1ST EDITION

Cloud Observability with Azure Monitor

A practical guide to monitoring your Azure infrastructure and applications using industry best practices

JOSÉ ÁNGEL FERNÁNDEZ
MANUEL LÁZARO RAMÍREZ

Cloud Observability with Azure Monitor

A practical guide to monitoring your Azure infrastructure and applications using industry best practices

José Ángel Fernández

Manuel Lázaro Ramírez

‹packt›

Cloud Observability with Azure Monitor

The author acknowledges the use of cutting-edge AI, in this case, Built-in AI tools in MS Word, with the sole aim of enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for readers. It's important to note that the content itself has been crafted by the author and edited by a professional publishing team.

Group Product Manager: Preet Ahuja

Publishing Product Manager: Surbhi Suman

Book Project Manager: Ashwin Kharwa

Senior Editor: Mohd Hammad

Technical Editor: Rajat Sharma

Copy Editor: Safis Editing

Proofreader: Mohd Hammad

Indexer: Tejal Soni

Production Designer: Shankar Kalbhor

DevRel Marketing Coordinator: Rohan Dobhal

First published: November 2024

Production reference: 1251024

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83588-118-7

www.packtpub.com

"Every man should plant a tree, have a child, and write a book. These all live on after us, ensuring a measure of immortality."

In the countryside home my wife and I chose, I've planted several trees—symbols of growth and lasting connection. Our greatest joy is our son, Leo, whose laughter and curiosity inspire me daily. With the completion of this book, I've fulfilled that timeless saying. This work represents years of dedication, and I hope it will guide and inspire others, just as my trees will grow and Leo will thrive.

I want to thank my wife for her support, and Leo for being my greatest inspiration. To all who have supported this journey—colleagues, mentors, and friends—thank you for making this book possible.

– José Ángel Fernández

"Life is like riding a bicycle. To keep your balance, you must keep moving."

We often seek a life of calm, with few disruptions and challenges that come easily and are quickly resolved. However, over time, we realize that true growth is found in facing difficulties head-on. Completing this book has been one such challenge, allowing me to grow a little more along the way. This book represents years of effort and learning and a hope that it will help others navigate their challenges and continue their journeys.

I want to express my deepest gratitude to my wife and father for their unwavering support during the writing process, and to my grandmother for her constant reminder that we must keep pushing forward. A heartfelt thanks also goes to my friends, colleagues, and mentors—your support has been invaluable in bringing this book to life.

– Manuel Lázaro Ramírez

Contributors

About the authors

José Ángel Fernández has worked as a Microsoft Specialist and Cloud Solution Architect, specializing in advanced cloud migrations, with extensive technical expertise and a deep understanding of Azure solutions. He has been focused on the cloud for the last 11 years at Microsoft, starting at the same time virtual machines reached general availability and Azure Monitor was not yet a product.

José Ángel graduated with a degree in telecommunications engineering from the Technical University of Madrid in 2013. He later earned a degree in big data analytics from the Graduate School of Engineering and Basic Sciences of Charles III University of Madrid in 2020.

He resides in Madrid, Spain with his wife, his three-year-old child, and an adopted black cat that has never brought him bad luck.

Manuel Lázaro Ramírez is a Microsoft Cloud Solution Architect with a wide technical breadth and deep understanding of Azure solutions. He has been focused on designing and implementing cloud architectures in different industries for the last 10 years.

Manuel graduated with a degree in pure and applied mathematics from Complutense University of Madrid in 2013 and later earned a master's degree in pure and applied mathematics from Complutense University of Madrid in 2014.

He resides in Madrid, Spain, with his wife, and his passion is developing code with their friends and working and solving real-world business problems with cloud technology to deliver real value.

About the reviewers

Jim Szubryt has been in the computer industry for more than 30 years and has a wide breadth of technology expertise. His experience with complex architectures in the three major clouds spans industries including CPG, financial services, insurance, and manufacturing. Jim has been with Microsoft since 2022, working daily with customers to lead them in achieving strategic business goals with Azure. This is the fourth time he has been a technical reviewer and enjoys assisting with books that deliver great technical value to the reader. Jim is a former Microsoft Application Lifecycle Management MVP (2013-2018) and is active in the Michigan technology user group community.

Yi Wei has worked in software development and cloud computing for more than 11 years. He is currently a principal software engineer manager at Microsoft. During his time at Microsoft, he has built software products and cloud services to provide data analytics and cloud observability capabilities. He has a Bachelor of Computer Science from Huazhong University of Science and Technology, a Master of Science in computer science from the University of Delaware, and a PhD in computer science from the University of Notre Dame.

Table of Contents

3

Exploring Azure Monitor Data Sources and the Ingestion Pipeline 43

Part 2: Working with Azure Monitor

4

Analyzing Your Data Using Logs and Metrics 81

5

Responding to Monitoring Events 113

6

Visualizing Your Logs and Metrics 155

7

Application Observability and Performance Monitoring with Application Insights 189

Part 3: The Road Ahead with Azure Observability

8

9

10

Preface

In an era where digital transformation is at the forefront of organizational strategies, the ability to monitor, manage, and optimize IT infrastructure has never been more critical. As organizations migrate to the cloud, leveraging its scalability and flexibility, they are also confronted with unprecedented complexity. Traditional approaches to monitoring systems are often inadequate in this dynamic and distributed environment. This book is your guide to navigating this new landscape by learning the details of Azure Monitor—a comprehensive suite of monitoring services provided by Microsoft Azure.

Azure Monitor is designed to maximize the availability and performance of your applications and services by delivering a robust monitoring solution. It offers deep visibility into every layer of your IT environment, from infrastructure and network to applications and databases. Azure Monitor helps organizations ensure their systems' reliability, performance, and security through an array of tools described in detail in this book.

The cloud's dynamism requires not just monitoring but true observability—where systems can be examined from multiple perspectives to understand their current state and predict potential issues. Azure Monitor embodies this principle, offering features, such as Application Insights for detailed application monitoring, Log Analytics for deep data analysis, and Azure Metrics for real-time performance tracking.

Who this book is for

This book is for IT professionals and developers working in cloud environments, particularly those utilizing Microsoft Azure. It is designed for individuals seeking a comprehensive understanding of cloud observability and practical skills in implementing and optimizing monitoring solutions on Azure.

Whether you are an IT professional responsible for ensuring the performance and reliability of cloud services, a developer looking to gain deeper insights into your applications' behavior, or a systems engineer focused on optimizing infrastructure, this book provides the essential knowledge and tools you need. It is especially relevant for those working in roles related to the following:

- **Cloud engineering**: Professionals who design, implement, and manage cloud environments will benefit from understanding how to effectively monitor and optimize these environments using Azure Monitor
- **Infrastructure and systems engineering**: Engineers responsible for maintaining the underlying infrastructure of cloud-based services will find the chapters on infrastructure monitoring and hybrid cloud observability particularly valuable

- **Application development**: Developers who want to ensure their applications are performing optimally and are easily maintainable will benefit from the deep dive into application observability and performance monitoring with tools such as Azure Application Insights

This book is suitable for both beginners looking to establish a strong foundation in cloud observability and seasoned professionals seeking to refine their monitoring strategies and stay ahead of industry trends. By the end of this book, you will have the skills and knowledge necessary to implement robust observability practices that align with your organization's goals and help you maintain high-performing, secure, and reliable cloud environments.

What this book covers

Chapter 1, *Introduction to Observability with Azure Monitor*, lays the foundation by exploring the concept of observability, its evolution, and how Azure Monitor serves as a pivotal tool in achieving comprehensive observability in cloud environments.

Chapter 2, *Understanding Azure Monitor Components and Functions*, explores the fundamental building blocks of Azure Monitor, including its core components and how they function together to provide a cohesive monitoring solution.

Chapter 3, *Exploring Azure Monitor Data Sources and the Ingestion Pipeline*, describes the various data sources Azure Monitor can integrate with and explains the processes involved in data ingestion and analysis.

Chapter 4, *Analyzing Your Data Using Logs and Metrics*, covers the methodologies and tools available within Azure Monitor for analyzing logs and metrics, helping you to derive actionable insights.

Chapter 5, *Responding to Monitoring Events*, focuses on the critical aspect of responding to monitoring alerts and events, emphasizing the importance of automated responses and proactive management.

Chapter 6, *Visualizing Your Logs and Metrics*, provides a guide to the visualization tools within Azure Monitor, enabling you to create dashboards that offer a clear view of your system's health and performance.

Chapter 7, *Application Observability and Performance Monitoring with Application Insights*, details how to use Application Insights to monitor the performance and behavior of your applications, ensuring they meet your users' expectations.

Chapter 8, *Hybrid and Multi-Cloud Monitoring*, addresses the challenges and strategies involved in monitoring systems across multiple cloud environments, integrating tools such as Azure Arc and System Center Operations Manager (SCOM). Previous knowledge of those services is recommended to extract the maximum value of this chapter.

Chapter 9, *Integrating with Third-Party Tools*, explores how Azure Monitor can be extended and enhanced by integrating with third-party tools, providing a comprehensive monitoring solution.

Chapter 10, *Building Your Monitoring Strategy*, synthesizes the book's lessons into actionable strategies for building a robust and scalable monitoring framework that aligns with your organization's goals and industry best practices.

Chapter 11, *Cost Management and Optimization*, covers the essential aspects of managing and optimizing costs associated with Azure Monitor. It provides strategies to efficiently utilize resources, control expenses, and maximize the return on your monitoring investments.

Chapter 12, *Future Trends and Looking Ahead*, concludes the book by exploring the future of cloud observability. This chapter discusses emerging trends, including AI-driven analytics for predictive observability, the challenges of serverless architectures, and evolving standards in cloud monitoring.

The *Appendix* offers a practical, in-depth guide for customizing monitoring pipelines within Azure Monitor. It walks you through the configuration options for tailoring these pipelines to specific monitoring needs, with a focus on optimizing data flow for enhanced performance and efficiency.

To get the most out of this book

To fully benefit from the content and practical guidance provided in this book, it is recommended that readers have access to an Azure subscription with sufficient permissions to create and manage resources.

Having the ability to interact directly with Azure Monitor and related services will allow you to follow along with the examples, implement the strategies discussed, and gain hands-on experience.

All of our examples have been tested in the Azure Cloud Shell environment. If you don't have access, the following tools would be required to be installed in your OS.

Software/hardware covered in this book	**OS requirements**
Azure PowerShell	Windows, macOS, or Linux
Azure CLI	Windows, macOS, or Linux
.Net 8	Windows, macOS, or Linux

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Cloud-Observability-with-Azure-Monitor`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

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

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `mv-expand` operator expands multivalued fields into multiple rows, making it easier to work with arrays or lists stored in a single column."

A block of code is set as follows:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddApplicationInsightsTelemetry();
builder.Services.AddRazorPages();
```

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

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DataContracts;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
```

