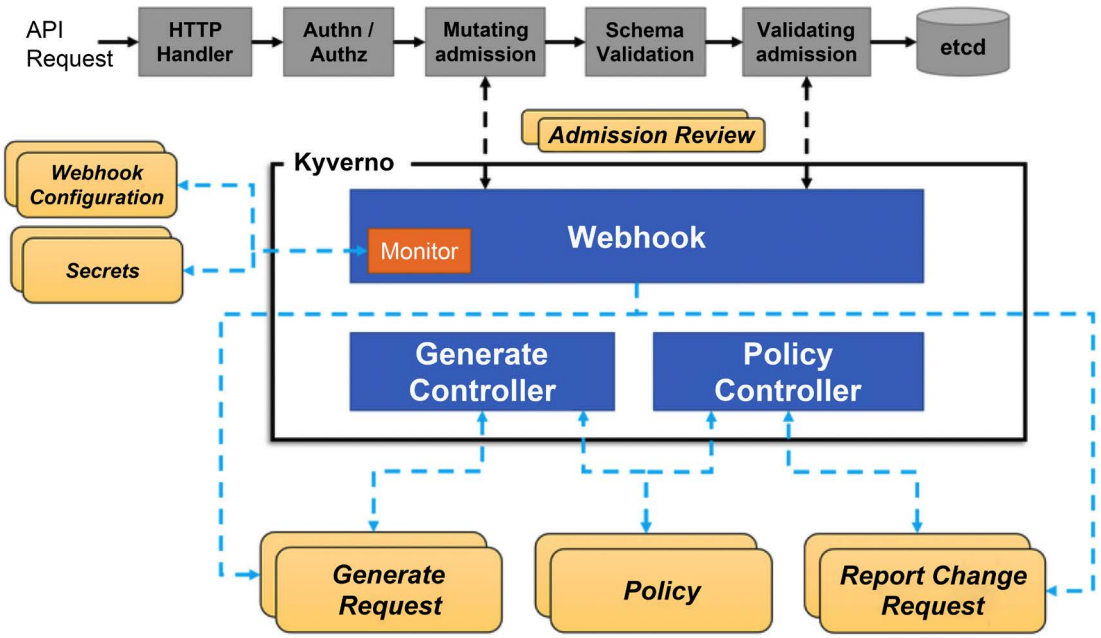


Figure 16.4: Kyverno architecture

Kyverno covers a lot of ground and has many features:

- GitOps for policy management
- Resource validation (to reject invalid resources)
- Resource mutation (to modify invalid resources)
- Resource generation (to generate additional resources automatically)
- Verifying container images (important for software supply chain security)
- Inspecting image metadata
- Using label selectors and wildcards to match and exclude resources (Kubernetes-native)
- Using overlays to validate and mutate resources (similar to Kustomize!)
- Synchronizing configurations across namespaces
- Operating in reporting or enforcing mode
- Applying policies using a dynamic admission webhook
- Applying policies at CI/CD time using the Kyverno CLI

- Testing policies and validating resources ad hoc using the Kyverno CLI
- High-availability mode
- Fail open or closed (allowing or rejecting resources when the Kyverno admission webhook is down)
- Policy violation reports
- Web UI for easy visualization
- Observability support

This is an impressive list of features and capabilities. The Kyverno developers keep evolving and improving it. Kyverno has made big strides in scalability, performance, and the ability to handle a large number of policies and resources.

Let's install Kyverno and configure it.

Installing and configuring Kyverno

Kyverno follows a similar upgrade policy as Kubernetes itself, where the node components version must be at most two minor versions below the control plane version. At the time of writing, Kyverno 1.8 is the latest version, which supports Kubernetes versions 1.23–1.25.

We can install Kyverno using kubectl or Helm. Let's go with the Helm option:

```
$ helm repo add kyverno https://kyverno.github.io/kyverno/
"kyverno" has been added to your repositories

$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "kyverno" chart repository
Update Complete. ⎈Happy Helming!⎈
```

Let's install Kyverno, using the default single replica, into its own namespace. Using one replica is NOT recommended for production, but it's okay for experimenting with Kyverno. To install it in high-availability mode, add the `--set replicaCount=3` flag:

```
$ helm install kyverno kyverno/kyverno -n kyverno --create-namespace
NAME: kyverno
LAST DEPLOYED: Sat Dec 31 15:34:11 2022
NAMESPACE: kyverno
STATUS: deployed
REVISION: 1
NOTES:
Chart version: 2.6.5
Kyverno version: v1.8.5

Thank you for installing kyverno! Your release is named kyverno.
```

⚠ **WARNING: Setting replicas count below 3 means Kyverno is not running in high availability mode.**

Note: There is a trade-off when deciding which approach to take regarding Namespace exclusions. Please see the documentation at https://kyverno.io/docs/installation/#security-vs-operability to understand the risks.

Let's observe what we have just installed using the ketall kubectl plugin: (`https://github.com/corneliusweig/ketall`):

```
$ k get-all -n kyverno
NAME                                                  NAMESPACE  AGE
configmap/kube-root-ca.crt                            kyverno    2m27s
configmap/kyverno                                     kyverno    2m26s
configmap/kyverno-metrics                             kyverno    2m26s
endpoints/kyverno-svc                                 kyverno    2m26s
endpoints/kyverno-svc-metrics                         kyverno    2m26s
pod/kyverno-7c444878f7-gfht8                          kyverno    2m26s
secret/kyverno-svc.kyverno.svc.kyverno-tls-ca         kyverno    2m22s
secret/kyverno-svc.kyverno.svc.kyverno-tls-pair       kyverno    2m21s
secret/sh.helm.release.v1.kyverno.v1                  kyverno    2m26s
serviceaccount/default                                kyverno    2m27s
serviceaccount/kyverno                                kyverno    2m26s
service/kyverno-svc                                   kyverno    2m26s
service/kyverno-svc-metrics                           kyverno    2m26s
deployment.apps/kyverno                               kyverno    2m26s
replicaset.apps/kyverno-7c444878f7                    kyverno    2m26s
lease.coordination.k8s.io/kyverno                     kyverno    2m23s
lease.coordination.k8s.io/kyverno-health              kyverno    2m13s
lease.coordination.k8s.io/kyvernopre                  kyverno    2m25s
lease.coordination.k8s.io/kyvernopre-lock             kyverno    2m24s
endpointslice.discovery.k8s.io/kyverno-svc-7ghzl      kyverno    2m26s
endpointslice.discovery.k8s.io/kyverno-svc-metrics-qflr5  kyverno  2m26s
rolebinding.rbac.authorization.k8s.io/kyverno:leaderelection  kyverno  2m26s
role.rbac.authorization.k8s.io/kyverno:leaderelection  kyverno    2m26s
```

As you can see, Kyverno installed all the expected resources: deployment, services, roles and role bindings, config maps, and secrets. We can tell that Kyverno exposes metrics and uses leader election too.

In addition, Kyverno installed many CRDs (at the cluster scope):

```
$ k get crd
NAME                                        CREATED AT
admissionreports.kyverno.io                 2022-12-31T23:34:12Z
```

```
backgroundscanreports.kyverno.io           2022-12-31T23:34:12Z
clusteradmissionreports.kyverno.io         2022-12-31T23:34:12Z
clusterbackgroundscanreports.kyverno.io    2022-12-31T23:34:12Z
clusterpolicies.kyverno.io                 2022-12-31T23:34:12Z
clusterpolicyreports.wgpolicyk8s.io        2022-12-31T23:34:12Z
generaterequests.kyverno.io                2022-12-31T23:34:12Z
policies.kyverno.io                        2022-12-31T23:34:12Z
policyreports.wgpolicyk8s.io               2022-12-31T23:34:12Z
updaterequests.kyverno.io                  2022-12-31T23:34:12Z
```

Finally, Kyverno configures several admission control webhooks:

```
$ k get validatingwebhookconfigurations
NAME                                      WEBHOOKS   AGE
kyverno-policy-validating-webhook-cfg     1          40m
kyverno-resource-validating-webhook-cfg   1          40m

$ k get mutatingwebhookconfigurations
NAME                                      WEBHOOKS   AGE
kyverno-policy-mutating-webhook-cfg       1          40m
kyverno-resource-mutating-webhook-cfg     0          40m
kyverno-verify-mutating-webhook-cfg       1          40m
```

The following diagram shows the result of a typical Kyverno installation:

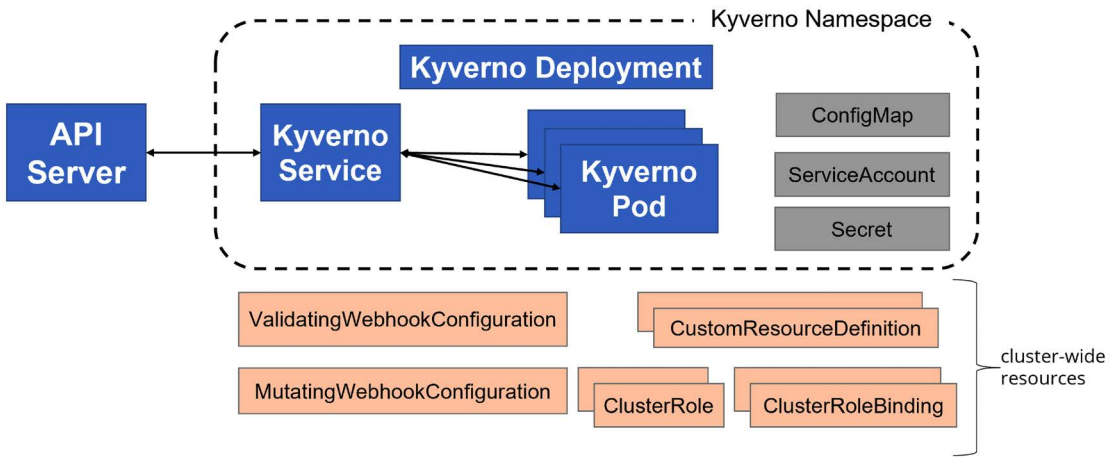


Figure 16.5: Typical Kyverno installation

Installing pod security policies

Kyverno has an extensive library of pre-built policies. We can install the pod security standard policies (see `https://kyverno.io/policies/pod-security/`) using Helm too:

```
$ helm install kyverno-policies kyverno/kyverno-policies -n kyverno-policies
--create-namespace
NAME: kyverno-policies
LAST DEPLOYED: Sat Dec 31 15:48:26 2022
NAMESPACE: kyverno-policies
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
Thank you for installing kyverno-policies 2.6.5 😀

We have installed the "baseline" profile of Pod Security Standards and set them in
audit mode.

Visit https://kyverno.io/policies/ to find more sample policies.
```

Note that the policies themselves are cluster policies and are not visible in the namespace `kyverno-policies`:

```
$ k get clusterpolicies.kyverno.io
NAME                            BACKGROUND   VALIDATE ACTION   READY
disallow-capabilities           true         audit             true
disallow-host-namespaces        true         audit             true
disallow-host-path              true         audit             true
disallow-host-ports             true         audit             true
disallow-host-process           true         audit             true
disallow-privileged-containers  true         audit             true
disallow-proc-mount             true         audit             true
disallow-selinux                true         audit             true
restrict-apparmor-profiles      true         audit             true
restrict-seccomp                true         audit             true
restrict-sysctls                true         audit             true
```

We will review some of these policies in depth later. First, let's see how to configure Kyverno.

Configuring Kyverno

You can configure the behavior of Kyverno by editing the Kyverno config map:

```
$ k get cm kyverno -o yaml -n kyverno | yq .data
resourceFilters: '[*,kyverno,*][Event,*,*][*,kube-system,*][*,kube-
public,*][*,kube-node-lease,*][Node,*,*][APIService,*,*][TokenReview,*,*]
[SubjectAccessReview,*,*][SelfSubjectAccessReview,*,*][Binding,*,*]
[ReplicaSet,*,*][AdmissionReport,*,*][ClusterAdmissionReport,*,*]
[BackgroundScanReport,*,*][ClusterBackgroundScanReport,*,*][ClusterRole,*,kyverno:*]
[ClusterRoleBinding,*,kyverno:*][ServiceAccount,kyverno,kyverno]
[ConfigMap,kyverno,kyverno][ConfigMap,kyverno,kyverno-metrics]
[Deployment,kyverno,kyverno][Job,kyverno,kyverno-hook-pre-delete]
[NetworkPolicy,kyverno,kyverno][PodDisruptionBudget,kyverno,kyverno]
[Role,kyverno,kyverno:*][RoleBinding,kyverno,kyverno:*][Secret,kyverno,kyverno-svc.
kyverno.svc.*][Service,kyverno,kyverno-svc][Service,kyverno,kyverno-svc-metrics]
[ServiceMonitor,kyverno,kyverno-svc-service-monitor][Pod,kyverno,kyverno-test]'
webhooks: '[{"namespaceSelector": {"matchExpressions":
[{"key":"kubernetes.io/metadata.name","operator":"NotIn","values":["kyverno"]}]}}]'
```

The `resourceFilters` flag is a list in the format `[kind,namespace,name]`, where each element may be a wildcard too, that tells Kyverno which resources to ignore. Resources that match any of the filters will not be subject to any Kyverno policy. This is good practice if you have a lot of policies to save the evaluation effort against all policies.

The `webHooks` flag allows you to filter out whole namespaces.

The `excludeGroupRole` flag is a string of comma-separated roles. It will exclude requests, where a user has one of the specified roles from Kyverno admission control. The default list is `system:serviceaccounts:kube-system,system:nodes,system:kube-scheduler`.

The `excludeUsername` flag represents a string consisting of Kubernetes usernames separated by commas. When a user enables `Synchronize` in `generate policy`, Kyverno becomes the only entity capable of updating or deleting generated resources. However, administrators have the ability to exclude specific usernames from accessing the delete/update-generated resource functionality.

The `generateSuccessEvents` flag is a Boolean parameter used to determine whether success events should be generated. By default, this flag is set to `false`, indicating that success events are not generated.

Furthermore, the Kyverno container provides several container arguments that can be configured to customize its behavior and functionality. These arguments allow for fine-tuning and customization of Kyverno's behavior within the container. You can edit the list of args in the Kyverno deployment:

```
$ k get deploy kyverno -n kyverno -o yaml | yq '.spec.template.spec.containers[0].
args'
- --autogenInternals=true
- --loggingFormat=text
```

In addition to the pre-configured `--autogenInternals` and `--loggingFormat`, the following flags are available:

- `admissionReports`
- `allowInsecureRegistry`
- `autoUpdateWebhooks`
- `backgroundScan`
- `clientRateLimitBurst`
- `clientRateLimitQPS`
- `disableMetrics`
- `enableTracing`
- `genWorkers`
- `imagePullSecrets`
- `imageSignatureRepository`
- `kubeconfig`
- `maxQueuedEvents`
- `metricsPort`
- `otelCollector`
- `otelConfig`
- `profile`
- `profilePort`
- `protectManagedResources`
- `reportsChunkSize`
- `serverIP`
- `splitPolicyReport` (deprecated – will be removed in 1.9)
- `transportCreds`
- `webhookRegistrationTimeout`
- `webhookTimeout`

All the flags have a default value, and you only need to specify them if you want to override the defaults.

Check out `https://kyverno.io/docs/installation/#container-flags` for details on each flag.

We installed Kyverno, observed the various resources it installed, and looked at its configuration. It's time to check out the policies and rules of Kyverno.

Applying Kyverno policies

At the user level, the unit of work in Kyverno is policies. You can apply policies as Kubernetes resources, write and edit your own policies, and test policies using the Kyverno CLI.

Applying a Kyverno policy is as simple as applying any other resource. Let's take a look at one of the policies we installed earlier:

```
$ k get clusterpolicies.kyverno.io disallow-capabilities
NAME                    BACKGROUND   VALIDATE ACTION   READY
disallow-capabilities   true         audit             true
```

The purpose of the policy is to prevent pods from requesting extra Linux capabilities beyond the allowed list (see `https://linux-audit.com/linux-capabilities-101/`). One of the capabilities that is not allowed is `NET_ADMIN`. Let's create a pod that requests this capability:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - name: some-container
    command: [ "sleep", "999999" ]
    image: g1g1/py-kube:0.3
    securityContext:
      capabilities:
        add: ["NET_ADMIN"]
EOF
pod/some-pod created
```

The pod was created, and we can verify that it has the `NET_ADMIN` capability. I use a kind cluster, so the cluster node is just a Docker process we can exec into:

```
$ docker exec -it kind-control-plane sh
#
```

Now that we're in a shell inside the node, we can search for the process of our container, which just sleeps for 999,999 seconds:

```
# ps aux | grep 'PID\|sleep' | grep -v grep
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root        4549  0.0  0.0 148276  6408 ?        Ssl  02:54   0:00 /usr/bin/
qemu-x86_64 /bin/sleep 999999
```

Let's check the capabilities of our process, 4549:

```
# getpcaps 4549
4549: cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_
setuid,cap_setpcap,cap_net_bind_service,cap_net_admin,cap_net_raw,cap_sys_chroot,cap_
mknod,cap_audit_write,cap_setfcap=ep
```

As you can see the `cap_net_admin` is present.

Kyverno didn't prevent the pod from being created because the policy operates in audit mode only:

```
$ k get clusterpolicies.kyverno.io disallow-capabilities -o yaml | yq .spec.
validationFailureAction
audit
```

Let's delete the pod and change the policy to "enforce" mode:

```
$ k delete po some-pod
pod "some-pod" deleted

$ k patch clusterpolicies.kyverno.io disallow-capabilities --type merge -p '{"spec":
{"validationFailureAction": "enforce"}}'
clusterpolicy.kyverno.io/disallow-capabilities patched
```

Now, if we try to create the pod again, the result is very different:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - name: some-container
    command: [ "sleep", "999999" ]
    image: g1g1/py-kube:0.3
    securityContext:
      capabilities:
        add: ["NET_ADMIN"]
EOF

Error from server: error when creating "STDIN": admission webhook "validate.kyverno.
svc-fail" denied the request:

policy Pod/kyverno-policies/some-pod for resource violation:

disallow-capabilities:
  adding-capabilities: Any capabilities added beyond the allowed list (AUDIT_WRITE,
    CHOWN, DAC_OVERRIDE, FOWNER, FSETID, KILL, MKNOD, NET_BIND_SERVICE, SETFCAP,
SETGID,
    SETPCAP, SETUID, SYS_CHROOT) are disallowed.
```

The Kyverno admission webhook enforced the policy and rejected the pod creation. It even tells us which policy was responsible (`disallow-capabilities`) and displays a nice message that explains the reason for the rejection, including a list of the allowed capabilities.

It is pretty simple to apply policies. Writing policies is much more involved and requires an understanding of resource requests, Kyverno matching rules, and the JMESPath language. Before we can write policies, we need to understand how they are structured and what their different elements are.

Kyverno policies in depth

In this section, we will learn all the fine details about Kyverno policies. A Kyverno policy has a set of rules that define what the policy actually does and several general settings that define how the policy behaves in different scenarios. Let's start with the policy settings and then move on to rules and different use cases, such as validating, mutating, and generating resources.

Understanding policy settings

A Kyverno policy may have the following settings:

- `applyRules`
- `validationFailureAction`
- `validationFailureActionOverrides`
- `background`
- `schemaValidation`
- `failurePolicy`
- `webhookTimeoutSeconds`

The `applyRules` setting determines if only one or multiple rules apply to matching resources. The valid values are "One" and "All" (the default). If `applyRules` is set to "One" then the first matching rule will be evaluated and other rules will be ignored.

The `validationFailureAction` setting determines if a failed validation policy rule should reject the admission request or just report it. The valid values are "audit" (default – always allows and just reports violations) and "enforce" (blocks invalid requests).

The `validationFailureActionOverrides` setting is a `ClusterPolicy` attribute that overrides the `validationFailureAction` for specific namespaces.

The `background` setting determines if policies are applied to existing resources during a background scan. The default is "true".

The `schemaValidation` setting determines if policy validation checks are applied. The default is "true".

The `failurePolicy` setting determines how the API server behaves if the webhook fails to respond. The valid values are "Ignore" and "Fail" (the default). If the setting is "Fail" then even valid resource requests will be denied, while the webhook is unreachable.

The `webhookTimeoutSeconds` determines the maximum time in seconds that the webhook is allowed to evaluate a policy. The valid values are between 1 and 30 seconds. The default is 10 seconds. If the webhook failed to respond in time, the `failurePolicy` (see above) determines the fate of the request.

Understanding Kyverno policy rules

Each Kyverno policy has one or more rules. Each rule has a `match` declaration, an optional `exclude` declaration, an optional `preconditions` declaration, and exactly one of the following declarations:

- `validate`
- `mutate`
- `generate`
- `verifyImages`

The diagram below demonstrates the structure of a Kyverno policy and its rules (policy settings are omitted):

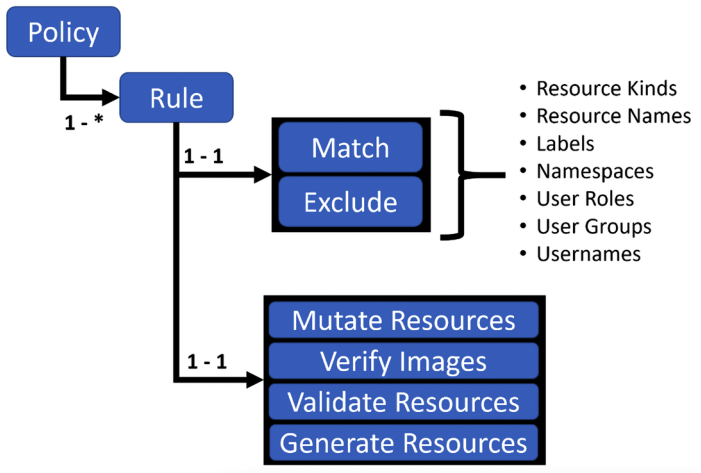


Figure 16.6: Kyverno rule structure

Let's go over the different declarations and explore some advanced topics too.

Matching requests

When a resource request arrives, the Kyverno webhook needs to determine for each policy if the requested resource and/or operation is relevant for the current policy. The mandatory `match` declaration has several filters that determine if the policy should evaluate the current request. The filters are:

- `resources`
- `subjects`
- `roles`
- `clusterRoles`

A `match` declaration may have multiple filters grouped under an `any` statement or an `all` statement. When filters are grouped under `any`, then Kyverno will apply OR semantics to match them, and if any of the filters match the request, the request is considered matched. When filters are grouped under `all`, then Kyverno will apply AND semantics, and all the filters must match in order for the request to be considered a match.

This can be a little overwhelming. Let's look at an example. The following policy spec has a single rule called `some-rule`. The rule has a `match` declaration with two resource filters under an `any` statement. The first resource filter matches resources of kind `Service` with names `service-1` or `service-2`. The second resource filter matches resources of kind `Service` in the namespace `ns-1`. This rule will match any Kubernetes service named `service-1` or `service-2` in any namespace, as well as any service in the namespace `ns-1`.

```
spec:
  rules:
  - name: some-rule
    match:
      any:
      - resources:
          kinds:
          - Service
          names:
          - "service-1"
          - "service-2"
      - resources:
          kinds:
          - Service
          namespaces:
          - "ns-1"
```

Let's look at another example. This time we add a cluster role filter. The following rule will match requests where the kind is a service named `service-1` and the requesting user has a cluster role called `some-cluster-role`.

```
  rules:
    - name: some-rule
      match:
        all:
          - resources:
              kinds:
                - Service
              names:
                - "service-1"
```

```
            clusterRoles:
              - some-cluster-role
```

The admission review resource contains all the roles and cluster roles bound to the requesting user or service account.

Excluding resources

Excluding resources is very similar to matching. It is very common to set policies that disallow all requests to create or update some resources unless they are made in certain namespaces or by users with certain roles. Here is an example that matches all services but excludes the `ns-1` namespace:

```
rules:
- name: some-rule
  match:
    any:
      - resources:
          kinds:
            - Service

  exclude:
    any:
      - resources:
          namespaces:
            - "ns-1"
```

Another common exclusion is for specific roles like `cluster-admin`.

Using preconditions

Limiting the scope of a policy using `match` and `exclude` is great, but in many cases it is not sufficient. Sometimes, you need to select resources based on fine-grained details such as memory requests. Here is an example that matches all pods that request memory of less than 1 GiB.

The syntax for the key value uses JMESPath (`https://jmespath.org`) on the built-in request object:

```
  rules:
  - name: memory-limit
    match:
      any:
      - resources:
          kinds:
          - Pod
    preconditions:
      any:
```

```
      - key: "{{request.object.spec.containers[*].resources.requests.memory}}"
        operator: LessThan
        value: 1Gi
```

Validating requests

The primary use case of Kyverno is validating requests. Validating rules have a `validate` statement. The `validate` statement has a `message` field that will be displayed if the request fails to validate. A validating rule has two forms, pattern-based validation and deny-based validation. Let's examine each of them. As you may recall, the result of a resource failing to validate depends on the `validationFailureAction` field, which can be `audit` or `enforce`.

Pattern-based validation

A rule with pattern-based validation has a `pattern` field under the `validate` statement. If the resource doesn't match the pattern, the rule failed. Here is an example of pattern-based validation, where the resource must have a label called `app`:

```
    validate:
      message: "The resource must have a label named `app`."
      pattern:
        metadata:
          labels:
            some-label: "app"
```

The validation part will only be applied to requests that comply with the `match` and `preconditions` statements and are not excluded by the `exclude` statement, if there are any.

You can also apply operators to values in the pattern – for example, here is a validation rule that requires that the number of replicas of a deployment will be at least 3:

```
  rules:
    - name: validate-replica-count
      match:
        any:
        - resources:
            kinds:
            - Deployment
      validate:
        message: "Replica count for a Deployment must be at least 3."
        pattern:
          spec:
            replicas: ">=3"
```

Deny-based validation

A rule with deny-based validation has a `deny` field under the `validate` statement. Deny rules are similar to the preconditions that we saw earlier for selecting resources. Each deny condition has a key, an operator, and a value. A common use for the deny condition is disallowing a specific operation such as `DELETE`. The following examples use deny-based validation to prevent the deletion of Deployments and StatefulSets. Note the use of request variables both for the message and the key. For `DELETE` operations the deleted object is defined as `request.oldObject` and not `request.object`:

```
rules:
  - name: block-deletes-of-deployments-and-statefulsets
    match:
      any:
      - resources:
          kinds:
            - Deployment
            - Statefulset
    validate:
      message: "Deleting {{request.oldObject.kind}}/{{request.oldObject.
metadata.name}} is not allowed"
      deny:
        conditions:
          any:
          - key: "{{request.operation}}"
            operator: Equals
            value: DELETE
```

There is more to validation, which you can explore here: `https://kyverno.io/docs/writing-policies/validate/`

Let's turn our attention to mutations.

Mutating resources

Mutation may sound scary, but all it is is modifying the resource in a request in some way. Note that the mutated request will still go through validation even if it matches any policy. It is not possible to change the kind of the requested object, but you can change its properties. The benefit of mutation is that you can automatically fix invalid requests, which is typically a better user experience instead of blocking invalid requests. The downside (especially if the invalid resources were created as part of a CI/CD pipeline) is that it creates a dissonance between the source code and the actual resources in the cluster. However, it is great for use cases where you want to control some aspects that users don't need to be aware of, as well as during migration.