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

```
az deployment group create \
--name {deployment_name} \
--resource-group {resource_group_name} \
--template-file ./dcr.bicep
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "After providing all the necessary details, click **Review + create** to review your configuration. Then, click **Create** to create the Application Insights resource."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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 would report this to us. Please visit `www.packtpub.com/support/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@packt.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 `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Cloud Observability with Azure Monitor*, 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/9781835881187`

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

Part 1: Fundamentals of Observability and Azure Monitor

In this part, you will gain a solid understanding of cloud observability principles and Azure Monitor fundamentals, laying the groundwork for effective cloud monitoring and management.

It starts with a general overview of what observability is, what its pillars are, how it can help obtain real-time insights for performance optimization, and how it can enhance our reliability and security through observability. After that, you would be able to understand how Azure Monitor aligns with those pillars and learn the fundamentals of Azure Monitor services and data sources.

This part includes the following chapters:

- *Chapter 1, Introduction to Observability with Azure Monitor*
- *Chapter 2, Understanding Azure Monitor Components and Functions*
- *Chapter 3, Exploring Azure Monitor Data Sources and the Ingestion Pipeline*

1

Introduction to Observability with Azure Monitor

In the current technology landscape, **cloud computing** has become the standard for many organizations, presenting new challenges and opportunities for managing and optimizing IT operations. One crucial aspect of cloud computing is **observability**, which enables you to view and comprehend the inner workings of your cloud environment. Observability goes beyond mere **monitoring**, the process of collecting data, by providing profound insights into your cloud resources, applications, and services. These insights enable you to optimize performance, improve reliability, and ensure security.

This chapter covers the basics of cloud observability, how it diverges from conventional monitoring, and its importance in contemporary IT operations. Additionally, we will introduce Azure Monitor, Microsoft's observability platform, and explore how its features and services contribute to observability.

The learnings from this chapter will provide a comprehensive understanding of cloud observability and its significance in modern IT operations. You will gain insight into the differences between monitoring and observability, and how observability can help you optimize performance, improve reliability, and ensure security in your cloud environments. Furthermore, you will become familiar with Azure Monitor, Microsoft's observability platform.

We'll cover the following topics:

- Are observability and monitoring the same?
- Understanding cloud observability fundamentals
- Real-time insights for performance optimization
- Enhancing reliability through observability
- Securing cloud environments with observability
- Azure Monitor, your cloud observability platform

Are observability and monitoring the same?

Observability and monitoring are related concepts, but they are not exactly the same. The key differences between them are as follows:

- **Purpose**: Monitoring is primarily focused on detecting and responding to anomalies, while observability is focused on providing a deeper understanding of the system.
- **Approach**: Monitoring typically involves setting up thresholds and alerts, while observability involves collecting data from multiple sources and presenting it in a way that allows operators to explore and analyze the data.
- **Timeframe**: Monitoring is often in real time, while observability can involve analyzing data over a longer period of time to identify trends and patterns.

Here's a brief overview of each concept and how they differ:

	Monitoring	Observability
Definition	Monitoring refers to the process of collecting data about a system's performance, health, and behavior over time.	Observability refers to the ability to gain insight into a system's internal workings and understand why it behaves in certain ways.
Goal	The primary goal of monitoring is to detect and respond to anomalies, issues, and unexpected changes in the system.	The primary goal of observability is to provide a deeper understanding of the system, its components, and their relationships so that developers and operators can make informed decisions about how to improve, debug, or optimize the system.
Activities	Monitoring typically involves setting up thresholds, alerts, and notifications to notify operators when something goes wrong.	Observability typically involves collecting data from multiple sources, including logs, metrics, traces, and dumps, and presenting it in a way that allows operators to explore and analyze the data.

Table 1.1 – Differences between observability and monitoring

In the end, both monitoring and observability are complementary practices that serve different purposes. Monitoring is essential for detecting and responding to immediate issues, while observability provides a deeper understanding of the system, which can help operators identify potential issues before they become problems and make informed decisions about how to improve, debug, or optimize the system.

After clarifying the differences between monitoring and observability, let's continue with the fundamentals of cloud observability.

Understanding cloud observability fundamentals

Observability is a fundamental concept in science and engineering that refers to the ability to perceive and understand the internal workings of a system or process. According to the *Oxford English Dictionary* (`https://www.oed.com`), observability is defined as *"the quality of being observable, a specific property of a system that allows an external entity to observe or take notice of the system."*

The origins of observability can be traced back to the scientific method, which relies heavily on observation and measurement to understand natural phenomena. Throughout history, scientists have developed increasingly sophisticated tools and techniques to observe and measure various aspects of the physical world, from the behavior of celestial bodies to the properties of subatomic particles. Instruments such as telescopes, microscopes, and sensors have enabled scientists to collect data and make observations that were previously impossible.

However, it was not till 1960 that a proper definition of the concept was established by Rudolf E. Kálmán in his paper *On the general theory of control systems* (R. Kálmán, *On the general theory of control systems*, IRE Transactions on Automatic Control, vol. 4, no. 3, pp. 110–110, Dec. 1959, doi: `https://doi.org/10.1109/tac.1959.1104873`). Although observability seems like an original concept associated with the evolution or expansion of the cloud, it is inherited from the field of **Control Theory**. Control Theory is *a field of engineering and mathematics that is focused on the control of dynamical systems in engineered processes and machines* (Wikipedia Contributors, *Control theory*, Wikipedia, Mar. 16, 2019, `https://en.wikipedia.org/wiki/Control_theory`). Kálmán introduced this concept as the property of a system that allows us to understand its status based only on the measurements of its outputs.

In engineering, observability has played an essential role in the design and optimization of systems. Engineers use observability to understand how their creations behave under different operating conditions, identify potential problems, and optimize performance. For example, in mechanical engineering, observability is crucial for understanding the dynamics of machines, while in electrical engineering, it helps engineers analyze circuits and ensure they are functioning correctly.

The concept of observability has been gaining traction in recent years, especially in the context of modern software development and deployment practices. As systems become more complex and distributed, it becomes increasingly difficult to monitor and manage them effectively. Observability provides a way to gain insights into the internal workings of these systems, allowing teams to understand how they behave and respond to different conditions.

Developers need to see inside their code and understand how it interacts with other components and the environment to fix bugs, improve performance, and add new features. Similarly, in DevOps, observability tools help teams collaborate more effectively by providing visibility into the entire software delivery pipeline. Not only that but cloud computing also introduces new ways in which the hardware and software can be used; ephemerality of the resources and their distributed nature is a new paradigm compared with not so many years ago. **Virtual machines**, **containers**, and **serverless functions** can spin up and down quickly, making it harder to monitor and troubleshoot issues.

The cloud has enabled the easy gathering of vast amounts of data from diverse sources. This data can then be analyzed using machine learning algorithms, statistical models, and visualization techniques to extract valuable insights. Overall, the principles of observability that Kálmán defined for manufacturing plants are indeed applicable to modern software development and deployment practices. By adopting observability strategies, teams can build more robust, scalable, and reliable systems that meet the needs of their users.

Now that we've discussed the origins of observability, let's take a closer look at its main components when it comes to cloud environments. They are usually known as the three pillars.

The three pillars of observability

Observability can sometimes feel like an abstract or vague concept. Observing something implies having a clear view of it, but when we're talking about complex systems and infrastructures, it can be hard to know where to start.

Cindy Sridharan, in their book *Distributed Systems Observability* (C. Sridharan, *Distributed Systems Observability*. O'Reilly Media, Inc., 2018), defines the idea of the three pillars of observability: **metrics**, **logs**, and **traces**. These pillars provide a framework for thinking about observability in a more structured way, helping us to break down the nebulous concept of observability into smaller, more manageable pieces.

By looking at our systems through the lens of these three pillars, we can start to identify specific areas where we can improve our observability and get a clearer view of what's happening in our environment.

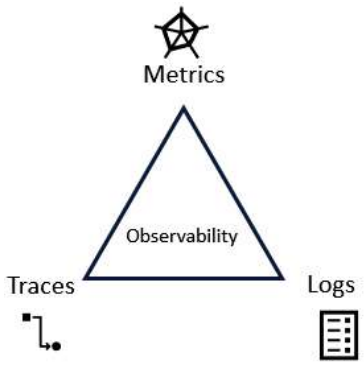

Figure 1.1 – The three pillars of observability

Let's go into more detail:

- **Metrics**: Metrics are quantitative measurements of system behavior, such as response time, error rate, and throughput. They provide a numerical view of system performance and help answer questions such as *How fast?* and *How many?*. Metrics are often collected using tools such as New Relic, Prometheus, or Datadog.

Metrics provide a high-level view of system performance and help teams identify trends and patterns. They're useful for monitoring resource utilization, response times, and error rates. Without metrics, teams might struggle to identify issues or optimize system performance.

- **Logs**: Logs are qualitative information about system behavior, such as events, errors, and warnings. They provide context and help answer questions such as *What happened?* and *Why did it happen?*. Logs are often collected using tools such as Elasticsearch, Logstash, Kibana (ELK), Splunk, or Sumo Logic.

  Logs provide context and detail about system behavior, helping teams understand the reasons behind metric fluctuations. They're useful for investigating issues, identifying edge cases, and understanding how systems interact with each other. Without logs, teams might struggle to diagnose issues or understand system behavior.

- **Traces**: Traces are detailed, step-by-step records of system behavior, such as the path a request takes through a distributed system. They provide a complete picture of system behavior and help answer questions such as *How did it happen?* and *What was the sequence of events?*. Traces are often collected using tools such as OpenTelemetry, Jaeger, or Zipkin.

  Traces provide a complete picture of system behavior, showing the sequence of events and how they relate to each other. They're useful for understanding complex distributed systems, identifying bottlenecks, and optimizing system performance. Without traces, teams might struggle to understand how systems interact with each other or identify performance bottlenecks.

Together, these three pillars provide a comprehensive view of system behavior, enabling teams to quickly identify issues, understand their root cause, and optimize system performance.

Cloud observability tools and techniques

To overcome the challenges introduced at the beginning of this chapter, cloud observability tools and techniques focus on collecting data from various sources, including the following:

- **Application performance monitoring** (**APM**): Tracks the performance and latency of application transactions, identifying bottlenecks and issues affecting user experience. APM tools typically rely on agents embedded within the application code or infrastructure. Those tools focus on collecting metrics and traces of our applications.
- **Log management**: Collects and analyzes logs from various sources, such as application servers, databases, and load balancers. Logs provide rich contextual information about system behavior, helping operators diagnose issues and investigate security breaches. As the name says, these tools focus on logs.
- **Network monitoring**: Uses tools such as packet captures, network taps, and flow records to monitor network traffic, detect anomalies, and troubleshoot connectivity issues. Network monitoring tools may also employ machine learning algorithms to baseline normal behavior and alert on unusual patterns. Those tools focus on metrics and logs.

- **Infrastructure monitoring**: Focuses on monitoring the health and utilization of virtual machines, containers, and other infrastructure components. This includes tracking CPU usage, memory consumption, disk I/O, and network bandwidth. Those tools focus on metrics and logs.
- **Security monitoring**: Encompasses the detection and response to security threats, such as intrusion attempts, unauthorized access, and data breaches. Security monitoring tools often leverage artificial intelligence and machine learning to identify suspicious activity and reduce false positives. Those tools collect all three pillars to provide a global overview of the current security status.
- **Service monitoring**: Ensures that cloud-based services, such as AWS Lambda, Azure Functions, or Google Cloud Functions, operate smoothly and efficiently. Service monitoring tools track service availability, response times, error rates, and other key performance indicators. Those tools are focused on metrics.
- **Container monitoring**: With the rise of containerization, observability tools must now monitor container performance, resource usage, and interactions between containers. Kubernetes, Docker, and other container orchestration platforms provide built-in monitoring capabilities, but third-party tools can offer additional functionality. Those tools focus on metrics and logs.
- **Serverless monitoring**: As serverless architectures become more popular, observability tools need to adapt to the unique characteristics of these environments. This involves monitoring event-driven functions, tracking request/response cycles, and profiling code execution. Those tools focus on metrics and traces.
- **Cloud provider monitoring**: Cloud providers offer monitoring tools that allow you to monitor their platform itself. Those tools allow you to understand whether the provider is having any issue that is producing a disruption to your service. For example, Azure exposes that information through Azure Service Health.

This is not an exhaustive list but covers the most relevant scenarios in a common cloud enterprise environment. It shows that observability in cloud computing requires careful planning, implementation, and maintenance of all those monitoring tools and practices.

It's essential to choose the right combination tailored to your specific use case, workload, and cloud environment.

Now the concept of observability is well understood, let's analyze the three primary cloud scenarios where an effective observability strategy provides value: performance, reliability, and security.

Real-time insights for performance optimization

The performance of our cloud applications and services is an important consideration from our perspective because it directly impacts the user experience and the efficiency of the organization. Slow-performing applications or systems can lead to frustrated users, decreased productivity, and lost business opportunities. Moreover, poor performance can also result in increased costs, as it can lead to unnecessary expenses on hardware, software, and personnel.

Real-time insights can be used in various industries, such as finance, healthcare, retail, manufacturing, and transportation, to name a few. In finance, they can be used to monitor trading activity, detect fraud, and optimize portfolio performance. In healthcare, they can be used to monitor patient vital signs, detect disease outbreaks, and optimize treatment plans. In retail, they can be used to monitor sales, optimize pricing, and personalize customer experiences. In manufacturing, they can be used to monitor production lines, optimize workflows, and predict maintenance needs. In transportation, they can be used to monitor traffic patterns, optimize routes, and predict maintenance needs for vehicles.

In our cloud environment, they provide the ability to gain immediate visibility into the performance and behavior of a system, process, or application, and to analyze and act on that data in real time. Real-time insights are typically obtained using analytics tools and technologies that can process and analyze large amounts of data quickly and efficiently, providing actionable information to stakeholders in a timely manner.

With real-time insights, you can do the following:

- **Detect issues proactively**: By continuously monitoring your systems and applications, you can detect issues before they become critical. This enables you to take preventative measures to avoid downtime and improve overall system performance.
- **Identify root causes quickly**: When an issue does arise, real-time insights help you quickly identify the root cause. You can see exactly what's happening in your system when the issue occurs, which makes it easier to determine the cause and take corrective action.
- **Optimize system performance**: Real-time insights allow you to monitor system performance in real time, so you can adjust performance as needed. For example, you can adjust resource allocation, modify caching settings, or tweak database queries to improve responsiveness.
- **Reduce mean time to detect and mean time to recovery**: Real-time insights enable you to detect issues faster and respond to them more quickly. This reduces the **mean time to detect** (**MTTD**) and **mean time to recovery** (**MTTR**), which can help you meet the **service-level agreements** (**SLAs**) and improve overall system reliability.
- **Better capacity planning**: Real-time insights help you understand how your systems and applications are performing under different loads and conditions. This information can be used to plan capacity more effectively, ensuring that you have sufficient resources to handle peak demand and avoid unnecessary expenses.
- **Improve DevOps collaboration**: Real-time insights can be shared across teams, which fosters collaboration and communication between development, operations, and quality assurance teams. This leads to faster resolution of issues and improved overall system performance.
- **Improve customer satisfaction**: By monitoring system performance in real-time, you can ensure that your customers receive the best possible experience. You can quickly identify and resolve issues that might impact customer satisfaction, leading to increased loyalty and retention.

After covering the impact of observability on the performance of your services and solutions, let's continue now with the second cloud scenario where an effective observability strategy provides value: reliability.

Enhancing reliability through observability

Reliability is the ability of a system or solution to perform its intended function consistently and accurately over time. In the context of cloud solutions, reliability is a measure of how dependable and trustworthy a cloud service or platform is in delivering the promised features and performance.

Reliability is an important aspect of cloud computing services because it directly impacts the success and profitability of businesses that rely on cloud solutions. A reliable cloud service or platform should be able to provide consistent uptime, fast response times, and minimal errors or downtime. This ensures that businesses can operate smoothly and efficiently, without interruptions or disruptions caused by technical issues.

It is possible that you have not used the term *reliability* directly, but you are familiar with several factors that contribute to the reliability of a cloud solution, such as the following:

- **Uptime**: The amount of time that a cloud service or platform is available and accessible to users. High uptime is critical for businesses that rely on cloud solutions for critical operations.
- **Downtime**: Just the opposite of the previous factor. The amount of time that a cloud service or platform is unavailable due to planned or unplanned maintenance, updates, or technical issues. Minimal downtime is crucial.
- **Redundancy**: The duplication of critical components and processes in a cloud service or platform to minimize the risk of failures and ensure high availability.
- **Disaster recovery**: The ability of a cloud service or platform to recover from disasters or major outages quickly and efficiently, minimizing the impact on business operations.

When developing cloud solutions, it's important to define the measures that ensure they meet the expected levels of performance and availability. Several metrics can be used to evaluate its reliability providing insights into their effectiveness and helping identify areas for improvement. Some of those key metrics are as follows:

- **Mean time between failures** (**MTBF**): MTBF measures the average time between failures of a cloud service. It calculates the duration between the last failure and the next failure, providing an estimate of the service's stability and reliability. A higher MTBF indicates fewer failures and therefore greater reliability. For instance, if an application has an MTBF of 100 hours, it means that on average, the service experiences one failure every 100 hours.

- **Mean time to recovery** (**MTTR**): MTTR measures the average time it takes for a cloud service to recover from a failure. It calculates the time elapsed from the moment a failure occurs until the service is fully restored. A lower MTTR indicates faster recovery times and higher reliability. For example, if a service has an MTTR of 30 minutes, it means that the service can recover from a failure within 30 minutes on average.

> **Tip**
> MTTR is used for mean time to recovery, repair, respond, or resolve, depending on the context.

- **Availability ratio**: The availability ratio measures the percentage of time that a cloud service is available and accessible compared to the total period. It provides an overview of the service's uptime and helps identify periods of downtime or low availability. A higher availability ratio indicates greater reliability, as the service is more likely to be accessible when needed. For instance, if a cloud service has an availability ratio of 95%, it means that the service was available 95% of the time during a given period.
- **Error rate**: Error rate measures the number of errors or failures per unit of time. It helps identify the frequency of failures and provides insights into the service's overall reliability. A lower error rate indicates higher reliability, as there are fewer instances of failures or errors. For example, if a cloud service has an error rate of 1%, it means that the service experiences one error per 100 transactions on average.
- **Downtime cost**: Downtime cost measures the financial impact of downtime or outages on the organization. It estimates the revenue loss, productivity loss, and other expenses incurred due to service disruptions. A higher downtime cost indicates the potential negative impact of unreliable service, emphasizing the importance of investing in reliability improvements. For instance, if a cloud service has a downtime cost of $10,000 per hour, it means that the organization loses $10,000 in revenue and productivity every hour the service is down.

Observability can play a crucial role in enhancing the reliability of complex systems. The following are some of the common scenarios where observability can help:

- **Early detection of failures**: Like the previous section, with observability, you can monitor the system's behavior and detect anomalies or deviations from expected behavior. By detecting issues early, you can take corrective action promptly, reducing the likelihood of cascading failures and improving the overall reliability of the system.
- **Continuous monitoring**: Observability enables continuous monitoring of the system's behavior, allowing you to track performance metrics, identify trends, and detect anomalies in real time. This means that you can catch issues before they become major problems, taking proactive steps to maintain the system's stability and reliability.

- **Predictive maintenance**: Observability can help you predict when maintenance will be required, allowing you to schedule maintenance during less busy periods. By monitoring the system's behavior and identifying patterns that indicate impending failure, you can perform maintenance before a failure occurs, reducing downtime and improving the system's overall reliability.
- **Improved troubleshooting**: Observability provides valuable insights into the system's behavior, making it easier to troubleshoot issues when they do occur. By examining the data collected from various sources, you can quickly identify the root cause of the problem and take appropriate action, reducing the time spent on troubleshooting and improving the system's reliability.
- **Enhanced transparency**: Observability provides stakeholders with real-time visibility into the system's behavior, enhancing transparency and trust. This means that stakeholders can see the system's performance and reliability in real time, enabling them to make informed decisions and take appropriate action if necessary.
- **Better decision-making**: Observability enables data-driven decision-making, allowing you to make informed decisions based on real-time data. By analyzing the data collected from various sources, you can identify areas where the system can be optimized, improved, or upgraded, leading to better reliability and performance.

The last but certainly not least important topic is security – a particularly pertinent topic in today's landscape with the rise in attacks from malicious actors and the growing number of publicly exposed services.

Securing cloud environments with observability

When using cloud services, users entrust their data and applications to third-party providers, who are responsible for securing and managing the infrastructure. Security in cloud environments is critical because it helps protect sensitive data and applications from unauthorized access, theft, damage, or disruption. Cloud environments introduce new security challenges that are not present in traditional on-premises environments, such as the following:

- **Multitenancy**: In a multitenant environment, multiple customers share the same physical hardware and infrastructure. This increases the risk of data breaches, as a single vulnerability could potentially expose multiple customers' data.
- **Shared responsibility**: Cloud providers are accountable for securing their infrastructure; however, customers retain responsibility for safeguarding their own data and applications. This shared responsibility model requires both parties to work together to ensure security.
- **Lack of control**: Customers have limited control over the underlying infrastructure in a cloud environment, which can make it difficult to implement security measures.
- **Dynamic scalability**: Cloud environments are designed to scale dynamically to meet changing demands. This makes it challenging to maintain consistent security controls across all instances.

- **Complexity**: Cloud environments can be highly complex, with many moving parts and interactions between services. This complexity can make it difficult to identify and mitigate security risks.

Observability can also contribute to your cloud environment security, as it allows you to monitor and analyze the behavior of your cloud infrastructure and applications in real time. By leveraging observability, you can identify potential security threats and mitigate them before they become incidents.

Here are some ways in which observability can help optimize cloud environment security:

- **Anomaly detection**: Observability tools can help detect unusual patterns in cloud usage, network traffic, or system logs that may indicate a security threat. By setting up alerts and automated responses, you can quickly identify and respond to potential threats before they escalate.
- **Compliance monitoring**: Cloud environments are subject to various compliance regulations, such as the **Payment Card Industry Data Security Standard** (**PCI DSS**), **Health Insurance Portability and Accountability Act** (**HIPAA**), **General Data Protection Regulation** (**GDPR**), and so on. Observability tools can help monitor compliance with these regulations by collecting log data, network traffic, and system configurations. This helps identify gaps in compliance and remediate them before they become issues.
- **Incident response**: In the event of a security incident, observability tools provide real-time data to help investigate and respond to the incident. Log data, network traffic, and system configurations can be used to identify the root cause of the incident, contain it, and remediate it.

After covering the three main scenarios where observability in the cloud can help us, let's analyze in more detail how Azure Monitor can help us with them.

Azure Monitor, your cloud observability platform

Azure Monitor is a comprehensive monitoring and analytics platform that offers a centralized approach to monitoring and analyzing the performance and health of applications, services, and infrastructure deployed on Azure, as well as on-premises environments. As the cornerstone of Microsoft's cloud observability strategy, Azure Monitor enables organizations to gain profound insights into their Azure-based resources.

In addition to supporting the three pillars of observability – metrics, logs, and traces – Azure Monitor also incorporates a fourth data type: changes. This broadens the scope of observability, providing a more complete view of the system's behavior.

Azure Monitor's ability to ingest and process large amounts of data from various sources, combined with its robust storage and visualization capabilities, makes it an ideal choice for addressing a wide range of monitoring and analytics needs.

Before diving into the details of Azure Monitor's alignment with the previously discussed scenarios, let's take a moment to learn more about the evolution of the service since its inception. Understanding the history of Azure Monitor will provide valuable context for grasping its current architecture and features.

A brief history of Azure Monitor

If this is the first time you are reading about Azure Monitor, the service was first introduced as **Operational Insights** in 2014 as a standalone service outside the Azure portal. It was described as *an analysis service designed to provide IT administrators with deep insight into their on-premises and cloud environments. It helps you interact with real-time and historical computer data for rapid development of custom insights, while providing Microsoft- and community-developed patterns for data analysis.*

Its objective was to *help empower operations teams to effortlessly collect, store, and analyze log data from virtually any Windows Server or Linux source – regardless of volume, format, or location. Access real-time operational intelligence with improved troubleshooting, operational visibility, and fast search, so you can explore, investigate, and fix incidents quickly* (`https://azure.microsoft.com/en-us/updates/general-availability-azure-operational-insights/`).

It provided an interesting collection of resources called **solution packs** that enabled users in their proactive decision-making around data configuration, best practices, security, and auditing. It went into general availability in April 2015. The following figure shows the initial user interface of the service: a predefined collection of colorful blocks with the information provided by each solution pack. Customization was minimal and creating your own visuals or solution packs was not straightforward.

Figure 1.2 – The origin of Azure Monitor, the Operational Insights main page

It evolved, one year later, into **Operational Management Suite** (**OMS**), which brought together several Azure services, including Operational Insights, under a single umbrella. OMS provided a comprehensive solution for monitoring and managing Azure resources and on-premises infrastructure and applications. Actual services such as Azure Automation, Azure Backup, Azure Site Recovery, and

Defender for Cloud have their root in it. OMS continued being a standalone portal outside the main management one.

It was the first time that an Azure service went multi-cloud. OMS collected information and details from Azure services, **Amazon Web Services** (**AWS**), OpenStack, and VMware environments.

Operational Insights continued to evolve and expand its capabilities, adding new features such as APM and **network performance monitoring** (**NPM**). However, as Microsoft's cloud offerings grew, it became clear that a more integrated approach to monitoring and management was needed.

In 2019, Microsoft announced the preview of Azure Monitor, which consolidated the monitoring and analytics capabilities of Operational Insights and OMS into a single service. Azure Monitor provided a unified view of Azure resources, on-premises infrastructure, and custom applications, along with advanced analytics and machine learning capabilities.

With the release of Azure Monitor, Microsoft began to phase out Operational Insights and OMS, encouraging customers to migrate to the newer, more comprehensive service. Today, Azure Monitor remains a core component of Microsoft's Azure suite, offering robust monitoring and analytics capabilities that help organizations optimize their cloud and on-premises environments.

Throughout its evolution, Azure Monitor has maintained a focus on delivering deep insights and analytics capabilities, while also integrating closely with other Azure services, such as Azure Advisor, Azure Policy, and Azure Security Center. This integration enables customers to gain a holistic understanding of their Azure environments, optimize resource utilization, and strengthen security and compliance postures.

Understanding the history of Azure Monitor's evolution from Operations Insights and OMS is important to appreciate its current status and capabilities. Azure Monitor inherited many of the powerful features of OMS and it's possible to see reminiscences of those services in the actual naming of different agents and services used by Azure Monitor.

With this background knowledge, let's now examine how Azure Monitor can assist in tackling the main scenarios we identified earlier, including monitoring performance optimization, reliability, and security.

Real-time insights and performance optimization using Azure Monitor

By leveraging Azure Monitor's real-time monitoring and analysis capabilities, you can gain valuable insights into their application's performance and identify areas for optimization. With Azure Monitor, it's possible to monitor your application's performance in real time, identify bottlenecks and issues, and take corrective actions before they impact end users.

Additionally, Azure Monitor's ability to track performance metrics and log data over time allows organizations to identify trends and patterns in their application's performance, enabling them to make informed decisions about capacity planning and performance optimization. By using Azure

Monitor, organizations can ensure that their applications are performing optimally, resulting in faster response times, lower latency, and improved user satisfaction.

Information is available directly through the Azure portal using the metrics explorer, Azure Workbooks, or querying the log repository. For your web applications, features such as live metrics allow you to select and filter metrics and performance counters to watch in real time, without any direct impact on the execution of your service. It is also possible to check stack traces from sample failed requests and exceptions.

Enhancing reliability through observability with Azure Monitor

Azure Monitor provides advanced analytics and machine learning algorithms that can detect anomalies and potential issues before they impact your application's performance. It also simplifies troubleshooting by providing a clear view of your application's performance and health.

It offers features such as automatic baseline creation, anomaly detection, and forecasting, which can help you anticipate and resolve issues proactively, along with tracing, logging, and error reporting, which can help you quickly identify the root cause of issues. Additionally, Azure Monitor integrates with other Azure services such as Azure Advisor, which can provide recommendations for optimizing your Azure resources, and Azure DevOps, which can provide additional tools for debugging and troubleshooting.

Securing cloud environments with Azure Monitor

Azure Monitor can help secure cloud environments with observability by providing real-time monitoring and analysis of cloud infrastructure and applications. One way it can do this is by detecting anomalies in cloud usage, network traffic, or system logs that may indicate a security threat. By setting up alerts and automated responses, you can quickly identify and respond to potential threats before they escalate.

Additionally, Azure Monitor can help monitor compliance with various regulations such as PCI DSS, HIPAA, GDPR, and so on by collecting log data, network traffic, and system configurations. This helps identify gaps in compliance and remediate them before they become issues.

In the event of a security incident, Azure Monitor can provide real-time data to help investigate and respond to the incident. Log data, network traffic, and system configurations can be used to identify the root cause of the incident, contain it, and remediate it.

Azure Monitor can also help monitor and analyze user behavior and identity management in cloud environments. This includes tracking user activities, login attempts, and access requests. By monitoring IAM, you can identify potential security threats and mitigate them.

Furthermore, Azure Monitor can analyze network traffic in real-time to identify suspicious activity, such as unexpected protocols, IP addresses, or geographic locations. By monitoring network traffic, you can detect potential security threats and block malicious traffic before it reaches your cloud environment.

Finally, Azure Monitor can help monitor and secure containerization and serverless architectures, which are increasingly being adopted in cloud environments. This includes monitoring container and serverless workloads, identifying potential security threats, and mitigating them. Automated remediation tools can also be integrated with Azure Monitor to automatically respond to potential security threats. For example, if Azure Monitor detects a suspicious login attempt, it can trigger an automated response to block the IP address or shut down the affected instance.

Before starting the next chapter, let's take a moment to summarize and review what we have learned in this one and refer to the additional resources provided for further information if needed.

Summary

This chapter introduced the basic concepts of observability and monitoring, a critical aspect of cloud computing, as it allows developers and operators to make informed decisions about how to improve, debug, or optimize a system. Although observability and monitoring are closely related, we clarified the key differences between the two.

Learning about observability in the cloud requires understanding its three pillars – metrics, logs, and traces. This chapter has introduced those concepts that would be further developed along the following chapters aligned with the Azure Monitor service. Understanding those three pillars will provide you with a comprehensive view of system behavior, enabling you and your team teams to quickly identify issues, understand their root cause, and optimize system performance.

This chapter has provided fundamental knowledge about observability, setting the stage for a deeper dive into Azure Monitor in the next chapter. It goes into the various components upon which Azure Monitor is built, establishing a solid foundation for the remainder of the book.

2

Understanding Azure Monitor Components and Functions

In the previous chapter, we introduced the core concepts of **observability** and **monitoring** and how Azure Monitor provides an end-to-end solution for these scenarios. In this chapter, we'll cover the various components and functionalities that make up Azure Monitor. We'll explore the key features of Azure Monitor, which will be expanded upon in subsequent chapters. By the end of this chapter, you'll have a solid foundation in the fundamentals of Azure Monitor and be ready to dive deeper into the specifics of how it works.

In this chapter, we're going to cover the following main topics:

- Introduction to Azure Monitor components
- Metrics and performance monitoring
- Log Analytics and data insights
- Alerting and actionable intelligence
- Configuring and deploying Azure Monitor – a hands-on example with an Azure VM

Technical requirements

At the end of this chapter, we have included a hands-on step-by-step exercise where we configure the monitoring and alerting for a **virtual machine** (**VM**). To complete it successfully, you need access to an Azure subscription or resource group with enough permissions to deploy an Azure VM and a Log Analytics workspace. We recommend using the Contributor or Owner roles to avoid permission errors while completing the exercise in your development environment.

Introduction to Azure Monitor components

In the context of monitoring and observability, most commercially available solutions are divided or organized into three different layers: **data collection**, **data storage**, and **data consumption**.

- **Data collection layer**: This tier comprises the places where data is generated and collected. Each of those places is called a **data source** or simply a **source**. Data sources can include servers, databases, applications, services, and so on. Essentially, any system or device that generates data that you want to monitor would be considered a data source.
- **Data storage layer**: Once data is collected from various sources, it needs to be stored somewhere so that it can be accessed and analyzed later. Data storage solutions can include relational databases, NoSQL databases, data warehouses, streaming services, or cloud storage services such as Azure Blob Storage. The choice of data storage solution depends on factors such as the volume of data, the type of data, the desired level of scalability, and cost.
- **Data consumption layer**: This tier involves the tools and systems that consume the data stored in the second layer and present it in a meaningful way to users. This tier usually provides a great amount of flexibility when consuming the available data. It can include tools such as dashboards, charts, graphs, alerts, and reports. Not only that, but it also provides mechanisms to respond to changes in the information collected or interoperate with external systems that depend on the observability platform. These tools help users understand what's happening with their systems and apps, identify potential problems, and make informed decisions.

The following figure summarizes the different layers and the key components that integrate the Azure Monitor service.

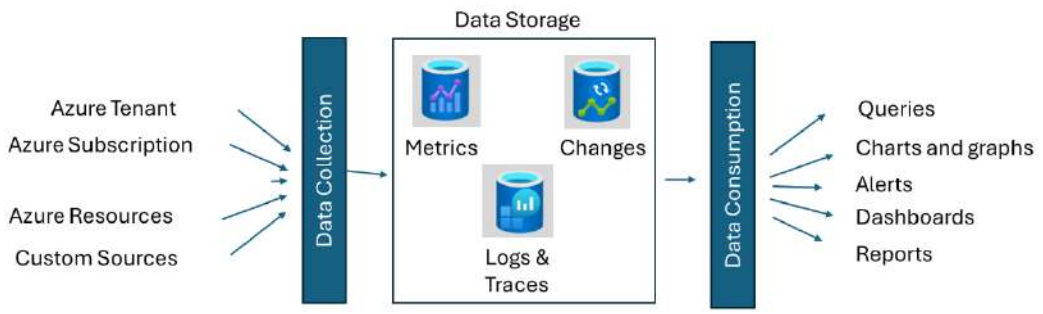

Figure 2.1 – Azure Monitor layers

Let's go into more detail on how Azure Monitor aligns with this architecture and its relevant components, breaking down by each tier in more detail.

Data sources

Data sources are the foundation of monitoring and observability. Without data, there's nothing to monitor or observe. Azure Monitor is aligned with the three pillars of observability discussed in *Chapter 1*, supporting **metrics**, **logs**, and **traces** as sources. However, it also adds a fourth type of data type: **changes**.

In the context of Azure Monitor, its official documentation (`https://learn.microsoft.com/en-us/azure/azure-monitor/data-platform`) defines those concepts as follows:

- **Metrics**: These are numerical values that describe some aspect of a system at a particular point in time. They're collected at regular intervals and are identified with a timestamp, a name, a value, and one or more defining labels. Metrics can be aggregated by using various algorithms. They can be compared to other metrics and analyzed for trends over time.
- **Logs**: These are events that occur within the system. They can contain different kinds of data and might be structured or freeform text with a timestamp. They might be created sporadically as events in the environment generate log entries. A system under heavy load typically generates more log volume.
- **Traces**: These are a series of related events that follow a user request through a distributed system. They can be used to determine the behavior of application code and the performance of different transactions. While logs will often be created by individual components of a distributed system, a trace measures the operation and performance of your application across the entire set of components.
- **Changes**: These are a series of events that occur in your Azure application, from the infrastructure layer through application deployment.

These data sources can be classified into the two main groups discussed in the following subsections.

Azure Platform data sources

Azure Platform data sources provide information about the platform where your applications are running to help customers monitor and optimize their Azure resources. Those logs, measurements, or statistics are collected and provided by Azure by default. From the top to the bottom of Azure's resource hierarchy, they are as follows:

- **Azure tenant**: An **Azure tenant** represents the highest level of abstraction in the Azure hierarchy. It is essentially a container that holds all the subscriptions, resources, and applications associated with a particular organization or customer. Think of it as a virtual private cloud that isolates the resources belonging to one organization from those of another. Azure tenant data sources collect the data of tenant-level services such as Microsoft Entra ID (previously known as Azure Active Directory).

- **Azure subscription**: An **Azure subscription** is a logical grouping of resources that share a common billing and ownership model. A subscription can contain multiple resource groups, which in turn can contain multiple resources. Subscriptions are isolated from one another, meaning that resources in one subscription cannot communicate directly with resources in another subscription without explicit permission. Each subscription has its own unique identifier, known as the **subscription ID**. Azure subscription data sources collect information related to the health and management of resources deployed inside a specific subscription.
- **Azure resources**: **Azure resources** are the individual components that make up an Azure subscription. Examples of resources include VMs, storage accounts, networks, and databases. Resources are allocated and managed by **Azure Resource Manager** (**ARM**), which assigns a unique identifier, known as the **resource ID**, to each resource. Azure resources data sources collect information related to the performance and operations of a specific resource deployed in your subscription.

What happens with resources outside Azure?

Microsoft Azure Monitor supports multi-cloud and hybrid deployments through Azure Arc and its agents. Resources outside Azure that use native monitoring agent capabilities, such as VMs, can be considered like Azure resources with some limitations. For example, monitoring a VM on AWS would not allow us to retrieve information outside of the VM itself, such as metrics, from the underlying hypervisor like Azure does.

Other custom sources

It could be possible that any of the previous data sources don't satisfy your requirements of monitoring or observability. In those cases, Azure Monitor offers the opportunity to collect custom metrics or logs from your application or infrastructure resources through their publicly available APIs.

Two options are available based on the type of data you are ingesting into the platform:

- **Logs Ingestion API**: The Logs Ingestion API in Azure Monitor lets you send data to a Log Analytics workspace using either a REST API call [1] or client libraries [2].
- **Custom metrics API**: The custom metrics API in Azure Monitor lets you send data to a Metrics database using either a REST API call [3] or through Application Insights, as explained in *Chapter 7*.

The next chapter will cover in more detail these data sources and the configuration options available inside Azure Monitor.

Data Storage

Once data is collected from various sources, it needs to be stored in a scalable and reliable manner. Common data storage solutions come in different flavors, including relational databases, NoSQL

databases, data warehouses, and other storage solutions. Azure Monitor stores all its information inside its data platform. This is a custom solution built on top of different storage platforms optimized to ingest, process, and retrieve the data types managed by the service: metrics, logs, traces, and changes.

Although each data type has its own optimized storage, Azure Monitor enables the correlation and analysis of data from all supported data sources. This allows for the utilization of a standardized toolset for analysis and correlation, streamlining the process of gaining insights from the accumulated data. Moreover, data stored inside Azure Monitor is not restricted to a single Azure subscription or tenant thereby offering a comprehensive perspective on the overall state of your solutions and applications running on-cloud or on-premises.

Azure Monitor data storage solutions are discussed in the next subsections.

Azure Log Analytics workspace

In Azure Log Analytics, a **workspace** serves as a centralized repository for storing and analyzing logs generated by various data sources. This includes logs from Azure resources and third-party services such as Microsoft Security solutions such as Sentinel or Defender.

Designed to handle large volumes of data, a workspace is organized into tables, with each table consisting of rows and columns. Like a traditional database, each row represents a single log entry, while columns represent the different attributes or properties of the log data. While many Azure services use predefined tables with specific schemas, custom tables can also be created to meet specific business needs.

To store logs in Azure Log Analytics, you must first create a workspace. The workspace is accessible through the Azure portal, just like any other Azure resource. Once the workspace is created, you can configure various resources to send their logs to it.

Azure Metrics database

The **Azure Metrics database** is a feature of Azure Monitor that provides a centralized repository for storing and querying metrics from various sources. By default, Azure resources provide a set of native platform metrics that help monitor performance, health, and status. Additionally, the Azure Metrics database supports the storage of custom metrics generated by Azure Monitor agents, Application Insights, or the service REST API.

Metrics information in Azure Monitor is stored in a time-series database optimized for analyzing data with timestamps. Each entry in the database contains the following elements: a timestamp with the time at which the metric was recorded, the ID of the resource that generated the metric, the name of the metric itself, and its value. Additionally, multidimensional metrics are also supported, allowing you to capture more detailed information about the metrics.

Unlike a Log Analytics workspace, the Azure Metrics database is not exposed as a separate resource and is instead associated with the subscription under which the resource was created.

Data consumption

Now that we have understood where data is stored in Azure Monitor, let's explore the various features that enable us to make sense of this data and extract valuable insights. Azure Monitor provides several data consumption scenarios that help us understand our data better and respond to changing conditions in our environment.

Azure Monitor data can be consumed from the following:

- **Queries**: Queries are a powerful way to extract specific data from Azure Monitor's Log Analytics workspaces. Using a simple query language called **Kusto Query Language** (**KQL**), we can filter, group, and aggregate data to answer specific questions or identify trends. Queries can be saved and shared, making it easy to reuse frequently used queries or collaborate with colleagues.
- **Charts and graphs**: Charts and graphs are an excellent way to visualize numerical data and identify trends, outliers, and correlations. Azure Monitor provides a range of chart types, including line charts, bar charts, pie charts, and scatter plots through Azure Metrics Explorer.
- **Dashboards**: Dashboards provide a visual representation of data, allowing us to quickly spot trends and patterns. They often display real-time data and offer drill-down capabilities for deeper analysis. In Azure Monitor, dashboards can be created through the workbooks feature. It provides both predefined dashboards and the option of creating custom visual reports customizing the layout and design. It supports log queries, metrics charts, and custom text and parameters to filter the information.
- **Reports**: Reports provide a summary of data over a specified period; they're helpful for sharing data insights with others and performing historical analysis. In Azure Monitor, you can export your log queries into **Microsoft Power BI** to create your custom reports, integrating the monitoring information from Azure into your enterprise reporting platform. With Azure Monitor, you can easily create reports that provide a comprehensive view of your environment. You can select the data you want to include in the report, choose a template, and customize the layout and design to suit your needs.
- **Alerts**: Alerts notify us when specific conditions are met, such as when CPU usage exceeds 90% or when a server goes offline. They help IT teams stay proactive and address issues before they become incidents. In Azure Monitor, we can create alerts based on data from any source, including metrics and logs. You can customize the alert criteria and actions to suit your needs.

In addition to these core data consumption features, Azure Monitor also provides advanced functionalities such as data export, data retention, and data encryption. These features help us manage our data more effectively, ensure compliance with regulations, and protect sensitive information.

Having gained an understanding of Azure Monitor components, encompassing data sources, repositories, and consumption solutions, we now transition to the next phase of configuring and deploying Azure Monitor.

Metrics and performance monitoring

Azure Monitor provides comprehensive monitoring capabilities for cloud resources running in Microsoft Azure. One essential aspect of Azure Monitor is its support of metrics. **Metrics** represent numerical values that describe different aspects of a resource's operation or behavior over time.

As mentioned in the *The three pillars of observability* section in *Chapter 1*, they provide the answer to *How many/much?* and *When?* Examples of typical metrics include the following:

- **CPU utilization**: How many cores were in use? How much percentage of the CPU was in use? How many threads was the CPU running?
- **Network bandwidth**: How many bytes were flowing through the network card interface? How many of those bytes were flowing in? How many TCP packages were flowing out?
- **Storage consumption**: How much free space was available? How many input/output operations per second were handled?

Metrics focus on providing insights into system health and performance and provide support on capacity planning. Azure Monitor categorizes metrics based on their origin, which can be either platform-generated (built-in) or custom; we will briefly describe them as follows:

- **Platform-generated metrics**: Also known as **built-in metrics** or **native metrics**, these are collected by default from all supported Azure services without requiring any additional configuration. They offer real-time visibility into the operational state of your resources. For instance, some common examples of platform-generated metrics include the following:
  - **CPU usage**: This represents the percentage of CPU used by a VM during a specific period
  - **Memory usage**: This indicates the amount of physical memory consumed by a process within a VM
  - **Request count**: This tracks the number of incoming requests processed by an application hosted on Azure App Service
- **Custom metrics**: Customers define and collect custom metrics tailored to their unique requirements. To send custom metrics data to Azure Monitor, it is possible to use one of the available APIs or **Software Development Kits** (**SDKs**) provided by Microsoft or use the monitoring agents installed inside the VMs. This allows users to monitor non-standard telemetry related to their applications or infrastructure.

What happens with the Prometheus metrics type?

Microsoft Azure has offered a managed service for Prometheus, an open source monitoring solution with its own suite of querying, visualization, and alerting tools since May 2023. This service is specifically designed for **Azure Kubernetes Service** (**AKS**) as an alternative monitoring option to the standard Azure Monitor stack, given the widespread adoption of Prometheus within the open source community as the de facto monitoring solution for Kubernetes environments. The metrics gathered by this managed service are known as **Prometheus metrics** and represent the third type of metrics available. As this managed service has limited scope, we won't go deeper into its features in this book.

In Azure Monitor, metrics are typically represented as name-value pairs, where the name is the metric name, and the value is the corresponding numerical value. However, in certain scenarios, for example monitoring the health status of backend services behind a load balancer, a single metric or non-dimensional metric may not suffice due to the one-to-many relationship between load balancers and backend services.

To address this challenge, Azure Monitor supports multidimensional metrics, which enable the association of multiple pieces of information within a single metric name. Each dimension represents a set of name-value pairs grouped together, providing a richer representation of the data.

For instance, consider the load balancer scenario mentioned earlier. The metric name is `Health Probe Status`, with the following dimensions: `ProtocolType`, `BackendPort`, `FrontendIPAddress`, `FrontendPort`, and `BackendIPAddress`. With these dimensions, you can easily obtain an aggregated view of the health status across all endpoints associated with the load balancer or drill down into specific details for each dimension.

Dimensions in Azure Monitor allow organizing and segmenting metric data according to meaningful categories, thereby improving contextual insights while reducing cardinality. Dimensions essentially act as labels assigned to individual data points, affording flexible querying, and slicing perspectives.

After describing the basic pieces that Azure Monitor is built upon, let's dig into the features it exposes.

Log Analytics and data insights

Azure Monitor's **Log Analytics** service plays a crucial role in collecting, analyzing, and deriving insights from structured and semi-structured logs generated by various sources within the Azure ecosystem.

Log analytics refers to the practice of examining event records and trace information emanating from diverse systems, networks, or applications. Azure Monitor's Log Analytics component excels at ingesting, processing, indexing, storing, and searching large volumes of data efficiently. Some notable advantages of employing Log Analytics in Azure Monitor encompass the following:

- Centralized management of disparate logging sources
- High-performance search queries executed at scale

- Real-time correlations among events dispersed across heterogeneous environments
- Integrated machine learning models delivering intelligent predictions and prescriptive guidance

This experience is powered by a strong backend storage system, an Azure Log Analytics workspace; and a powerful query language, KQL.

As previously discussed in the *Introduction to Azure Monitor components* section, Azure Log Analytics workspaces organize data into tables with rows and columns, much like traditional databases. These tables are automatically generated based on the various Azure services in use and their diagnostic configuration settings. Each table has a predefined schema that defines the expected columns and data types for the stored entries. Users can enhance the contents of these tables by adding extra columns with custom information. For example, in the lab at the end of this chapter, we would monitor a VM. All the performance data is automatically stored inside the `Perf` table and can be queried as shown in the following example.

A simple KQL query to retrieve a list of the top 5 VMs by CPU usage over the past 24 hours might look like this:

```
Perf
| where TimeGenerated > ago(24h)
| where CounterName == "% Processor Time" and InstanceName == "_Total"
| project TimeGenerated, Computer, ObjectName, CounterName,
InstanceName, round(CounterValue, 2)
| summarize arg_max(TimeGenerated, *) by Computer
| top 5 by CounterValue
```

Chapter 4 goes deeper into KQL and query structure, but here's a quick rundown of this sample query:

- `Perf`: The first line specifies the source table for data extraction. In this case, it's the performance table, which stores information about various performance counters and is created automatically when enabling performance monitoring within VMs.
- `where`: Apply filters to narrow down the data relevance. This query uses two conditions: data points older than 24 hours are excluded, and only a specific performance counter is selected.
- `project`: If all available columns aren't needed, use this operator to choose relevant ones for the query.
- `summarize`: Aggregate the selected and filtered data using an aggregation function, such as `arg_max` in this case.
- `top`: Filter the output table by a specified number of entries based on a filtering property, in this case, `CounterValue`.
- `|`: Finally, the element that joins all the commands together: the pipe symbol. It provides a connector to pass the information from one command to the other. Each command is written in a new line for readability purposes.

Furthermore, custom tables can be created in the workspace to store data from external sources or custom data sources, allowing users to define the schema manually and tailor the storage format to their needs. We will cover this in more detail in *Chapter 5*.

Once the data is stored in the workspace, users can interact with it using KQL commands, which enable powerful filtering, sorting, aggregation, transformation, and join operations.

Mastering KQL is crucial for effective cloud observability with Azure Monitor. As mentioned earlier, KQL is a versatile query language that allows you to search for and manipulate data to produce desired outputs. Its importance extends beyond interactive querying, as it's also used for building alert-triggering queries and creating visualizations and dashboards for third-party consumption. However, don't be scared; the new simple visualization mode [4] provides an intuitive experience to get access to common Azure Monitor Logs functionality without advanced KQL knowledge.

KQL queries are the basic element of your alerting strategy. If you can build specific queries to monitor the resources and systems relevant to you, it is possible to create alerts based on those queries to act if something unexpected happens.

Alerting and actionable intelligence

A core tenet of effective cloud observability entails establishing robust alert mechanisms that allow us to find deviations from expected behaviors promptly. Azure Monitor includes a versatile alert framework supporting multi-tiered notification cascades and automated corrective actions. Alerts are supported on Azure Monitor for both metrics and logs.

The cornerstone of Azure Monitor's alerting capability is the **alert rule**, a configurable definition that outlines the conditions and actions for generating alerts based on metrics, logs, or activity logs collected from Azure resources. This rule specifies criteria such as threshold values or specific events that trigger an alert when met or detected.

To create an alert rule, you typically specify the following elements:

- **Condition**: This defines the criteria that must be met for the alert to be triggered. Examples include setting a condition based on CPU usage exceeding a certain threshold or a specific error occurring in application logs.
- **Thresholds**: Thresholds determine when the condition is considered met. For numeric metrics such as CPU usage or memory utilization, you might specify thresholds such as greater than, less than, or equal to a certain value.
- **Frequency**: You can specify how often the alert rule is evaluated against the data. This frequency can vary depending on the metric or log source.

- **Actions**: Actions define what happens when the alert is triggered. This could include sending notifications via email or SMS or integrating with other services, such as Azure Logic Apps, to automate responses.

When the conditions of an alert rule are satisfied, the alert is triggered, which in turn initiates the associated action group to notify or automatically remediate the alert based on the configuration established in the alert rule.

Azure Monitor offers a wide range of alert types that enable you to stay informed about critical events and conditions within your Azure environment. These alert types cater to various scenarios, including monitoring numerical metrics such as CPU usage and analyzing logs for specific keywords or patterns:

- **Metric alerts**: These allow you to set threshold values for quantitative measurements, ensuring proactive monitoring of resource health. When thresholds are exceeded or unmet, alerts are triggered, prompting you to take necessary actions.
- **Log alerts**: These, on the other hand, enable you to detect and respond to specific events or anomalies within extensive log datasets. By analyzing log data, you gain valuable insights into application behavior and system performance.
- **Activity log alerts**: These are triggered by changes to Azure resources' metadata, access controls, or policies. This helps you maintain security and compliance, ensuring that your Azure environment remains secure and well managed.
- **Smart detection alerts**: These leverage machine learning algorithms to automatically detect and notify you of anomalies or performance issues. Unlike traditional alerts, smart detection alerts do not require explicit threshold configurations. Instead, they rely on machine learning models to identify unusual patterns in your data, ensuring that you stay informed about critical issues without manual intervention.

What happens with the Prometheus alert type?

As mentioned in *Chapter 1*, Microsoft Azure has offered a managed service for Prometheus, since May 2023. This service is specifically designed for AKS as an alternative monitoring option to the standard Azure Monitor stack. Prometheus alerts are used to monitor metrics stored in this service. Due to the restricted usage of Prometheus alerts inside Azure AKS, we will not be focusing on it.

After reviewing all the theoretical fundamentals of the service, let's jump to a practical example of configuring the monitoring and alerting for an Azure VM.

Configuring and deploying Azure Monitor – a hands-on example with an Azure VM

Azure Monitor has undergone significant evolution over time, transforming from a collection of standalone services into a comprehensive platform that integrates them into a single solution. As a result, there isn't a uniform approach to configuring and deploying Azure Monitor; rather, the process varies depending on the specific features and services you wish to leverage. In this section, we will outline the introductory steps to activate the standard functionality of Azure Monitor.

The standard process to enable its monitoring capabilities is as follows:

1. Determine the services and features you want to use based on your monitoring requirements.
2. Create an Azure Monitor Log Analytics workspace in your subscription or identify an existing workspace to serve as the destination for your data.
3. Configure data sources to collect logs and metrics.
4. Set up metrics and dashboards aligned with your monitoring objectives.
5. Configure alerts and notifications for anomalies, thresholds, or other conditions that require attention.
6. Integrate with other Azure services, such as Azure Automation, Azure Logic Apps, or Power BI, to enhance your monitoring and automation capabilities.
7. Review and refine your monitoring setup as needed after establishing the basic configuration.

In this section, we will follow the previous steps to configure and deploy the required elements to monitor a Windows VM running on Azure. Let's start:

1. Our objective is to register the relevant metrics and logs to configure an alert rule if the VM is running out of disk space, the available memory is low, or the CPU is highly utilized.

   The first two options require the usage of the Azure Monitor agent inside the VM because those details are not available to the hypervisor outside the VM; however, the last one is provided by default by the platform metrics.

VM deployment

It is possible to use any VM on Azure that you have already deployed. If you don't have any VM yet, you can use the provided template, `vm.json`, available in the GitHub repository of the book inside the `chapter02` folder.

2. After the initial requirements are defined, the next step is to create the workspace where our information would be stored. Using the *Create a new resource* button represented by a green plus sign, we should search for `Log Analytics Workspace` and select the first option as shown in the following figure.

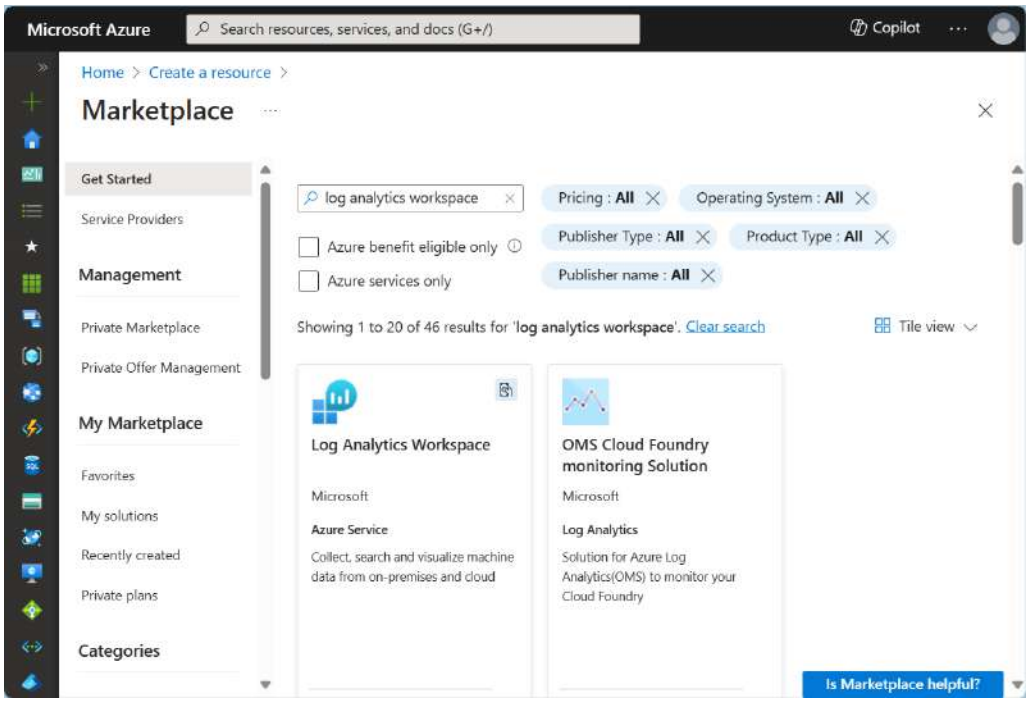

Figure 2.2 – Create a new Log Analytics workspace through the Azure portal

Every Azure resource requires a name and a resource group to be defined before it can be created. Azure Log Analytics workspaces are regional services so a region should be defined too. You can select the values you prefer. For example, we selected the name `packt-book` for **Resource group**, `packtbook-la-ws` for **Name** under **Instance details**, and `Sweden Central` for **Region** as shown in the following figure.

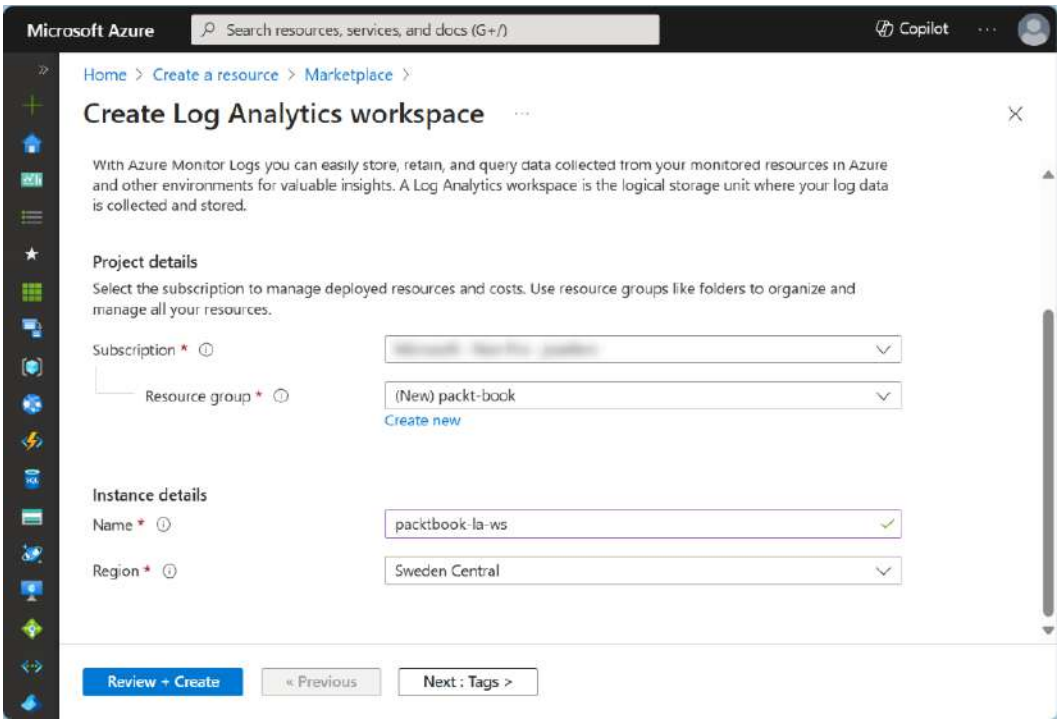

Figure 2.3 – Define the required parameters to create the workspace

After clicking on the **Review + Create** button, all the details introduced in the previous steps will be shown. If everything is correct, click the **Create** button. The workspace would be available to start collecting our monitoring data.

3. Monitoring requirements were established in the first step: we want to be alerted when we are running out of disk space, the available memory is low, or the CPU is highly utilized.

   Information about the CPU utilization is provided directly by the platform as we discuss later in *Chapter 3*. It is possible to access those details through the **Metrics** blade available in the left menu of the VM main page, as shown in the following figure.

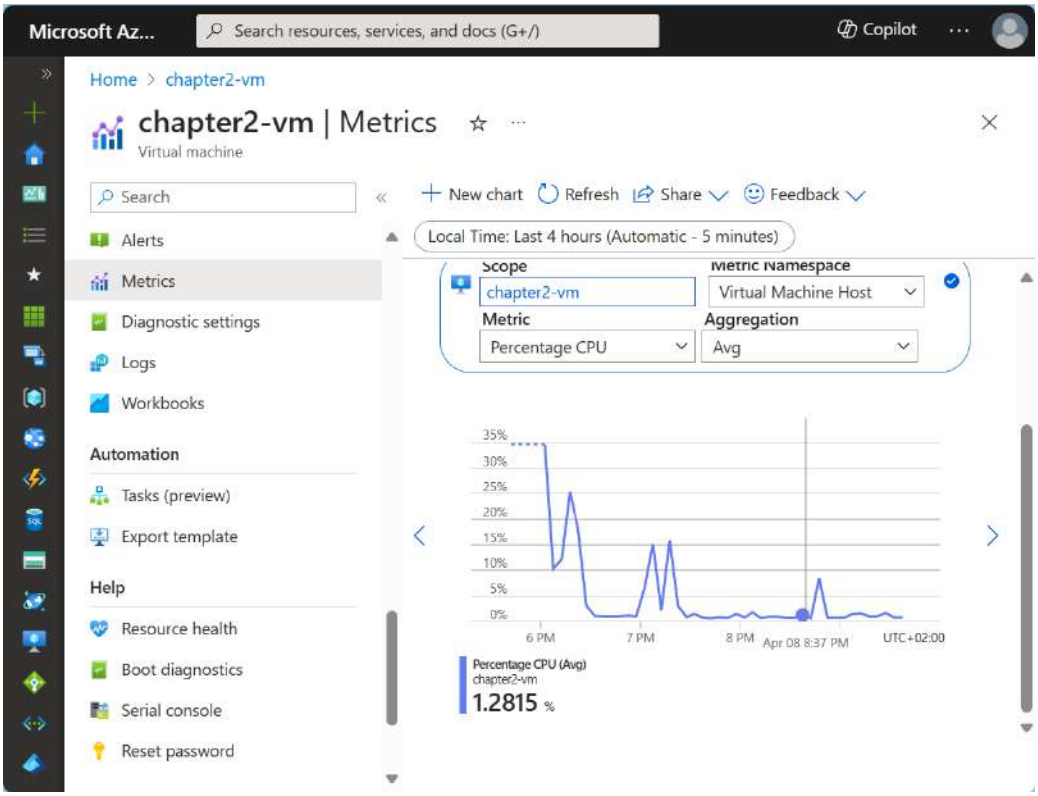

Figure 2.4 – Viewing VM CPU utilization

After being able to check native metrics directly provided by Azure, we would continue capturing the VM memory usage or operating system disk utilization through the usage of the Azure Monitor agent. The agent can be installed through a VM extension, through Azure Policy, or directly using Azure Monitor **data collection rules** (**DCRs**).

In this example, we will use the last option, DCRs. *Chapter 3* will go into more detail about what they are and how they are used inside Azure Monitor to customize the ingestion experience. To create a DCR in Azure Monitor, follow these steps:

I. Use the search box on the top to look for `Monitor` to open the service menu. On the left sidebar, under the **Settings** category, select **Data Collection Rules** and **Create data collection rule**, as shown in the following figure.

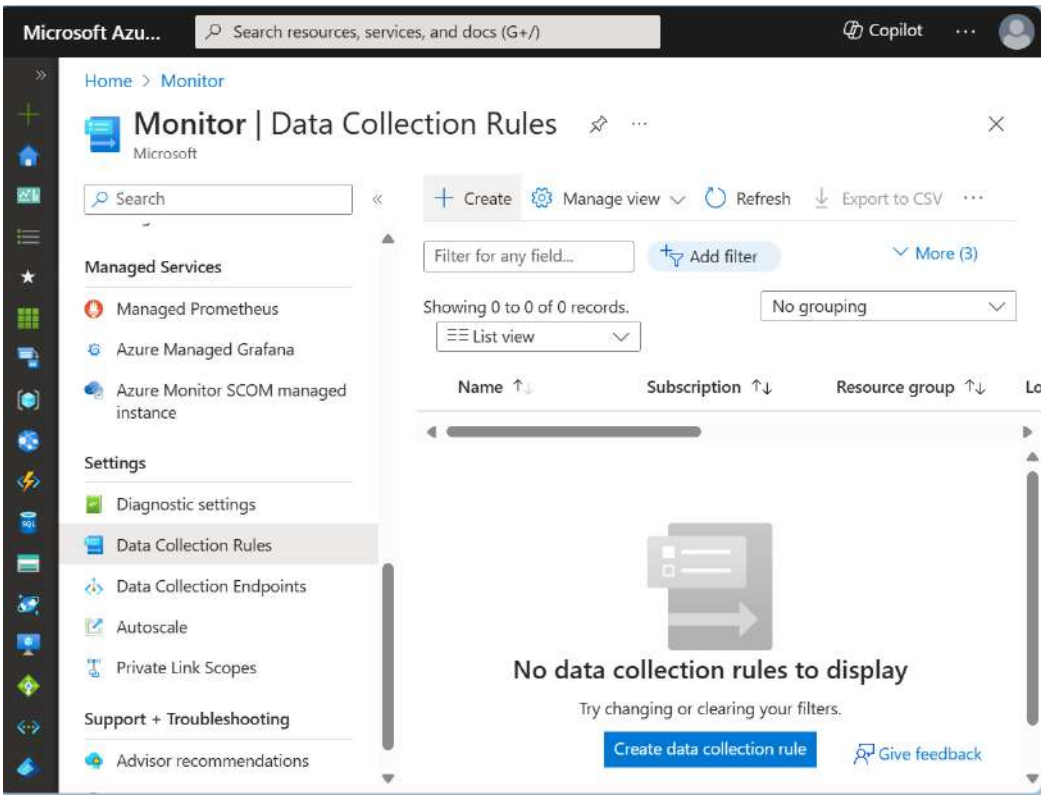

Figure 2.5 – Create a new DCR

II. As mentioned in *Step 2* of the main list of steps, every Azure resource requires a name and a resource group to be defined before it can be created. Enter a **Rule Name** value and specify a **Subscription**, **Resource Group**, **Region**, and **Platform Type** value. **Region** specifies where the DCR will be created. **Platform Type** specifies the type of resources this rule can apply to, Windows in this case. The following figure shows the details of the creation process.

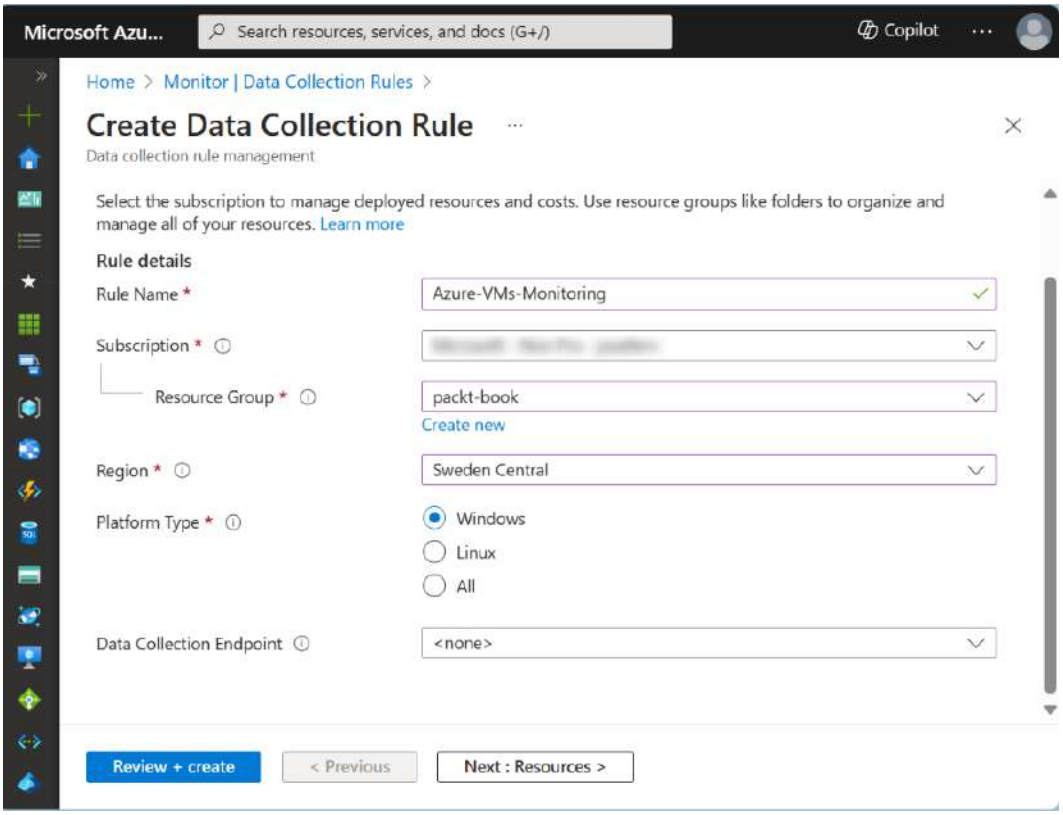

Figure 2.6 – Create a DCR

III. After the rule is created, the next step is to assign the rule to the resources it would be applied. Supported resources are VMs, VM Scale Sets, and Azure Arc for servers. Following this option, the Azure Monitor agent is installed automatically on resources that don't already have it installed. It simplifies the process of onboarding VMs at scale with a single rule that covers both the installation process and the recollection of the monitoring data we are interested in. To complete this step, select the **Add resources** option and find your VM. Before jumping into the next section, select **Enable Data Collection Endpoints**. After you add a resource, you will see something like the following.

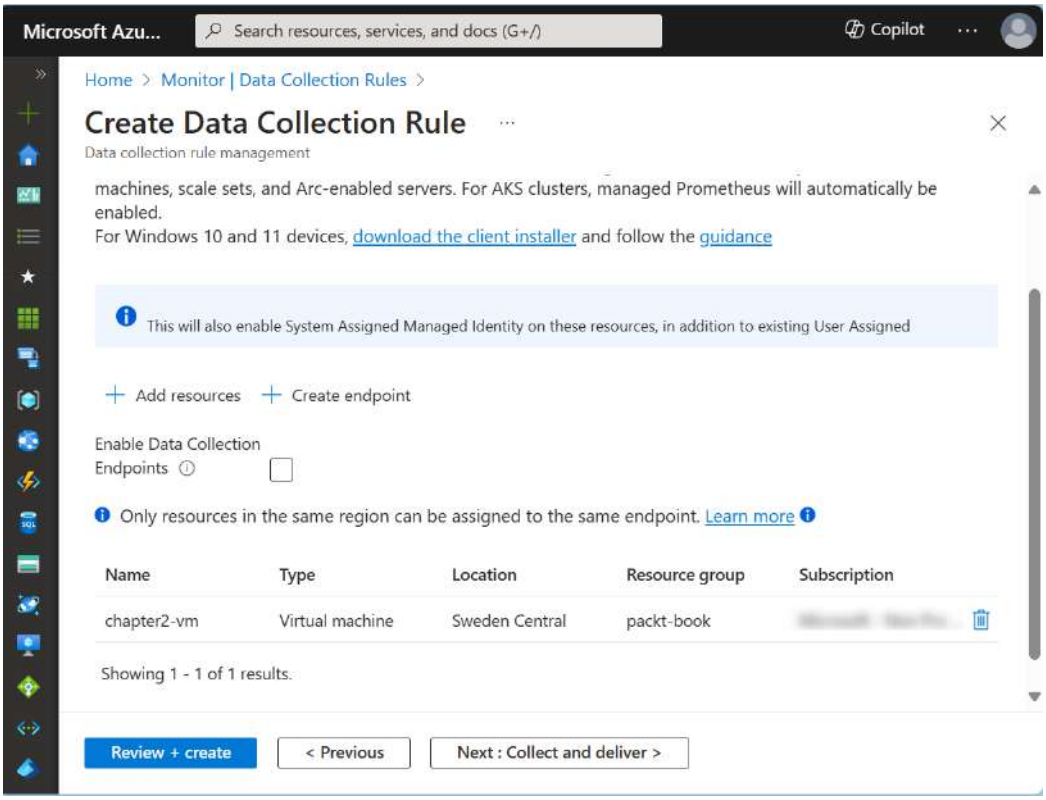

Figure 2.7 – Add resources to the DCR

IV. After we have selected the resource that we will collect the information from, the next step is to define which information we want to collect and where we want to store it. In the **Collect and deliver** tab, select **Add data source** to add a data source and set a destination. Select **Performance Counters** as the **Data source** type. The information we need would be collected through Windows performance counters. The **Basic** configuration would be enough for our objective, as shown in the following figure.

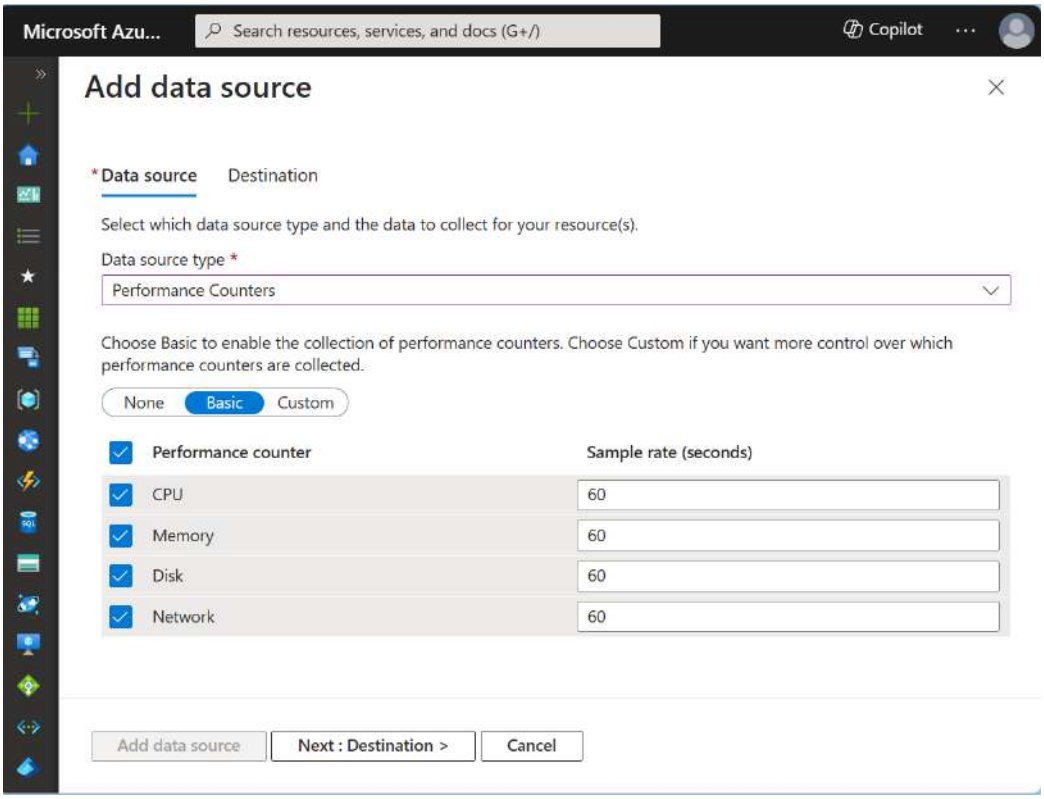

Figure 2.8 – Define the data to be collected

V. On the **Destination** tab, we would add **Azure Monitor Metrics** as the **Destination type** option. We want to store these metrics in the same repository as Azure platform metrics.

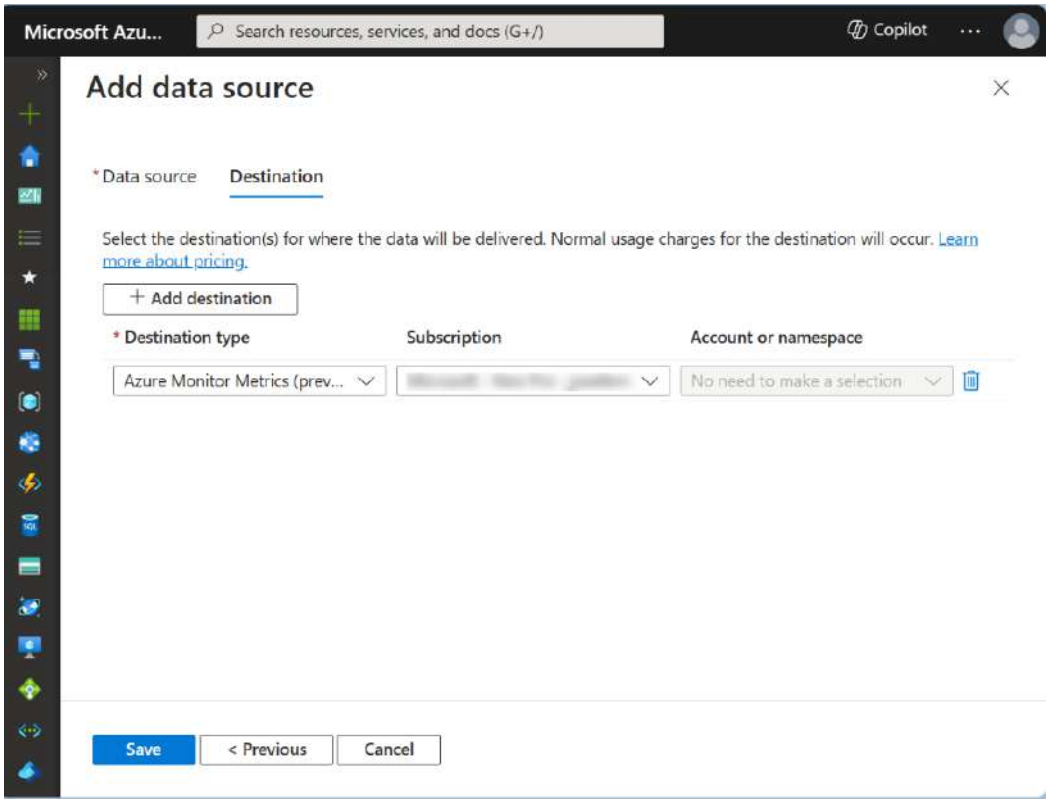

Figure 2.9 – Add Azure Metrics as a destination

VI. Save all the changes and complete the creation of the DCR.

4. After the collection rule changes are applied, information collected will be available through the **Metrics** blade inside the VM resource page, as shown in *Figure 2.4*.

5. The next step would be configuring alerts and notifications for the thresholds defined in *Step 1*. They are configured on the **Alerts** blade inside the VM resource page. To start with the process, select the **Create custom alert rule** option to start with the wizard. We would configure the alert by choosing **Available Memory Bytes** as the **Signal name** value. The process is the same for the other metric.

   The condition for our alert would be that the available memory would be less than 1 GB on average for a 5-minute period evaluated every minute. This is translated into the specified options shown in the following figure.

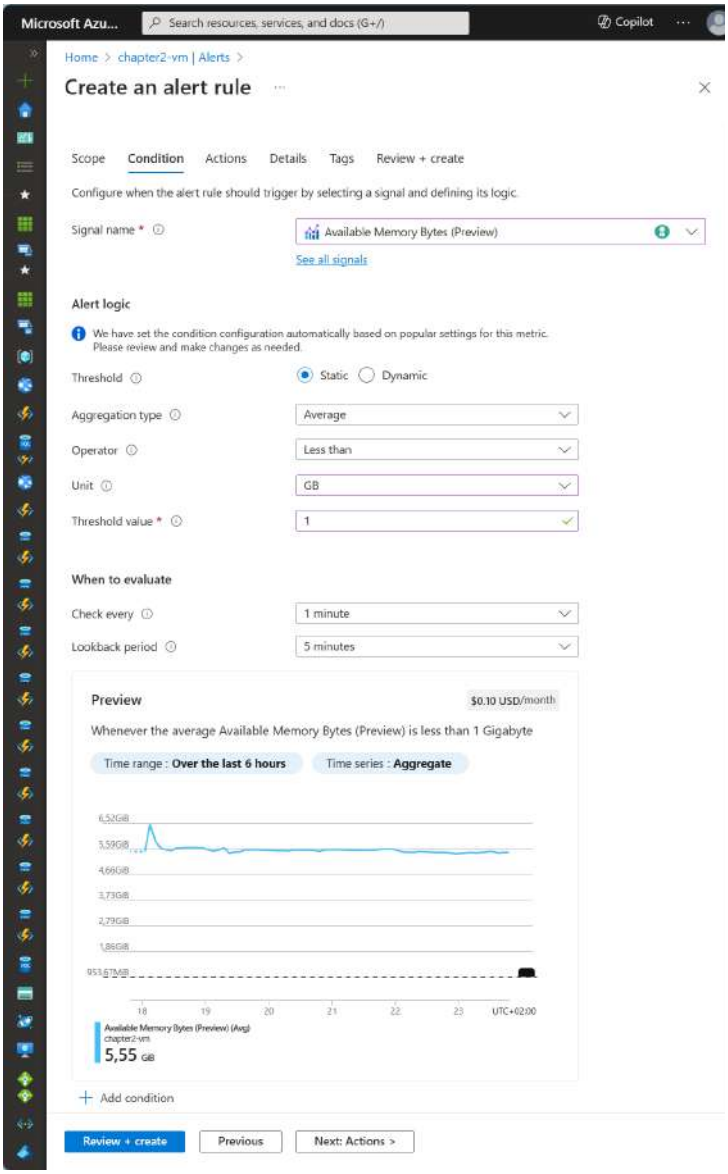

Figure 2.10 – Create an alert for low memory

6. In this simple example, we will not integrate our alert with other Azure services, such as Azure Automation, Azure Logic Apps, or Power BI. This will be explained in more detail in *Chapter 5*. However, you can review the options available on the **Action** tab. Jump into the **Details** tab to add a name to the alert we are currently creating and complete the wizard by clicking on the **Create** button after reviewing the details of the alert rule.