Enough theory – let's see what mutation looks like in Kyverno. You still need to select the resources to mutate, which means that the `match`, `exclude`, and `precondition` statements are still needed for mutation policies.

However, instead of a `validate` statement you will have a `mutate` statement. Here is an example that uses the `patchStrategicMerge` flavor to set the `imagePullPolicy` of containers that use an image with the `latest` tag. The syntax is similar to Kustomize overlays and merges with the existing resource. The reason the `image` field is in parentheses is because of a JMESPath feature called anchors (`https://kyverno.io/docs/writing-policies/validate/#anchors`), where the rest of the subtree is applied only if the given field matches it. In this case it means the `imagePullPolicy` will only be set for images that satisfy the condition:

```
mutate:
  patchStrategicMerge:
    spec:
      containers:
        # match images which end with :latest
        - (image): "*:latest"
          # set the imagePullPolicy to "IfNotPresent"
          imagePullPolicy: "IfNotPresent"```
```

The other flavor of mutation is JSON Patch (`http://jsonpatch.com`), which is specified in RFC 6902 (`https://datatracker.ietf.org/doc/html/rfc6902`). JSON Patch has similar semantics to preconditions and deny rules. The patch has an operation, path, and value. It applies the operation to the patch with the value. The operation can be one of:

- `add`
- `remove`
- `replace`
- `copy`
- `move`
- `test`

Here is an example of adding some data to a config map using JSON Patch. It adds multiple fields to the `/data/properties` path and a single value to the `/data/key` path:

```
spec:
  rules:
    - name: patch-config-map
      match:
        any:
        - resources:
            names:
              - the-config-map
            kinds:
              - ConfigMap
      mutate:
```

```
        patchesJson6902: |-
          - path: "/data/properties"
            op: add
            value: |
              prop-1=value-1
              prop-2=value-2
          - path: "/data/key"
            op: add
            value: some-string
```

Generating resources

Generating resources is an interesting use case. Whenever a request comes in, Kyverno may create new resources instead of mutating or validating the request (other policies may validate or mutate the original request).

A policy with a `generate` rule has the same `match` and/or `exclude` statements as other policies. This means it can be triggered by any resource request as well as existing resources. However, instead of validating or mutating, it generates a new resource when the origin resource is created. A `generate` rule has an important property called `synchronize`. When `synchronize` is true, the generated resource is always in sync with the origin resource (when the origin resource is deleted, the generated resource is deleted as well). Users can't modify or delete a generated resource. When `synchronize` is false, Kyverno doesn't keep track of the generated resource, and users can modify or delete it at will.

Here is a `generate` rule that creates a `NetworkPolicy` that prevents any traffic when a new `Namespace` is created. Note the `data` field, which defines the generated resource:

```
spec:
  rules:
  - name: deny-all-traffic
    match:
      any:
      - resources:
          kinds:
          - Namespace
    generate:
      kind: NetworkPolicy
      apiVersion: networking.k8s.io/v1
      name: deny-all-traffic
      namespace: "{{request.object.metadata.name}}"
      data:
        spec:
          # select all pods in the namespace
          podSelector: {}
```

```
          policyTypes:
          - Ingress
          - Egress
```

When generating resources for an existing origin resource instead of a `data` field, a `clone` field is used. For example, if we have a config map called `config-template` in the `default` namespace, the following `generate` rule will clone that config map into every new namespace:

```
spec:
  rules:
  - name: clone-config-map
    match:
      any:
      - resources:
          kinds:
          - Namespace
    generate:
      kind: ConfigMap
      apiVersion: v1
      # Name of the generated resource
      name: default-config
      namespace: "{{request.object.metadata.name}}"
      synchronize: true
      clone:
        namespace: default
        name: config-template
```

It's also possible to clone multiple resources by using a `cloneList` field instead of a `clone` field.

Advanced policy rules

Kyverno has some additional advanced capabilities, such as external data sources and autogen rules for pod controllers.

External data sources

So far we've seen how Kyverno uses information from an admission review object to perform validation, mutation, and generation. However, sometimes additional data is needed. This is done by defining a context field with variables that can be populated from an external config map, the Kubernetes API server, or an image registry.

Here is an example of defining a variable called `dictionary` and using it to mutate a pod and add a label called `environment`, where the value comes from the config map variable:

```
    rules:
    - name: configmap-lookup
```

```
      context:
      - name: dictionary
        configMap:
          name: some-config-map
          namespace: some-namespace
      match:
        any:
        - resources:
            kinds:
            - Pod
      mutate:
        patchStrategicMerge:
          metadata:
            labels:
              environment: "{{dictionary.data.env}}"
```

The way it works is that the context named "dictionary" points to a config map. Inside the config map there is a section called "data" with a key called "env".

Autogen rules for pod controllers

Pods are one of the most common resources to apply policies to. However, pods can be created indirectly by many types of resources: Pods (directly), Deployments, StatefulSets, DaemonSets, and Jobs. If we want to verify that every pod has a label called "app" then we will be forced to write complex match rules with an `any` statement that covers all the various resources that create pods. Kyverno provides a very elegant solution in the form of autogen rules for pod controllers.

The auto-generated rules can be observed in the status of the policy object. We will see an example in the next section.

We covered in detail a lot of the powerful capabilities Kyverno brings to the table. Let's write some policies and see them in action.

Writing and testing Kyverno policies

In this section, we will actually write some Kyverno policies and see them in action. We will use some of the rules we explored in the previous section and embed them in full-fledged policies, apply the policies, create resources that comply with the policies as well as resources that violate the policies (in the case of validating policies), and see the outcome.

Writing validating policies

Let's start with a validating policy that disallows services in the namespace `ns-1` as well as services named `service-1` or `service-2` in any namespace:

```
$ cat <<EOF | k apply -f -
apiVersion: kyverno.io/v1
```

```
kind: ClusterPolicy
metadata:
  name: disallow-some-services
spec:
  validationFailureAction: Enforce
  rules:
    - name: some-rule
      match:
        any:
          - resources:
              kinds:
                - Service
              names:
                - "service-1"
                - "service-2"
          - resources:
              kinds:
                - Service
              namespaces:
                - "ns-1"
      validate:
        message: >-
         services named service-1 and service-2 and
         any service in namespace ns-1 are not allowed
        deny: {}
EOF
clusterpolicy.kyverno.io/disallow-some-services created
```

Now that the policy is in place, let's try to create a service named "service-1" in the default namespace that violates the policy. Note that there is no need to actually create resources to check the outcome of admission control. It is sufficient to run in dry-run mode as long as the dry-run happens on the server side:

```
$ k create service clusterip service-1 -n default --tcp=80 --dry-run=server
error: failed to create ClusterIP service: admission webhook "validate.kyverno.svc-
fail" denied the request:

policy Service/default/service-1 for resource violations:

disallow-some-services:
  some-rule: services named service-1 and service-2 and  any service in namespace
    ns-1 are not allowed
```

```
exclude-services-namespace:
  some-rule: services are not allowed, except in the ns-1 namespace
```

As you can see, the request was rejected, with a nice message from the policy that explains why.

If we try to do the dry-run on the client side, it succeeds (but doesn't actually create any service), as the admission control check happens only on the server:

```
$ k create service clusterip service-1 -n default --tcp=80 --dry-run=client
service/service-1 created (dry run)
```

Now that we have proved the point, we will use only a server-side dry-run.

Let's try to create a service called `service-3` in the default namespace, which should be allowed:

```
$ k create service clusterip service-3 -n default --tcp=80 --dry-run=server
service/service-3 created (server dry run)
```

Let's try to create `service-3` in the forbidden `ns-1` namespace:

```
$ k create ns ns-1
$ k create service clusterip service-3 -n ns-1 --tcp=80 --dry-run=server

error: failed to create ClusterIP service: admission webhook "validate.kyverno.svc-
fail" denied the request:

policy Service/ns-1/service-3 for resource violation:

disallow-some-services:
  some-rule: services named service-1 and service-2 and  any service in namespace
    ns-1 are not allowed
```

Yep. That failed as expected. Let's see what happens if we change the `validationFailureAction` from `Enforce` to `Audit`:

```
$ k patch clusterpolicies.kyverno.io disallow-some-services --type merge -p '{"spec":
{"validationFailureAction": "Audit"}}'
clusterpolicy.kyverno.io/disallow-some-services patched

$ k create service clusterip service-3 -n ns-1 --tcp=80 --dry-run=server
service/service-3 created (server dry run)
```

However, it generated a report of validation failure:

```
$ k get policyreports.wgpolicyk8s.io -n ns-1

NAME                          PASS   FAIL   WARN   ERROR   SKIP   AGE
cpol-disallow-some-services   0      1      0      0       0      2m4s
```

Now, the service passes the admission control, but a record of the violation was captured in the policy report. We will look at reports in more detail later in the chapter.

For now, let's look at mutating policies.

Writing mutating policies

Mutating policies are a lot of fun. They quietly modify incoming requests to comply with the policy. They don't cause failures like validating policies in "enforce" mode, and they don't generate reports you need to scour through like validating policies in "audit" mode. If an invalid or incomplete request comes in, you just change it until it's valid.

Here is a policy that sets the `imagePullPolicy` to `IfNotPresent` when the tag is `latest` (by default it is `Always`).

```
$ cat <<EOF | k apply -f -
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: set-image-pull-policy
spec:
  rules:
    - name: set-image-pull-policy
      match:
        any:
          - resources:
              kinds:
                - Pod
      mutate:
        patchStrategicMerge:
          spec:
            containers:
              # match images which end with :latest
              - (image): "*:latest"
                # set the imagePullPolicy to "IfNotPresent"
                imagePullPolicy: "IfNotPresent"
EOF
clusterpolicy.kyverno.io/set-image-pull-policy created
```

Let's see it in action. Note that for a mutating policy, we can't use dry-run because the whole point is to actually mutate a resource.

The following pod matches our policy and doesn't have `imagePullPolicy` set:

```
$ cat <<EOF | k apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
    - name: some-container
      image: g1g1/py-kube:latest
      command:
        - sleep
        - "9999"
EOF
pod/some-pod created
```

Let's verify that the mutation worked and check the container's `imagePullPolicy`:

```
$ k get po some-pod -o yaml | yq '.spec.containers[0].imagePullPolicy'
IfNotPresent
```

Yes. It was set correctly. Let's confirm that Kyverno was responsible for setting the `imagePullPolicy` by deleting the policy and then creating another pod:

```
$ k delete clusterpolicy set-image-pull-policy
clusterpolicy.kyverno.io "set-image-pull-policy" deleted

$ cat <<EOF | k apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: another-pod
spec:
  containers:
    - name: some-container
      image: g1g1/py-kube:latest
      command:
        - sleep
        - "9999"
EOF
pod/another-pod created
```

The Kyverno policy was deleted, and another pod called `another-pod` with the same image `g1g1/py-kube:latest` was created. Let's see if its `imagePullPolicy` is the expected `Always` (the default for images with the `latest` image tag):

```
$ k get po another-pod -o yaml | yq '.spec.containers[0].imagePullPolicy'
Always
```

Yes, it works how it should! Let's move on to another type of exciting Kyverno policy – a generating policy, which can create new resources out of thin air.

Writing generating policies

Generating policies create new resources in addition to the requested resource when a new resource is created. Let's take our previous example of creating an automatic network policy for new namespaces that prevents any network traffic from coming in and out. This is a cluster policy that applies to any new namespace except the excluded namespaces:

```
cat <<EOF | k apply -f -
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: deny-all-traffic
spec:
  rules:
  - name: deny-all-traffic
    match:
      any:
      - resources:
          kinds:
          - Namespace
    exclude:
      any:
      - resources:
          namespaces:
          - kube-system
          - default
          - kube-public
          - kyverno
    generate:
      kind: NetworkPolicy
      apiVersion: networking.k8s.io/v1
      name: deny-all-traffic
      namespace: "{{request.object.metadata.name}}"
      data:
```

```
        spec:
          # select all pods in the namespace
          podSelector: {}
          policyTypes:
          - Ingress
          - Egress
EOF
clusterpolicy.kyverno.io/deny-all-traffic created
```

The `deny-all-traffic` Kyverno policy was created successfully. Let's create a new namespace, `ns-2`, and see if the expected NetworkPolicy is generated:

```
$ k create ns ns-2
namespace/ns-2 created

$ k get networkpolicy -n ns-2
NAME               POD-SELECTOR   AGE
deny-all-traffic   <none>         15s
```

Yes, it worked! Kyverno lets you easily generate additional resources.

Now that we have some hands-on experience in creating Kyverno policies, let's learn about how to test them and why.

Testing policies

Testing Kyverno policies before deploying them to production is very important because Kyverno policies are very powerful, and they could easily cause outages and incidents if misconfigured by blocking valid requests, allowing invalid requests, improperly mutating resources, and generating resources in the wrong namespaces.

Kyverno offers tooling as well as guidance about testing its policies.

The Kyverno CLI

The Kyverno CLI is a versatile command-line program that lets you apply policies on the client side and see the results, run tests, and evaluate JMESPath expressions.

Follow these instructions to install the Kyverno CLI: `https://kyverno.io/docs/kyverno-cli/#building-and-installing-the-cli`.

Verify that it was installed correctly by checking the version:

```
$ kyverno version
Version: 1.8.5
Time: 2022-12-20T08:41:43Z
Git commit ID: c19061758dc4203106ab6d87a245045c20192721
```

Here is the help screen if you just type kyverno with no additional command:

```
$ kyverno
Kubernetes Native Policy Management

Usage:
  kyverno [command]

Available Commands:
  apply       applies policies on resources
  completion  Generate the autocompletion script for the specified shell
  help        Help about any command
  jp          Provides a command-line interface to JMESPath, enhanced with Kyverno
specific custom functions
  test        run tests from directory
  version     Shows current version of kyverno

Flags:
      --add_dir_header           If true, adds the file directory to the header of
the log messages
  -h, --help                     help for kyverno
      --log_file string          If non-empty, use this log file (no effect when
-logtostderr=true)
      --log_file_max_size uint   Defines the maximum size a log file can grow to (no
effect when -logtostderr=true). Unit is megabytes. If the value is 0, the maximum
file size is unlimited. (default 1800)
      --one_output               If true, only write logs to their native
severity level (vs also writing to each lower severity level; no effect when
-logtostderr=true)
      --skip_headers             If true, avoid header prefixes in the log messages
      --skip_log_headers         If true, avoid headers when opening log files (no
effect when -logtostderr=true)
  -v, --v Level                  number for the log level verbosity

Use "kyverno [command] --help" for more information about a command.
```

Earlier in the chapter we saw how to evaluate the results of a validating Kyverno policy without actually creating resources, using a dry-run. This is not possible for mutating or generating policies. With `kyverno apply` we can achieve the same effect for all policy types.

Let's see how to apply a mutating policy to a resource and examine the results. We will apply the `set-image-pull-policy` to a pod stored in the file `some-pod.yaml`. The policy was defined earlier, and is available in the attached code as the file `mutate-image-pull-policy.yaml`.

First, let's see what the result would be if we just created the pod without applying the Kyverno policy:

```
$ k apply -f some-pod.yaml -o yaml --dry-run=server | yq '.spec.containers[0].
imagePullPolicy'
Always
```

It is `Always`. Now, we will apply the Kyverno policy to this pod resource and check the outcome:

```
$ kyverno apply mutate-image-pull-policy.yaml --resource some-pod.yaml

Applying 1 policy rule to 1 resource...

mutate policy set-image-pull-policy applied to default/Pod/some-pod:
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
  namespace: default
spec:
  containers:
  - command:
    - sleep
    - "9999"
    image: g1g1/py-kube:latest
    imagePullPolicy: IfNotPresent
    name: some-container

---

pass: 1, fail: 0, warn: 0, error: 0, skip: 2
```

As you can see, after the mutating policy is applied to `some-pod`, the `imagePullPolicy` is `IfNotPresent` as expected.

Let's play with the `kyverno jp` sub-command. It accepts standard input or can take a file.

Here is an example that checks how many arguments the command of the first container in a pod has. We will use this pod manifest as input:

```
$ cat some-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
```

```
  containers:
    - name: some-container
      image: g1g1/py-kube:latest
      command:
        - sleep
        - "9999"
```

Note that it has a command called `sleep` with a single argument, "9999". We expect the answer to be 1. The following command does the trick:

```
$ cat some-pod.yaml | kyverno jp 'length(spec.containers[0].command) | subtract(@,
`1`)'
1
```

How does it work? First it pipes the content of `some-pod.yaml` to the `kyverno jp` command with the JMESPath expression that takes the length of the command of the first container (an array with two elements, "sleep" and "9000"), and then it pipes it to the `subtract()` function, which subtracts 1 and, hence, ends up with the expected result of 1.

The Kyverno CLI commands `apply` and `jp` are great for ad hoc exploration and the quick prototyping of complex JMESPath expressions. However, if you use Kyverno policies at scale (and you should), then I recommend a more rigorous testing practice. Luckily Kyverno has good support for testing via the `kyverno test` command. Let's see how to write and run Kyverno tests.

Understanding Kyverno tests

The `kyverno test` command operates on a set of resources and policies governed by a file called `kyverno-test.yaml`, which defines what policy rules should be applied to which resources and what the expected outcome is. It then returns the results.

The result of applying a policy rule to a resource can be one of the following four:

- `pass` – the resource matches the policy and doesn't trigger the `deny` statement (only for validating policies)
- `fail` – the resource matches the policy and triggers the deny statement (only for validating policies)
- `skip` – the resource doesn't match the policy definition and the policy wasn't applied
- `warn` – the resource doesn't comply with the policy but has an annotation: `policies.kyverno.io/scored: "false"`

If the expected outcome of the test doesn't match the result of applying the policy to the resource, then the test will be considered a failure.

For mutating and generating policies, the test will include `patchedResource` and `generatedResource` respectively.

Let's see what the `kyverno-test.yaml` file looks like:

```
name: <some name>
policies:
  - <path/to/policy.yaml>
  - <path/to/policy.yaml>
resources:
  - <path/to/resource.yaml>
  - <path/to/resource.yaml>
variables: variables.yaml # optional file for declaring variables
userinfo: user_info.yaml # optional file for declaring admission request
information (roles, cluster roles and subjects)
results:
- policy: <name>
  rule: <name>
  resource: <name>
  resources: # optional, primarily for `validate` rules. One of either
`resource` or `resources[]` must be specified. Use `resources[]` when a number
of different resources should all share the same test result.
  - <name_1>
  - <name_2>
  namespace: <name> # when testing for a resource in a specific Namespace
  patchedResource: <file_name.yaml> # when testing a mutate rule this field is
required.
  generatedResource: <file_name.yaml> # when testing a generate rule this field
is required.
  kind: <kind>
  result: pass
```

Many different test cases can be defined in a single `kyverno-test.yaml` file. The file has five sections:

- `policies`
- `resources`
- `variables`
- `userInfo`
- `results`

The `policies` and `resources` sections specify paths to all the policies and resources that participate in the tests. The `variables` and `userInfo` optional sections can define additional information that will be used by the test cases.

The `results` section is where the various test cases are specified. Each test case tests the application of a single policy rule to a single resource. If it's a validating rule, then the `result` field should contain the expected outcome.

If it's a mutating or generating rule, then the corresponding `patchedResource` or `generatedResource` should contain the expected outcome.

Let's write some Kyverno tests for our policies.

Writing Kyverno tests

All the files mentioned here are available in the `tests` sub-directory of the code attached to the chapter.

Let's start by writing our `kyverno-test.yaml` file:

```
name: test-some-rule
policies:
  - ../disallow-some-services-policy.yaml
resources:
  - test-service-ok.yaml
  - test-service-bad-name.yaml
  - test-service-bad-namespace.yaml
results:
  - policy: disallow-some-services
    rule: some-rule
    resources:
      - service-ok
    kind: Service
    result: skip
  - policy: disallow-some-services
    rule: some-rule
    resources:
      - service-1
    kind: Service
    result: fail
  - policy: disallow-some-services
    rule: some-rule
    resources:
      - service-in-ns-1
    kind: Service
    namespace: ns-1
    result: fail
```

The `policies` section contains the `disallow-some-services-policy.yaml` file. This policy rejects services named `service-1` or `service-2` and any service in the `ns-1` namespace.

The `resources` section contains three different files that all contain a Service resource:

- `test-service-ok.yaml`
- `test-service-bad-name.yaml`
- `test-service-bad-namespace.yaml`

The `test-service-ok.yaml` file contains a service that doesn't match any of the rules of the policy. The `test-service-bad-name.yaml` file contains a service named `service-1`, which is not allowed. Finally, the `test-service-bad-namespace.yaml` file contains a resource named `service-in-ns-1`, which is allowed. However, it has the `ns-1` namespace, which is not allowed.

Let's look at the `results` section. There are three different test cases here. They all test the same rule in our policy, but each test case uses a different resource name. This comprehensively covers the behavior of the policy.

The first test case verifies that a service that doesn't match the rule is skipped. It specifies the policy, the rule name, the resources the test case should be applied to, and most importantly, the expected result, which is `skip` in this case:

```
  - policy: disallow-some-services
    rule: some-rule
    resources:
      - service-ok
    result: skip
```

The second test case is similar except that the resource name is different and the expected result is `fail`:

```
  - policy: disallow-some-services
    rule: some-rule
    resources:
      - service-1
    kind: Service
    result: fail
```

There is one more slight difference. In this test case, the kind of the target resource is explicitly specified (`kind: Service`). This may seem redundant at first glance because the `service-1` resource defined in `test-service-bad-name.yaml` already has the `kind` listed:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: service-1
  name: service-1
  namespace: ns-2
spec:
```

```
  ports:
    - name: https
      port: 443
      targetPort: https
  selector:
    app: some-app
```

The reason the `kind` field is needed is to disambiguate which resource is targeted, in case the resource file contains multiple resources with the same name.

The third test case is the same as the second test case, except it targets a different resource and, as a consequence, a different part of the rule (disallowing services in the `ns-1` namespace):

```
- policy: disallow-some-services
  rule: some-rule
  resources:
    - service-in-ns-1
  kind: Service
  namespace: ns-1
  result: fail
```

OK. We have our test cases. Let's see how to run these tests.

Running Kyverno tests

Running Kyverno tests is very simple. You just type `kyverno test` and the path to the folder containing a `kyverno-test.yaml` file or a Git repository and a branch.

Let's run our tests:

```
$ kyverno test .

Executing test-some-rule...
applying 1 policy to 3 resources...

|---|------------------------|-----------|-----------------------------------|--------|
| # | POLICY                 | RULE      | RESOURCE                          | RESULT |
|---|------------------------|-----------|-----------------------------------|--------|
| 1 | disallow-some-services | some-rule | ns-2//service-ok                  | Pass   |
| 2 | disallow-some-services | some-rule | ns-2/Service/service-1            | Pass   |
| 3 | disallow-some-services | some-rule | ns-1/Service/service-in-ns-1      | Pass   |
|---|------------------------|-----------|-----------------------------------|--------|

Test Summary: 3 tests passed and 0 tests failed
```

We get a nice output that lists each test case and then a one-line summary. All three tests passed, so that's great.

When you have test files that contain a lot of test cases and you try to tweak one specific rule, you may want to run a specific test case only. Here is the syntax:

```
kyverno test . --test-case-selector "policy=disallow-some-services, rule=some-rule,
resource=service-ok"
```

The `kyverno test` command has very good documentation with a lot of examples. Just type `kyverno test -h`.

So far, we have written policies, rules, and policy tests and executed them. The last piece of the puzzle is viewing reports when Kyverno is running.

Viewing Kyverno reports

Kyverno generates reports for policies with `validate` or `verifyImages` rules. Only policies in `audit` mode or that have `spec.background: true` will generate reports.

As you recall Kyverno can generate two types of reports in the form of custom resources. `PolicyReports` are generated for namespace-scoped resources (like services) in the namespace the resource was applied. `ClusterPolicyReports` are generated for cluster-scoped resources (like namespaces).

Our `disallow-some-services` policy has a `validate` rule and operates in `audit` mode, which means that if we create a service that violates the rule, the service will be created, but a report will be generated. Here we go:

```
$ k create service clusterip service-3 -n ns-1 --tcp=80
service/service-3 created
```

We created a service in the forbidden `ns-1` namespace. Kyverno didn't block the creation of the service because of audit mode. Let's review the report (that `polr` is shorthand for `policyreports`):

```
$ k get polr -n ns-1
NAME                          PASS   FAIL   WARN   ERROR   SKIP   AGE
cpol-disallow-some-services   0      1      0      0       0      1m
```

A report named `cpol-disallow-some-services` was created. We can see that it counted one failure. What happens if we create another service?

```
$ k create service clusterip service-4 -n ns-1 --tcp=80
service/service-4 created

$ k get polr -n ns-1
NAME                          PASS   FAIL   WARN   ERROR   SKIP   AGE
cpol-disallow-some-services   0      2      0      0       0      2m
```

Yep. Another failure is reported. The meaning of these failures is that the resource failed to pass the `validate` rule. Let's peek inside. The report has a `metadata` field, which includes an annotation for the policy it represents. Then there is a `results` section where each failed resource is listed. The info for each result includes the resource that caused the failure and the rule it violated. Finally, the `summary` contains aggregate information about the results:

```
$ k get polr cpol-disallow-some-services -n ns-1 -o yaml

apiVersion: wgpolicyk8s.io/v1alpha2
kind: PolicyReport
metadata:
  creationTimestamp: "2023-01-22T04:01:12Z"
  generation: 3
  labels:
    app.kubernetes.io/managed-by: kyverno
    cpol.kyverno.io/disallow-some-services: "2472317"
  name: cpol-disallow-some-services
  namespace: ns-1
  resourceVersion: "2475547"
  uid: dadcd6ae-a867-4ec8-bf09-3e6ca76da7ba
results:
- message: services named service-1 and service-2 and  any service in namespace ns-1
    are not allowed
  policy: disallow-some-services
  resources:
  - apiVersion: v1
    kind: Service
    name: service-4
    namespace: ns-1
    uid: 4d473ac1-c1b1-4929-a70d-fad98a411428
  result: fail
  rule: some-rule
  scored: true
  source: kyverno
  timestamp:
    nanos: 0
    seconds: 1674361576
- message: services named service-1 and service-2 and  any service in namespace ns-1
    are not allowed
  policy: disallow-some-services
  resources:
  - apiVersion: v1
```

```
    kind: Service
    name: service-3
    namespace: ns-1
    uid: 62458ac4-fe39-4854-9f5a-18b26109511a
  result: fail
  rule: some-rule
  scored: true
  source: kyverno
  timestamp:
    nanos: 0
    seconds: 1674361426
summary:
  error: 0
  fail: 2
  pass: 0
  skip: 0
  warn: 0
```

This is pretty nice, but it might not be the best option to keep track of your cluster if you have a lot of namespaces. It's considered best practice to collect all the reports and periodically export them to a central location. Check out the Policy reporter project: `https://github.com/kyverno/policy-reporter`. It also comes with a web-based policy reporter UI.

Let's install it:

```
$ helm repo add policy-reporter https://kyverno.github.io/policy-reporter
"policy-reporter" has been added to your repositories

$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "policy-reporter" chart repository
...Successfully got an update from the "kyverno" chart repository
Update Complete. ⎈Happy Helming!⎈

$ helm upgrade --install policy-reporter policy-reporter/policy-reporter --create-
namespace -n policy-reporter --set ui.enabled=true
Release "policy-reporter" does not exist. Installing it now.
NAME: policy-reporter
```

```
LAST DEPLOYED: Sat Jan 21 20:39:42 2023
NAMESPACE: policy-reporter
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

The policy reporter has been installed successfully in the `policy-reporter` namespace, and we enabled the UI.