> **Important note**
>
> In this initial example, this step would not be required. However, it is very important that you continue reviewing and refining your monitoring setup after the initial configuration to make sure that it is aligned with your needs and expectations.

In summary, this hands-on exercise has demonstrated the key steps involved in setting up and deploying Azure Monitor to monitor a Windows VM on Azure. By following a structured methodology, we have defined our monitoring requirements, created an Azure Monitor Log Analytics workspace as our data destination, and configured data collection for metrics. We have also leveraged the Azure Monitor agent within the VM to gather important metrics such as disk space, memory usage, and CPU utilization.

Furthermore, we have explored the process of creating a DCR using Azure Monitor, allowing us to collect specific performance data and store it in our Log Analytics workspace. Lastly, we have configured alerts and notifications based on defined thresholds to proactively address critical resource conditions.

Although this example focuses on basic configurations, it lays the groundwork to configure more sophisticated monitoring and automation rules offered by Azure Monitor in future examples, ensuring that your monitoring setup stays adaptable to your needs and expectations.

Summary

In this chapter, we described the fundamental components of Azure Monitor. We started by discussing the core concepts of data collection, storage, and consumption in observability platforms, underscoring the importance of data sources, data repositories, and data consumption solutions. Azure Monitor aligns with this architecture by supporting various data sources, including metrics, logs, traces, and changes, from Azure platform services and external resources through custom ingestion methods.

We also examined the role of Azure Log Analytics workspace as a centralized repository for logs and how the Azure Metrics database efficiently stores and analyzes metric data. Moreover, we explored the various data consumption scenarios offered by Azure Monitor, such as queries, charts, dashboards, reports, and alerts, which enable users to extract valuable insights and take proactive actions based on observed data.

Finally, we outlined the next steps in configuring and deploying Azure Monitor, emphasizing the flexibility and customization options available to customize monitoring solutions to specific organizational needs. This chapter provides a foundational understanding of Azure Monitor, setting the stage for more advanced exploration and practical implementation in subsequent chapters where each of the previous points will be explained in more detail.

Chapter 2 has been a general overview of the different components that Azure Monitor integrates. From *Chapter 3* to *Chapter 6*, we will go into more detail starting with the Azure Monitor data sources and the ingestion pipeline.

Further reading

Here, you can find the links to expand your knowledge about the specific concepts not covered in this book but referenced in this chapter:

- [1] Details of the REST API to ingest logs inside a Log Analytics workspace: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/logs-ingestion-api-overview#rest-api-call`.
- [2] Details of the client libraries to ingest logs inside a Log Analytics workspace: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/logs-ingestion-api-overview#client-libraries`.
- [3] Details of the REST API to ingest custom metrics inside Azure Monitor: `https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/metrics-store-custom-rest-api?tabs=rest`.
- [4] Analyze data using Log Analytics Simple mode: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/log-analytics-simple-mode`.

3

Exploring Azure Monitor Data Sources and the Ingestion Pipeline

Building on the foundation laid in the previous chapter, where we introduced the components and functions of Azure Monitor, we will now go deeper into the details of Azure Monitor data sources and the data ingestion pipeline.

This chapter introduces the varied data sources that Azure Monitor supports, including system metrics, custom logs, and external telemetry. You'll gain insight into how each data source contributes to the observability of Azure resources, applications, and services, and learn how to configure and optimize them to suit your specific monitoring needs.

Furthermore, we'll introduce the details of the data ingestion pipeline, a vital component of Azure Monitor that facilitates the collection, processing, and analysis of telemetry data. To learn about the customization options for tailoring the pipeline for diverse monitoring scenarios and ensuring optimized data flow for enhanced performance and efficiency, we have added a section named *Exploring customization options for tailored monitoring* in the *Appendix* at the end of this book.

Upon completing this chapter, you'll get a deep understanding of Azure Monitor's data sources and ingestion pipeline, empowering you to harness the platform's full potential for comprehensive monitoring and analytics in your cloud environment.

We will cover the following topics:

- Azure Monitor data sources
- System logs and metrics – monitoring the infrastructure backbone
- Custom metrics – tailoring monitoring to your needs
- External telemetry – integrating insights from external sources
- Understanding the data ingestion pipeline in Azure Monitor

Technical requirements

To follow up on all the examples available in this chapter, we recommend downloading the associated files available in the GitHub repository of the book. You will find the relevant files in the `chapter03` folder (`https://github.com/PacktPublishing/Cloud-Observability-with-Azure-Monitor/tree/main/chapter03`).

Azure CLI will be used to deploy and configure the resources. You can install it following the instructions available in the Azure official documentation (`https://learn.microsoft.com/en-us/cli/azure/install-azure-cli`) or using Microsoft Azure Cloud Shell (`https://learn.microsoft.com/en-us/azure/cloud-shell/overview`), which has all the tools required already installed.

Azure Monitor data sources

Azure Monitor gathers and consolidates data from every layer and component of your system and this data is stored in a common data platform for consumption by a shared set of tools capable of correlating, analyzing, visualizing, and responding to the data.

This section outlines the prevalent sources of monitoring data amassed by Azure Monitor, supplementing the data generated by Azure resources.

Azure Monitor enables the integration of monitoring data generated by Azure resources and data from applications, infrastructure, and custom sources outside of Azure, whether on-premises or in other clouds. This is why the monitoring data sources for an application that can be collected in Azure Monitor can be classified as follows:

- **Custom application or Infrastructure as a Service (IaaS) workload data**: This data can originate from Azure, on-premises, or other clouds:
  - In the case of a custom application, it refers to the performance, health, and activity data of the application
  - For an IaaS workload, such as a SQL or Postgres server on a **virtual machine** (**VM**), the data includes information generated by database engines, and so on
- **Infrastructure data**: There are two subtypes of data in this category, and both can originate from Azure, on-premises, or other clouds:
  - Data from containers and applications running within the containers
  - Data from the operating system on which the application runs

  In this book, we will focus exclusively on describing the second type in depth.

- **Azure platform data**: This is also known as **platform logs**, and it encompasses three subtypes of data created by Azure resources:
  - **Tenant data**: Data from Azure services at the tenant level, mainly, Microsoft Entra ID
  - **Azure subscription**: Data from Azure subscription activity
  - **Azure resources**: Data from resource operations within an Azure subscription
- **Custom data sources**: This is data that can be ingested into Azure Monitor through REST API calls or client libraries.

Each of these data source origins has a specific data ingestion pipeline with different types of destinations and collection methods in Azure Monitor. The following figure summarizes it; it would be useful as a global overview.

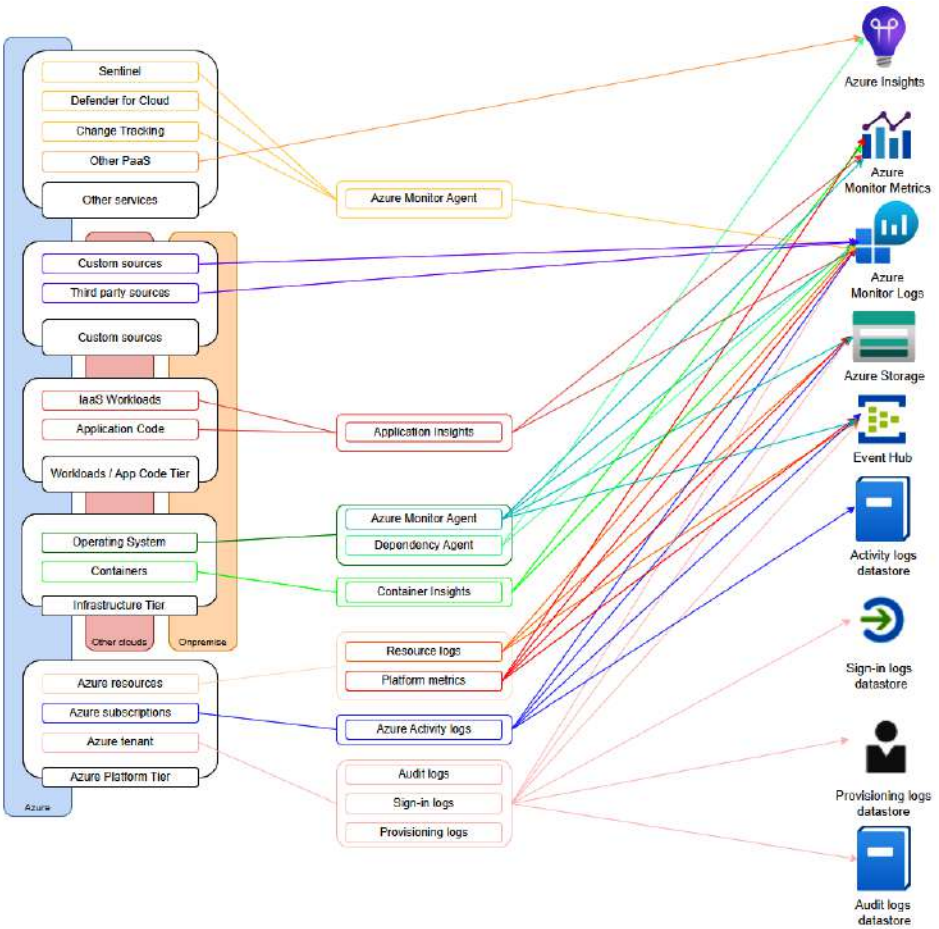

Figure 3.1 – Data source origins

In the following sections, we will describe the different data sources in more detail.

System logs and metrics – monitoring the infrastructure backbone

Our analysis of Azure Monitor data sources will start with the system logs and metrics, the core monitoring capabilities of Azure Monitor. These elements provide indispensable insights into the health and performance of the underlying infrastructure backbone.

Azure tenant data – Microsoft Entra ID activity logs

Microsoft Azure provides a robust identity and access management solution through Microsoft Entra ID. It plays a pivotal role in managing identities, enforcing security policies, and facilitating seamless authentication and authorization across Azure services. Collecting logs related to Entra ID activities and authentication events is essential for maintaining a secure and compliant cloud environment. These logs provide valuable insights into user access patterns, authentication attempts, and security-related events, enabling organizations to detect and mitigate security threats in real time.

Analyzing Microsoft Entra ID logs can help organizations achieve several goals:

- **Monitor user activity**: Gain visibility into user sign-ins, sign-outs, failed login attempts, and other authentication events to detect suspicious or unauthorized activity
- **Detect anomalies**: Identify unusual access patterns or behaviors that may indicate potential security incidents, such as brute force attacks or account compromise
- **Investigate incidents**: Conduct forensic analysis and investigations by tracing user activities, correlating events, and understanding the scope and impact of security incidents
- **Enforce compliance**: Ensure compliance with regulatory requirements and internal security policies by auditing user access and maintaining an audit trail of authentication events

These activity logs come in three types:

- **Audit logs**: These logs contain a comprehensive history of operations and changes made in your tenant, including your users, groups, licenses, or applications.
- **Sign-in logs**: These logs include information about all sign-in attempts by your users and applications.
- **Provisioning logs**: These logs provide information about users provisioned in your tenant through a third-party service.

Microsoft Entra ID activity logs are generated by default, and they do not require a specific configuration to enable them; however, collecting them is required to configure the option to send them to different destinations through the usage of the **Diagnostic setting** configuration blade [1]. Destinations supported by Entra ID logs are as follows:

- **Azure Monitor Logs**: This stores the collected data in one or more **Log Analytics workspaces**, organizing logs and performance data for later analysis using the **Kusto Query Language** (**KQL**), creating alerts, visualization, and so on. The following figure shows the configuration required to enable Azure Monitor Logs as a destination. A subscription and a Log Analytics workspace need to be selected to properly forward the entries.

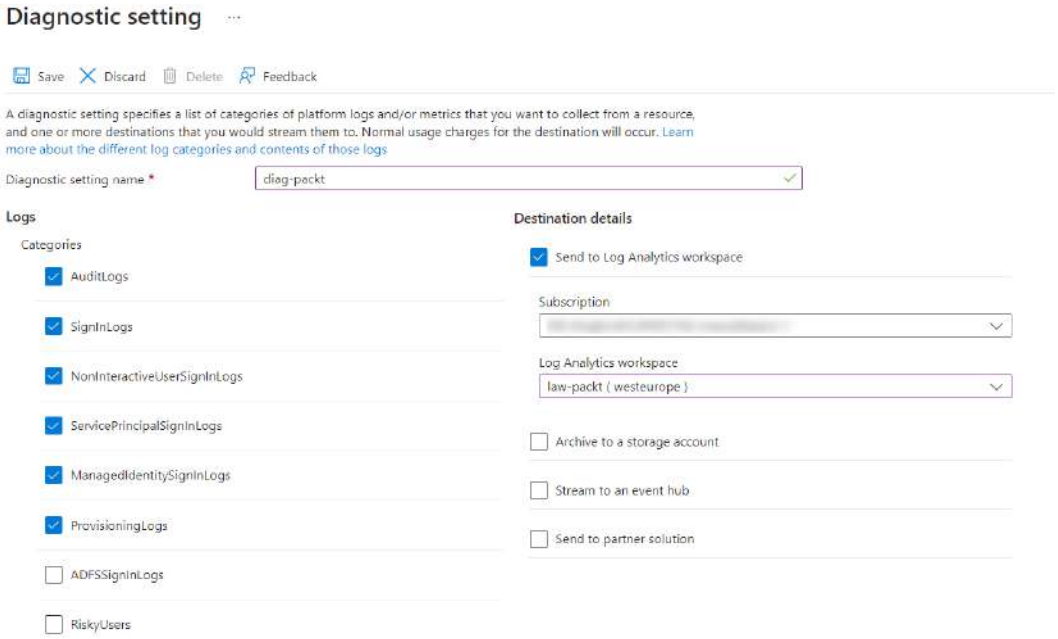

Figure 3.2 – Diagnostic setting for Microsoft Entra ID activity logs with a Log Analytics workspace as the destination

- **Azure Storage**: By default, activity logs have a retention period, but there are situations where it is necessary to extend this retention period due to compliance or security requirements. The recommendation is to use a general storage account for archiving these logs and employ life cycle policies for retention management. It provides a cost-effective solution compared with the previous option if extra analysis is not required. The next figure shows the configuration required to enable Azure Storage as a destination. Similar to the previous scenario, it is required to select an Azure subscription and a storage account to send the log details.

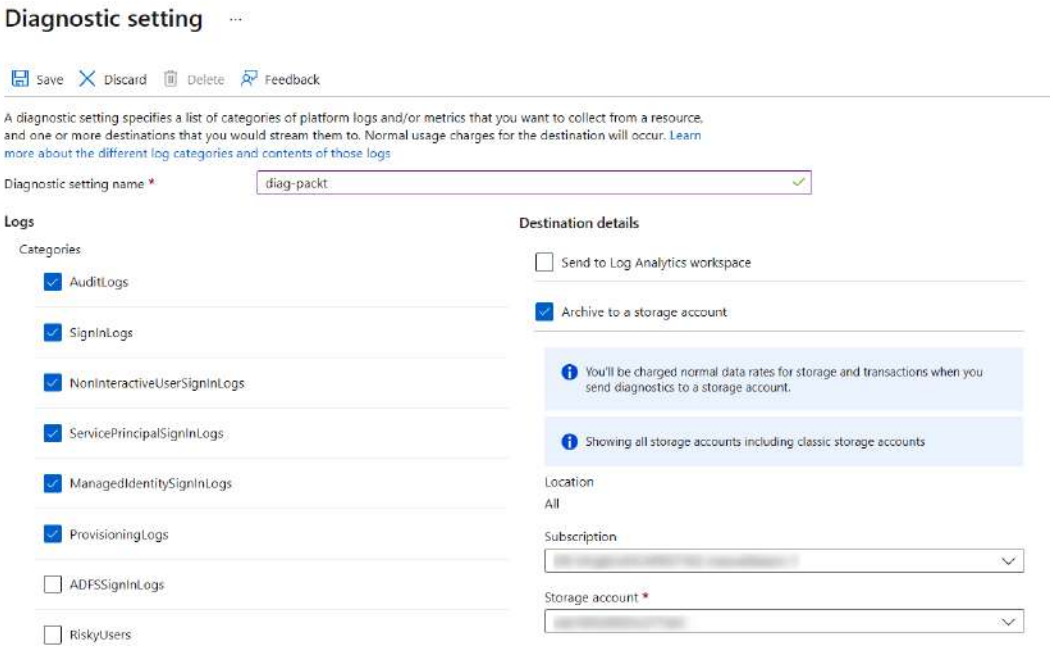

Figure 3.3 – Diagnostic setting for Microsoft Entra ID activity logs with a storage account as the destination

- **Event Hub**: The previous two options allow us to store the data inside Azure; however, in some scenarios, the information is required to be transmitted to an external system. For example, this could be a **security information and event management** (**SIEM**) or third-party solution that processes that information. In this case, an Event Hubs sink is the ideal option. The following figure shows the configuration required to enable Azure Event Hubs as a destination. Like the other scenarios, it is required to select an Azure subscription and an Event Hubs namespace to forward the log details. If no Event Hubs name is selected, logs will be forwarded to the default instance inside the namespace.

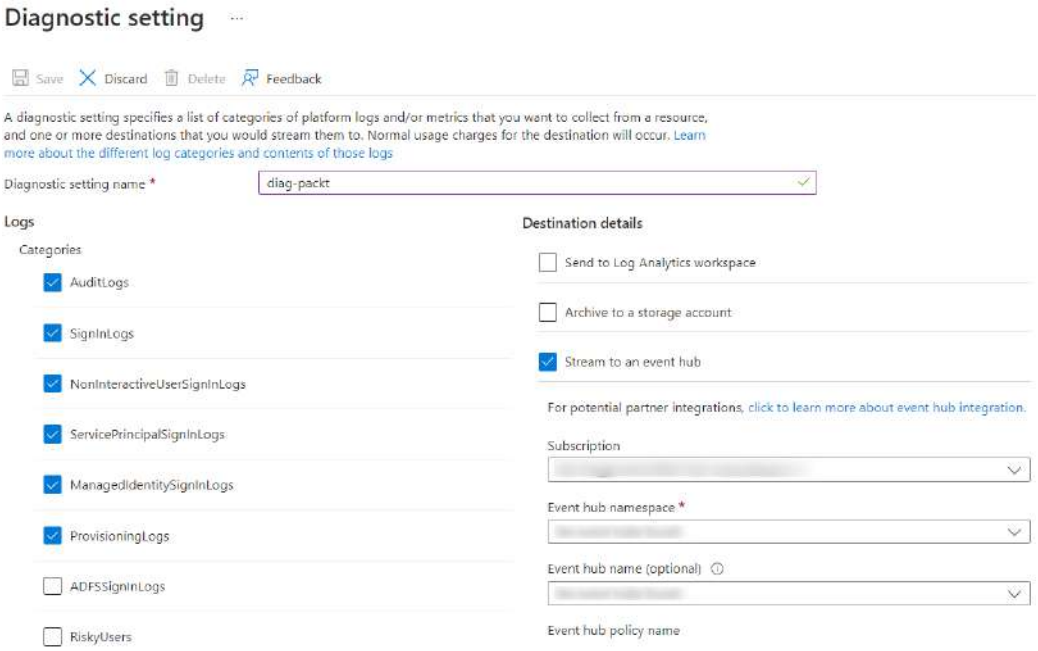

Figure 3.4 – Diagnostic setting for Microsoft Entra ID activity logs with Event Hubs as the destination

Collecting Microsoft Entra ID logs allows us to understand who is interacting with our cloud resources from a pure identity perspective. This information is not only limited to Azure, but to any resource protected by Microsoft Entra ID. For a comprehensive record of operations performed on our Azure resources, Azure Activity logs are the right solution.

Azure subscription – Azure activity logs

Azure activity logs encompass all records related to operations occurring in the Azure management plane at the subscription level. The activity log includes operations and changes made to each resource within a subscription. Examples of entries collected by the activity log include the creation of a key vault or the startup of a VM. The following figure shows the standard view of Azure activity logs inside the Azure portal. Information can be filtered and ordered to find relevant information.

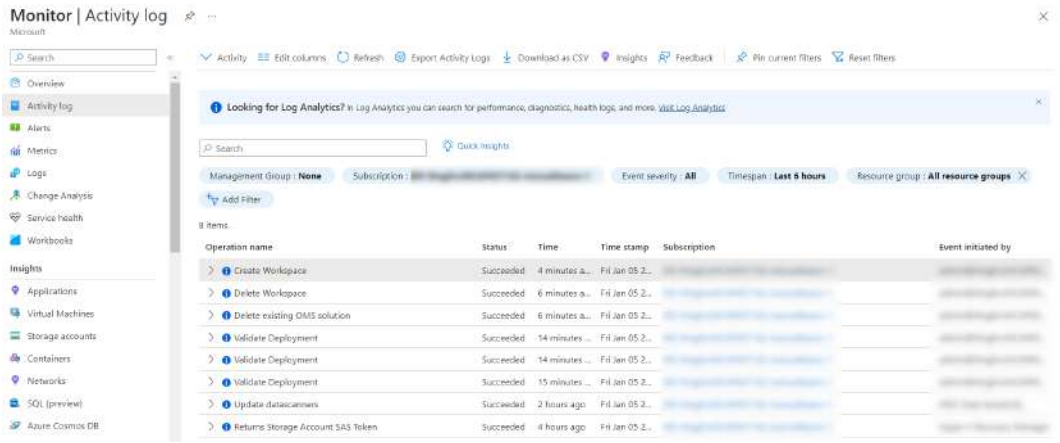

Figure 3.5 – Azure activity logs

Similar to Microsoft Entra ID activity logs, Azure activity logs do not require a specific configuration to collect them; they are gathered automatically by Azure in their own data store. However, they are kept for a maximum of 90 days. If you need a longer retention period, sending them to an external data store is required. These destinations are the same as those mentioned earlier for Microsoft Entra ID Logs and they are configured also through the specific **Diagnostic setting** associated with the activity logs [1]. The next figure shows the configuration options available on the settings blade for Azure activity logs. It is possible to check that the options are similar to the previous scenario.

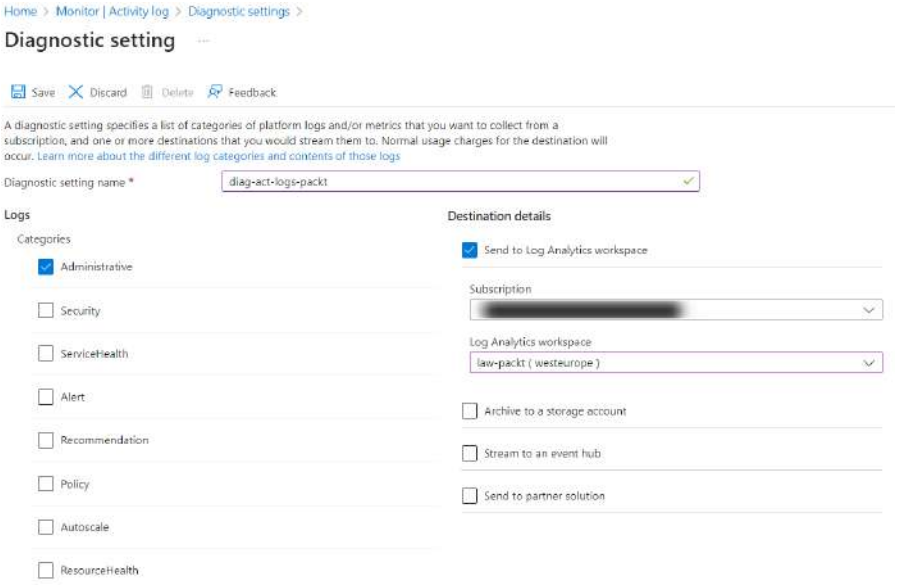

Figure 3.6 – Diagnostic setting for Azure activity logs

As we continue to explore the depths of monitoring Azure resources, our attention turns to the logs generated by individual Azure services. While Azure activity logs offer a broad view of operations performed on Azure resources, we now shift our focus to the detailed, service-specific logs that provide a finer-grained understanding of resource-level activities, diagnostics, and operational insights.

Azure resources – resource logs and platform metrics

The **resource logs** and **platform metrics** are two types of data that provide information about the operations and internal functioning of resources. The content of these logs depends on the specific Azure resource in use. It's not possible to cover all of them in detail in this book, so we will give a general overview of them. Further details are available on the specific articles for each service in the Microsoft Azure documentation:

- **Resource logs**: These logs offer information about the operations performed within a resource. Unlike Azure activity logs and Microsoft Entra ID logs, resource logs are not collected by default. Their collection and forwarding to another destination for further analysis or retention depend on the configuration of the resource diagnostic settings [1]. As mentioned before, its content varies based on the type of resource and log category. Available destinations in the diagnostic settings for resource logs include Azure Monitor Logs, Azure Storage, and Event Hubs, as shown in the next figure. As you have noticed, these options are repeated from the previous log types. Azure Monitor tries to normalize the destination options across all the resources although it's possible that not all of the resources support all the destinations at this time.

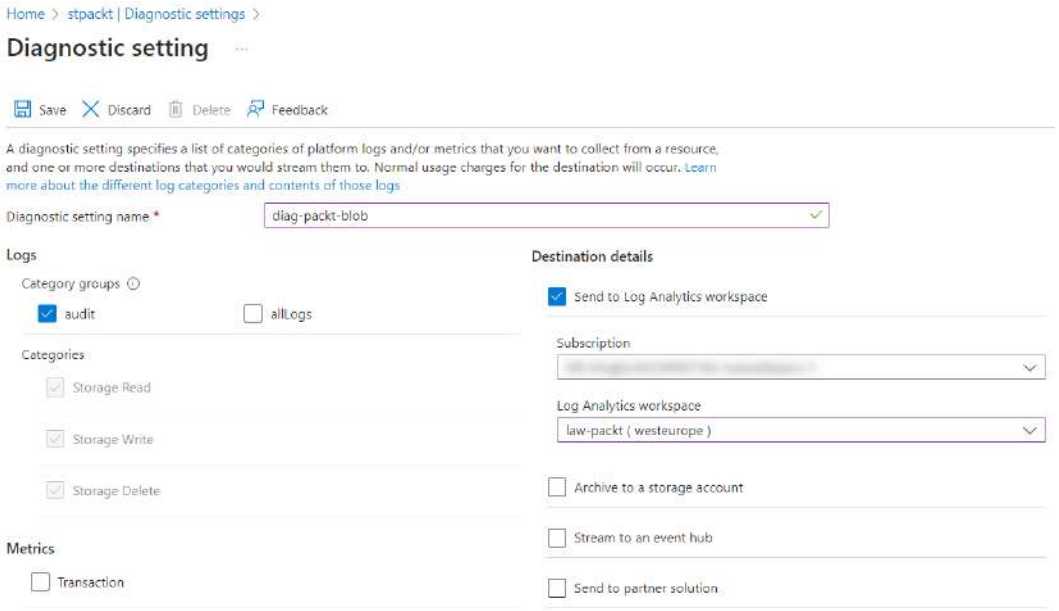

Figure 3.7 – Diagnostic setting for Azure Blob

- **Platform metrics**: These metrics provide information about the health and performance of resources. By default, each type of resource creates a set of metrics that are stored automatically with a retention period of 93 days in a time-series database called **Azure Monitor Metrics**, located within the Azure Monitor platform. Like before, if an extended retention period is needed for long-term trend analysis, exporting the platform metrics from Azure is required.

  There are two options for this:

  - **Azure Metrics REST API**: This API provides access to external applications and services to export any of the metrics collected by Azure. However, it requires specific customization for integrating the metric structure defined by Azure with the expected structure by the destinations.
  - **Diagnostic settings**: This is the quickest and simplest method but does not support the export of multidimensional metrics, and there are certain limitations on which metrics of specific resources can be exported. For further information, it's recommended to check the supported metrics with the Azure Monitor article in the documentation. The next figure shows the configuration of the diagnostic settings for a storage account. It is possible to see the different categories it supports for exporting metrics and logs. In this case, supported destinations include Azure Monitor Logs, Azure Storage, Event Hubs, or a third-party partner solution that could receive this information directly, such as Datadog, Elastic, or Dynatrace, among others.

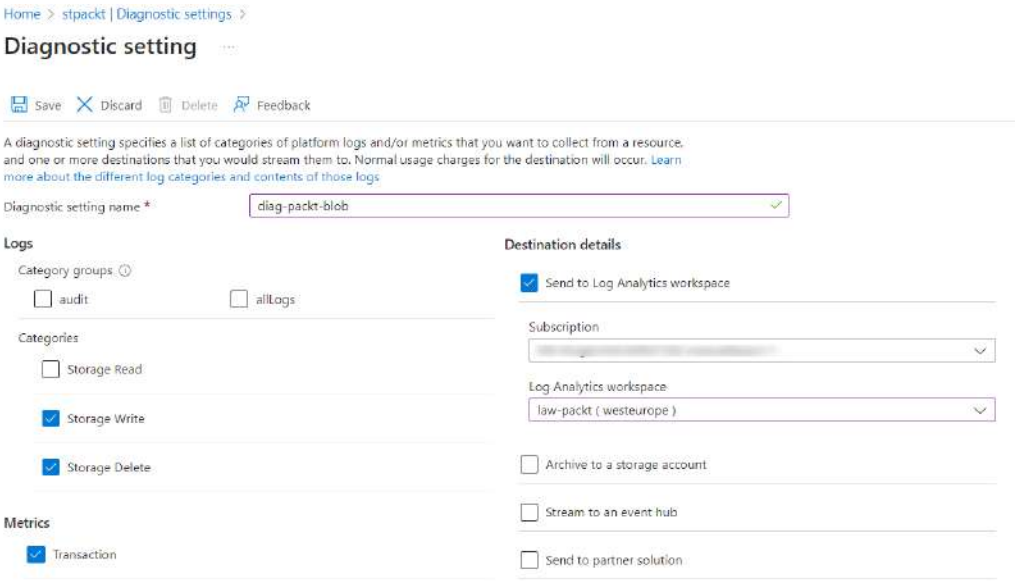

Figure 3.8 – Diagnostic setting of platform metrics for Azure Blob

In our continued path from top to bottom to analyze Azure monitoring capabilities, we will now turn our attention to the specific information that we can collect from the inside of Azure VMs through the **Azure Monitor Agent** (**AMA**).

Operating system (guest) – AMA and Dependency Agent

Until now, we have covered the options that Azure offers to collect diagnostic information from their managed services. However, the application or resources to be monitored may consist of computing resources in Azure, on-premises, or other clouds. It will require the collection of logs and metrics from the guest **operating system** (**OS**) of those resources.

For instance, in the specific case of an Azure VM, platform metrics provide host OS metrics related to the hypervisor session hosting the guest OS session. However, they do not provide information on guest OS metrics, which include memory usage or percentage of CPU utilization.

For those scenarios, Azure Monitor offers an alternative through the installation of a guest agent inside the compute resources to extract securely the required metrics or logs. The main agents are listed after the following note:

> **Evolution of AMAs**
>
> Along with the evolution of Azure Monitor, multiple individual agents were provided to collect different monitoring information. Microsoft has focused on unifying those agents into one; however, at the time of writing this book, work is still in progress to get a single monitoring agent supporting all Azure Monitor features.

- **AMA**: It is the primary agent of the Azure Monitor platform responsible for collecting telemetry from Azure VMs, on-premises, or other clouds. There are five types of data sources that can be collected with AMA:
  - **Performance counters**: Numeric values to measure the OS's performance
  - **Windows event logs**: Data sent to the Windows event logging system
  - **Syslog**: Data sent to the Linux event logging system
  - **Text/JSON logs**: Application information logged in the form of text or JSON files
  - **Internet Information Service (IIS) logs**: Information from the IIS logged on to the local disk of the Windows VM

  Regarding the destination, AMA allows sending performance counters to the Azure Monitor Metrics database, while the remaining types of data can be sent to one or more Log Analytics workspaces. Also, this is in preview and only for Azure VMs, AMA supports sending to Event Hubs and Azure Storage. Azure offers preview features to you for evaluation purposes. A preview may include preview, beta, or other pre-release features, services, software, or regions. Previews are subject to reduced or different service terms, as set forth in your service agreement and the preview supplemental terms `https://azure.microsoft.com/en-us/support/legal/preview-supplemental-terms/`. Previews are made available to you on the condition that you agree to these terms of use, which supplement your agreement governing the use of Azure.

The collection method used by AMA is **data collection rules** (**DCRs**), which, as we will see in *Creating a DCR* section, specify which type of data should be collected, how it should be transformed, and to which destination it should be sent.

- **Dependency Agent**: It is responsible for gathering data from the processes running on a VM as well as external process dependencies. This data is used by the map feature of the VM Insights service. In the case of the data collected by the Dependency Agent, the transmission of this data to Azure Monitor is through AMA and a DCR created by VM Insights.

Azure Insights service

VM Insights is a feature provided by the Azure Insights solution that collects and analyzes a set of data collected through Azure Monitor to provide out-of-the-box visualizations and alerting rules. This service will be covered in more detail in *Chapter 6*, not only for VMs but also for other Azure first-party services.

Now that we have explored the built-in options within Azure Monitor, let's shift gears and check the alternatives available to customize the information collected from your applications and services thanks to Azure Monitor Application Insights and Azure Monitor REST APIs. They will help to create a tailored solution for our unique demands.

Custom application data – Azure Monitor Application Insights

Application Insights is a solution integrated inside Azure Monitor that provides a complete **application performance management** (**APM**) platform for your applications. It provides not only a monitoring solution to collect the logs and metrics of your custom applications; if not a full suite of tools to understand the usage of your applications by the final users, the behavior of your app and dependencies through their investigation features, and code analysis tool to diagnose the performance issues and optimizations in your environment.

This service supports applications running inside Azure, on-premises, or in another cloud provider.

Chapter 6 of this book goes into more detail regarding application observability and performance monitoring with Application Insights.

Custom data sources – Azure Monitor REST API

Up until now, we have discussed how it is possible to work with the logs and metrics that the application generates from different data sources. However, there may be situations where a broader view of the application's behavior is required; this can be achieved by adding logs and metrics from external data sources or simply monitoring resources that generate logs and metrics that cannot be collected with

the previously described data sources. In such cases, the Azure Monitor platform provides a REST API that allows directly sending metrics and logs through a standard REST interface, such as the following:

> **Tip**
> The REST endpoints will depend on the type of information you are submitting.

- **Logs Ingestion API**: This is the REST API endpoint I used for storing logs in a Log Analytics workspace. In the *External telemetry – integrating insights from external sources* section, we will delve into the details of how this REST API works.
- **Custom metrics API**: On the other hand, this REST API is used for sending custom metrics to the Azure Monitor Metrics datastore for an Azure resource. The details of how this REST API operates will be addressed in the next section.

In this initial section, we have been able to understand how Azure Monitor provides a comprehensive platform for monitoring within the Microsoft Azure ecosystem. It aggregates data from various layers and components of systems, storing it in a centralized data platform for analysis and response. We have explored the primary sources of monitoring data collected by Azure Monitor, extending beyond Azure resources to encompass applications, infrastructure, and custom sources both within and outside of Azure environments.

In the next section, we will go into more detail on how you can use the Azure Monitor APIs to integrate external telemetry as a valuable data source extending monitoring capabilities beyond the default solutions provided by Azure.

Custom metrics – tailoring monitoring to your needs

In the previous section, we covered how we can monitor our resources on Azure through the platform metrics it provides for each type of resource. However, there are situations where the user needs metrics not available within the platform metrics. Two alternatives were introduced to send these custom metrics to Azure Monitor Metrics based on the scenario: using Azure Monitor Application Insights or the custom REST APIs.

In this section, we will show an example of how we can use these REST APIs to submit a custom metric to Azure Monitor. Let's consider a microservices application that heavily uses a messaging service such as Azure Service Bus for information exchange between different microservices. The architecture team is interested in creating a custom metric, *Percentage of Capacity Used*, at the entity level, indicating when a queue or topic has reached a threshold in percentage capacity. This way, with this custom metric, they will not need to know the maximum underlying storage capacity or manually set thresholds for alerting. Once the metric is created, the architecture team can set up an alert that proactively notifies capacity issues through Azure Monitor alerting capabilities, and even perform auto-remediations.

Next, let's look into how the custom metrics API works. In summary, the first step is to obtain the required permissions to submit the data through the custom API. Once the token that guarantees access is obtained, a definition of our custom metric needs to be established to help Azure Monitor properly handle that data. After that, we submit the metric to the API endpoint and visualize the data through the Azure portal like any other platform metric.

Authentication and authorization when sending custom metrics

To send custom metrics to Azure Monitor Metrics, it is necessary to use a **managed identity** or a **service principal** depending on the scenario. Azure services that support authentication through Microsoft Entra can request a managed identity when connecting to those resources without the developer or system administrator managing secrets or passwords explicitly. In the other case, Azure services that don't support this type of authentication, or custom applications that require access to those resources, could use a service principal to authenticate themselves and get access.

Additionally, the chosen identity type must be assigned the **Monitoring Metrics Publisher** role at the required scope. This way, the identity will request an access token from Microsoft Entra ID for the endpoint (`https://monitoring.azure.com/`) and will be able to submit the custom metric. This endpoint is also called the **audience** of the token.

Let's assume we have an Azure VM, `PacktVM`, and we want to send the `PacktCustomMetric` custom metric to Azure Monitor Metrics. Azure VM supports the usage of managed identities; in the next figure, you can see how the identity can be enabled when creating the VM.

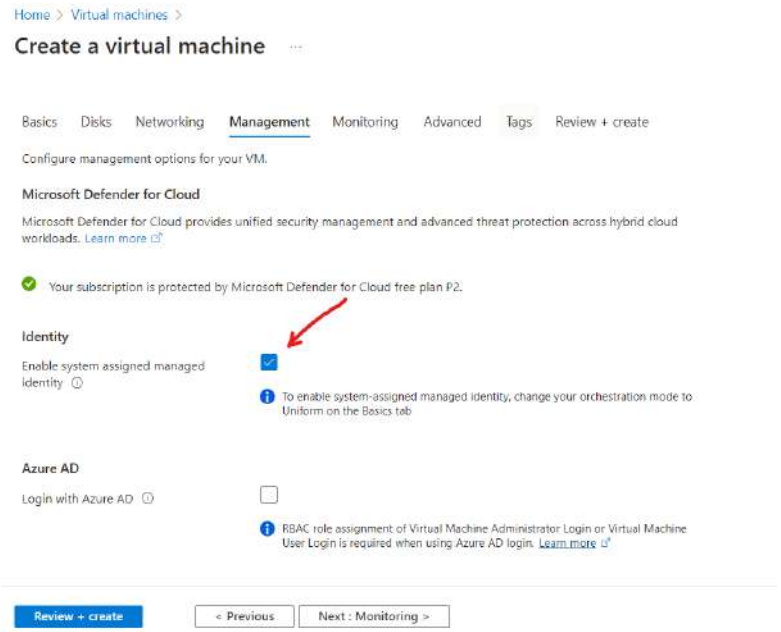

Figure 3.9 – Enable a system-assigned managed identity

After the VM is created and its system-managed identity associated, we will assign it the **Monitoring Metrics Publisher** role, as shown in the following figure.

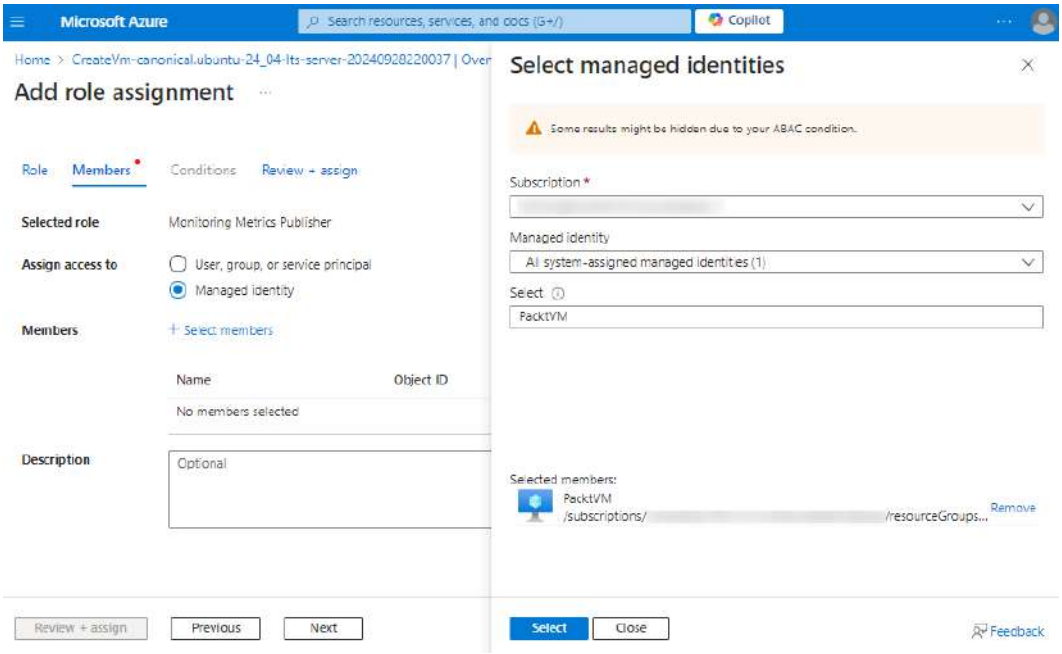

Figure 3.10 – Monitoring Metrics Publisher role assignment

Using the managed identity, we can connect to the Azure **Instance Metadata Service** (**IMDS**) (`https://learn.microsoft.com/en-us/azure/virtual-machines/instance-metadata-service?tabs=linux`) endpoint to obtain a `bearer` token for the endpoint mentioned earlier. Subsequently, we can use this token to send data to Azure Monitor Metrics, using the following command on your terminal as an example. In this case, the example is executed on a Linux bash shell through the Azure Cloud Shell service. It is possible to use your local shell or any other shell-like PowerShell with the proper adjustments to the command:

```
response=$(curl 'http://169.254.169.254/metadata/identity/oauth2/
token?api-version=2018-02-01&resource=https%3A%2F%2Fmonitoring.azure.
com%2F' -H Metadata:true -s)
```

Figure 3.11 shows the response received from the IMDS service with the `bearer` token available on its body.

Figure 3.11 – Response body with access token

The token is inside the JSON response. We need to extract it to reuse it later by defining a shell variable:

```
access_token=$(echo $response | python -c 'import sys, json; print
(json.load(sys.stdin)["access_token"])')
```

Most Linux distributions come with Python installed out of the box. That is the reason that we decided to extract the token through this path. Alternative programs such as `jq` (`https://jqlang.github.io/jq/`) simplify the process although it requires a previous step to install them. With this access token, we can send custom metrics to Azure Monitor Metrics. However, the service would not be able to process them without specifying its schema.

Custom metric schema

Let's now create a custom metric named `PacktCustomMetric` in the `PacktMetrics` namespace for this VM. To do this, we will create the following JSON file:

```
time=$(date -u +"%Y-%m-%dT%H:%M:%S")
echo '{
  "time": "'$time'",
  "data": {
      "baseData": {
          "metric": "PacktCustomMetric",
          "namespace": "PacktMetrics",
          "dimNames": [
            "PacktProcess"
          ],
          "series": [
            {
              "dimValues": [
```