The next step is to do port-forwarding to access the UI:

```
$ k port-forward service/policy-reporter-ui 8080:8080 -n policy-reporter
Forwarding from 127.0.0.1:8080 -> 8080
Forwarding from [::1]:8080 -> 8080
```

Now, we can browse `http://localhost:8080` and view policy reports visually.

The dashboard shows the failing policy reports. We can see our 20 failures in the `kube-system` namespace and our 2 failures in the `ns-1` namespace.

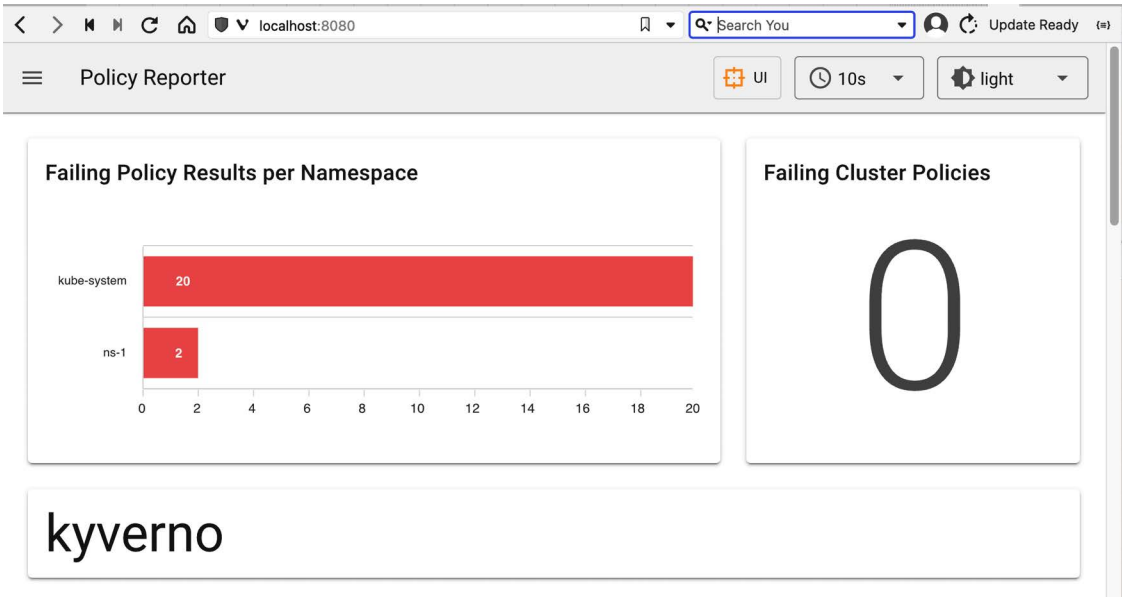


Figure 16.7: Policy Reporter UI – Dashboard

The failures in kube-system are due to the best practice security policies we installed with Kyverno.

We can scroll down and see more details about the failures:

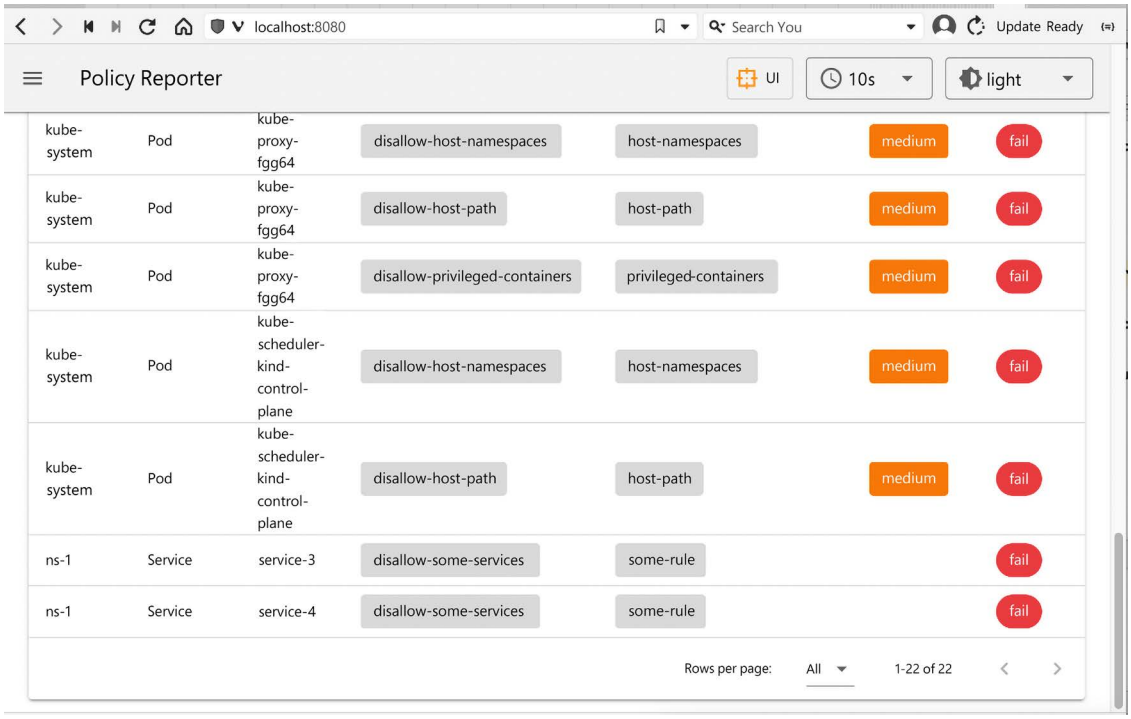


Figure 16.8: Policy Reporter UI – Results

We can also select from the sidebar the "Policy Reports" option and then see passing results. We can also filter policy reports using different criteria like policies, kinds, categories, severities, and namespaces:

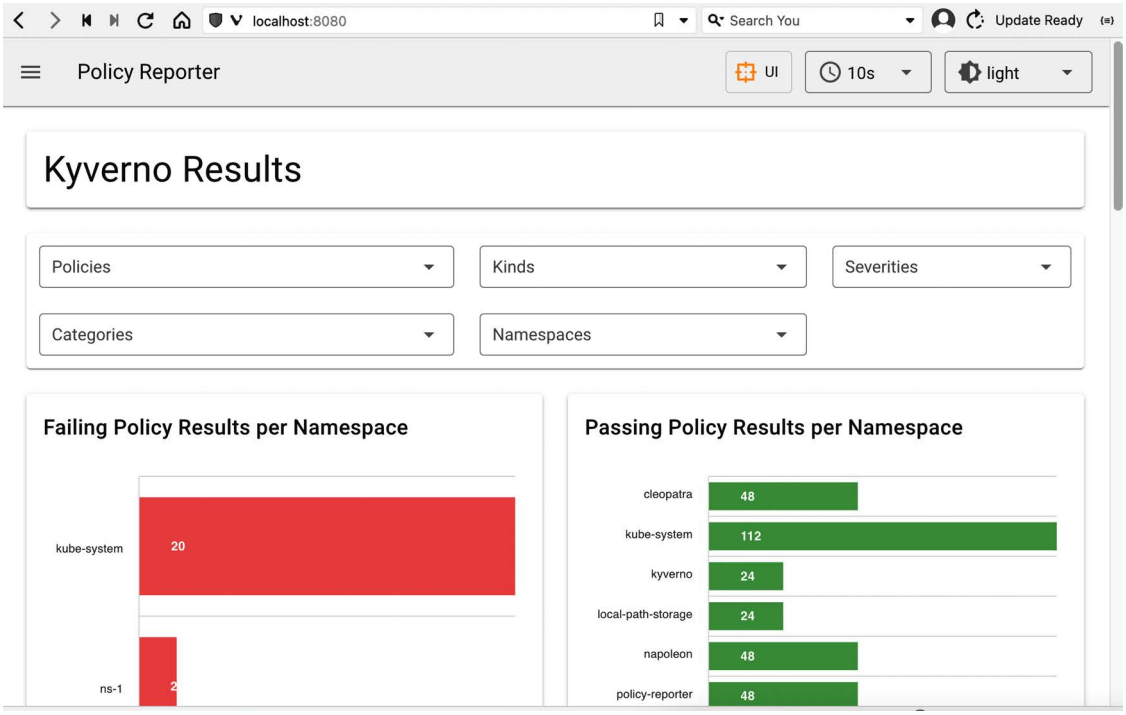


Figure 16.9: Policy Reporter UI – Policy Reports

Overall, the policy reporter UI has a slick look and provides a great option for exploring, filtering, and searching policy reports.

Summary

In this chapter, we covered the increasing adoption of Kubernetes in large enterprise organizations and the importance of governance in managing these deployments. We looked at the concept of policy engines and how they are built on top of the Kubernetes admission control mechanism. We discussed how policy engines are used to address security, compliance, and governance concerns. We also provided a review of popular policy engines. Finally, we did a deep dive into Kyverno, in which we explained in detail how it works. Then, we jumped in, wrote some policies, tested them, and reviewed policy reports. If you run a non-trivial production system on Kubernetes, you should very seriously consider having Kyverno (or another policy engine) as a core component. This is a perfect segue to the next chapter where we will discuss Kubernetes in production.

17

Running Kubernetes in Production

In the previous chapter, we discussed governance and policy engines. This is an important part of managing large-scale Kubernetes-based systems in production. However, it is only one part. In this chapter, we will turn our attention to the overall management of Kubernetes in production. The focus will be on running multiple Managed Kubernetes clusters in the cloud.

The topics we will cover are:

- Understanding Managed Kubernetes in the cloud
- Managing multiple clusters
- Building effective processes for large-scale Kubernetes deployments
- Handling infrastructure at scale
- Managing clusters and node pools
- Upgrading Kubernetes
- Troubleshooting
- Cost management

Understanding Managed Kubernetes in the cloud

Managed Kubernetes is a service provided by cloud providers such as **Amazon Web Services** (**AWS**), **Google Cloud Platform** (**GCP**), and **Microsoft Azure** that simplifies the deployment, management, and scaling of containerized applications in the cloud. With Managed Kubernetes, organizations can focus on developing and deploying their applications without worrying too much about the underlying infrastructure.

Managed Kubernetes provides a pre-configured and optimized environment for deploying containers, eliminating the need for the manual setup and maintenance of a Kubernetes cluster. This allows organizations to quickly deploy and scale their applications, reducing time to market and freeing up valuable resources.

Additionally, Managed Kubernetes integrates with the cloud providers' other services, such as databases, networking, storage solutions, security, identity, and observability features, making it easier to manage and secure the entire application stack. This also enables organizations to leverage the providers' expertise in managing large-scale infrastructure, ensuring high availability, and reducing downtime.

Overall, Managed Kubernetes provides a simplified and efficient way to deploy and manage containerized applications in the cloud, reducing operational overhead and improving time to market. This makes it an attractive option for organizations of all sizes looking to take advantage of the benefits of containers and the cloud.

Deep integration

Cloud providers utilize the extensibility of Kubernetes to offer deep integration of their Managed Kubernetes solutions with their cloud services via the CNI, the CSI, and authentication/authorization plugins. The cloud providers also implement the **Cloud Controller Interface** (**CCI**) to allow their compute infrastructure to serve Kubernetes nodes.

However, the integration runs deeper. The cloud providers often configure the kubelet, control the container runtime that runs on every node, and deploy various DaemonSets on every node.

For example, AKS leverages many Azure services:

- **Azure Compute**: AKS leverages Azure Compute resources such as **virtual machines** (**VMs**), availability sets, and scale sets to provide a managed Kubernetes experience.
- **Azure Virtual Network**: AKS integrates with Azure Virtual Network, allowing users to create and manage their own virtual networks and subnets. This provides users with control over their network layout and the ability to tightly control network traffic.
- **Azure Blob Storage**: AKS integrates with Azure Blob Storage, allowing users to store and manage their application data in the cloud. This provides users with scalable, secure, and highly available storage for their applications.
- **Azure Key Vault**: AKS integrates with Azure Key Vault, allowing users to securely manage and store secrets such as passwords, keys, and certificates. This provides users with secure storage for their application secrets.
- **Azure Monitor**: AKS integrates with Azure Monitor, allowing users to collect and analyze metrics, logs, and traces from their applications. This provides users with the ability to monitor and troubleshoot their workloads.
- **Azure Active Directory** (**AAD**): AKS integrates with AAD to provide a secure, reliable, and highly available platform for running Kubernetes clusters. AAD provides an efficient and secure way to authenticate and authorize users and applications to access the cluster. AAD can also be integrated with Kubernetes **RBAC** (**role-based access control**) to provide granular control over access to cluster resources.

Let's move on and discuss one of the key elements of successfully managing a production Kubernetes-based system in the cloud.

Quotas and limits

Cloud infrastructure has revolutionized the way organizations store and manage their data and run their workloads. However, one major issue that requires consideration and attention is the use of quotas and limits by cloud service providers. These quotas and limits, while necessary for ensuring the stability and security of the cloud infrastructure, can be a major source of frustration and even outages for users.

Quotas and limits are restrictions placed on the number of resources that a user can consume. For example, there may be a limit on the number of VMs of a particular type that can be created in each region environment, or a quota on the amount of storage space that can be used. These quotas and limits are put in place to prevent a single user from consuming too many resources and potentially disrupting the overall performance of the cloud infrastructure. It also protects users from inadvertently provisioning a huge quantity of resources that they don't really need but will have to pay for.

The cloud is in theory infinitely scalable and elastic. In practice, this is true only within the quotas and limits.

Let's look at some real-world examples in the next sections.

Real-world examples of quotas and limits

On GCP, quotas can generally be increased, while limits are fixed. Also, each service has its own page of quotas and limits. The **virtual private cloud** (**VPC**) page is found at `https://cloud.google.com/vpc/docs/quota`.

You can view the quotas in the GCP console as well as request increases: `https://console.cloud.google.com/iam-admin/quotas`.

There are currently 9,441 quotas!

Here is a screenshot that shows some quotas for the GCP Compute Engine service:

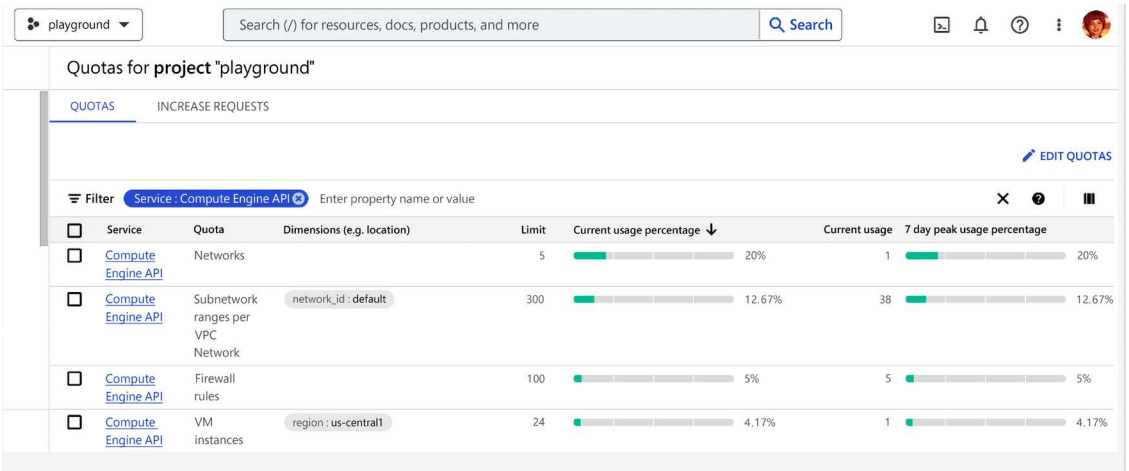


Figure 17.1: Screenshot of GCP compute engine quotas

Now that we understand what quotas and limits are, let's discuss capacity planning.

Capacity planning

In the olden days, capacity planning meant thinking about how many servers you needed in your data centers, how big the disks should be, and the bandwidth of your network. This was based on the usage of your workloads as well as keeping healthy headroom for redundancy as well as growth. Then, you had to consider upgrades and how to phase out obsolete hardware. In the cloud, you don't need to worry about hardware. However, you do need to plan around the quotas and limits. This means you need to monitor the quotas and limits and, whenever you get close to the current quota, request an increase. For quotas such as the number of VM instances of a particular VM family, I recommend staying below 50%-60% if possible. This should give you ample room for disaster recovery and growth, as well blue-green deployments where you run your new version and old version side by side for a while.

When should you not use Managed Kubernetes?

Managed Kubernetes is great, but it is not a panacea. There are several situations and use cases where you may prefer to manage Kubernetes yourself, such as:

- The obvious use case is if you run Kubernetes on-prem and a managed solution is simply not available. However, you can run a similar stack to cloud-managed Kubernetes via platforms like GKE Anthos, AWS Outposts, and Azure Arc.
- You require extreme control over the control plane, the node components, and the daemonsets that run on each node.
- You already have in-house expertise in running Kubernetes yourself and using Managed Kubernetes will require a steep learning curve and might cost more.

- You manage highly sensitive information that you must fully control and can't trust the cloud provider with.
- You run Kubernetes on multiple cloud providers and/or hybrid environments and prefer to have a uniform way to manage Kubernetes in all environments.
- You want to make sure you are not locked into a particular cloud provider.

In short, there are various situations where you may take on managing Kubernetes yourself. Let's look at the various ways you may deploy and manage multiple clusters of Kubernetes in different environments.

Managing multiple clusters

A Kubernetes cluster is powerful and can manage a lot of workloads (thousands of nodes, and hundreds of thousands of pods). As a startup, you may get pretty far with just one cluster. However, at enterprise scale, you'll need more than one cluster. Let's consider some use cases.

Geo-distributed clusters

Geo-distributed clusters are clusters that run in different locations. There are three main reasons for using geo-distributed clusters:

- Keeping your data and workloads close to their consumers.
- Compliance and data privacy laws where data must remain in its country of origin.
- High availability and disaster recovery in case of a regional outage.

Multi-cloud

If you run on multiple clouds, then naturally you need at least one cluster per cloud provider.

Running on multiple clouds can be complicated, but at enterprise scale, it may be unavoidable and sometimes desirable. For example, your company may run Kubernetes on cloud X and acquire a company that runs Kubernetes on cloud Y. Migrating from Y to X might be too risky and expensive.

Another valid reason to run on multiple clouds is to have leverage against the cloud providers to secure better discounts and ensure you are not fully locked in. Finally, it may allow you to tolerate a complete cloud provider outage (this is not trivial to pull off).

Hybrid

Hybrid Kubernetes means running some Kubernetes clusters in the cloud (with a single or multiple cloud providers) and some Kubernetes clusters on-prem.

This situation may arise as before because of an acquisition or even if you are in the process of migrating from on-prem Kubernetes to the cloud. Large systems might take years to migrate and during the migration, you'll have to run a mix of Kubernetes clusters running in a hybrid environment.

You may also adopt patterns like burst to cloud where most of your Kubernetes clusters run on-prem, but you have the flexibility to deploy workloads to Kubernetes clusters running in the cloud, which can scale quickly if you are hit with unanticipated load or if your on-prem infrastructure is having issues.

Kubernetes on the edge

Most enterprise data (around 90%) is generated in the cloud and private data centers; however, that number will drop to just 25% by 2025 according to Gartner.

This is mind-blowing. Edge computing (AKS IoT) will be responsible for this massive shift.

A lot of devices spread all over the place will generate constant streams of data. Some of that data will be sent back to the backend for processing, aggregation, and storage. However, it makes a lot of sense to perform various forms of data processing close to the data instead of sending the raw data as it is. In some cases, you can even run workloads locally close to the data and serve users completely on the edge.

This is the promise of edge computing. Running Kubernetes at the edge allows organizations to bring the processing of data closer to the source of data generation, reducing the latency and bandwidth requirements of sending data to a centralized data center or cloud. This results in improved response times and real-time processing of data, making it an ideal solution for use cases such as industrial IoT, autonomous vehicles, and other real-time data processing applications.

However, running Kubernetes at the edge comes with its own set of challenges. Edge devices are typically resource-constrained, making it necessary to optimize the deployment of Kubernetes for the edge environment. Organizations must also consider the network connectivity and reliability of edge devices, as well as the security and privacy implications of deploying a Kubernetes cluster at the edge.

Projects like CNCF KubeEdge (`https://kubeedge.io`) can get you started.

However, we will focus for the rest of this chapter on large-scale Kubernetes-based systems in the cloud.

Building effective processes for large-scale Kubernetes deployments

To run multi-cluster Kubernetes systems in production, you must develop a set of effective processes and best practices that encompass every aspect of the operation. Here are some of the critical areas to address.

The development lifecycle

The development lifecycle of a multi-cluster Kubernetes-based system in production can be a complex process, but it is possible to streamline it with the right approach.

You should absolutely implement a CI/CD pipeline that automatically builds, tests, and deploys code changes. This pipeline should be integrated with version control systems such as Git, and it should also include automated testing to ensure code quality.

It's important to manage the ownership and approval process for different areas of the code base.

Kubernetes namespaces can provide a convenient way to organize workloads and corresponding software assets and associate them with teams and stakeholders.

You should have a complete track of changes, ongoing builds, and deployments, and the ability to freeze activity and roll back changes of each workload.

It's also important to control the gradual deployment to different clusters and regions to avoid a situation where a bad change is deployed simultaneously across the board and brings the entire system down.

Environments

Code review and careful incremental deployment while monitoring the outcome are required, but they are insufficient for large enterprise systems with multiple Kubernetes clusters. Some changes might display a negative impact only after running for a while or under certain conditions, which will escape the control mechanisms we mentioned earlier. The best practice is to have multiple runtime environments such as production, staging, and development. The exact division of environments can vary, but you typically need at least a production environment, which is the actual system that manages all the data and your users interact with, and a staging environment, which mimics the production system and where you can test changes and new versions without impacting your users and risking bringing the production environment down.

Let's consider some aspects of using multiple environments.

Separated environments

It is critical that the staging environment can't accidentally contaminate and impact the production environment. For example, if you run a stress test in staging and some workloads in the staging environment are misconfigured and hit production endpoints, you will have a very unpleasant time untangling the mess.

Rigid network segmentation where the staging environment is unable to reach the production environment is a good first step. You will still need to be mindful of the interaction between staging and production through public endpoints. The staging workloads should not have secrets and identities that allow production access.

Staging environment fidelity

The primary motivation for the staging environment is to test changes and interactions with other sub-systems before deploying a change to production. This means that the staging environment should mimic the production environment as much as possible. However, running an exact replica of the production environment is prohibitively expensive.

The staging environment should be configured and set up using the same automated CI/CD pipeline that is able to deploy staging and production.

Staging infrastructure and resources should also be provisioned using the same tools as production although there will typically be fewer resources allocated to staging.

You may want to be able to temporarily scale down or even completely shut down some parts of the staging environment and be able to bring them back up only when necessary for running large-scale tests in staging.

Resource quotas

Resource quotas in staging and production can ensure that misconfiguration or even an attack doesn't cause excessive resource usage.

Promotion process

Once a change has been thoroughly tested in staging, there should be a clear promotion process for deploying it to production. The process may be different for different components depending on the scope and impact of the change. The promotion may be completely automatic where the CI/CD pipeline detects that staging tests are completed successfully and moves ahead with production deployment or involve extra steps and possibly another explicit deployment to production.

Permissions and access control

When you manage a constellation of Kubernetes clusters running on cloud infrastructure, you need to pay a lot of attention to the permission model and your access control. This builds on the previous best practices of the development lifecycle and environments.

The principle of least privilege

The principle of least privilege comes from the security field, but it is useful even beyond security for reliability, performance, and cost. Actors – either humans or workloads – should not have more permissions than necessary to accomplish their tasks. By reducing access to the bare minimum, you ensure that no accidental or malicious activity occurs for forbidden resources.

Also, when investigating an incident, it automatically narrows down the possible culprits to those who had the permissions to act on the misconfigured resource or take the invalid action.

If you follow the GitOps model, it is possible to create a workflow where every change to your clusters and your infrastructure is done by CI/CD and dedicated tooling. Human engineers have only read-only access. Some special exceptions can be made (see the *Break glass* section in this chapter).

Assign permissions to groups

It is highly recommended to assign permissions to groups or teams as opposed to individuals. Even if just a single person is currently carrying out a task that requires some set of permissions, you should define a group where this person is the single member. That will make it easier to add other people later or replace the person.

Fine-tune your permission model

However, sometimes too strict a permission model can be detrimental. You'll have to maintain a very fine-grained set of permissions to a large set of resources. Whenever the slightest change occurs such that another action is needed on some resource, you'll have to modify the permissions.

Find the golden path between granting super-admin permissions to everyone and painstakingly creating hundreds and thousands of roles for each and every resource.

In particular, consider relaxing the permission model in the development environment and potentially in the staging environment too. This is where a lot of experiments take place and where you discover what actions you need to perform and what permissions are actually necessary and then you can tweak your permissions model before deploying to production.

Break glass

Sometimes, your CI/CD pipeline itself will be broken or, due to incomplete coverage, some resources may have been provisioned manually. In these cases, human engineers must intervene and, so to speak, "*break the glass*" and take direct action against Kubernetes or cloud infrastructure.

It is recommended to have a formal process of acquiring **break glass** access, who is allowed to have it, how long it lasts, and record who had it.

This brings us to the next section about observability.

Observability

A comprehensive observability stack is an absolute must. Complex systems composed of multiple Kubernetes clusters can be reasoned about and understood theoretically. You must have a complete record of events from cloud providers, Kubernetes itself, and workloads. Your CI/CD pipeline and other tools must also be fully observable.

Let's look at some elements of multi-cluster observability.

One-stop shop observability

Cloud providers and Kubernetes itself provide a lot of observability in the form of logs and metrics out of the box. However, those are typically organized at the cluster level. If you are dealing with some widespread issue across multiple clusters, it is very difficult, and at a certain scale impossible, to go into each individual cluster, extract the observability data, and try to make sense of it. You must ship all observability data into a single centralized system where it can be aggregated, summarized, and be ready for multi-cluster analysis and response.

Troubleshooting your observability stack

Your observability stack is an indispensable component of your system. If it is down or degraded, you may be flying blind and unable to effectively respond to issues. Moreover, a cross-cluster issue may impact your observability stack as it impacts your entire system. Consider this scenario very carefully and make sure you have plenty of redundancies and observability alternatives if your primary observability stack is not up to the task temporarily. For example, you may rely on in-cluster observability solutions if your centralized observability stack is compromised. If you want complete redundancy, you may have a parallel observability stack on two cloud providers.