```
                "PacktApp1.exe"
              ],
              "min": 2,
              "max": 12,
              "sum": 14,
              "count": 2
            }
            {
              "dimValues": [
                "PacktApp2.exe"
              ],
              "min": 4,
              "max": 10,
              "sum": 19,
              "count": 3
            }
          ]
      }
  }
}' > PacktCustomMetric.json
```

Let's analyze the structure of the `schema` file to be able to customize it for your specific use case:

- `date`: The timestamp associated with the moment the information regarding your custom metric was collected.

> **Important note**
>
> We have parameterized the timestamp since Azure Monitor only accepts metrics with timestamps in ISO 8601 format, not older than 20 minutes, and not later than 5 minutes.

- `data`: A JSON object containing the relevant information associated with our custom metric. It only contains a `baseData` object where the specific data is included.
- `baseData.metric`: This property identifies the name of our custom metric. It identifies the metric throughout Azure when it is analyzed or visualized.
- `baseData.namespace`: Metrics can be grouped by logical associations called **namespaces**. In this case, we are defining a specific namespace that would contain all the metrics related to our examples under the `Packt Metrics` name.
- `baseData.dimNames`: In *Chapter 2*, we discussed the differences between single and multidimensional metrics. This parameter allows us to define which type of metric we are sending to the service. In this case, it will be a single-dimensional metric.

- `baseData.dimValues`: For a specific dimension, we would include the associated values of the metric. In this case, we are sending information about two processes running on our VM.

Azure Monitor only supports one-minute granularity. There are two possible options when sending information below that granularity interval. It is possible to aggregate the information before submitting the data to the platform, saving costs of storing all those extra points that would not be accessible later disaggregated. Or submit the information directly as it is generated and let Azure Monitor do the aggregation later for you.

The `min` property contains the minimum observed value during that minute for our custom metric. In the same way, the `max` property stores the maximum observed value. The `sum` property contains the aggregation of all the values observed during that time and `count` is the number of samples collected.

In our example, for `Queue1`, we have collected two samples with a minimum value of `2`, a maximum value of `12`, and a total sum of `14`. If our application does not pre-aggregate the values before submitting them, we will have two messages. Both messages would have a `count` value of `1` and the rest of the properties equal to `2` in the first case and `12` in the second one.

Sending a custom metric

After the previous steps are completed, we are ready to send the custom metric to the custom metrics API. The endpoint URL would be similar to `https://{azure_region}.monitoring.azure.com/{azure_resource_id}/metrics`, for example, `https://westeurope.monitoring.azure.com/subscriptions/XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX/resourceGroups/{resource_group_name}/providers/Microsoft.Compute/virtualMachines/{vm_name}/metrics`.

> **Important note**
>
> `azure_region` is the region where the resource, in this case, `PacktVM`, has been deployed because the resource's metrics are sent to the ingestion endpoint in the same region as the resource.

With all the information provided, we can now make the following HTTP `POST` request to send `PacktCustomMetric` to Azure Monitor Metrics:

```
curl -X POST 'https://$azure_region.monitoring.azure.com/$azure_
resource_id/metrics' \
-H 'Content-Type: application/json' \
-H "Authorization: Bearer $access_token" \
-d @PacktCustomMetric.json
```

If everything has been properly configured as described before, the REST API should return `200 HTTP`, indicating that the information was successfully received. The simplest way to validate it is to check if the custom metric is visible through the Azure portal.

Viewing custom metrics

Once the data has been sent, we can view both the custom metrics and platform metrics for `PacktVM` from the portal. You first need to open the VM blade inside the Azure portal and select **PacktVM**. After that, as shown in the following figure, by opening **Metrics** and choosing the namespace selected previously for your custom metrics, you will be able to see the *Percentage of Capacity Used* metric (`PacktCustomMetric`) ingested.

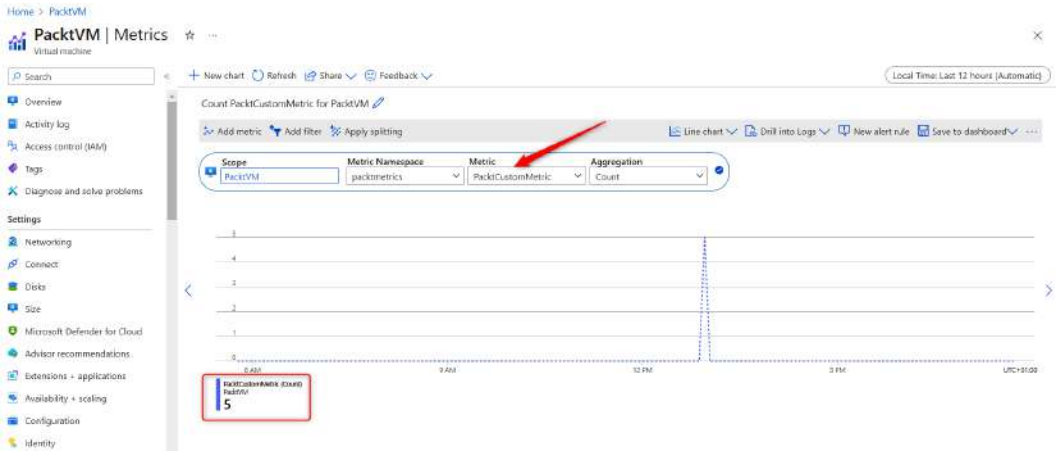

Figure 3.12 – Metric Namespace and the custom Metric name

Note that since the created metric is multidimensional, there is an option to split the data by the dimension name, as shown next.

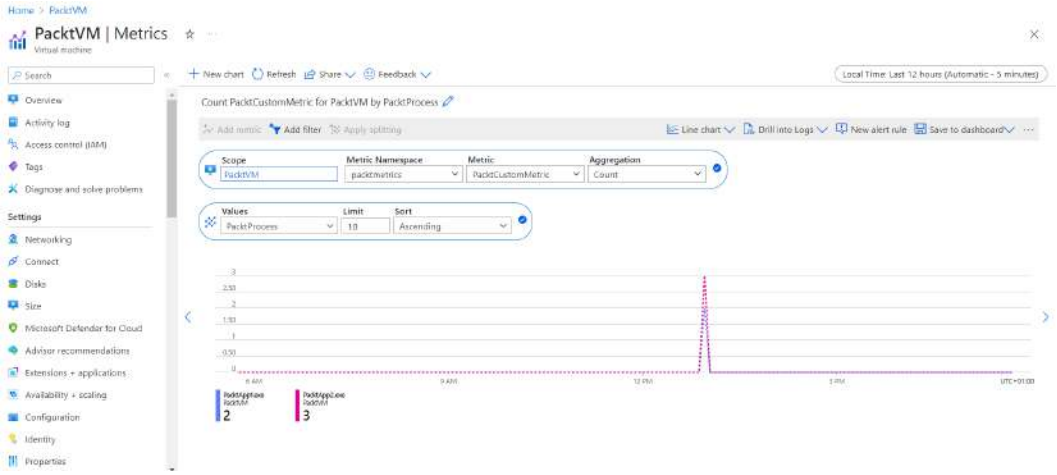

Figure 3.13 – Dimension metric values (count by PacktProcess)

In this section, we have covered how custom metrics can be integrated into Azure Monitor to have the same experience as native platform and system metrics. The next step is to check how the same process can be achieved for custom logs.

External telemetry – integrating insights from external sources

In the *Custom data sources – Azure Monitor REST API* section, we mentioned that Azure Monitor provides a REST API called Logs Ingestion API for sending logs to the Log Analytics workspace. This can be done using a REST API call or client libraries (.NET, Go, Java, JavaScript, and Python), but the ingestion is supported for a limited set of Azure tables and any custom table created in the Log Analytics workspace.

The next diagram illustrates the necessary steps for sending data to the Logs Ingestion API:

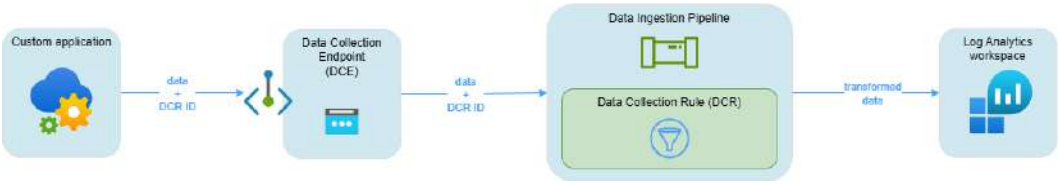

Figure 3.14 – The Logs Ingestion API flow

While submitting a custom metric was a straightforward task, submitting custom logs requires a more complex process. Like the previous case, it starts with the obtention of the required authentication tokens to be able to submit the information. After that, we need to enable a **data collection endpoint** (**DCE**) inside Azure Monitor that would be configured to receive our data. Our logs could go into a specific processing stage before the information is stored inside Log Analytics; these transformations are defined through a DCR. After the information is ready, we can decide whether to store that information inside an existing Azure Monitor Log Analytics table or use a custom one. In this case, we would go with the second option to show all the possibilities inside Azure Monitor. Let's start.

Creating an app registration and secret

The first step is to create an application registration inside Microsoft Entra ID that will authenticate against the Azure Monitor REST API, obtaining an access token that grants the required permissions.

> **Important note**
>
> If the source is an Azure VM wanting to send logs via the Logs Ingestion API instead of using an app registration and secret, you can simply use an Azure-managed identity like in the previous section.

Make note of the **Application (client) ID**, **Object ID**, and **Directory (tenant) ID** values, as they will be needed in subsequent steps. The next figure shows where those values can be obtained through the Azure portal.

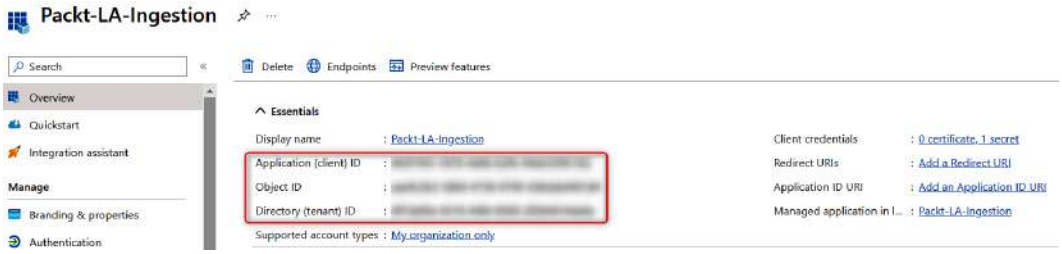

Figure 3.15 – Application registration for Logs Ingestion API call

You can create the secret associated with this application registration using the **Certificates & secrets** blade on the overview page, or by using this command:

```
az ad app credential reset --id <appID> --append --display-name
<secret-name>
```

Creating a DCE

This second step involves the creation of a DCE. A **DCE** is a logical interface created by the Logs Ingestion API to allow our applications to send information for ingestion and processing.

A DCE can be created through the Azure portal or using code options such as Bicep or the REST API. We have selected Bicep, in this case, to help users follow infrastructure-as-code principles to define their environment. Create a file called `dce.bicep` with the following content:

```
param location string = resourceGroup().location
param dataCollectionEndpointName string = 'packt-logs-ingestion'

resource dataCollectionEndpoint 'Microsoft.Insights/
dataCollectionEndpoints@2022-06-01' = {
  name: dataCollectionEndpointName
  location: location
  properties: {
    networkAcls: {
      publicNetworkAccess: 'Enabled'
    }
  }
}

output dataCollectionEndpointId string = dataCollectionEndpoint.id
```

Once the template is ready, we can use the Azure command-line tool to create the resource. You need to replace the name and resource group properties before submitting it:

```
az deployment group create \
--name {deployment_name} \
--resource-group {resource_group_name} \
--template-file ./dce.bicep
```

Opening the Azure portal and looking for DCEs in the top search box will open the details of all DCEs available in your subscription. You can then verify that our DCE appears in the list as shown next. That means that it was successfully created.

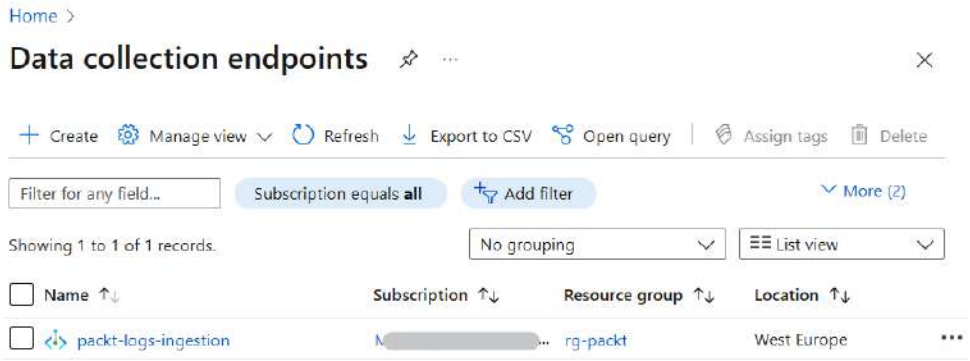

Figure 3.16 – DCE created

Creating a custom table in a Log Analytics workspace

We have decided that our custom logs will be stored in a custom table inside the Log Analytics workspace instead of using the default one. In our example, we will use a previously created Log Analytics workspace named `law-packt`.

Our custom table will be named `PacktTable_CL`. It is important to mention that the `_CL` suffix is mandatory by design when selecting the name of our custom table.

A custom table is defined by its schema. Azure Monitor uses the schema to understand the data submitted and process it accordingly when analyzing or visualizing the logs. We will use the following schema and save it as `schema.json`. The content of the JSON file is self-defined in this case:

```
{
    "properties": {
        "schema": {
            "name": "PacktTable_CL",
            "columns": [
                {
```

```
                    "name": "TimeGenerated",
                    "type": "datetime",
                    "description": "The time at which the event
occurred."
                },
                {
                    "name": "hostname",
                    "type": "string",
                    "description": "The computer name that generated
the event."
                },
                {
                    "name": "eventMessage",
                    "type": "dynamic",
                    "description": "Additional information of the
event."
                }
            ]
        }
    }
}
```

It is important to mention that all tables in the workspace require the `TimeGenerated` column with the timestamp of the logged event. In the case of creating this table through the Azure portal, it is not necessary to include this column in the schema data table source. Instead, you can configure a KQL transformation during the creation of the custom table, indicating that this column will be created or included. This way, if, for example, the data source has the column timestamp but not `TimeGenerated`, the process from the Azure portal detects that there is a transformation that will modify the schema to include the `TimeGenerated` column and, therefore, understands that it should create the custom table schema with the `TimeGenerated` column. However, when the table schema is explicitly created through the REST API, as we will do next, we do not specify any transformations at the same time. Therefore, the API will return a `500` error since it is not possible to create the custom table without a `TimeGenerated` column.

Once the schema file is defined, the next step is to execute the following REST API call to create the custom table inside the Log Analytics workspace. In the previous section, we used the IMDS with our managed identity to obtain the access token required to submit our request. In this case, we are using a service principal instead. In order to get the access token, you need to log in from your shell using the `az login` command and the ID and secrets obtained in the first step. After that, the Azure command-line interface simplifies the process of obtaining the access token through the `az account get-access-token` command:

```
curl -X PUT 'https://management.azure.com/subscriptions/<subscription_
id>/resourceGroups/rg-packt/providers/Microsoft.OperationalInsights/
workspaces/law-packt/tables/PacktTable_CL?api-version=2022-10-01' \
```

```
-H "Authorization: Bearer <you_access_token>" \
-H 'Content-Type: application/json' \
-d @schema.json
```

You should receive a similar response body after the table has been successfully created inside the Log Analytics workspace.

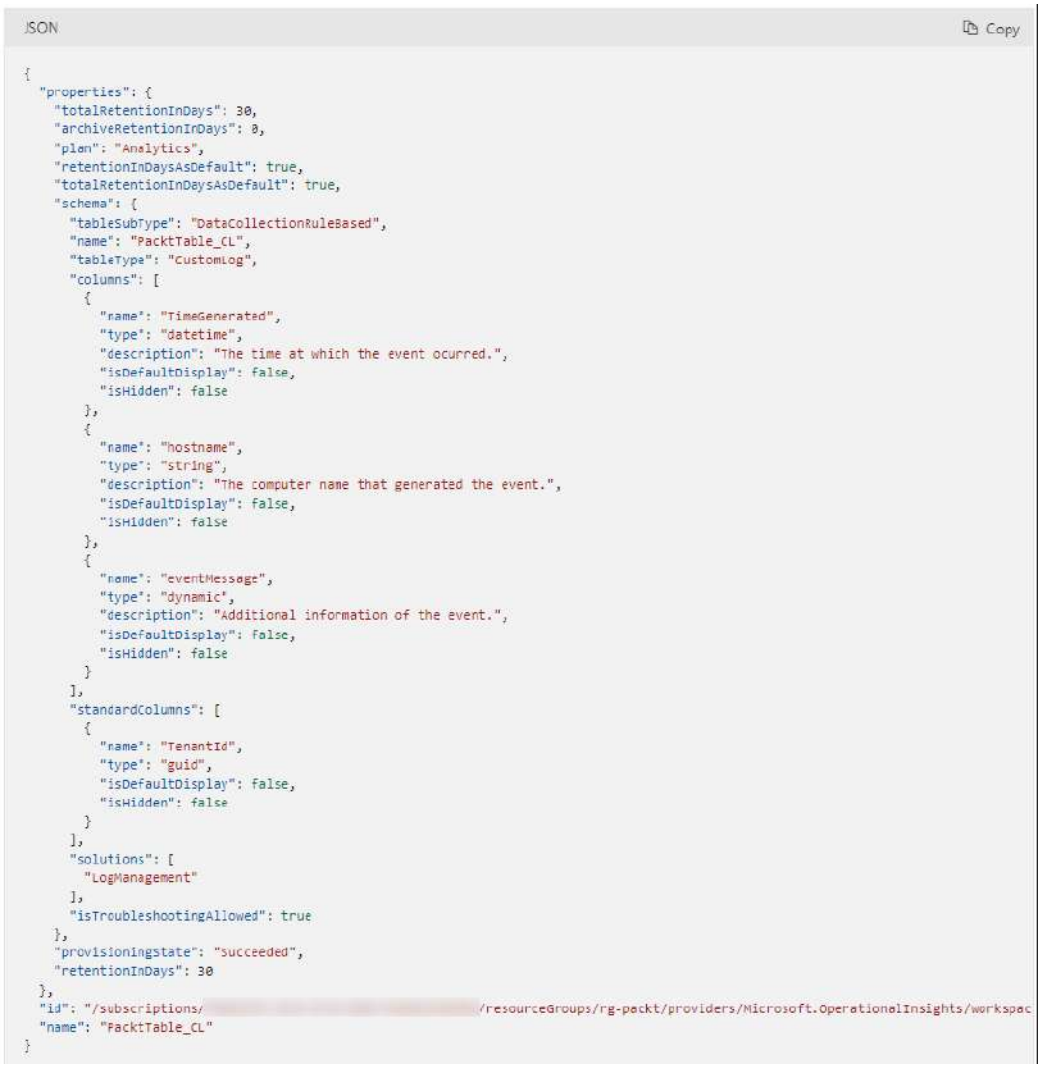

Figure 3.17 – Response body for the PacktTable_CL creation

As an alternative solution, you can also verify the same from the Azure portal by opening our Log Analytics workspace and selecting the **Tables** option on the left menu, as shown in the next figure.

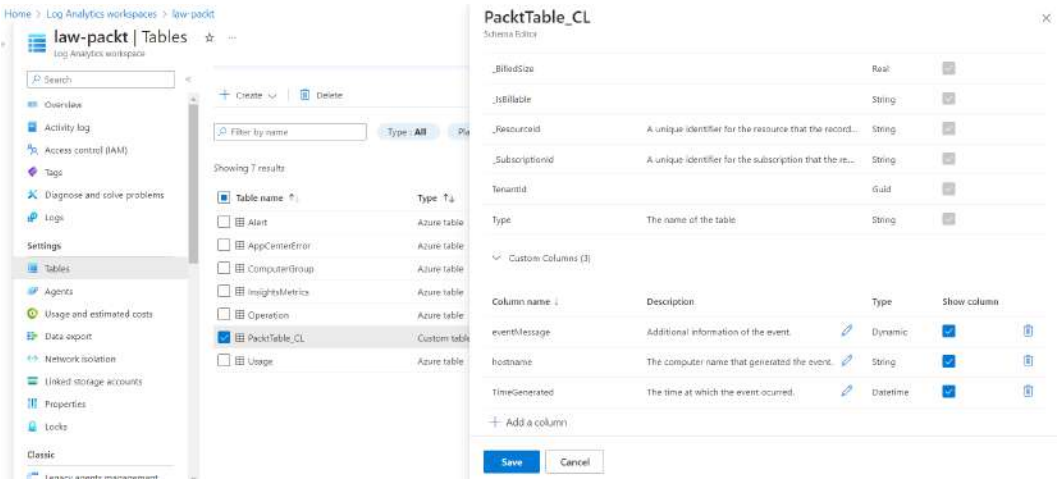

Figure 3.18 – Response body for PacktTable_CL creation

Creating a DCR

In the *Understanding the data ingestion pipeline in Azure Monitor* section, we will go into more detail about what DCRs are, their types, and their structure. As a summary, a DCR defines how data is ingested, transformed, and sent to a Log Analytics workspace.

The REST API call we will make for sending our logs to the platform will connect first with the previously defined DCE and process the data received based on the DCR associated with it in the ingestion pipeline.

The structure of the ingestion pipeline executed by this type of DCR is as follows:

- **Input data structure**: This refers to the schema of the source data intended to be sent to the workspace
- **KQL transformations on data**: This allows modification of the structure of the ingested data to adapt it to the schema of the destination table
- **Loading data into a destination**: This involves sending and storing the transformed data in the destination table of the workspace

To create the schema of the DCR, copy the template available in the GitHub repository of the book inside the `chapter03` folder under the file named `dcr.bicep`.

Let us detail the schema for the Logs Ingestion API DCR:

- `location`: The region where the DCR is created must be the same as that of the DCE and the workspace.
- `dataCollectionEndpointId`: This is the resource ID of the DCE.
- `streamDeclarations`: This contains one or multiple streams that define the structure of the data sent to the ingestion endpoint. The streams must include the `Custom-` prefix in the name. As mentioned earlier, this input structure does not necessarily match the structure of the destination table since it is possible to apply a KQL transformation to make them match. The output of the KQL transformation must match the structure of the destination table.
- `destinations`: This indicates the destinations to which the data should be sent. In the case of this API, the destination is Log Analytics, and it is specified by including `workspaceResourceId` and the workspace name.
- `dataFlows`: This is responsible for matching streams with destinations and applying transformations if necessary. In our case, it consists of a `Custom-TablePackt` stream, a `workspaceName` destination, and a KQL transformation to match the input data schema with the destination table schema. In the end, we have `outputStream`, which indicates which table in the workspace will ingest the output of the KQL transformation. When dealing with a custom table, the nomenclature for `outputStream` is `Custom-<tableName>` so, in our case, the result is `Custom-PacktTable_CL`.

From our terminal or the Azure Cloud Shell, let's create a new deployment associated with the defined template to create the DCR, after filling in the required details in the `chapter03` template file:

```
az deployment group create \
--name {deployment_name} \
--resource-group {resource_group_name} \
--template-file ./dcr.bicep
```

Figure 3.19 shows that the creation of the DCR has been successful as the Azure portal shows. Pay specific attention to the `ImmutableID` value, which we will need in the final step for sending logs to the Logs Ingestion API.

Figure 3.19 – The JSON definition of the DCR and the immutable ID property

Assigning the Monitoring Metrics Publisher role to the app

In this step, we assign the Monitoring Metrics Publisher role to the application that we created in the first step of this section, *Creating an app registration and secret*. The required scope, in this case, would be the DCR just created so that the application can send data to the DCE, and it can be processed by the DCR. For this, we need the object ID or client ID of the enterprise app that we copied when it was created, or you can get it again using the following command:

```
az role assignment create --assignee "<objectID_or_clientID>" \
--role "Monitoring Metrics Publisher" \
--scope "/subscriptions/<subscription_id>/resourcegroups/rg-packt/
providers/ Microsoft.Insights/dataCollectionRules/dcr-packt"
```

With all the necessary resources now provisioned and ready, the final step is to submit your custom logs and verify their successful receipt. This marks the completion of the setup process, enabling your application to send logs to Azure Monitor for processing and analysis.

Sending data to the Logs Ingestion API

To send logs to the Logs Ingestion API, there are two options:

- **Client libraries**: Client libraries for Python, Java, JavaScript, .NET, and PowerShell. You can explore Logs Ingestion **Software Development Kits** (**SDKs**) in [2] the *Further reading* section.
- **REST API call**: The `POST` request to the `https://<Data Collection Endpoint URI>/dataCollectionRules/<DCR Immutable ID>/streams/{Stream Name}?api-version=2023-01-01` endpoint with a content body in JSON array format containing the data to be sent, whose data structure matches that defined in the DCR stream.

Following the previous examples, we will perform log sending using the second method.

To obtain the `Bearer` token required for the authorization header, we will use the v2.0 endpoint of Microsoft Entra ID and the application ID and secret from the *Creating an app registration and secret* section:

```
curl -X POST 'https://login.microsoftonline.com/<tenantID>/oauth2/
v2.0/token' \
-H 'Content-Type: application/x-www-form-urlencoded' \
-d 'grant_type=client_credentials' \
-d 'client_id=<client_id>' \
-d 'client_secret=<client_secret>' \
-d 'scope=https://monitor.azure.com/.default'
```

With the `Bearer` token from the response body, we construct the following `POST` request to send the logs to the `PacktTable_CL` previously created:

```
curl -X POST 'https://packt-logs-ingestion-8b6s.westeurope-1.ingest.
monitor.azure.com/dataCollectionRules/dcr-e90dad7c486943b7bbf117b9ed
dd7915/streams/Custom-PacktTable?api-version=2023-01-01' \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer eyJ0eXAiOiJKV1QiL....
J3GPGzOpp3wLiUu7HkGdNdQg' \
-d '  [
    {
        "timestamp": "2023-12-31T16:34:05Z",
        "hostname": "ABCD-LAPTOP",
        "eventMessage": {
            "src_ip": "10.0.0.27",
            "user_agent": "Mozilla/5.0 (compatible; MSIE 9.0; Windows
NT 6.3; Win64; x64; Trident/5.0)",
            "username": "superadmin",
            "result": "Successful Login",
            "password_hash": "d8651634062a4fdacdf1bed51ff2a3db",
            "description": "A user has attempted to log into the host
```

```
machine. The logon was successful."
        }
    },
    {
        "timestamp": "2023-12-31T11:24:43Z",
        "hostname": "ABCD-LAPTOP",
        "eventMessage": {
            "src_ip": "10.10.0.28",
            "user_agent": "Mozilla/5.0 (compatible; MSIE 9.0; Windows
NT 6.3; Win64; x64; Trident/5.0)",
            "username": "superadmin",
            "result": "Failed Login",
            "password_hash": "ccfe03f2a539273dfa6f294cc53b5faf",
            "description": "A user has attempted to log into the host
machine. The login failed because an incorrect password was provided."
        }
    },
    {
        "timestamp": "2023-12-31T10:32:40Z",
        "hostname": "ABCD-LAPTOP",
        "eventMessage": {
            "src_ip": "10.0.0.29",
            "user_agent": "Mozilla/5.0 (Windows NT 6.3; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.114
Safari/537.36",
            "username": "superadmin",
            "result": "Failed Login",
            "password_hash": "2e5374ab84081c04be8e4e8a6cab6fa0",
            "description": "A user has attempted to log into the host
machine. The login failed because an incorrect password was provided."
        }
    }
]'
```

Important note

The content body must be in a JSON array with a data structure that matches the schema expected by the `Custom_PacktTable` stream of the DCR. If the information doesn't match, an error will occur.

It is possible to verify that the logs have been properly ingested through the Log Analytics workspace interface inside the Azure portal. The first time data is ingested into a table, it can take up to 15 minutes for it to appear. *Figure 3.20* shows an example of retrieving the latest 10 results inside that specific table inside our Log Analytics instance.