Consider testing these harsh scenarios of CI/CD pipeline and observability stack outages and see how you operate.

Let's look more specifically into different types of infrastructure and how to handle them at scale.

Handling infrastructure at scale

One of the most demanding tasks when running large-scale multi-cluster Kubernetes in the cloud is dealing with the cloud infrastructure. In some respects, it is much better than being responsible for low-level compute, network, and storage infrastructure. However, you lose a lot of control, and troubleshooting issues is challenging.

Before diving into each category of infrastructure, let's look at some general cloud-level considerations.

Cloud-level considerations

In the cloud, you organize your resources in entities such as AWS accounts, GCP projects, and Azure subscriptions. An organization may have multiple such groups, and each one has its own limits and quotas. For the sake of brevity, let's call them accounts. Enterprise organizations' infrastructure requirements will exceed the capacity of a single account. It's critical to decide how to break down your infrastructure into different accounts. One good heuristic is to separate environments – production, staging, and development – into separate accounts. Account-level isolation is beneficial for these environments. However, this may not be sufficient and within a single environment, you might need more resources than can fit in one account.

Having a solid account management strategy is key. Accounts can also be a boundary of access control as even account administrators can't access other accounts.

Consult with your security team about security-motivated account breakdowns.

Another important aspect is the breakdown of regions. If you manage infrastructure across multiple regions and workloads in these regions communicate with each other, then this has severe latency and cost implications. In particular, cross-region egress is typically not free.

Let's look at each category of infrastructure.

Compute

Compute infrastructure for Managed Kubernetes includes the Kubernetes clusters themselves and their worker nodes. The worker nodes are typically grouped into node pools, which is not a Kubernetes concept. How you break down your system into Kubernetes clusters and what types of node pools exist in each cluster will greatly impact your ability to manage the system at scale.

Ideally, you can treat clusters like cattle, provision identical clusters, and easily add or remove clusters in different locations. Each cluster will have the same node pools.

This uniform and consistent organization of clusters is not always possible. There are sometimes reasons to have different clusters for particular purposes. You should still strive for a small number of cluster types that can be replicated easily.

Design your cluster breakdown

Clusters in the cloud typically allocate private IP addresses to nodes and pods from a virtual network that resides in one region. Yes, it's possible to have wide clusters that cross regions, but this is the exception and not the rule. It is highly recommended to manage clusters as cattle if possible and automatically provision clusters across all regions of operations.

Design your node pool breakdown

Node pools are groups of nodes that have the same instance type. They can typically autoscale to accommodate the needs of the cluster with the help of the cluster autoscaler. How you choose what node pools to provision in your clusters is a fundamental decision that impacts performance, cost, and operational complexity.

We will dive into a deeper discussion on this later in the chapter.

Networking

Networking is a very dynamic area of infrastructure. There are many degrees of freedom. The interplay between latency and bandwidth has nuances. Workloads can't request a certain amount of bandwidth or guaranteed latency. In addition, there are a lot of external factors that impact the connectivity, reachability, and performance of your network. Let's look at some of the important topics we have to consider and plan for.

IP address space management

When you run a multi-cluster Kubernetes-based system, every pod in a cluster gets a unique private IP address. However, if you want to connect multiple clusters and have workloads in one cluster reach workloads in other clusters via their private IP address, then two conditions must exist:

1. The networks of the different clusters must be peered together.
2. The private IP address of pods must be unique across all clusters.

This requires centralized management of the private IP address space and carefully assigning IP ranges to different clusters.

Cloud providers differ a little in their approach to assigning IP address ranges to clusters. AKS requires that each cluster belongs to a VNet with its own IP address range and then subnets are assigned to nodes and pods with sub-ranges from the VNet IP address range. GKE comes with a default network that has no IP address range of its own. Clusters are provisioned with subnets that have their own IP address ranges.

In addition, services require their own IP addresses too, and possibly some other components.

The entire private IPv4 address space consists of several blocks:

- `10.0.0.0/8` (class A)
- `172.16.0.0/12` (class B)
- `192.168.0.0/8` (class C)

At scale, the most important one is `10.0.0.0/8`, which consists of 2^24 IP addresses, which is more than 16 million addresses. That is a lot of IP addresses, but if you don't plan carefully, you may cause fragmentation and run out of large blocks even if you have plenty of unused IP addresses.

Here are some best practices for managing the IP address space:

- Allocate CIDR blocks to pods, nodes, and services that will be sufficient and utilized without too much space.
- Be aware of Kubernetes and cloud provider limits in terms of the number of supported nodes and pods.
- Consider network peering and service meshes spanning clusters.
- Make sure you use non-overlapping CIDR blocks for connected clusters.
- Use proper tools to manage the address space.
- Consider the impact of pod density on IP address space (e.g., on AKS, IP addresses are pre-allocated to the max number of pods on a node even if not utilized).
- Be aware of limits such as the maximum number of IP addresses available in a region.

Network topology

All cloud providers offer a virtual network or VPC concept. All cloud providers also have the concept of a region, which is a geographical area where cloud providers host resources. Specifically, virtual networks are always confined to a single region. Since a Kubernetes cluster in the cloud is associated with a virtual network, it follows that a single Kubernetes cluster can't span more than one region. This has implications for availability and reliability. If you want to survive a regional outage, you need to run each critical workload across multiple clusters in different regions. Moreover, all these clusters typically need to be connected to each other. We will discuss this more in the *Cross-cluster communication* section that follows. However, as far as network topology goes, it may be better to have multiple clusters in the same region share the same virtual network.

Network segmentation

Network segmentation is about dividing a network into smaller subnets. In the context of Kubernetes, the most important subnets are the subnets for nodes, pods, and services. In some cases, the nodes and pods share the same subnet and in other cases, there are separate subnets for nodes and pods. Regardless, you need to plan and understand how many nodes and pods your cluster can accommodate and size your cluster subnets accordingly.

Cross-cluster communication

When running multiple Kubernetes clusters, it is often beneficial to consider groups of clusters as a single conceptual cluster. This means that pods in any cluster can directly reach pods in other clusters via their private IP address. This flat IP address model is an extension of the standard Kubernetes networking model within a single cluster to multiple clusters. This requires two elements we discussed earlier:

1. Non-conflicting IP address ranges for pods across all connected clusters.
2. Network peering between clusters.

The network peering requirement might be tedious as clusters come and go. Having a single regional virtual network reduces the overhead and allows peering just the regional virtual networks. However, if you run a lot of clusters in the same region sharing the same virtual network, you might run into cloud provider limits that will stunt your growth. For example, on Azure, a VNet can have at most 64K unique IP addresses.

Cross-cloud communication

If your system spans multiple cloud providers, you need to consider how to connect your Kubernetes clusters across cloud providers. There are several ways, with different pros and cons.

First, you may decide to avoid direct communication between clusters on different cloud providers. If Kubernetes clusters deployed on different clouds need to communicate with each other, they can do so through public APIs. This approach is simple but eliminates the idea of a unified conceptual cluster where pods can communicate directly with each other regardless of where they are.

A site-to-site VPN is a communication method where different cloud providers can connect systems via BGP and establish a VPN connection to networks managed by another cloud provider via a VPN gateway that sits in front of the virtual networks. This establishes a secure channel; however, a VPN gateway is not trivial to set up and incurs a significant overhead.

Direct connect (AKA direct peering) is another option that requires installing a router in a cloud provider point of presence. This method allows connecting Kubernetes clusters running in private data centers to clusters in the cloud. In addition, the performance is much better because there is no VPN gateway in the middle. The downside is that it is quite complicated to set up and you might have to comply with various requirements. It's a good option for organizations with deep low-level networking expertise.

Carrier or partner peering is similar to direct connect; however, you take advantage of the expertise of an established third party that specializes in providing this service and already has an established relationship and is certified with the cloud provider. You will have to pay for the service, of course.

Cross-cluster service meshes

Service meshes bring tremendous value to Kubernetes, as we discussed in *Chapter 14*, *Utilizing Service Meshes*. When running multiple Kubernetes clusters in production, it is arguably even more important to connect all the clusters via a service mesh. The advanced capabilities and policies of a service mesh can be applied and manage connectivity and routing across all clusters.

There are two approaches to consider here:

1. A single fully connected service mesh.
2. Divide your clusters into multiple meshes.

The single fully connected single mesh aligns conceptually with the single conceptual Kubernetes cluster approach. Everything is straightforward. New clusters just join the mesh, and the mesh is configured to allow every pod to talk to every other pod (as long as the routing policies allow it).

However, you might eventually run into scalability barriers as a single mesh means that the mesh control plane needs to handle policies for all workloads in all the clusters, and updating sidecars for all pods can put a lot of burden on your clusters.

An alternative approach is to have multiple independent meshes. Pods in clusters that belong to a particular mesh can directly talk to pods in all other clusters in the same service mesh but must go through public endpoints to access clusters in other service meshes.

The multi-mesh approach is more scalable but much more complicated. You need to consider how to divide your system into different service meshes and when new clusters join or leave the system, and how it impacts your overall architecture.

The private IP address space management in the multi-service mesh case can be more nuanced too where different service meshes can have conflicting IP addresses. This means that you can manage the IP address space for each mesh separately.

Service meshes offer another interesting solution to the cross-cluster connectivity story, which is the east-west gateway. With the east-west gateway approach, workloads in different clusters communicate indirectly through dedicated gateways in each cluster. This means that the private IP addresses of each cluster are unknown and there is an extra hop for each cross-cluster communication.

Managing egress at scale

Some systems need to access external systems aggressively. Maybe you frequently fetch data from external systems or maybe the purpose of your system is to manage some external systems via APIs.

There may be unique issues for egress traffic that require special attention. Some third-party organizations or even countries may have policies that block or throttle traffic coming from certain geographical areas or specific IP CIDR blocks. For example, China and its great firewall are famous for blocking and censoring a vast number of companies, such as Google and Facebook. If you run on GCP and need to access China, it might be a serious issue.

Beyond total blocking, there may be limits and throttling in place if you try to access some third-party APIs at scale.

If you persist in accessing those third-party APIs, you could even be reported, and your cloud provider could potentially impose various sanctions.

Let's consider some solutions to deal with these real-world problems.

If your current cloud provider is not allowed to access your target destination, then you must establish an egress presence outside your cloud provider. This can be on another cloud provider or via an intermediate organization in good standing. This proxy approach can take many shapes, which is beyond the scope of this section.

If you are getting throttled, then the issue may be that you send too many requests from the same source IP address. A good solution here is to create a pool of egress nodes with different public IP addresses and distribute your requests through multiple different IP addresses. It can also help if you rotate your public IP addresses periodically, which is pretty easy in the cloud by just re-creating instances, which receive new public IP addresses.

The opposite issue is if you have an agreement with some third-party company and they specifically allow traffic from your organization by whitelisting some IP addresses you provide. In this case, you need to manage static public IP addresses that don't change and ensure that all requests to that third-party organization go out through the whitelisted IP addresses.

Finally, to address the risk of being reported and flagged by your cloud provider, you may need to isolate your egress access to a separate account. Most cloud provider sanctions are at the account level. If your egress account is disabled by your cloud provider, at least it will not bring down the entire enterprise.

Managing the DNS at the cluster level

Large-scale clusters with lots of pods and services may put a high load on CoreDNS, which is the internal DNS server of Kubernetes. It's important to ensure sufficient DNS capacity since most internal communication between workloads in the cluster uses DNS names for addressing and not direct IP addresses.

It is recommended to use DNS autoscaling, which is often not enabled by default. See `https://kubernetes.io/docs/tasks/administer-cluster/dns-horizontal-autoscaling/` for more details.

Storage

Storage is arguably the most critical element of your infrastructure. This is where your persistent data lives, which is the long-term memory of organizations.

Choose the right storage solutions

There are many storage solutions available for Kubernetes clusters in the cloud, such as cloud-native blob storage, managed storage services, managed databases, and managed file systems. You should develop a deep understanding of the performance, durability, and cost of each storage solution and match them against your storage use cases.

Your baseline should always be cloud-native blob storage (AKA buckets) like AWS S3, GCP Google Cloud Storage, or Azure Blob Storage. It's hard to imagine a large-scale Managed Kubernetes enterprise that doesn't use buckets.

Then, consider more structured or high-level storage solutions. If you aim to stay cloud-agnostic, you may ignore cloud-based managed storage solutions and deploy your own solutions, as we saw in *Chapter 6*, *Managing Storage*.

At an enterprise scale, it may be worthwhile considering different levels of access speed and cost for data at different levels of importance.

Data backup and recovery

Plan for data backup and recovery. Your data is valuable. Data backup and recovery are crucial for production environments. Consider implementing data backup and recovery processes that are reliable and scalable, and make sure they are regularly tested and updated.

You should also consider data retention policies and not automatically assume that all data must be kept forever.

Of course, to comply with data privacy laws and regulations like the GDPR, you will need to build the ability to selectively delete data too.

Storage monitoring

Set up storage monitoring. Period. Monitoring storage performance, usage, and capacity is essential for identifying and resolving issues before they impact the availability or performance of your applications. Set up monitoring and alerting for storage utilization, latency, and throughput. This is important for managed storage, but also for node storage where logs can easily accumulate and render a node un-operational.

Data security

Implement data security measures. Protecting sensitive data and ensuring compliance with data protection regulations is critical for production environments. Implement access controls, encryption, and data security policies to safeguard your data.

Optimize storage usage

Kubernetes clusters in the cloud can be expensive, and storage costs can add up quickly. Optimize your storage usage by deleting unused data, using data compression or deduplication, and setting up storage tiering.

Test and validate storage performance

Before deploying applications in a production environment, test and validate the performance of your storage solution to ensure it meets the performance requirements of your workloads.

By considering these factors and implementing best practices for managing storage in production for Kubernetes clusters in the cloud, you can ensure reliable and scalable storage performance for your applications.

Now that we have covered a lot of guidelines and best practices for managing cloud infrastructure at scale for Kubernetes, let's shine a spotlight on the management of clusters and node pools, which is at the heart of managing multi-cluster Kubernetes in production.

Managing clusters and node pools

Managing your clusters and node pools is the top infrastructure administration activity for a large-scale Kubernetes-based enterprise. In this section, we will look at several crucial aspects, including provisioning, bin packing and utilization, upgrades, troubleshooting, and cost management.

Provisioning managed clusters and node pools

There are different methods for provisioning clusters and node pools. You should choose the method that is best for your use case wisely because failure here can result in devastating outages. Let's review some options. All cloud providers offer cluster and node pool provisioning via APIs, CLIs, and UIs. I highly recommend avoiding directly using any of these methods and instead using GitOps-based declarative approaches. Here are some solid options to consider.

The Cluster API

The Cluster API is an open-source project from the Cluster Lifecycle SIG. Its goal is to make provisioning, upgrading, and operating multiple Kubernetes clusters easier. It is focused on clusters' and node pools' lifecycles. However, it started mostly as a way to provision clusters using kubeadm. Managed cluster support on different cloud providers was added later, and it is still young. In particular, GKE is not supported (although you can provision Kubernetes clusters on GCP as an infrastructure layer). AKS and EKS are supported.

The Cluster API has a lot of momentum, and if you don't operate GKE, you should definitely look into it.

Terraform/Pulumi

Terraform and Pulumi are similar in their approach. They can provision clusters and node pools on all cloud providers. However, these tools on their own can't respond to out-of-band changes and don't monitor the state of the infrastructure after provisioning. Their internal state can deviate from the real world and that can cause difficult to recover from situations that require careful "surgery." In particular, node pools often need to be provisioned or updated, and Terraform or Pulumi might not be up to the task. If you have a lot of experience with these tools and are aware of their quirks and special requirements, they may still be a good option for you.

Kubernetes operators

Another alternative is to use Kubernetes operators that reconcile CRDs with cluster and node pool specs with the cloud provider. Under the covers, the operator will invoke Managed Kubernetes APIs from the cloud provider. This requires non-trivial work and expertise in writing Kubernetes operators but gives you the ultimate control.

You may try to use Crossplane instead of writing your own operator; however, Crossplane support seems pretty basic and incomplete at the moment. One option to expand the scope is to use the Upjet project (`https://github.com/upbound/upjet`) to generate Crossplane controllers from Terraform providers.

Utilizing managed nodes

You can also try to use managed nodes, so you never need to deal with provisioning node pools and nodes directly. All cloud providers offer some form of managed nodes such as GKE AutoPilot, EKS + Fargate, and AKS + ACI. For enterprise use cases, I believe you will need more control than fully managed node pools provide. It may be a good option for a subset of your workloads. However, at scale, you will want to optimize your resource usage and performance and the limitations of managed node pools might be too severe.

Once you have figured out how to provision and manage your clusters and node pools, you should turn your attention to effectively using the resources you provisioned.

Bin packing and utilization

Cloud resources are expensive. Efficient usage of resources on Kubernetes has two parts: efficiently scheduling pods to nodes based on their resource requests, and pods actually using the resources they requested.

Bin packing means ensuring that the total sum of resource requests is as close as possible to the allocatable resources on the target node. Once a workload is scheduled to a node, it will not be evicted under normal conditions even if the node is highly underutilized, but components like the cluster autoscaler can help here.

Resource utilization measures what percentage of the requested resource is actually used. Resource utilization is in general not fixed as the resource usage of workloads may vary widely throughout their lifetimes.

There are a lot of nuances to bin packing, resource utilization, and the interplay between them. For example, there are different resources such as CPU, memory, disk, and network. A node may have 100% bin packing for CPU, but only 20% bin packing memory. Network and non-ephemeral disks on the node are shared resources that pods can request to ensure they will always have a certain amount. This complicates operation and reliability. Let's discuss some principles and concepts that can assist in navigating this complicated topic.

Understanding workload shape

Workload shape is the ratio between the workload CPU requests and its memory requests. In the cloud, there is a standard ratio of 1 CPU to 4 GiB of memory. As a result, most VM types that you can choose offer resource capacities with this ratio. Some workloads need more memory or more CPU than this ratio. All cloud providers also offer high-memory VM types with a ratio of 1 CPU to 8 GiB of memory as well as high-CPU VM types with 1 CPU to 2 GiB of memory.

Understanding the resource shape of your workloads is necessary to inform the VM types you choose to optimize your resource usage.

For example, if a workload requires 1 CPU and 8 GiB of memory and you schedule it on a VM type with a ratio of 1:4, you will need to run it on a node that has 2 CPUs and 8 GiB of memory. No other pod can run on this node since the original workload uses all the 8 GiB of memory. However, 1 CPU out of 2 is not used at all. It would have been much better to schedule the workload on a node with a VM type of 1:8 ratio, which ensures optimal bin packing.

Setting requests and limits

Setting requests and limits for your workloads is a key for proper resource utilization. As you recall when you set requests for your pod's containers, it will be scheduled to a node that has at least the requested number of resources available for the total sum of the requests of all containers. The requested resources are allocated for the exclusive use of each container for as long as the pod running on the node. The containers may use more resources than the requests if available. If you specify CPU limits and the container tries to use more CPU than the limit, then the pod may get throttled. If you specify a memory limit and the container tries to use more memory than the limit, then the container will be OOMKilled and restarted.

It is best practice to set resource requests for CPU, memory, and even ephemeral storage if the container uses any. How do you know how much to request? You can start with a rough estimate and monitor the actual resource usage over time and fine-tune it later.

But even this straightforward method has some subtleties. Suppose a workload uses between 2 CPUs and 4 CPUs with an average of 3 CPUs. Should you request 4 CPUs and know for sure that the workload will never get throttled? But then, you waste a whole CPU because the average usage is just 3 CPUs. If you request 3 CPUs, are you going to get throttled every time the workload needs more than 3 CPUs? That depends on the available CPU on the node the pod is scheduled to. If the overall CPU on the node is saturated because all the pods need a lot of CPU, then it is possible.

On top of plain requests, you can also assign priorities to workloads, which allow you to control the destiny of high-priority workloads and ensure they take precedence over non-prioritized or low-priority workloads.

Yes, scheduling is far from trivial. If you need a refresher, check out the *Understanding the design of the Kubernetes scheduler* section in *Chapter 15*, *Extending Kubernetes*.

Let's turn our attention to limits. A simple approach is to set limits equal to the requests. This ensures in general that containers will not use more resources than they requested, which makes bin packing easy. However, in real-world situations, the resource usage of workloads varies. It is often more economical to request less than the maximal usage or sometimes even less than the average usage. In this case, you may opt not to set limits at all or set the limits higher than the requests. For example, if a workload uses 1 to 4 CPUs, then you may decide to request 2 CPUs and set the limits to 4 CPUs. Requesting just 2 CPUs will allow packing more pods into the same node or schedule the pod into a smaller node. So, why set limits at all? Well, setting some limits ensures the pods don't get out of control, hog all the CPU, and starve all other pods that may also set lower requests but actually need additional CPU.

Setting memory high limits is even more important, especially for workloads that are more sensitive and that shouldn't be restarted often since any attempt to use more allocated memory than the limit will result in a container restart.

Utilizing init containers

Some workloads need to do a lot of work when they just start and then their resource requirements are lower. For example, a workload needs 10 GiB of memory and 4 CPUs to fetch some data and process it in memory before it is ready to handle requests. However, once it's running, it doesn't need more than 1 CPU and 4 GiB. It would be pretty wasteful to request 4 CPUs and 10 GiB if the pod is a long-running one. This is where init containers are very useful. You can split your workload into two containers. All the initialization work that requires 4 CPUs and 10 GiB can be done by an init container and then the main container can request just 1 CPU and 4 GiB.

Shared nodes vs. dedicated nodes

When designing your node pools, you have two fundamental choices to make. Shared node pools have multiple different workloads scheduled and run side by side on the same node. Dedicated node pools have a single workload taking over a single node (possibly multiple instances of the same workload).

Shared node pools are simple. The extreme case is that you have just a single node pool and all pods are scheduled to nodes from this node pool. If you have multiple shared node pools (e.g., one with regular nodes and one with spot instances), then you need to assign taints to node pools and tolerations to workloads as well as dealing with node and pod affinity.

Since you don't know exactly which combination of pods will end up on which node, there might be inefficiencies with bin packing. However, as long as the overall average workload shape matches the resource ratio of your nodes, bin packing at a large scale should be close to optimal.

Workloads can request the CPU, memory, and ephemeral storage they need. However, there are some shared resources on the node, like network and disk I/O, that you can't easily carve out for your workload when other workloads on the same node might try to use the same resource.

This is where dedicated node pools come in. Critical workloads like databases or event queues require predictable network and disk I/O. Scheduling such workloads on a dedicated node ensures the workload doesn't have to worry about other workloads cannibalizing the shared resources.

It makes sense in this case for the workload to request more than 50% of the standard resources like CPU or memory to ensure exactly one pod of the critical workload is scheduled on the node.

Remember that system daemons will also run on the node and have higher priority. If your dedicated workload requests too many resources, it might become unschedulable.

I have run into this issue after an upgrade where the daemonsets on the node required more resources and caused the dedicated workload to be unschedulable until it reduced its resource requests.

Large nodes vs, small nodes

In the cloud, nodes come in a variety of sizes, from 1 core to tens or even hundreds of cores. Should you have lots of small nodes or fewer large nodes?

First and foremost, you must have nodes that your largest workloads fit into. For example, if a workload requests 8 CPUs, then you must have a node with at least 8 CPUs available.

But what about much bigger nodes? There are advantages in terms of efficiency for larger nodes. In the cloud, when you provision (for example) a node with 1 CPU core and 4 GIB of memory, you don't really get all these resources. First, the OS, the container runtime, and kube-proxy take their resources, then the additional processes the cloud provider decides to run on each node, then various sys daemonsets and your own daemonsets. Finally, what's left is available for your workloads. All these processes and workloads that always run on every node need a lot of resources. However, the resources they require are not proportional to the size of the node. This means that on small nodes, a much smaller percentage of the resources you pay for will be available for your pods. Let's look at an example.

Here is the resource breakdown for a real node running on an AKS production cluster. It has a VM type of **Standard_F2s_v2** (`https://learn.microsoft.com/en-us/azure/virtual-machines/fsv2-series`). It has 2 CPUs and 4 GIB of memory. However, the allocatable CPU and memory is 1.9 CPU and 2.1 GiB. Yes, this is correct. You barely get a little more than 50% of the memory available on the node:

```
Capacity:
  cpu:                2
  memory:             4019488Ki
Allocatable:
  cpu:                1900m
  memory:             2202912Ki
```

But the story doesn't end here. There are system daemonsets running in kube-system. You can find them with the following command:

```
$ k get po --field-selector spec.nodeName=<node-name> -n kube-system
```

Let's look at the resources requested by these workloads on our node:

```
  Namespace     Name                          CPU Requests  Memory Requests
  ---------     ----                          ------------  ---------------
  kube-system   azure-ip-masq-agent-8pkqx     100m (5%)     50Mi (2%)
  kube-system   azure-npm-twjlx               250m (13%)    300Mi (13%)
  kube-system   cloud-node-manager-fv5gs      50m (2%)      50Mi (2%)
  kube-system   csi-azuredisk-node-kqnn7      30m (1%)      60Mi (2%)
  kube-system   csi-azurefile-node-h8zpw      30m (1%)      60Mi (2%)
  kube-system   kube-proxy-lgzcf              100m (5%)     0 (0%)
```

That's a total of 0.56 CPU and 520Mi of memory. If we subtract it from the allocatable CPU and memory, we end up with 1.4 CPU and 1.58 GiB of memory.

This is quite eye-opening. On a small node with 2 CPUs and 4 GIB of memory, we end up with 70% of the CPU and less than 40% of the memory. Beyond the cost implications, if you miscalculate and assume you can schedule, for example, a pod that requests 2 GIB of memory on a 4 GiB node, you'll have an unpleasant surprise when your pod remains pending because it doesn't fit on this node.

Let's look at large nodes. A Standard_D64ads_v5 Azure VM has a whopping 64 cores and 256 GiB of memory. It is undoubtedly a beast. Let's look at its capacity and allocatable resources:

```
Capacity:
  cpu:                64
  memory:             263932684Ki
Allocatable:
  cpu:                63260m
  memory:             250707724Ki
```

Here, we lost 740 mcpu (as opposed to 100 mcpu on the small node) and 17 GiB of memory. This sounds like a lot, but proportionally, it is much better. Let's look at system workloads to get the full picture:

```
Namespace     Name                        CPU Requests  Memory Requests
---------     ----                        ------------  ---------------
kube-system   azure-ip-masq-agent-lgzxq   50m (0%)      50Mi (0%)
kube-system   azure-npm-s5gsd             100m (0%)     300Mi (0%)
kube-system   cloud-node-manager-jntvt    10m (0%)      50Mi (0%)
kube-system   csi-azuredisk-node-srcfg    30m (0%)      60Mi (0%)
kube-system   csi-azurefile-node-gx247    75m (0%)      60Mi (0%)
kube-system   kube-proxy-xgppg            100m (0%)     0 (0%)
```

That's a total of 0.365 CPU and 520Mi of memory. Surprisingly, less CPU is requested than the small node and the memory requests are the same. If we subtract it from the allocatable CPU and memory, we end up with 62.9 CPU and 238.48 GiB of memory.

On a large node with 64 cores and 256 GiB of memory, we end up with more than 98% of the CPU and more than 93% of the memory.

This is a pretty clear victory for large nodes in terms of resource provisioning efficiency and getting more resources available for your workloads.

However, there are additional nuances and considerations to consider. Let's consider small and short-lived workloads.

Small and short-lived workloads

Suppose we use large nodes, and our cluster is bin-packed very efficiently. Some deployment needs to scale up and create a new pod. If there is no room for the new pod in any of the existing nodes, then a new node must be provisioned. However, if the new pod is small, then we actually waste a lot of resources by running just one small pod on a large node. At scale, and when a lot of pods come and go pretty quickly, this may not be a problem. However, consider the following scenario – our cluster is normally running on 100 large nodes. During a temporary spike of activity, our clusters scaled up to 200 large nodes and then the activity went back to normal. Our resource utilization is now 50% (the cluster needs 100 nodes out of 200). In an ideal world, the cluster autoscaler will eventually scale down empty nodes until we have 100 properly bin-packed nodes. But, in the real world, especially in the presence of small short-lived pods, new pods may get scheduled arbitrarily to all 200 nodes and the autoscaler might have a difficult time scaling down. We will see later, in the custom scheduling section, some options.

Another issue with short-lived workloads is that even if they have room on an existing node, they can still waste resources if they take a while to get ready. Consider a pod that takes 1 minute to get ready and runs, on average, for 1 minute. This pod with optimal utilization of its resources still can do better than 50% because after it gets scheduled, it reserves its resources on the node for 2 minutes, but actually does work for only 1 minute. If the pod needs to pull its image, then it can easily take several minutes to get ready.

The Kubernetes scheduler is very sophisticated and can be extended too, as we covered in *Chapter 15*, *Extending Kubernetes*. The issues with the inefficient scheduling of pods in the different use cases we mentioned could potentially be addressed by choosing a different scoring strategy. The default scoring strategy of **RequestedToCapacityRatio** is intended to evenly distribute workloads across all nodes. This is not ideal for tight bin packing. The **MostAllocated** scoring strategy may be preferable here.

Check out `https://kubernetes.io/docs/concepts/scheduling-eviction/resource-bin-packing` for more details.

Pod density

Pod density is the maximum number of pods per node (the Kubernetes default is 110). As mentioned earlier, some resources like private IP addresses or system daemon CPU and memory may be correlated with the pod density. If your pod density is too high, then you may waste the resources that were pre-allocated to support many pods on each node. However, if you set the pod density too low, then you may not be able to schedule enough pods to run on the node. Let's consider a large node with 64 CPU cores and 256 GiB of memory. If the pod density is 100, then at most 100 pods can run on this node. Suppose we have a lot of small pods that use only 10 mcpu and 100 MiB of memory. 100 pods need only 10 CPU cores and 10 GiB of memory combined. If 100 such pods get scheduled to one large node, the node will be highly underutilized. 54 CPU cores and 246 GIB of memory will be wasted.

If you go with the shared node pool model, then it's an arbitrary mix of pods with different workload shapes, and resource requirements can get scheduled to nodes.

Fallback node pools

Cloud providers suffer from temporary capacity issues from time to time, and as a result, are unable to provision new nodes. In addition, spot instances may disappear at any time if there is a lot of demand for regular nodes. The good news is that these outages are a zonal affair and also are typically limited to a specific instance type or VM family.

A good strategy to address this issue is to use fallback node pools.

A fallback node pool is an empty node pool with autoscaling disabled that has the same labels and taints as another active node pool but with a different VM family or a different node type (e.g., regular vs. spot). If the active node pool is unable to provision more nodes and there are pending pods, then the back node pool can be resized and/or become auto-scaling. This will allow the pending pods to be scheduled to the backup node pool until the situation with the native node pool is resolved.

If you choose this path, you need to come up with a proper procedure to activate the backup node pool, which includes detection of issues in the active node pool, a manual or automated process for backup node pool activation, and a scale-back process when the active node pool is back to normal.

It is very important to ensure the backup node pool has enough quota to replace its active node pool when needed.

This was a very thorough treatment of bin packing and resource utilization. Let's turn our attention to upgrades.

Upgrading Kubernetes

Upgrading Kubernetes can be a very stressful operation. A hasty upgrade might remove support for resource versions, and if you have unsupported resources deployed, you will encounter catastrophic failures. Using Managed Kubernetes has its pros and cons. When it comes to upgrades, there is, at any point in time, a range of supported versions.

You may upgrade to more recent supported versions. However, if you delay and neglect to upgrade, then the cloud provider will upgrade your clusters and node pools automatically once you fall behind the cutting edge of supported versions. Let's look at the various elements of upgrading Kubernetes you must be on top of.

Know the lifecycle of your cloud provider

Cloud providers can't support just any Kubernetes version in existence. It is critical to know when the current version of your clusters and node pools is going to be defunct. All cloud providers have a methodical process and share the information broadly. Here are the locations for each of the three major cloud providers:

- AWS EKS: `https://docs.aws.amazon.com/eks/latest/userguide/kubernetes-versions.html`
- Azure AKS: `https://learn.microsoft.com/en-us/azure/aks/supported-kubernetes-versions`
- Google GKE: `https://cloud.google.com/kubernetes-engine/docs/release-schedule`

For example, AKS, at the time of writing, supports versions 1.23 through 1.26. In addition, each version has an official end-of-life date. For example, the end-of-life date for 1.23 is April 2023. If your cluster is still on 1.23, then AKS may upgrade your cluster automatically to version 1.24. The process of cloud provider upgrade is gradual, done region by region, and might take several weeks.

All cloud providers offer an API and CLI to check the exact list of versions (including patch versions) in every region.

For example, at the moment, these are versions supported on AKS for the Central US region:

```
$ az aks get-versions --location centralus --output table
KubernetesVersion    Upgrades
-------------------  ----------------------
1.26.0(preview)      None available
1.25.5               1.26.0(preview)
1.25.4               1.25.5, 1.26.0(preview)
1.24.9               1.25.4, 1.25.5
1.24.6               1.24.9, 1.25.4, 1.25.5
1.23.15              1.24.6, 1.24.9
1.23.12              1.23.15, 1.24.6, 1.24.9
```

As you can see, for each minor version, there are several patch versions. It is even nice enough to mention which versions you may upgrade to yourself. Due to security concerns, the cloud provider may drop support for patch versions at any time.

Let's talk about the upgrade process of the control plane.

Upgrading clusters

When using Managed Kubernetes in the cloud, you are not responsible for the operation of the control plane, but you still need to manage the upgrade process. You have two options:

- Auto upgrade
- Manual upgrade

In an auto upgrade, the cloud provider will update your cluster according to their schedule, but you still must ensure that the versions of resources in your cluster are compatible with the new version. A manual upgrade requires you to upgrade yourself but gives you more control over timing. For example, you may choose to update earlier to benefit from some new features.

Remember that a manual upgrade doesn't mean you can stay on the same Kubernetes version forever. The cloud provider will forcefully upgrade you if you fall behind the minimal supported version.

Kubernetes releases a new version roughly every 3 months. Cloud providers support roughly 4 versions. This means that if you just upgraded to the latest supported version, you may hold off for about a year on upgrades, but then you will be on the minimal supported version, which means you will now have to upgrade every 3 months.

Note that you should upgrade the control plane one minor version at a time. If you are on version 1.24, and you want to upgrade to 1.26, you have to upgrade to 1.25 first and then from 1.25 to 1.26.

The bottom line is that upgrading the Kubernetes control plane is a standard operation that takes place multiple times per year. You should have a streamlined process for it.

Let's look at what is involved.

Planning an upgrade

You should plan your upgrades and coordinate them with cluster users. Control plane upgrades typically take 20-45 minutes. This is a non-disruptive operation for your workloads. Your workloads will keep running, and new pods will be scheduled to existing nodes. However, node pool operations might be blocked during the control plane upgrade.

If you're running multiple clusters with a redundancy scheme, it is best to perform the upgrades gradually and start with non-critical clusters (e.g., a development or staging environment).

I recommend having owners (engineers or teams) for every namespace. Notify all owners about upcoming upgrades so they can reserve time for converting incompatible resources.

Detecting incompatible resources

The main concern with an upgrade is that the functionality of your system will be compromised or completely broken because it uses resources that are not supported anymore. In most cases, a specific version of a resource will be removed, and a newer version will be available.

But you don't have to wait until the last minute to scramble and replace removed resources or versions. Kubernetes has a deprecation policy and resources will be deprecated for several versions before they are fully removed. I suggest making sure before each upgrade that all deprecated resources are updated or replaced. This will ensure that the upgrade process is not stressful because, even if you didn't manage to update all resources, the deprecated resources are still going to be supported by the new version and you will have some extra time to update them before they are fully removed.

Kubernetes publishes a migration guide with details about deprecated and removed APIs in each version. See `https://kubernetes.io/docs/reference/using-api/deprecation-guide`.

For example, Kubernetes 1.25 stopped serving the `CronJob` resource with the API version of `batch/v1beta1`. Instead, the `batch/v1` `CronJob` resource has been available since Kubernetes 1.21. Ideally, after you upgrade to Kubernetes 1.21, you have updated all your `CronJob` resources to use `batch/v1`, and by the time you upgrade to Kubernetes 1.25, the fact that `batch/v1beta1` is removed is not an issue because you are already on the supported version.

There are several ways to make sure you detect all deprecated and/or removed resources that you currently use. You can use the manual method of reading the deprecation guide and just scanning your code and detecting incompatible resources. Most releases don't have a lot of deprecations or removals. However, some releases may have up to ten different resources that are being deprecated or removed. For example, Kubernetes 1.25 stopped serving seven different resource versions.

A more systematic way is to use a tool like `kube-no-trouble` (`https://github.com/doitintl/kube-no-trouble`), which scans your clusters and can output a list of deprecated resources.

Here is how to install it:

```
$ sh -c "$(curl -sSL 'https://git.io/install-kubent')"
>>> kubent installation script <<<
> Detecting latest version
> Downloading version 0.7.0
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100 11.7M  100 11.7M    0     0  4154k      0  0:00:02  0:00:02 --:--:-- 8738k
> Done. kubent was installed to /usr/local/bin/.
```

I have a 1.25 cluster that doesn't contain any deprecated resources at the moment. However, in Kubernetes 1.26, the **HorizontalPodAutoscaler** of version autoscaling/v2beta2 will be removed as it has been deprecated since Kubernetes 1.23. Let's create such a resource. There is a `kyverno` deployment in the cluster:

```
$ k get deploy -n kyverno
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
kyverno   1/1     1            1           64d
```

Here is an HPA that sets the min replicas to 1 and the max replicas to 3:

```
$ cat <<EOF | k apply -f -
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: kyverno
  namespace: kyverno
spec:
  maxReplicas: 3
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 80
        type: Utilization
    type: Resource
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: kyverno
EOF

Warning: autoscaling/v2beta2 HorizontalPodAutoscaler is deprecated in v1.23+,
unavailable in v1.26+; use autoscaling/v2 HorizontalPodAutoscaler
horizontalpodautoscaler.autoscaling/kyverno created
```

As you can see, kubectl gives a very nice warning when you create a deprecated resource version that tells when the resource was deprecated (1.23), when it will be removed (1.26), and which version to replace it with (autoscaling/v2).

This is nice, but it is not sufficient. You probably create your resources through CI/CD, which might not receive the same warning, and even if it does, might not surface it, because it is not an error. However, if you created the HPA when your cluster was on an earlier version of Kubernetes than 1.23, then you wouldn't get any warning because at the time it wasn't deprecated.

Let's see if kubent can detect the deprecated HPA:

```
$ kubent
6:27AM INF >>> Kube No Trouble `kubent` <<<
6:27AM INF version 0.7.0 (git sha d1bb4e5fd6550b533b2013671aa8419d923ee042)
6:27AM INF Initializing collectors and retrieving data
6:27AM INF Target K8s version is 1.25.3
```

```
6:28AM INF Retrieved 7 resources from collector name=Cluster
6:28AM INF Retrieved 109 resources from collector name="Helm v3"
6:28AM INF Loaded ruleset name=custom.rego.tmpl
6:28AM INF Loaded ruleset name=deprecated-1-16.rego
6:28AM INF Loaded ruleset name=deprecated-1-22.rego
6:28AM INF Loaded ruleset name=deprecated-1-25.rego
6:28AM INF Loaded ruleset name=deprecated-1-26.rego
6:28AM INF Loaded ruleset name=deprecated-future.rego

_____
>>> Deprecated APIs removed in 1.26 <<<
------------------------------------------------------------------------------
------
KIND                      NAMESPACE   NAME      API_VERSION           REPLACE_WITH
(SINCE)
HorizontalPodAutoscaler   kyverno     kyverno   autoscaling/v2beta2   autoscaling/v2
(1.23.0)
```

Yes, it does. You get the same information: when it will be removed, when it was deprecated, and what to replace it with.

Updating incompatible resources

Updating an incompatible resource may require some changes to your manifests. If the API change just adds new fields, then you may just change the API version and be done with it. However, sometimes it may require additional changes.

OK. We're about to upgrade our cluster, and we detected some incompatible resources. Kubectl and the kubectl-convert plugin can help here. Follow the instructions here to install the plugin: `https://kubernetes.io/docs/tasks/tools/#kubectl`. Let's convert our HPA manifest and see what it looks like:

```
$ k convert -f hpa.yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  creationTimestamp: null
  name: kyverno
  namespace: kyverno
spec:
  maxReplicas: 3
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: kyverno
```

```
  targetCPUUtilizationPercentage: 80
status:
  currentReplicas: 0
  desiredReplicas: 0
```

The conversion succeeded but created a few unnecessary fields. The `creationTimestamp: null` is useless as it will be updated on a live resource. Also, the status is useless as this is just a manifest file, and the status will be updated at runtime.

However, the main differences are that `apiVersion` was changed to `apiVersion: autoscaling/v1` and that the target CPU percentage is now specified as a single field:

```
targetCPUUtilizationPercentage: 80
in autoscaling/v1beta1 it was specified as:
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 80
        type: Utilization
```

Using kubectl-convert saves time, and it is a well-tested tool.

Dealing with removed features

There is one other situation we need to address, which is the complete removal of a feature without an upgrade path. Kubernetes 1.25 completely removed support for **Pod Security Policies** (**PSPS**). The application of PSPs to pods has caused confusion for many users who have attempted to utilize them. Check this link for more details: `https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/`.

If you used PSPs, then when the time comes to upgrade to Kubernetes 1.25, your PSPs will no longer work. The Kubernetes developers didn't just remove the feature with no alternative. There are two alternatives to PSPs:

- Pod security admission
- A third-party admission controller

The pod security admin is a simplified solution that may or may not be a complete replacement for PSPs. The Kubernetes developers published a detailed guide for migration. Check it out: `https://kubernetes.io/docs/tasks/configure-pod-container/migrate-from-psp/`.

If you choose a third party (e.g., Kyverno), then you should check its documentation. Kyverno comes with a lot of sample policies for pod security and the transition is pretty straightforward.

Upgrading node pools

Upgrading the node pools of multiple clusters can be a major undertaking. If you have tens to hundreds of clusters and in each cluster, you have multiple node pools (5-20 is not uncommon), then be ready for a serious adventure. Control plane upgrades (once you have ensured your workloads are compatible with the new version) are pretty quick and painless. Node pool upgrades are very difficult. Realistically, it can take several weeks to upgrade all the node pools in a large Kubernetes-based system with tens to hundreds of node pools with thousands of nodes.

Syncing with control plane upgrades

Kubernetes imposes constraints on the versions of the control plane and the worker nodes. The Kubernetes node components may be two minor versions behind the control plane. If the control plane is on version N, then the node pools may be on version N-2. Since node pool upgrades are much more disruptive and labor-intensive than control plane upgrades, I recommend upgrading node pools only to every other version of Kubernetes. For example, suppose we start with a Kubernetes cluster where both the control plane and the nodes are on version 1.24. When we upgrade to version 1.25, we upgrade only the control plane to 1.25 and keep the node pools on 1.24, which is compatible. Then, when it's time to upgrade to 1.26, we upgrade the control plane first from 1.25 to 1.26, and then we start upgrading all the node pools from 1.24 directly to 1.26. Let's see how to go about upgrading node pools.

How to perform node pool upgrades

Node pool upgrades require a new node pool. It is not possible to upgrade nodes within the node pool. It is not even possible to add new nodes with a new version to an existing node pool. The node version is one of the essential properties of a node pool. What it means is that you actually don't upgrade an existing node pool. You replace your node pool. First, you create a new node pool, scale it up, and start draining nodes from the old node pool until all the pods run on the new node pool and then you can delete the old (now empty) node pool.

If your original node pool was an autoscaling node pool, then before starting the upgrade process, you must turn off autoscaling; otherwise, the pods you evicted from a node in the old node pool might get scheduled right back to the old node pool. Let's list the exact steps you need to take to upgrade a node pool:

- Create a new node pool with the exact same specifications (instance type, labels, tolerations) as the existing node with the new version.
- You may pre-allocate some instances to the new pool, so they are ready to schedule pods from the old node pool.
- Turn autoscaling off in the old node pool.
- Cordon all the nodes in the old node pool to prevent the scheduling of new pods to the old pool.
- Drain the nodes of the old node pool.
- Observe and deal with problems.
- Wait for the cluster autoscaler to delete empty old pool nodes or delete them yourself to expedite the process.

Let's look at some problems that can delay or even hold up the upgrade process.

Concurrent draining

If you need to upgrade many node pools with lots of nodes in each, you may decide to provision new node pools and start draining all your node pools at once or in large batches. This can cause you to exceed your quota or hit cloud provider capacity issues.

You should pay attention to your quota and ensure you have a sufficient quota regardless of upgrades. If you're getting close to your quota ceiling, I suggest bumping it before engaging in a complex operation like a node pool upgrade. The last thing you want is to be in the middle of an upgrade when you need to scale your capacity due to business needs and realize you maxed out your quota.

A good strategy for handling capacity issues and ramping up speed (how fast the cloud provider provision can instances for your new nodes) is to pre-allocate those instances. Again, this requires that you have a sufficient quota for the old node pool and the new node pool at the same time.

Let's understand what happens if you don't pre-allocate nodes in the new node pool. When you drain a node from the old node pool, all the pods are evicted from the node and become pending pods. Kubernetes will try to schedule these pods to existing nodes if any are available. The old node pool is cordoned, so either Kubernetes can find suitable nodes on other existing node pools (it's a good thing that improves bin packing) or the cluster autoscaler will need to provision a new node. That takes several minutes. If you drain multiple nodes at the same time, then all the pods from these nodes will be pending for a few minutes until new nodes can be provisioned and your system's capacity is degraded. In addition, if the cloud provider has capacity issues, maybe it can't provision new nodes and your pods will remain pending until then.

Pre-allocating nodes means that the new node pool will have nodes ready to go. The moment a pod is evicted from the old node pool, it will immediately be scheduled to an available node in the new node pool.

Dealing with workloads with no PDB

When draining a node, Kubernetes is mindful of **pod disruption budgets** (**PDBs**). If a deployment has a PDB that says only one pod can be unavailable and there are two pods of this deployment on the drained node, then Kubernetes will evict just one pod and wait until it is eventually scheduled before evicting the other pod. However, if you have workloads without PDBs, then that means Kubernetes is allowed to evict all the pods of those workloads at the same time. For most workloads, this is unacceptable. You should identify these workloads and work with their owner to add a PDB. Note that in the scenario of draining all nodes at once, workloads with no PDB are vulnerable even if they have many pods running on different nodes.

Dealing with workloads with PDB zero

However, the opposite problem of unevictable pods is the bane of node pool upgrades. If a node contains an unevictable pod, then it can't be drained, and Kubernetes will wait forever (or until the pod is manually deleted) before it fully drains the node. This can halt the upgrade process indefinitely and typically requires coordinating with the workload owner to resolve it.

If a workload has a PDB with `minUnavailable: 0`, it means that Kubernetes is not allowed to evict even a single pod from the workload regardless of how many replicas the workload has.

Some workloads (usually stateful) are more sensitive than others and prefer not to be disrupted at all. This is, of course, an unrealistic expectation because the node itself might go bad or the underlying VM might disappear due to cloud provider issues, and then the pods scheduled on it will have to be evicted. It's best to work with workload owners and come up with a solution that minimizes disruption, but still allows upgrades to progress. This has to be worked out before the upgrade process starts. You don't want to be in a situation where a single workload holds a node pool upgrade process hostage, and you have to beg the workload owner to allow you to evict a pod.

But, in addition to strict PDB-zero workloads, you might run into effective PDB-zero situations. Consider a workload with a PDB of `minUnavailable: 1`. This is pretty common and means the workload allows one pod at a time to be unavailable. When draining the pods of this workload, Kubernetes is allowed to evict one pod as long as all the other pods are running. However, if even one of the pods of this workload is pending or unable to be ready due to any reason, then effectively the workload already has one pod unavailable, and the upgrade process will be halted again.

The best practice here is to identify these workloads before the upgrade process starts and ensure that all workloads are healthy and can participate in the node pool upgrade process.

However, even if you did all the preparation work ahead of time, some workloads might get into an unhealthy state during the upgrade process (remember we're talking about a process that can take weeks).

I recommend having strong monitoring of the progress of the upgrade process, detecting stuck pods, and working with owners to resolve issues. In the case of pods that are scheduled on the old node pool and can't be ready, it is a simple solution to just delete the pod, see it is scheduled to the new node pool, and let the workload owner resolve the problem on the new node pool.

Other cases might require more creative solutions.

Let's turn our attention to various problems that can occur in a cluster and how to handle them, especially at scale.

Troubleshooting

In this section, we will cover the troubleshooting process in a production cluster and the logical procession of actions to take. The pod lifecycle involves multiple phases and failures can occur at each phase. In addition, pod containers go through their own mini lifecycle where init containers are running to completion and then the main containers start running. Let's see what can go wrong along the way and how to handle it.

First, let's look at pending pods.

Handling pending pods

When a new pod was created, Kubernetes used to place it in the Pending state and try to find a node to schedule it on. However, since Kubernetes 1.26, there is an even earlier state where a pod can't be scheduled.

Let's create a new 1.26 kind cluster called "`trouble`" and enable the pod scheduling readiness feature. Here is the configuration file (`cluster-config.yaml`):

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: trouble
nodes:
  - role: control-plane
featureGates:
  "PodSchedulingReadiness": true
```

And here is how to create the kind cluster:

```
$ kind create cluster -n trouble --config cluster-config.yaml --image kindest/
node:v1.26.0
Creating cluster "trouble" ...
 ✓ Ensuring node image (kindest/node:v1.26.0) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-trouble"
You can now use your cluster with:

kubectl cluster-info --context kind-trouble
```

Next, we'll create a new namespace called `trouble` and take it from there.

```
$ k create ns trouble
namespace/trouble created
```

Let's create a pod with a scheduling gate called `no-scheduling-yet`:

```
$ cat <<EOF | k apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
  namespace: trouble
  labels:
    app: the-app
spec:
  schedulingGates:
```

```
    - name: no-schedule-yet
  containers:
    - name: pause
      image: registry.k8s.io/pause:3.8
EOF
pod/some-pod created

$ k get po -n trouble
NAME       READY   STATUS            RESTARTS   AGE
some-pod   0/1     SchedulingGated   0          10s
```

As you can see, the pod has a status of `SchedulingGated`.

The benefit of the scheduling gate is that if the pod can't be scheduled yet due to issues like the quota, which need to be resolved externally, then the pod in this state will not cause a lot of churns to the Kubernetes scheduler, which will ignore it. After the external issue is resolved, you (or more likely an operator) can remove the feature gate and the pod will become a pending pod ready to be scheduled.

Now, let's turn our attention to pending pods. It's okay for a pod to be pending; however, if a pod is pending for more than a few minutes, something is wrong and we need to investigate it. I suggest having an alert set up for pods pending for more than X minutes (reasonable values for X can be between 10 and 60 minutes).

There are two types of pending pods: temporarily pending pods and permanent pending pods. Temporarily pending pods may be scheduled to one of the existing node pools; however, there is currently no room on any of the nodes. If the node pool has autoscaling enabled, the cluster autoscaler will try to provision a new node. If the node pool has autoscaling disabled, then the pod will remain pending until some other pods complete or are evicted from a node to make room. Another category of temporarily unschedulable pods is if the target namespace has a resource quota that is maxed out at the moment. Here is an example, where the namespace has a resource quota of 1 CPU and a deployment with 3 replicas is created where each pod requests 0.5 CPU. Only 2 pods can fit with the namespace quota. The third pod will be pending:

```
cat <<EOF | k apply -f -
apiVersion: v1
kind: ResourceQuota
metadata:
  name: cpu-requests
  namespace: trouble
spec:
  hard:
    requests.cpu: "1"
EOF
resourcequota/cpu-requests created
```

Now, let's create the deployment and see what happens:

```
cat <<EOF | k apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: some-deployment
  namespace: trouble
  labels:
    app: the-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: the-app
  template:
    metadata:
      namespace: trouble
      labels:
        app: the-app
    spec:
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.8
          resources:
            requests:
              cpu: "0.5"
EOF
deployment.apps/some-deployment created
```

We end up with just two running pods as expected:

```
$ k get deploy -n trouble
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
some-deployment   2/3     2            2           5m3s

$ k get po -n trouble
NAME                               READY   STATUS    RESTARTS   AGE
some-deployment-7f876df998-fc7kq   1/1     Running   0          4m34s
some-deployment-7f876df998-htdsn   1/1     Running   0          4m34s
```

Note, that there is no third pending pod in this case. Kubernetes is smart enough to create only two pods in this case.

The reason is the namespace quota:

```
$ k get deploy some-deployment -o yaml  -n trouble | grep forbidden -A 2
    message: 'pods "some-deployment-7f876df998-84z9s" is forbidden: exceeded quota:
      cpu-requests, requested: requests.cpu=500m, used: requests.cpu=1, limited:
requests.cpu=1'
    reason: FailedCreate
```

Permanent pending pods are pods that can't be scheduled on any of the available node pools, so provisioning a new node will not help. There are several categories of such permanently unschedulable pods:

- All the node pools have taints, and the pod doesn't have the proper tolerations.
- The pod requests more resources than are available on any of the node pools.
- The pod is waiting for a persistent volume.
- The pod has incorrect `nodeSelector` or `nodeAffinity`.

Let's delete the previous deployment and resource quota and look at an example of a pod that just requests way too many resources (666 CPU cores):

```
$ k delete resourcequota cpu-requests -n trouble
resourcequota "cpu-requests" deleted

$ k delete deploy some-deployment –n troublet
deployment.apps "some-deployment" deleted

$ cat <<EOF | k apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: some-deployment
  namespace: trouble
  labels:
    app: the-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: the-app
  template:
    metadata:
      labels:
```

```
        app: the-app
    spec:
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.8
          resources:
            requests:
              cpu: "666"
              memory: 1Gi
EOF
deployment.apps/some-deployment created
```

If we look at the pod that was created now, we can see it is indeed pending:

```
$ k get po -n trouble
NAME                               READY   STATUS    RESTARTS   AGE
some-deployment-bbf88d559-pdf8p   0/1     Pending   0          2m37s
```

To understand why it is pending, we can look at the status of the pod:

```
$ k get po some-deployment-bbf88d559-pdf8p -o yaml -n trouble | yq .status
conditions:
  - lastProbeTime: null
    lastTransitionTime: "2023-03-26T05:35:30Z"
    message: '0/1 nodes are available: 1 Insufficient cpu. preemption: 0/1 nodes are
available: 1 No preemption victims found for incoming pod..'
    reason: Unschedulable
    status: "False"
    type: PodScheduled
phase: Pending
qosClass: Burstable
```

The message is pretty clear and explains that 0 out of 1 node is available to schedule. It even says that 1 node has insufficient CPU. If there were other nodes in the cluster with other reasons, it would list them too.

Pending pods don't use resources and don't take up space in nodes; however, they put some pressure on the API server and also it means that some workloads don't get to do their work and they are waiting for a node to be scheduled on, which might be very serious in production. Mind your pending pods and make sure to resolve any issues quickly.

The next category of problems is about pods that are scheduled to a node but are unable to run.

Handling scheduled pods that are not running

There may be several reasons why a scheduled pod is not running. One of the most common ones is failure to pull an image required by one of the pod's containers. The kubelet will just keep trying and the pod's status will show as `ErrImagePull`:

```
cat <<EOF | k apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: no-such-image
  namespace: trouble
  labels:
    app: no-such-image
spec:
  replicas: 1
  selector:
    matchLabels:
      app: no-such-image
  template:
    metadata:
      labels:
        app: no-such-image
    spec:
      containers:
        - name: no-such-image
          image: no-such-image:6.6.6
EOF
deployment.apps/no-such-image created
```

Let's check the pods:

```
$ k get po -l app=no-such-image -n trouble
NAME                              READY   STATUS         RESTARTS   AGE
no-such-image-77585fd5b4-tqpv2   0/1     ErrImagePull   0          27s
```

To see a more elaborate message, we can check the `containerStatuses` field of the status:

```
$ k get po no-such-image-77585fd5b4-tqpv2 -n trouble -o yaml | yq '.status.
containerStatuses[0].state'
waiting:
  message: Back-off pulling image "no-such-image:6.6.6"
  reason: ImagePullBackOff
```

Image pull errors could relate to a misconfigured image. The image name or the image tag might be wrong. However, the image may be correct but might have been deleted accidentally from the registry. If you try to pull from a private registry, then possibly you don't have the correct permissions. Finally, the image registry may be unavailable. For example, Docker Hub often has rate limits.

You may prefer to pull all your images from a single source you control, where you can scan and curate the images and ensure that images don't disappear from you. If you're on the cloud, then every cloud provider offers their image registry. This should be the preferred option in most cases. You may save some money by using a private registry, and you may prefer a different solution in hybrid cloud scenarios.

Another reason that a pod doesn't start running is an init container that doesn't complete:

```
cat <<EOF | k apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: infinite-init
  namespace: trouble
  labels:
    app: infinite-init
spec:
  replicas: 1
  selector:
    matchLabels:
      app: infinite-init
  template:
    metadata:
      name: infinite-init
      labels:
        app: infinite-init
    spec:
      initContainers:
        - name: init-pause
          image: registry.k8s.io/pause:3.8
      containers:
        - name: main-pause
          image: registry.k8s.io/pause:3.8
EOF
deployment.apps/infinite-init created
```

Checking the pod status, we can see that it is stuck in the init phase because our init container is in the pause container, which never completes:

```
$ k get po -l app=infinite-init -n trouble
NAME                             READY   STATUS     RESTARTS   AGE
infinite-init-d555945fb-dccbk    0/1     Init:0/1   0          46s
```

Sometimes the pod starts running, but the container keeps failing. In this case, you need to check your pod's logs or your Dockerfile for the reason. Here is a pod that keeps crashing because its Dockerfile command just exists with exit code 1:

```
cat <<EOF | k apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: run-container-error
  namespace: trouble
spec:
  containers:
    - name: run-container-error
      image: bash
      command:
        - exit
        - "1"
EOF

pod/run-container-error created
```

The result is that the pod will have a status of `RunContainerError`. Kubernetes will keep restarting the pod (assuming the default restart policy of always):

```
$ k get po run-container-error -n trouble
NAME                  READY   STATUS              RESTARTS     AGE
run-container-error   0/1     RunContainerError   4 (8s ago)   100s
```

Getting your pods and containers to a running state is a good start, but it is not enough. In order for Kubernetes to send requests to your pods, all the containers must be ready. Let's see what problems you might run into.

Handling running pods that are not ready

If all your init containers are completed and all your main containers are running with no errors, then probes come into play. Kubernetes considers a pod with running containers ready to receive requests if no probes are defined or if all probes for all containers succeed. The startup probe is checked initially until it succeeds. If it fails, the pod is not considered ready. If your container has a hung startup pod, it will never be ready.

Kubernetes will eventually kill and restart your container and the startup probe will have another chance.

Here is a deployment where the main container has a startup probe that will always fail (the pause container doesn't even listen on port 80). The pod will never get to the ready state. After some retries and delays defined by the startup probe, Kubernetes will restart the container and the cycle will repeat:

```
Cat <<EOF | k apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bad-startup-probe
  namespace: trouble
  labels:
    app: bad-startup-probe
spec:
  replicas: 1
  selector:
    matchLabels:
      app: bad-startup-probe
  template:
    metadata:
      name: bad-startup-probe
      labels:
        app: bad-startup-probe
    spec:
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.8
          startupProbe:
            httpGet:
              path: /
              port: 80
            failureThreshold: 3
            periodSeconds: 10
            initialDelaySeconds: 5
            timeoutSeconds: 2
EOF

deployment/bad-startup-probe created
```

Checking the pod shows that the pod is in `CrashloopBackoff` and Kubernetes keeps restarting the container:

```
$ k get po -l app=bad-startup-probe –n trouble

NAME                                 READY   STATUS             RESTARTS          AGE
bad-startup-probe-66fb7cf4fd-qw8kp   0/1     CrashLoopBackOff   160 (3m56s ago)   8h
```

Note that the delay between restarts grows exponentially to avoid a bad container putting a lot of stress on the API server and the kubelet having to keep restarting often.

If there is no startup probe or it succeeds, the hurdle is the readiness probe. It works very similar to a startup probe, except Kubernetes will not restart the container. It will just keep checking the readiness probe. When a readiness probe fails, the pod will be removed from the endpoints list of any service that matches its labels, so it doesn't get to handle any requests. However, the pod remains alive and consumes resources on the node.

Let's see it in action:

```
cat <<EOF | k apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bad-readiness-probe
  namespace: trouble
  labels:
    app: bad-readiness-probe
spec:
  replicas: 1
  selector:
    matchLabels:
      app: bad-readiness-probe
  template:
    metadata:
      name: bad-readiness-probe
      labels:
        app: bad-readiness-probe
    spec:
      containers:
        - name: pause
          image: registry.k8s.io/pause:3.8
          readinessProbe:
            httpGet:
              path: /
```

```
              port: 80
            failureThreshold: 3
            periodSeconds: 10
            initialDelaySeconds: 5
            timeoutSeconds: 2
EOF

deployment.apps/bad-readiness-probe created
```

As you can see, the pod is running for an hour, it never gets ready, and it is not restarted:

```
$ k get po -l app=bad-readiness-probe –n trouble
NAME                                   READY   STATUS    RESTARTS   AGE
bad-readiness-probe-7458b65d98-5jrjq   0/1     Running   0          60m
```

The last type of probe is the liveness probe. It works just like the startup probe (the container will get restarted and the pod will be in `CrashloopBackoff`), except it is checked periodically, even if it succeeds while the startup probe is a one-time deal. Once it succeeds, it is never checked again.

The reason both startup and liveness probes are needed is that some containers need a longer startup period, but once they are initialized, then periodic liveness checks should be shorter.

That covers troubleshooting the pod lifecycle. When pods are scheduled but are not running or are not ready, it has cost implications. Let's move on and consider cost management.

Cost management

When running Kubernetes at a large scale in the cloud, one of the major concerns is the cost of the infrastructure. Cloud providers offer a variety of infrastructure options and services for your Kubernetes clusters. These are expensive. To harness your costs, make sure you follow best practices such as:

- Having a cost mindset
- Cost observability
- The smart selection of resources
- Efficient usage of resources
- Discounts, reserved instances, and spot instances
- Invest in local environments

Let's review them one by one.

Cost mindset

Engineers often neglect cost or put it way down the priority list. I often think in this order: make it work, make it fast, make it last, make it secure, and only then make it cheap. This is not necessarily a bad thing, especially for startup companies or new projects. Growth and velocity are often the top priorities. After all, if you don't have a good product, and you don't have customers, then the business will fail even if your costs are zero.

In addition, when the system is small, the absolute cost might be relatively small even if there is a lot of waste. Add to that the fact that cloud providers lure companies in with generous credits.

However, if you are part of a large enterprise or your startup succeeds and grows, at some point, cost will become a significant concern. At this point, you need to shift your thinking and have cost as the primary concern for everything you do. Cost may or may not be aligned with other initiatives. For example, everybody loves Pareto improvements. If you just manage to use 20% fewer VMs to accomplish the same task, then you will automatically save a lot of money without negatively impacting any other aspect.

But those easy wins and low-hanging fruit will eventually dry up. Then, you get to harder decisions. For example, caching the last week of data in memory will give you excellent response times, but at a large cost. What if you just cache one day?

Availability and redundancy are often at odds with cost as well. Do you really need a full-fledged zero-downtime active-active setup across multiple availability zones, regions, and cloud providers or can you get by with some downtime and recovery from backups in the event of catastrophic failure?

You may end up choosing the more expensive option, but you should do it with an explicit understanding of how much you pay for it and ensure the value you receive is greater.

That takes us to the next topic of cost observability.

Cost observability

To manage your infrastructure cost on Kubernetes and in the cloud in general, you must have strong cost-oriented observability. Let's look at some of the ways to accomplish it.

Tagging

Tagging is associating every resource with a set of tags or labels. From a cost perspective, tags should enable you to attribute the cost of any infrastructure resource to the relevant stakeholders. For example, you may have a team tag or an owner tag. If the resources provisioned by a specific team suddenly grow rapidly, you can narrow down the issue more quickly. The specific tags are up to you. Common tags may include environment (production, staging, and development), release, and git sha. Many resources in the cloud come with cloud-provider tags that you can take advantage of.

Policies and budgets

Policies and budgets let you rein in wild spending on infrastructure. Some policies are implicit, such as a namespace resource quota that will block the provisioning of excess resources. However, other policies may be more cost-specific and be informed by cost tracking. Budgets let you set a hard limit on spending at different scopes. All cloud providers offer budgets as part of their cost management solutions:

1. Tutorial: Create and manage Azure budgets: `https://learn.microsoft.com/en-us/azure/cost-management-billing/costs/tutorial-acm-create-budgets`
2. Managing your costs with AWS Budgets: `https://docs.aws.amazon.com/cost-management/latest/userguide/budgets-managing-costs.html`

3. Create, edit, or delete budgets and budget alerts in GCP: `https://cloud.google.com/billing/docs/how-to/budgets`

Policies and budgets are great, but sometimes they are not sufficient, or you don't have the cycles to specify and update them. This is where alerting comes into play.

Alerting

Budgets are often the last resort to mitigate against rogue infrastructure provisioning or accidental runaway provisioning. For example, you may set broad budgets for several overall categories, like no more than $500,000 of compute spending. These budgets of course need to align with business growth and make sure that they don't cause an incident if you temporarily need to provision more infrastructure to handle a temporary spike in demand. Budgets are often set or modified only with top management approval.

Fine-grained and day-to-day cost management alerts are much more nimble and practical. If you cross or come close to some cost limit, you can set alerts that will let you know and escalate as necessary. The alerts rely on proper tagging, so you can have meaningful information to evaluate the cause of the cost increase and the responsible party.

As you can see, managing costs is a dynamic and complex activity. You need good tools to help you.

Tools

All the cloud providers have strong cost management tools. Check out:

1. AWS Cost Explorer: `https://aws.amazon.com/aws-cost-management/aws-cost-explorer/`
2. Azure Cost Analyzer: `https://www.serverless360.com/azure-cost-analysis`
3. GCP Cost Management: `https://cloud.google.com/cost-management/`

In addition, you may opt to use a multi-cloud open-source tool like kubecost (`https://www.kubecost.com`) or a paid product like cast.ai (`https://cast.ai`). Of course, you can do everything yourself, ingest cost metrics from the cloud provider into Prometheus, and build your Grafana dashboards and alerts on top of them.

Remember that picking the right set of tools can literally pay for itself very quickly.

The smart selection of resources

The cloud offers a plethora of choices and combinations for resources like VMs, networking, and disks. We covered all the considerations in the *Bin packing and utilization* section earlier in this chapter. However, with a focus on cost, you should ensure that you understand that you may be able to get the job done with a cheaper alternative and reduce your costs significantly. Familiarize yourself with the inventory of resources and stay up to date as cloud providers update their offerings and may change prices too.

Efficient usage of resources

When you're cutting costs, any unused resources are a red flag. You leave money on the table. It is often necessary to build in flexibility. For example, the industry average CPU utilization is in the range of 40%-60%. This might seem low, but it is not easy to improve on that in a very dynamic environment with constraints like high availability, quick recovery, and the ability to scale up quickly.

Discounts, reserved instances, and spot instances

One of the best ways to reduce costs is to just pay less money to the cloud providers for the same resources. The common ways to accomplish this are discounts, reserved instances, and spot instances.

Discounts and credits

Discounts are the best. They only have an upside. You simply pay less than the sticker price. That's it. Well, it's not that easy. You will need to negotiate to get the best prices and often show promise for growth and commitment to stay longer on the platform.

Credits are another great way to offset initial cloud spending. All the major cloud providers offer various credit programs, and you may be able to negotiate more credits too.

AWS has the Activate program, targeted mostly at startups, where you can get up to $100,000 of AWS credit. See `https://aws.amazon.com/activate/`.

Azure has the Microsoft for Startups program, which offers up to $150,000 of Azure credit. See `https://www.microsoft.com/en-us/startups`.

GCP has the Google for Startups cloud program, which offers (like AWS) up to $100,000 of GCP credit. See `https://cloud.google.com/startup`.

These programs are designed to boost young startups without a lot of revenue. Let's move on to options for established enterprise organizations that still want to reduce their cloud spending.

Reserved instances

Reserved instances are a very good way to reduce your costs. They require that you purchase capacity in bulk and commit for a long period (one year or three years). The longer period carries better discounts. Overall, the discounts are significant and can vary from 30% to 75% compared to on-demand prices.

Beyond the price, reserved instances also ensure capacity compared to on-demand instances, which may temporarily be unavailable for particular instance types in a particular availability zone.

The downsides of reserved instances are that you typically have to prepay, and the commitment is often tied to specific instance types and regions. You may be able to exchange equivalent reservations, but you'll have to check the terms and conditions of the cloud provider. Also, if you are unable to use all your reserved capacity, you still pay for it (although at a very discounted price).

If you consider reserved instances (RIs), you may opt for a limited capacity of reserved instances, which you know you can always utilize, and then use on-demand and spot instances to handle spikes and take advantage of the elasticity of the cloud. If you discover later that you spend too much on on-demand, you can always reserve more instances and come up with a mix of three-year reserved instances, one-year reserved instances, on-demand, and spot instances. This is a great segue to the next item on the list, which is spot instances.

Spot instances

Cloud providers love reserved instances. They sell them, provision them, make their profit, and can forget about them (except for making sure they're up and running). The on-demand side is very different. Cloud providers have to make sure they can reasonably provision more capacity when their customers demand it. In particular, cloud providers should roughly have enough capacity to handle the quota of each customer even if the customer significantly underutilizes their quota. That means that, in practice, under normal conditions, cloud providers have a lot of idle or underutilized capacity. This is where spot instances come in. Cloud providers can sell this excess capacity, which is in theory allocated for on-demand use. If it is needed, then the cloud provider just takes away the spot instances and provides them to the on-demand customers.

Why should you use spot instances? Well, because they are much cheaper. Due to their potentially ephemeral nature, they carry significant discounts of up to 90%. Remember, from the cloud provider's point of view, this is free money. Those instances are already accounted for and paid for by the markup of on-demand instances in use and the quota ratio of each customer.

In practice, spot instances don't disappear from under you very quickly. The Kubernetes way advocates that workloads shouldn't care too much about specific nodes. If you run on a spot instance and it goes away, all your pods will be evicted and scheduled to other nodes. If you have sensitive workloads that don't handle eviction from their node well, then these workloads are not good Kubernetes citizens in the first place. Nodes become unhealthy all the time, regardless, and your workloads should be able to handle eviction.

However, there is one situation that needs to be addressed, specifically if you run critical workloads on spot instances. In case of a zonal outage of a specific instance type, it is possible that many spot instances will be taken by the cloud provider. I suggest having empty fallback on-demand node pools with the same or similar instance type and the same labels and taints. If a node pool using spot instances suddenly loses a lot of nodes and is unable to scale up (because the spot instances are unavailable), then you can scale up your empty on-demand node pool and your pods will be scheduled there until spot instances become available again.

Make sure you have enough quota for the fallback node pools to pick up the slack if the equivalent spot instances are unavailable.

Next, let's talk about local environments and how they can help us save money.

Invest in local environments

Organizations practice different protocols of development and testing. Some organizations do a lot of testing in the cloud in staging and development environments. Sometimes, engineers provision infrastructure for various experiments and tests. Such development and test environments can be difficult to manage effectively as infrastructure is often provisioned in an ad hoc manner. There are solutions like disposable Kubernetes clusters and virtual clusters. Another direction is to invest in local development environments that engineers can run on their local machines. The advantages are that these environments are often very quick to set up and discard, and they don't incur any expensive cloud costs. The downside is that such environments might not be fully representative of the staging or production environments. I suggest looking into local environments and finding use cases that will save cloud costs without compromising other critical aspects of the system.

Summary

In this chapter, we covered in depth what it takes to run large-scale Managed Kubernetes systems in production. We looked at managing multiple clusters, building effective processes, handling infrastructure at scale, managing clusters and node pools, bin packing and utilization, upgrading Kubernetes, and troubleshooting and cost management. That's a lot, but even that is just the tip of the iceberg. There is no substitute for in-depth familiarity with your use cases and special concerns. The bottom line is that large-scale enterprise systems are complex to manage, but Kubernetes gives you a lot of industrial-strength tools to accomplish it.

The next chapter will conclude the book. We will look at the future of Kubernetes and the road ahead. Spoiler alert: the future is very bright. Kubernetes has established itself as the gold standard for cloud-native computing. It is being used across the board, and it keeps evolving responsibly. An entire support system has developed around Kubernetes, including training, open-source projects, tools, and products. The community is amazing, and the momentum is very strong.

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`



18

The Future of Kubernetes

In this chapter, we will look at the future of Kubernetes from multiple angles. We'll start with the momentum of Kubernetes since its inception across dimensions such as community, ecosystem, and mindshare. Spoiler alert – Kubernetes won the container orchestration wars by a landslide. As Kubernetes grows and matures, the battle lines shift from beating competitors to fighting against its own complexity. Usability, tooling, and education will play a major role as container orchestration is still new, fast-moving, and not a well-understood domain. Then we will take a look at some very interesting patterns and trends, and finally, we will review my predictions from the second edition, and I will make some new predictions.

The covered topics are as follows:

- The Kubernetes momentum
- The importance of CNCF
- Kubernetes extensibility
- Service mesh integration
- Serverless computing on Kubernetes
- Kubernetes and VMs
- Cluster autoscaling
- Ubiquitous operators
- Kubernetes for AI
- Kubernetes challenges

The Kubernetes momentum

Kubernetes is undeniably a juggernaut. Not only did Kubernetes beat all the other container orchestrators, but it is also the de facto solution on public clouds, utilized in many private clouds, and even VMware – the virtual machine company – is focused on Kubernetes solutions and integrating its products with Kubernetes.

Kubernetes works very well in multi-cloud and hybrid-cloud scenarios due to its extensible design.

In addition, Kubernetes makes inroads on the edge, too, with custom distributions that expand its broad applicability even more.

The Kubernetes project continues to release a new version every three months, like clockwork. The community just keeps growing.

The Kubernetes GitHub repository has almost 100,000 stars. One of the major drivers of this phenomenal growth is the **CNCF** (**Cloud Native Computing Foundation**).

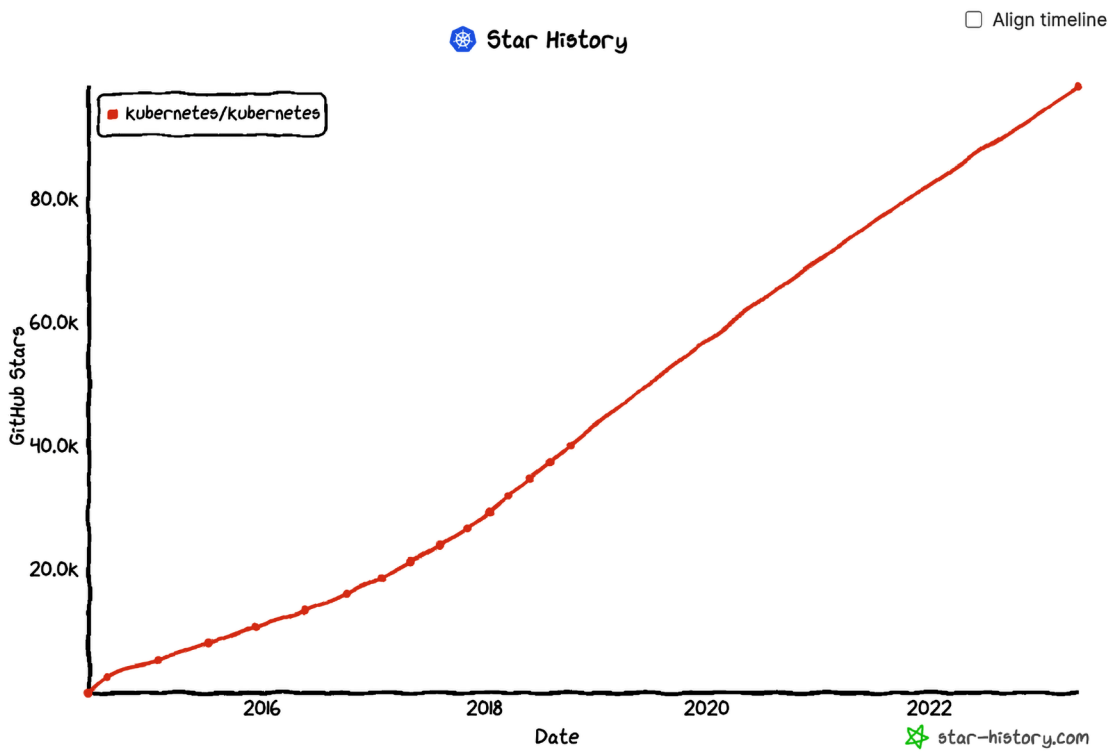


Figure 18.1: Star History chart

The importance of the CNCF

The CNCF has become a very important organization in the cloud computing scene. While it is not Kubernetes-specific, the dominance of Kubernetes is undeniable. Kubernetes is the first project to graduate, and most of the other projects lean heavily toward Kubernetes. In particular, the CNCF offers certification and training only for Kubernetes. The CNCF, among other roles, ensures that cloud technologies will not suffer from vendor lock-in. Check out this crazy diagram of the entire CNCF landscape: `https://landscape.cncf.io`.

Project curation

The CNCF assigns maturity levels to projects: **graduated**, **incubating**, and **sandbox**:

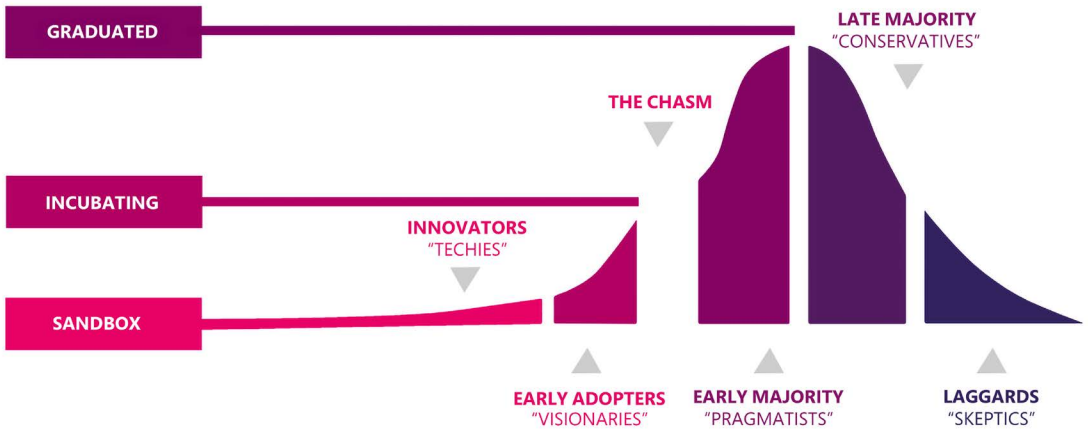


Figure 18.2: CNCF maturity levels

Projects start at a certain level – sandbox or incubating – and over time, can graduate. That doesn't mean that only graduated projects can be safely used. Many incubating and even sandbox projects are used heavily in production. For example, etcd is the persistent state store of Kubernetes itself, and it is just an incubating project. Obviously, it is a highly trusted component. Virtual Kubelet is a sandbox project that powers AWS Fargate and Microsoft ACI. This is clearly enterprise-grade software.

The main benefit of the CNCF curation of projects is to help navigate the incredible ecosystem that grew around Kubernetes. When you look to extend your Kubernetes solution with additional technologies and tools, the CNCF projects are a good place to start.

Certification

When technologies start to offer certification programs, you can tell they are here to stay. The CNCF offers several types of certifications:

- Certified Kubernetes for conforming Kubernetes distributions and installers (about 90).
- **Kubernetes Certified Service Provider** (**KCSP**) for vetted service providers with deep Kubernetes experience (134 providers).
- **Certified Kubernetes Administrator** (**CKA**) for administrators.
- **Certified Kubernetes Application Developer** (**CKAD**) for developers.
- **Certified Kubernetes Security Specialist** (**CKS**) for security experts.

Training

The CNCF offers training too. There is a free introduction to Kubernetes course and several paid courses that align with the CKA and CKAD certification exams. In addition, the CNCF maintains a list of Kubernetes training partners (`https://landscape.cncf.io/card-mode?category=kubernetes-training-partner&grouping=category`).

If you're looking for free Kubernetes training, here are a couple of options:

- VMware Kubernetes academy
- Google Kubernetes Engine on Coursera

Community and education

The CNCF also organizes conferences like KubeCon, CloudNativeCon, and meetups and maintains several communication avenues like Slack channels and mailing lists. It also publishes surveys and reports.

The number of attendees and participants keeps growing year after year.

Tooling

The number of tools to manage containers and clusters, the various add-ons, extensions, and plugins just keep growing. Here is a subset of the tools, projects, and companies that participate in the Kubernetes ecosystem:



Figure 18.3: Kubernetes tooling

The rise of managed Kubernetes platforms

Pretty much every cloud provider has a solid managed Kubernetes offering these days. Sometimes there are multiple flavors and ways of running Kubernetes on a given cloud provider.

Public cloud Kubernetes platforms

Here are some of the prominent managed platforms:

- Google GKE
- Microsoft AKS
- Amazon EKS
- Digital Ocean
- Oracle Cloud
- IBM Cloud Kubernetes service
- Alibaba ACK
- Tencent TKE

Of course, you can always roll your own and use the public cloud providers just as infrastructure providers. This is a very common use case with Kubernetes.

Bare metal, private clouds, and Kubernetes on the edge

Here, you can find Kubernetes distributions that are designed or configured to run in special environments, often in your own data centers as a private cloud or in more restricted environments like edge computing on small devices:

- Google Anthos for GKE
- OpenStack
- Rancher k3S
- Kubernetes on Raspberry PI
- KubeEdge

Kubernetes PaaS (Platform as a Service)

This category of offerings aims to abstract some of the complexity of Kubernetes and put a simpler facade in front of it. There are many varieties here. Some of them cater to the multi-cloud and hybrid-cloud scenarios, some expose a function-as-a-service interface, and some just focus on a better installation and support experience:

- Google Cloud Run
- VMware PKS
- Platform 9 PMK
- Giant Swarm
- OpenShift
- Rancher RKE

Upcoming trends

Let's talk about some of the technological trends in the Kubernetes world that will be important in the near future.

Security

Security is, of course, a paramount concern for large-scale systems. Kubernetes is primarily a platform for managing containerized workloads. Those containerized workloads are often run in a multi-tenant environment. The isolation between tenants is super important. Containers are lightweight and efficient because they share an OS and maintain their isolation through various mechanisms like namespace isolation, filesystem isolation, and cgroup resource isolation. In theory, this should be enough. In practice, the surface area is large, and there were multiple breakouts out of container isolation.

To address this risk, multiple lightweight VMs were designed to add a hypervisor (machine-level virtualization) as an additional isolation level between the container and the OS kernel. The big cloud providers already support these technologies, and the Kubernetes CRI interface provides a streamlined way to take advantage of these more secure runtimes.

For example, FireCracker is integrated with containerd via firecracker-containerd. Google gVisor is another sandbox technology. It is a userspace kernel that implements most of the Linux system calls and provides a buffer between the application and the host OS. It is also available through containerd via gvisor-containerd-shim.

Networking

Networking is another area that is an ongoing source of innovation. The Kubernetes CNI allows any number of innovative networking solutions behind a simple interface. A major theme is the incorportation of eBPF – a relatively new Linux kernel technology – into Kubernetes.

eBPF stands for **extended Berkeley Packet Filter**. The core of eBPF is a mini-VM in the Linux kernel that executes special programs attached to kernel objects when certain events occur, such as a packet being transmitted or received. Originally, only sockets were supported, and the technology was called just BPF. Later, additional objects were added to the mix and that's when the *e* for *extended* came along. eBPF's claim to fame is its performance due to the fact it runs highly optimized compiled BPF programs in the kernel and doesn't require extending the kernel with kernel modules.

There are many applications for eBPF:

- **Dynamic network control**: iptables-based approaches don't scale very well in a dynamic environment like a Kubernetes cluster where you have a constantly changing set of pods and services. Replacing iptables with BPF programs is both more performant and more manageable. Cilium is focused on routing and filtering traffic using eBPF.
- **Monitoring connections**: Creating an up-to-date map of TCP connections between containers is possible by attaching a BPF program kprobes that track socket-level events. WeaveScope utilizes this capability by running an agent on each node that collects this information and sends it to a server that provides a visual representation through a slick UI.

- **Restricting syscalls**: The Linux kernel provides more than 300 system calls. In a security-sensitive container environment, it is highly desirable. The original seccomp facility was pretty rudimentary. In Linux 3.5, seccomp was extended to support BPF for advanced custom filters.
- **Raw performance**: eBPF provides significant performance benefits, and projects like Calico took advantage and implemented a faster data plane that uses fewer resources.

Custom hardware and devices

Kubernetes manages nodes, networking, and storage at a relatively high level. But there are many benefits to integrating specific hardware at a fine-grained level. For example, GPUs, high-performance network cards, FPGAs, InfiniBand adapters, and other compute and networking and storage resources. This is where the device plugin framework comes in, which can be found here: `https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins`. It has been in GA since Kubernetes 1.26, and there is ongoing innovation in this area. For example, monitoring device plugin resources is also in beta since Kubernetes 1.15. It is very interesting to see what devices will be harnessed to Kubernetes. The framework itself follows modern Kubernetes extensibility practices by utilizing gRPC.

Service mesh

The service mesh is arguably the most important trend in networking over the last couple of years. We covered service meshes in depth in *Chapter 14*, *Utilizing Service Meshes*. The adoption is impressive, and I predict that most Kubernetes distributions will provide a default service mesh and allow easy integration with other service meshes. The benefits that service meshes provide are just too valuable. It makes sense to provide a default platform that includes Kubernetes with an integrated service mesh. That said, Kubernetes itself will not absorb some service mesh and expose it through its API. This goes against the grain of keeping the core of Kubernetes small.

Google Anthos is a good example where Kubernetes + Knative + Istio are combined to provide a unified platform that provides an opinionated best-practices bundle that would take an organization a lot of time and resources to build on top of vanilla Kubernetes.

Another push in this direction is the sidecar container KEP; information about it can be found here: `https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/753-sidecar-containers/README.md`.

The sidecar container pattern has been a staple of Kubernetes from the get-go. After all, pods can contain multiple containers. But there was no notion of a main container or a sidecar container. All containers in the pod have the same status. Most service meshes use sidecar containers to intercept traffic and perform their jobs. Formalizing sidecar containers will help those efforts and push service meshes even further.

It's not clear at this stage if Kubernetes and the service mesh will be hidden behind a simpler abstraction on most platforms or if they will be front and center.

Serverless computing

Serverless computing is another trend that is here to stay. We discussed it at length in *Chapter 12*, *Serverless Computing on Kubernetes*. Kubernetes and serverless can be combined on multiple levels. Kubernetes can utilize serverless cloud solutions like AWS Fargate and **AKS Azure Container Instances** (**ACI**) to save the cluster administrator from managing nodes. This approach also caters to integrating lightweight VMs transparently with Kubernetes since the cloud platforms don't use naked Linux containers for their container-as-a-service platforms.

Another avenue is to reverse the roles and expose containers as a service powered by Kubernetes under the covers. This is exactly what Google Cloud Run is doing. The lines blur here as there are multiple products from Google to manage containers and/or Kubernetes ranging from just GKE, through Anthos GKE (bring your own cluster to the GKE environment for your private data center), Anthos (managed Kubernetes + service mesh), and Anthos Cloud Run.

Finally, there are functions-as-a-service and scale-to-zero projects running inside your Kubernetes cluster. Knative may become the leader here, as it is already used by many frameworks, and is deployed heavily through various Google products.

Kubernetes on the edge

Kubernetes is the poster boy of cloud-native computing, but with the **Internet of Things** (**IoT**) revolution, there is more need to perform computation at the edge of the network. Sending all data to the backend for processing suffers from several drawbacks:

- Latency
- Need for enough bandwidth
- Cost

With edge locations collecting a lot of data via sensors, video cameras, etc., the amount of edge data grows, and it makes more sense to perform more and more sophisticated processing at the edge. Kubernetes grew out of Google's Borg, which was definitely not designed to run at the edge of the network. But Kubernetes' design proved to be flexible enough to accommodate it. I expect that we will see more and more Kubernetes deployments at the edge of the network, which will lead to very interesting systems that are composed of many Kubernetes clusters that will need to be managed centrally.

KubeEdge is an open-source framework that is built on top of Kubernetes and Mosquito – an open-source implementation of MQTT message broker – to provide a foundation for networking, application deployment, and metadata synchronization between the cloud and the edge.

Native CI/CD

For developers, one of the most important questions is the construction of a CI/CD pipeline. There are many options and choosing between them can be difficult. The CD Foundation is an open source foundation that was formed to standardize concepts like pipelines and workflows, and define industry specifications that will allow different tools and communities to interoperate better.

The current projects are:

- Jenkins
- Tekton
- Spinnaker
- Jenkins X
- Screwdriver.cd
- Ortelius
- CDEvents
- Pyrsia
- Shipwright

Note that only Jenkins and Tekton are considered graduated projects. The rest are incubating projects (even Spinnaker).

One of my favorite native CD projects, Argo CD, is not part of the CD Foundation. I actually opened a GitHub issue asking to submit Argo CD to the CDF, but the Argo team has decided that CNCF is a better fit for their project.

Another project to watch is CNB – Cloud Native Buildpacks. The project takes the source and creates OCI (think Docker) images. It is important for FaaS frameworks and native in-cluster CI. It is also a CNCF sandbox project.

Operators

The **Operator** pattern emerged in 2016 from CoreOS (acquired by RedHat, acquired by IBM) and gained a lot of success in the community. An Operator is a combination of custom resources and a controller used to manage an application. At my current job, I write operators to manage various aspects of infrastructure, and it is a joy. It is already the established way to distribute non-trivial applications to Kubernetes clusters. Check out `https://operatorhub.io/` for a huge list of existing operators. I expect this trend to continue and intensify.

Kubernetes and AI

AI is the hottest trend right now. **Large language models** (**LLMs**) and **Generative Pre-trained Transforms** (**GPT**) surprised most professionals with their capabilities. The release of ChatGPT 3.5 by OpenAI was a watershed moment. AI suddenly excels in areas that were considered strongholds of human intelligence, such as creative writing, painting, understanding, answering nuanced questions, and, of course, coding. My perspective is that advanced AI is the solution to the big data problem. We learned to collect a lot of data, but analyzing and extracting insights from the data is a difficult and labor-intensive process. AI seems like the right technology to digest all the data and automatically understand, summarize, and organize it into a useful form to be used by humans and other systems (most likely AI-based systems).

Let's see why Kubernetes is such a great fit for AI workloads.

Kubernetes and AI synergy

Modern AI is all about deep learning networks and huge models with billions of parameters trained on massive datasets, often using dedicated hardware. Kubernetes is a perfect fit for such workloads as it quickly adapts to the workload's needs, takes advantage of new and improved hardware, and provides strong observability.

The best evidence is the field. Kubernetes is at the core of the OpenAI pipeline, and additional companies are developing and deploying massive AI applications. Check out this article that shows how OpenAI pushes the envelope with Kubernetes and runs huge clusters with 7,500 nodes: `https://openai.com/research/scaling-kubernetes-to-7500-nodes`.

Let's consider training AI models on Kubernetes.

Training AI models on Kubernetes

Training large AI models is potentially slow and very expensive. Organizations that partake in training AI models on Kubernetes benefit from many of its properties:

- **Scalability**: Kubernetes provides a highly scalable infrastructure for deploying and managing AI workloads. With Kubernetes, it is possible to quickly scale resources up or down based on demand, enabling organizations to train AI models quickly and efficiently.
- **Resource utilization**: Kubernetes allows for efficient resource utilization, enabling organizations to train AI models using the most cost-effective infrastructure. With Kubernetes, it is possible to automatically allocate and manage resources, ensuring that the right resources are available for the workload.
- **Flexibility**: Kubernetes provides a high degree of flexibility in terms of the infrastructure used for training AI models. Kubernetes supports a wide range of hardware, including GPUs, FPGAs, and TPUs, making it possible to use the most appropriate hardware for the workload.
- **Portability**: Kubernetes provides a highly portable infrastructure for deploying and managing AI workloads. Kubernetes supports a wide range of cloud providers and on-premises infrastructure, making it possible to train AI models in any environment.
- **Ecosystem**: Kubernetes has a vibrant ecosystem of open-source tools and frameworks that can be used for training AI models. For example, Kubeflow is a popular open-source framework for building and deploying machine learning workflows on Kubernetes.

Running AI-based systems on Kubernetes

Once you have trained your models and built your application on top of your models, you need to deploy and run it. Kubernetes is, of course, a great platform for deploying workloads in general. AI-based workloads are often designed for a reliable and quick super-human response. The high availability that Kubernetes offers and the ability to quickly scale up and down based on demand satisfy these requirements.

In addition, if the system is designed to keep learning (as opposed to fixed pre-trained systems like GPTs), then Kubernetes offers strong security and control that support safe operation.

Let's look at the emerging field of AIOps.

Kubernetes and AIOps

AIOps is a paradigm that leverages AI and machine learning to automate and optimize the management of infrastructure. AIOps can help organizations improve the reliability, performance, and security of their IT infrastructure while reducing the burden on human engineers.

Kubernetes is a perfect target for practicing AIOps. It is fully accessible programmatically. It is often deployed with deep observability. Those two conditions are necessary and sufficient to enable AI to scrutinize the state of the system and take action when necessary.

The future of Kubernetes seems bright, but it has some challenges too.

Kubernetes challenges

Is Kubernetes the answer to everything to do with infrastructure? Not at all. Let's look at some challenges, such as Kubernetes' complexity, and some alternative solutions for the problem of developing, deploying, and managing large-scale systems.

Kubernetes complexity

Kubernetes is a large, powerful, and extensible platform. It is mostly un-opinionated and very flexible. It has a huge surface area with a lot of resources and APIs. In addition, Kubernetes has a huge ecosystem. That translates to a system that is extremely difficult to learn and master. What does it say about Kubernetes' future? One likely scenario is that most developers will not interact with Kubernetes directly. Simplified solutions built on top of Kubernetes will be the primary access point for most developers.

If Kubernetes is fully abstracted, then it may become a threat to its future since Kubernetes, as the underlying implementation, might be replaced by the solution provider. The final users may not need to make any changes at all to their code or configuration.

Another scenario is that more and more organizations negatively weigh the costs of building on top of Kubernetes compared to lightweight container orchestration platforms such as Nomad. This could lead to an exodus from Kubernetes.

Let's look at some technologies that may compete with Kubernetes in different areas.

Serverless function platforms

Serverless function platforms offer organizations and developers similar benefits to Kubernetes using a simpler (if less powerful) paradigm. Instead of modeling your system as a set of long-running applications and services, you just implement a set of functions that can be triggered on demand. You don't need to manage clusters, node pools, and servers. Some solutions offer long-running services, too, either pre-packaged as containers or directly from source. We covered it thoroughly in *Chapter 12*, *Serverless Computing on Kubernetes*. As the serverless platforms get better and Kubernetes become more complicated, more organizations may prefer to at least start using serverless solutions and possibly migrate to Kubernetes later.

First and foremost, all cloud providers offer various serverless solutions. The pure cloud functions models are:

- AWS Lambda
- Google Cloud Functions
- Azure Functions

There are also multiple strong and easy-to-use solutions out there that are not associated with the big cloud providers:

- Cloudflare Workers
- Fly.io
- Render
- Vercel

That concludes our coverage of Kubernetes challenges. Let's summarize the chapter.

Summary

In this chapter, we looked at the future of Kubernetes, and it looks great! The technical foundation, the community, the broad support, and the momentum are all very impressive. Kubernetes is still young, but the pace of innovation and stabilization is very encouraging. The modularization and extensibility principles of Kubernetes let it become the universal foundation for modern cloud-native applications. That said, there are some challenges to Kubernetes and it might not dominate each and every scenario. This is a good thing. Diversity, competition, and inspiration from other solutions will just make Kubernetes better.

At this point, you should have a clear idea of where Kubernetes is right now and where it's going from here. You should be confident that Kubernetes is not just here to stay, but that it will be the leading container orchestration platform for many years to come and integrate with any major offering and environment you can imagine, from planet-scale public cloud platforms, private clouds, data centers, edge locations and all the way down to your development laptop and Raspberry Pi.

That's it! This is the end of the book.

Now it's up to you to use what you've learned and build amazing things with Kubernetes!

Join us on Discord!

Read this book alongside other users, cloud experts, authors, and like-minded professionals.

Ask questions, provide solutions to other readers, chat with the authors via. Ask Me Anything sessions and much more.

Scan the QR code or visit the link to join the community now.

`https://packt.link/cloudanddevops`





packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

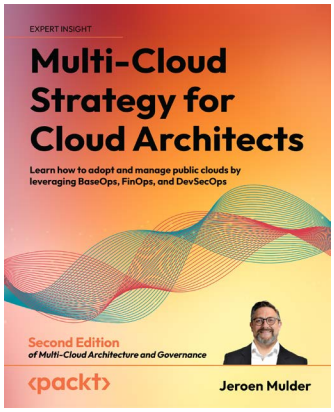
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

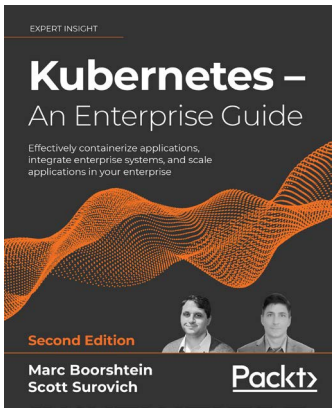


Multi-Cloud Strategy for Cloud Architects, Second Edition

Jeroen Mulder

ISBN: 9781804616734

- Choose the right cloud platform with the help of use cases
- Master multi-cloud concepts, including IaC, SaaS, PaaS, and CaC
- Use the techniques and tools offered by Azure, AWS, and GCP to integrate security
- Maximize cloud potential with Azure, AWS, and GCP frameworks for enterprise architecture
- Use FinOps to define cost models and optimize cloud costs with showback and chargeback



Kubernetes – An Enterprise Guide, Second edition

Marc Boorshtein

Scott Surovich

ISBN: 9781803230030

- Create a multinode Kubernetes cluster using KinD
- Implement Ingress, MetalLB, ExternalDNS, and the new sandbox project, K8GBConfigure a cluster OIDC and impersonation
- Deploy a monolithic application in Istio service mesh
- Map enterprise authorization to Kubernetes
- Secure clusters using OPA and GateKeeper
- Enhance auditing using Falco and ECK
- Back up your workload for disaster recovery and cluster migration
- Deploy to a GitOps platform using Tekton, GitLab, and ArgoCD

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Mastering Kubernetes, Fourth Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

access control webhooks
admission control webhook 559
authentication webhook 555, 556
authorization webhook 557-559
using 555
webhook admission controller, configuring 560, 561

ACK (Alibaba Container service for Kubernetes) 60

adapter pattern 11

Admiralty 391
architecture 392
features 392
reference link 392

advanced scheduling 157, 158

advanced storage features 229
CSI volume cloning 230
storage capacity tracking 231
volume health monitoring 231
volume snapshots 229, 230

advancing science
with Hue 182

AI-based systems
running, on Kubernetes 670

AI models
training, on Kubernetes 670

AIOps 671

AKS Arc 64

alerting 448

Alertmanager 464

Amazon Web Services (AWS) 57, 185, 613

ambassador pattern 10

annotation 7

API deprecation 267

API objects
serializing, with protocol buffers 92

API server
caching reads 90

API server, accessing 108
admission control plugins, using 114, 115
impersonation 111
request authorization 112, 113
user authentication 109-111

API server aggregation 544
advanced capabilities 544

AppArmor 117
pod, securing with 118, 119
profiles, writing for 119-121
requirements 117, 118

application error reporting 448

Argo CD 669

Aspecto
URL 470

autoscaling
based on custom metrics 80

instances 87
rolling updates, performing with 276-278
AWS Activate
URL 658
AWS App Mesh 484
URL 484
AWS Cost Explorer
reference link 657
AWS EKS 58
AWS Elastic Block Store (EBS) 210-212
AWS Elastic File System (EFS) 212-214
Azure 59
Azure Active Directory (AAD) 614
Azure Blob Storage 614
Azure Compute 614
Azure Container Instance (ACI) 59, 406, 407, 668
Azure Cost Analyzer
reference link 657
Azure data disk 217
Azure file 218
Azure Key Vault 614
Azure Kubernetes Service (AKS) 59, 88
Azure Monitor 614
Azure Virtual Network 614

B

bare metal cluster 665
creating, considerations 62
creating 61
custom cluster, building with Kubespray 63
custom cluster, building with Rancher RKE 63
managed Kubernetes, running 63
managing, with Cluster API 62
process 62
use cases 61
virtual private cloud infrastructure, using 63
basic Linux networking 334
bridges 334
CIDRs 334
IP address 334
maximum transmission unit (MTU) 335
netmasks 334
network namespaces 334
pod networking 335
ports 334
routing 334
subnets 334
Virtual Ethernet (veth) devices 334
best practices, high availability 69
cluster state, protecting 72
data protecting 72
etcd, clustering 72
highly available clusters, creating 70, 71
leader election, running with Kubernetes 73, 74
nodes, making reliable 71, 72
redundant API servers, running 73
staging environment, making highly available 74
big library approach
issues 480
bin packing 630
blue-green deployments 265, 266
BookInfo
installing 491-494
bootstrap provider 376
bridge plugin
reviewing 364-368
bridges 334
broker 420
budgets 656, 657

build.sh script 246
built-in objects
embedding 322
Buoyant 484
URL 484

C

caching reads
in API server 90
Calico 342
canary deployments 266
capacity planning 76, 616
CAP theorem 84
Cassandra 8, 243, 249
Cassandra cluster 244
running, in Kubernetes 243
Cassandra configuration file 252, 253
Cassandra Docker image 244-246
Cassandra headless service
creating 254, 255
cast.ai
URL 657
cattle approach
versus pets approach 4
CCE (Cloud Container Engine) 60
centralized logging 450
cluster-level central logging 452
log collection strategy, selecting 450
remote central logging 452, 453
sensitive log information, handling 453
Ceph 219
connecting to 221
using 221
Certified Kubernetes Administrator (CKA) 663
Certified Kubernetes Application Developer (CKAD) 663
Certified Kubernetes Security Specialist (CKS) 663
Chaos Mesh CNCF incubating project 75
chart dependencies
additional subfields, utilizing of dependencies field 316, 317
managing 315, 316
charts 313
deprecating 314
managing, with Helm 311, 312
metadata files 314, 315
testing 320, 321
troubleshooting 320, 321
versioning 313
Chart.yaml file 313
appVersion field 313
fields 313
optional fields 313
Chocolatey
URL 300
CI/CD pipeline 289
designing, for Kubernetes 290
Cilium 342
bandwidth management 343
efficient IP allocation and routing 342, 343
identity-based service-to-service communication 343
load balancing 343
observability 343
URL 344
Cilium Service Mesh 485
URL 485
Citadel 487
claims
mounting, as volumes 199
Classless Inter-Domain Routing (CIDR) notation 334

client IP addresses
preserving 349
Cloud Controller Interface (CCI) 614
cloud controller manager (CCM) 15-17, 534
Kubernetes, extending with 534
reference link 535
Cloud native Buildpacks 669
Cloud Native Computing Foundation (CNCF) 1, 445, 662
certification programs 663
community and education 664
project curation 663
reference link 662
significance 662
training 664
cloud-native security
4Cs model 100
cloud provider APIs
tooling 291
cloud-provider interface 56
cloud providers
AWS 57
Azure 59
Chinese Alibaba cloud 60
Digital Ocean 59
Google Cloud Platform (GCP) 57
Huawei cloud platform 60
IBM Kubernetes service 60
lifecycle 636
Oracle Container Service 61
Tencent 60
cloud quotas 87
Cluster API 374, 629
architecture 374, 375
bootstrap provider 376
custom resources 376, 377
infrastructure provider 376
management cluster 375
work cluster 375
Cluster API, control plane 376
external 376
machine-based 376
pod-based 376
cluster autoscaler (CAS) 76
installing 77-79
cluster breakdown
designing 623
cluster capacity
cost and response time, trading off 86
managing 85
multiple node configurations, using effectively 86
node types, selecting 85
selecting 85
storage solutions, selecting 85
clusterctl
reference link 375
cluster-level central logging 452
Clusternet 380
features 380
multi-cluster deployment 381
reference link 382
Clusternet architecture 380
Clusternet agent 381
Clusternet hub 381
Clusternet scheduler 381
Clusterpedia 382
advanced multi-cluster search 384, 385
clusters, importing 384
resource collections 385
Clusterpedia architecture 382
Clusterpedia API server 383
ClusterSynchro manager 383
storage component 384
storage layer 384

cluster resource category 14

clusters 5
cloud-provider interface 56
creating, in cloud 55
incompatible resources, detecting 637-639
incompatible resources, updating 640, 641
managing 628
removed features, handling 641
upgrade, planning 637
upgrading 637

cluster syncro 382

CNCF KubeEdge
URL 618

CNI 336-340, 358
skeleton, building 362-364
third-party plugins 336

CockroachDB 8

complex deployments 264

complex distributed system, troubleshooting 474
dashboards vs. alerts 476, 477
logs vs metrics vs. error reports 477
performance and root cause, detecting with distributed tracing 478
problem daemons 476
problems, detecting at node level 475, 476
staging environments, using 474

component logs 450

compute infrastructure 622
cluster breakdown, designing 623
node pool breakdown, designing 623

compute resource quota 280

config and storage resource category 14

ConfigMap
consuming, as environment variable 238
creating 237, 238

containerd 22

container logs 449

container-native solutions
considering 88, 89

container orchestration 3
physical machines 3
virtual machines 3

container runtime 337

Container Runtime Interface (CRI) 18-20, 536, 537

containers 3, 186, 666
benefits 3
in cloud 3, 4

Container Storage Interface (CSI) 8, 185, 202, 228, 563
architecture 229
ephemeral volumes 202
volume cloning 230

controller pattern 535

controller-runtime
reference link 527

control plane 6, 376
API server 15
cloud controller manager 15-17
DNS 17
etcd 15
Kube controller manager 15
Kube scheduler 17

control plane upgrades
syncing with 642

CoreDNS 332, 333

CoreV1Api group
dissecting 520-523

cost management 655
cost mindset 655
cost observability 656
discounts and credits 658
efficient usage, of resources 658

invest, in local environments 660
reserved instances 658
smart selection, of resources 657
spot instances 659
tools 657
cost observability 656
alerting 657
policies and budgets 656
tagging 656
CRI-O 22
capabilities 22
cron jobs
scheduling 174, 175
cross-cloud communication 625
cross-cluster communication 624, 625
cross-cluster service meshes 625, 626
Crossplane 294
URL 294
using 294
Cue 324
URL 324
custom container runtimes
Kubernetes, extending with 536, 537
custom metrics
providing, for horizontal pod autoscaling 562
custom operators 294
custom resource definitions (CRDs) 487
custom resources 376, 537
custom printer columns, adding 543
custom resource definitions, developing 538, 539
finalizing 542
integrating 539
unknown fields, handling 540-542
custom seed provider 253
custom storage
Kubernetes, extending with 563

D

DaemonSet
using, for redundant persistent storage 239
DaemonSet pods 180
dashboards 448
data-contract changes
managing 267
data migration 267
dedicated nodes
versus shared nodes 631, 632
deep integration 614
default compute quotas
limit ranges, using for 287, 288
dependencies
managing, with readiness probes 178
deployments
long-running processes, deploying with 148-150
updating 151
descheduler 163
reference link 164
development lifecycle, large-scale Kubernetes deployments 618
device plugin framework
reference link 667
Digital Ocean 59
directed acyclic graph (DAG) 447
Direct Server Return (DSR) 343
discovery and load balancing resource category 13
distributed data-intensive apps 236
distributed hash table (DHT) algorithm 243
distributed systems design patterns 10
adapter pattern 11
ambassador pattern 10
level-triggered infrastructure 11

multi-node patterns 11
reconciliation 11
sidecar pattern 10
distributed tracing 447
with Kubernetes 468
DNS 17
external data stores, accessing via 237
DNS autoscaling
reference link 627
Docker 20, 21
DOKS (Digital Ocean Kubernetes Service) 60
Domain Name System (DNS) 326, 330, 331
reference link 330

E

Earliest Departure Time (EDT)-based rate-limiting 343
ECI (Elastic Container Instances) 60
edge computing 618
EKS Anywhere 64
elastic cloud resources
autoscaling instances 87
benefiting from 87
cloud quotas 87
container-native solutions, considering 88, 89
regions, managing 87
Elastic Kubernetes Service (EKS) 88
emptyDir
using, for intra-pod communication 186-188
enterprise software
Kubernetes, using 565, 566
requirements 566
enterprise storage
integrating, into Kubernetes 227
environments, large-scale Kubernetes deployments 619
separated environments 619
staging environment 619
environment variables
ConfigMap, consuming as 238
external data stores, accessing via 237
Envoy 484
URL 484
Envoy proxies
tasks 487
workflow, in Kubernetes 487
etcd 15
etcd3 92
gRPC 92
leases, with TTLs 92
state storage 92
watch implementation 92
even external load balancing 349
event consumer 420
event registry 421
event source 420
event types 421
extended Berkeley Packet Filter (eBPF) 340, 666
applications 666, 667
URL 341
external data stores
accessing, via DNS 237
accessing, via environment variables 237
external load balancers 347
configuring 347
configuring, via kubectl 348
configuring, via manifest file 348
external services 152

F

FaaS solutions 405
characteristics 405

fallback node pool 635

Fargate 59, 408
limitations 409
URL 408

Federation V1 373

feeds, in Clusternet 381

FireCracker 666
reference link 405

Fission 436, 437
executors 437, 438
experimenting with 441, 442
NewDeploy executor 438
PoolManager executor 438
URL 436
workflows 438, 439

Fluentbit 454

Fluentd 453
URL 453
using, for log collection 453, 454

fully qualified domain name (FQDN) 331

G

Galley 488

Gardener project 393
conceptual model 394
extensibility feature 397-401
reference link 393
structure 394
terminology 393

Gardener project architecture 395
clusters, monitoring 396
cluster state, managing 395
control plane, managing 395
gardenctl CLI 396
infrastructure, preparing 395
Machine controller manager, using 395
networking across clusters 396

Gatekeeper 570
architecture 571
URL 570

GCE persistent disk 215, 216

generic ephemeral volumes 202, 203

geo-distributed clusters 617

GKE Anthos 64

GlusterFS 219
endpoints, creating 220
Kubernetes service, adding 220
pods, creating 220, 221
using 219

go-k8s
reference link 527

Google Anthos 667

Google Cloud Filestore 216

Google Cloud Platform (GCP) 57, 613

Google Cloud Run 409
models 410

Google Compute Engine (GCE) 185

Google for Startups cloud program
URL 658

Google gVisor 666

Google Kubernetes Engine (GKE) 57, 88
Autopilot project 57

GPT (Generative Pre-trained Transforms) 669

Grafana 465
metrics, visualizing with 465-467
URL 465

gRPC 92

H

Helm 297
alternatives 323
chart, customizing 307, 308
charts, finding 300, 301

charts, managing with 311, 312
functionalities 297
installation options 308
installation status, checking 305, 306
installing 300
packages, installing 304
release, deleting 310
release, rolling back 309, 310
release, upgrading 308, 309
repositories 311
repositories, adding 301-304
starter packs 312
supported use cases 298

Helm 2
versus Helm 3 299

Helm 3 architecture 298

Helm client 299
installing 300

Helm Hub 300

Helm library 299

Helm release secrets 298

high availability 68, 76
best practices 69-74
testing 74, 75

High-Availability (HA) Proxy 352
NodePort, utilizing 353
reference link 353
running, inside Kubernetes cluster 353, 354
with custom load balancer provider 353

high availability requirements
best effort 81
maintenance windows 81
performance and data consistency 84
quick recovery 82
site reliability engineering 84
zero-downtime 82, 83

Homebrew
URL 300

Honeycomb
URL 470

Horizontal Pod Autoscaler (HPA) 12, 76, 425, 638
creating 269-272

horizontal pod autoscaling 268
custom metrics 272
custom metrics, providing 562
with kubectl 274-276

HostPath
using, for intra-node communication 188-190

hot swapping 68

HTTPie
URL 519

HTTP trigger 436

Huawei cloud platform 60

Hue platform
advancing science 182
building, with Kubernetes 143
designing 138
directory structure, configuring 167, 168
for education 182
identity 138
managing, with Kubernetes 176
privacy 138
scope, defining of 138
security 138
smart reminders and notifications 138
utilizing, in enterprise 182

Hue platform, components
authorizer 140
data stores 141
event-driven interactions 141
external services 140
generic actuator 140
generic sensor 140
identity 139
microservices 140
plugins 141

serverless functions 141
stateless microservices 141
user graph 139
user learner 140
user profile 139
Hue platform, workflows 142
automatic workflows 142
budget-aware workflows 142
human workflows 142
hue-reminders service
creating 154, 155
hybrid Kubernetes 617, 618

I

IBM Kubernetes service 60
reference link 61
idempotency 69
infrastructure
cloud-level considerations 622
compute infrastructure 622
handling, at scale 622
networking 623
storage 627
infrastructure provider 376
reference link 376
ingress controllers 156, 157, 350-352
reference link 352
init containers
using, for orderly pod bring-up 179
utilizing 631
inside-the-cluster-network components 176
internal services 152
deploying 152, 153
inter-pod communication 326
inter-process communication (IPC) 257
intra-node communication
HostPath, using for 188-190
intra-pod communication 326
emptyDir, using for 186-188
IoT (Internet of Things) 668
IP address 334
IP address management (IPAM) plugin 337
IP address space management 623, 624
IQN (iSCSI Qualified Name) 227
Isovalent 485
URL 485
Istio 485
access logs 503-506
architecture 485, 486
distributed tracing 509-511
installing 488-490
metrics 506-508
minikube cluster, preparing 488
monitoring 503
observability 503
service mesh, visualizing with Kiali 512
traffic management 494-497
URL 485
working with 494
Istio architecture, components
Citadel 487
Envoy proxy 486, 487
Galley 488
Pilot 487
Istio authentication 499
peer authentication 499
request authentication 499
Istio authorization 500-502
Istio certificate management 499
Istio identity 498
Istio security 497
architecture 498

J

Jaeger 469
architecture 470
client libraries 471
features 470
installing 471-473
URL 469
Jaeger agent 471
Jaeger collector 471
Jaeger ingester 471
Jaeger query 471
Java Management Extensions (JMX) 251
JMESPath
URL 586
jobs
completed jobs, cleaning up 173
cron jobs, scheduling 174, 175
launching 171, 172
running, in parallel 172, 173
jq
URL 519
jsPolicy 572
architecture 572
URL 572

K

k3d 49, 50
comparing, with Minikube and KinD 54
installing 50
multi-node cluster, creating with 49-53
reference link 54
k3s 50
K9S 28
reference link 551
Kapp-controller 324
URL 324
Karmada 377
additional capabilities 379
architecture 378
OverridePolicy 379
PropagationPolicy 379
reference link 380
ResourceTemplate 379
URL 377
Kiali 512
used, for visualizing service mesh 512
KinD 44
comparing, with Minikube and k3d 54
installing 44
multi-node cluster, creating with 43-48
reference link 54
Klusterlet 386
Knative 410
scale to zero option 422
Knative channels 421
Knative eventing 419
architecture 421, 422
broker 420
channels 421
event consumer 420
event registry 421
event source 420
event types 421
subscriptions 421
trigger 420
Knative serving 410
Configuration object 417
quickstart environment, installing 411-413
revisions, creating 415
Route object 415, 416
Service object 413
Kong Mesh 484

kOps
reference link 57
Krew 552
reference link 552
used, for managing kubectl plugins 552, 553
Kubebuilder 536
reference link 536
Kube controller manager 15
kubecost
URL 657
kubectl 27, 143, 144, 274
alternative tools 28
benefits 529
invoking, from Python and Go 529
reference link 530
running, Python subprocess used 529-532
kubectl manifest files 144
apiVersion 144
container spec 145
kind 144
metadata 145
spec 145
kubectl plugins 552
custom plugin, creating 553, 554
flat namespace, for Krew plugins 555
issues 554
kubectl commands, overriding 554
managing, with Krew 552, 553
naming 554
writing 551
KubeEdge 668
kubelet 18
Kubemark 97
Kubemark cluster
comparing, to real-world cluster 98
setting up 97
Kubenet 335
MTU, setting 336
requirements 335
kube-prometheus
reference link 458
Kubernetes API 11, 12, 516-518
accessing, via Python client 520
additional extension points 562
controlling, controller-runtime used via go-k8s 527-529
controlling, Go and controller-runtime used 527
CoreV1Api group, dissecting 520-523
custom metrics, providing for horizontal pod autoscaling 562
exploring, Postman used 518
extending 533
extending, with cloud controller manager 534
extending, with controllers and operators 535
extending, with custom container runtimes 536, 537
extending, with custom storage 563
extending, with plugins 534
extending, with webhooks 535
extension points and patterns 534
extension, scheduling 536
kubectl, invoking from Python and Go 529
objects, creating 523, 524
objects, listing 523
objects, watching 525, 526
OpenAPI 515
output, filtering with HTTPie and jq 519, 520
pod, creating via 526
proxy, setting up 516
resource categories 12
working with 515
Kubernetes API server 15
Kubernetes architecture 5, 9, 10
annotation 7
cluster 5
control plane 6

distributed systems design patterns 10
label 6
label selectors 7
name 9
namespace 9
node 5
pod 6
replica set 8
replication controllers 8
secret 9
service 7
StatefulSet 8
volume 8

Kubernetes Certified Service Provider (KCSP) 663

Kubernetes Cluster Federation
history 373, 374

Kubernetes clusters
creating, in cloud 56

Kubernetes components 15
control plane components 15
node components 18

Kubernetes container runtimes 18
containerd 22
Container Runtime Interface (CRI) 18-20
CRI-O 22
Docker 20, 21
lightweight VMs 22

Kubernetes custom scheduler
design 545
manual pod scheduling 548
node filtering, factors 546
pods, assigning 550, 551
preparing 549
workflow 546, 547
writing 545

Kubernetes Event-Driven Autoscaling (Keda) 273
architecture 273
URLs 80, 274, 425, 562

Kubernetes, future challenges 671
complexity 671
serverless function platforms 671, 672

Kubernetes Gateway API 355
reference link 357
resources 355
routes, attaching to gateways 356
working 356, 357

Kubernetes governance 567
admission control 568
audit logs 569
configuration constraints 568
image management 567
network policy 567
pod security 567
policy management 568
policy validation and enforcement 568
RBAC (Role-Based Access Control) 568
reporting 569

Kubernetes-like control planes
building 545

Kubernetes networking model 325

Kubernetes networking solutions 341
bridging, on bare-metal clusters 341
Calico project 342
Cilium 342
Weave Net 342

Kubernetes network plugins 333
basic Linux networking 334
CNI 336
Kubenet 335

Kubernetes on EC2 57
features 58

Kubernetes operators 629

Kubernetes PaaS (Platform as a Service) 665

Kubernetes plugins
custom scheduler, writing 545
kubectl plugins, writing 551
writing 545

Kubernetes training partners
reference link 664

Kubernetes watch trigger 436

Kube scheduler 17

Kubespray 63
used, for building bare metal cluster 63

kube-state-metrics 461, 462

KubeWarden 572, 573
architecture 573
URL 572

kugo package
reference link 531

KUI 29
reference link 29

Kuma 484
URL 484

kustomizations
applying 168, 169
patching 169, 170
staging namespace 170, 171

kustomize 324
basics 166, 167
URL 166, 324

Kyverno 571, 573
architecture 574
configuring 577-580
features 574, 575
installing 575-577
pod security policies, installing 578
policies, applying 580-583
URL 571

Kyverno CLI 598-600
reference link 598

Kyverno policies 583
advanced policy rules 591
autogen rules, for pod controllers 592
external data sources 591, 592
generating policies, writing 597, 598
mutating policies, writing 595-597
preconditions, using 586
requests, matching 584-586
requests, validating 587
resources, excluding 586
resources, generating 590, 591
resources, mutating 588, 589
rules 584
rule structure 584
settings 583
testing 592, 598
validating policies, writing 592-595
writing 592

Kyverno reports
viewing 606-611

Kyverno tests 601, 602
running 605, 606
writing 603-605

L

labels 6
pods, decorating with 147

label selector 7

large language models (LLMs) 669

large nodes
versus small nodes 632-634

large-scale Kubernetes deployments
development lifecycle 618
environments 619
observability 621

permissions and access control 620
promotion process 620
leader election 68
Lens 29
reference link 30
Lens Desktop
reference link 551
level-triggered infrastructure 11
Lightstep
URL 470
lightweight VMs 22
limit ranges
using, for default compute quotas 287, 288
limits 615
real-world examples 615
scarce resources, handling with 279
Linkerd 484
Linkerd 2 484
URL 484
Liqo 392
architecture 393
reference link 393
live cluster updates 261
rolling updates 262-264
liveness probes
using 177
load balancer IP addresses
finding 348
load balancing 346
external load balancers 347
ingress controllers 350-352
Kubernetes Gateway API 355
service load balancers 350
local volumes 196
creating 196, 197
using, for durable node storage 190, 191
log collection
Fluentd, using for 453, 454
log collection strategy, selecting 450
direct logging, to remote logging service 450, 451
node agent 451
sidecar container 452
logging 446
logging, with Kubernetes 448
centralized logging 450
component logs 450
container logs 449
Fluentd, for log collection 453
logs 446
aggregation 446
formats 446
storage 446
Loki 467, 468
URL 467
long-running microservices
deploying, in pods 146
long-running processes
deploying, with deployments 148-150
long-running services
running, on serverless infrastructure 404
lookup and discovery 328
ingress object 329
loosely coupled connectivity with data stores 329
loosely coupled connectivity with queues 328
self-registration 328
services and endpoints 328
loopback plugin 358-362
LUN (Logical Unit Number) 227

M

macOS
Minikube installation 35, 36
Rancher Desktop installation 26

Mæsh 484
URL 484
maintenance windows 81
managed clusters
provisioning 628
Managed Kubernetes 613, 614
limitations 616
managed nodes
utilizing 629
management cluster 375
maximum transmission unit (MTU) 335
message queue trigger 436
metadata resource category 14
MetalLB 354
reference link 354
metrics 447, 454
collecting, with Kubernetes 454
visualizing, with Grafana 465-467
Metrics API 454
Metrics Server
monitoring with 455, 456
microservice
issues 479
Microsoft Azure 613
Microsoft for Startups program
URL 658
Minikube 31
advanced options 31
capabilities 31
comparing, with KinD and k3d 54
installation, troubleshooting 37
installing 31
installing, on macOS 35, 36
installing, on Windows 31-35
reference link 54
single-node cluster, creating with 30
minikube cluster
preparing, for Istio 488
minions 5
monitoring 446
monit process 72
MTTR (mean time to recovery) 82
multi-cloud 617
multi-cluster Kubernetes 372
benefits 373
disadvantages 373
running, reasons 372
multi-node cluster, with k3d
creating 49, 51-53
multi-node cluster, with KinD
creating 44-48
Docker contexts, handling 44, 45
echo service, accessing with proxy 49
echo service, deploying 48, 49
multi-node patterns 11
multiple clusters
managing 617
multi-tenant cluster
case 129
namespace pitfalls, avoiding 130, 131
namespaces, using 129, 130
running 128
virtual clusters, using 131-134

N

name 9
namespaces 9
using, to limit access 164-166
native CI/CD 668
netmasks 334
Network Address Translation (NAT) 325
networking, future trends 666
networking infrastructure 623
cross-cloud communication 625
cross-cluster communication 624, 625

cross-cluster service meshes 625, 626
DNS, managing at cluster-level 627
egress, managing at scale 626, 627
IP address space management 623, 624
network segmentation 624
network topology 624

network namespaces 334

network policies
CNI plugins 344
configuring 344
costs 124
cross-namespace policies 124
defining 122, 123
design 344
egress, limiting to external networks 124
implementing 345
managing 121
supported networking solution, selecting 122
using, effectively 344

network segmentation 624

network topology 624

node 5

node affinity 160
advantages 160
and anti-affinity 161

node agent 451

node-agent service mesh architecture 483

node components 18
kubelet 18
proxy 18

node pool breakdown
designing 623

node pools
concurrent draining 643
managing 628
provisioning 628
upgrades, performing 642
upgrading 642

node selector 158

non-cluster components
mixing 176

O

object count quota 281, 282

observability 446
alerting 448
application error reporting 448
dashboards and visualization 448
distributed tracing 447
logging 446
metrics 447

observability, large-scale Kubernetes deployments 621
observability stack, troubleshooting 621
one-stop shop observability 621

octant
reference link 551

OpenAPI 515
URL 516

Open Cluster Management (OCM) 385
application lifecycle 387, 388
architecture 386
cluster lifecycle 387
governance model 388
reference link 389
registration process 387

Open Container Initiative (OCI) 21

OpenFaaS 423
architecture 425
delivery pipeline 423
installing 426, 427
URL 423
using 429-436

OpenFaaS, features 424
autoscaling 425
function invocations and triggers 424

function management 424
metrics 424
web-based UI 425
Open Policy Agent (OPA) 570
architecture 571
Rego 571
URL 570
Open Service Mesh (OSM) 485
URL 485
OpenTelemetry 468
tracing concepts 469
URL 468
Open Virtual Networking (OVN) 337
operator pattern 536, 669
reference link 536
Oracle Container Service 61
reference link 61
outside-the-cluster-network components 176

P

PaaS (Platform as a Service) 2
patching 169, 170
pending pods
handling 644-649
permanent pending pods 646
temporarily pending pods 646
permanent pending pods 646
permissions and access control, large-scale Kubernetes deployments 620
break glass access 621
permission model, fine-tuning 620
permissions, assigning to groups 620
principle of least privilege 620
persistent volume claims
applying 239
making 197-199
persistent volumes
access modes 193
capacity 193
creating 192
mount options 194
provisioning 191
provisioning, dynamically 192
provisioning, externally 192
provisioning, statically 192
reclaim policy 194
storage class 194
volume node 193
volume type 194
persistent volume storage end to end
demonstrating 205-210
personally identifiable information (PII) 453
Pilot 487
plugins
Kubernetes, extending with 534
pod affinity
and anti-affinity 161, 162
pod density 635
Pod Lifecycle Event Generator (PLEG) 91
pod networking 335
pod readiness 180
pods 6, 186, 325
creating 146, 147
creating, via Kubernetes API 526, 527
decorating, with labels 147
long-running microservices, deploying 146
unique ID (UID) 6
pod security 115
AppArmor, using 118, 119
cluster, protecting with AppArmor 117
ImagePullSecrets 115
private image repository, using 115

security context, specifying for pods and containers 116, 117
standards 117
Pod Security Admission 121
pod topology spread constraints 162
pod-to-service communication 326, 327
policy engines 569
admission control 569
Gatekeeper 570, 571
jsPolicy 572
KubeWarden 572, 573
Kyverno 571
Open Policy Agent (OPA) 570, 571
responsibilities 570
reviewing 570
ports 334
Postman 518
URL 518
used, for exploring Kubernetes API 518
principle of least privilege 620
priority classes 282
private clouds 665
projected volumes 195
serviceAccountToken projected volumes 196
Prometheus 457, 458
alerting, with Alertmanager 463
architecture 458
custom metrics, incorporating 463
features 457
installing 458-460
kube-state-metrics 462
kube-state-metrics, incorporating 461
node exporter, utilizing 462
URL 457
working with 460, 461
protected health information (PHI) 453
protocol buffers
API objects, serializing with 92
proxy injection approach
attributes 481
public cloud Kubernetes platforms 665
public cloud storage volume types 210
AWS Elastic Block Store (EBS) 210-212
AWS Elastic File System (EFS) 212-214
Azure data disk 217
Azure file 218
GCE persistent disk 215, 216
Google Cloud Filestore 216
Pulumi 292, 293, 629
pure cloud functions models 672
Python subprocess
used, for running kubectl 529-532

Q

queues
benefits 328, 329
downsides 329
quick recovery 82
quotas 615
creating 283-287
namespace-specific context 283
real-world examples 615
scarce resources, handling with 279
scopes 282
working with 283

R

Rancher Desktop 26
installation methods 26, 27
installing, on macOS 26
installing, on Windows 26
Rancher Desktop Kubernetes cluster 54, 55
Rancher RKE 63
reference link 63
used, for building bare metal cluster 63

raw block volumes 200, 201
RBAC (Role-Based Access Control) 58
RBD
used, for connecting to Ceph 221
readiness gates 180
readiness probes
used, for managing dependencies 178
reconciliation 11
redundancy 68
redundant in-memory state
using 238, 239
redundant persistent storage
DaemonSet, using for 239
Rego 571
remote central logging 452, 453
remote logging service
direct logging to 450, 451
replica set 8
replication controllers 8
requests validation, Kyverno 587
deny-based validation 588
pattern-based validation 587
reserved instances 658
resource categories, Kubernetes APIs
cluster 14
config and storage 14
discovery and load balancing 13
metadata 14
workloads 12, 13
resource quotas 620
compute resource quota 280
enabling 279
limits 283
object count quota 281, 282
priority classes 282
requests 283
storage resource quota 280
resource utilization 630
rolling updates 262-264
blue-green deployments 265, 266
canary deployments 266
complex deployments 264
performing, with autoscaling 276-278
Rook 223
architecture 224
features 223
routing 334
running pods
handling 652-655
run.sh script 247-251

S

scalability 76
scarce resources
handling, with limits 279
handling, with quotas 279
scheduled pods
handling 650-652
scheduler
guiding principles 158
SchedulingStrategy 381
secrets 9, 125
creating 126
decoding 126
encryption, configuring at rest 125
managing, with Vault 128
storing, in Kubernetes 125
using 125
using, in container 127, 128
security challenges, Kubernetes 99, 100
configuration and deployment challenges 104
image challenges 103
network challenges 101-103
node challenges 100, 101

organizational, cultural, and process challenges 105, 106
pod and container challenges 105
self-healing 69
sensitive log information
handling 453
Sentry 448
URL 176
separated environments 619
serverless computing 403, 404, 668
serverless infrastructure
functions, running as service on 404, 405
long-running services, running on 404
serverless Kubernetes
in cloud 405
service 7
exposing, externally 155, 156
service accounts 107
managing 108
serviceAccountToken projected volumes 196
Service Level Objectives (SLOs) 84, 93
service load balancers 350
service mesh 479-483, 667
AWS App Mesh 484
Cilium Service Mesh 485
Envoy 484
Istio 485
Kuma 484
Linkerd 2 484
Mæsh 484
Open Service Mesh (OSM) 485
selecting 483
service mesh architecture
in Kubernetes 482
Service Mesh Interface (SMI) 485
URL 485
shared nodes
versus dedicated nodes 631, 632
short-lived workloads 634
sidecar container 10, 452, 667
sidecar container KEP
reference link 667
sidecar service mesh architecture 481
SigNoz
URL 470
single-node cluster, with Minikube
checking 38, 40
creating 30
examining, with dashboard 42, 43
working 40, 41
single point of failure (SPOF) 329
site reliability engineering 84
SLAs (service level agreements) 84
SLIs (service level indicators) 84
small nodes
versus large nodes 632-634
small workloads 634
smart load balancing 69
Span 469
Special Interest Group (SIG) 373
spot instances 659
staging environment 619
staging namespace
kustomizations, applying 170, 171
startup probes
using 178
state
managing, outside of Kubernetes 236
managing, reason 236
stateful application 235
StatefulSets 8
components 239-241
properties 8
usage scenarios 239
using, to create Cassandra cluster 255

utilizing 239
working with 241-243
StatefulSet YAML file 255-259
stateless application 235
storage capacity tracking 231
storage classes 204, 205
default storage classes 205
storage infrastructure 627
data backup and recovery 627
data security 628
right storage solutions, selecting 627
storage monitoring 628
storage performance, testing 628
storage performance, validating 628
storage usage, optimizing 628
storage providers 228
storage resource quota 280
stretched Kubernetes clusters 372
benefits 372
disadvantages 372
subnets 334
subscribers, Clusternet 381
subscriptions, Knative 421

T

tagging 656
taints 159, 160
Tekton
reference link 410
template files
writing 318, 319
writing, pipelines and functions used 319
templates
using 317
temporary pending pods 646
Tencent 60
tensile-kube 390, 395
architecture 390
features 390
Terraform 291, 629
issues 292
URL 291
workflow 292
timer trigger 436
Time to Live (TTL) 92
TKE (Tencent Kubernetes Engine) 60
tolerations 159, 160
Trace 469
tracing concepts, OpenTelemetry
Span 469
Trace 469
Traefik 354
features 354
reference link 355
Træfik 484
URL 484
trigger 420

U

upcoming trends 666
custom hardware and devices 667
KubeEdge 668
native CI/CD 668
networking 666
Operator pattern 669
serverless computing 668
service mesh 667
Upjet project
reference link 629

V

values
feeding, from file 322
using 317

Velero 82
URL 72
vertical pod autoscaler 79
limitations 80
requirements 80
Virtual Ethernet (veth) devices 334
Virtual Kubelet 389, 663
Admiralty 391, 392
features 390
Liqo 392
tensile-kube 390
visualization 448
volume health monitoring 231
volumes 8, 186
claims, mounting as 199
volume snapshots 229, 230
VPC (virtual private cloud) page
reference link 615

W

Weave Net 342
webhooks
Kubernetes, extending with 535
Windows
Minikube installation 31-35
Rancher Desktop installation 26
work cluster 375
workloads
limits, setting 631
requests, setting 630
workload shape 630
workloads resource category 12, 13
workloads, with no PDB
handling 643
workloads, with PDB zero
handling 643, 644
WSL (Windows System for Linux) 32

Y

YAML JMESPath 571
URL 571

Z

zero downtime 82, 83
Zipkin
URL 470

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781804611395`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly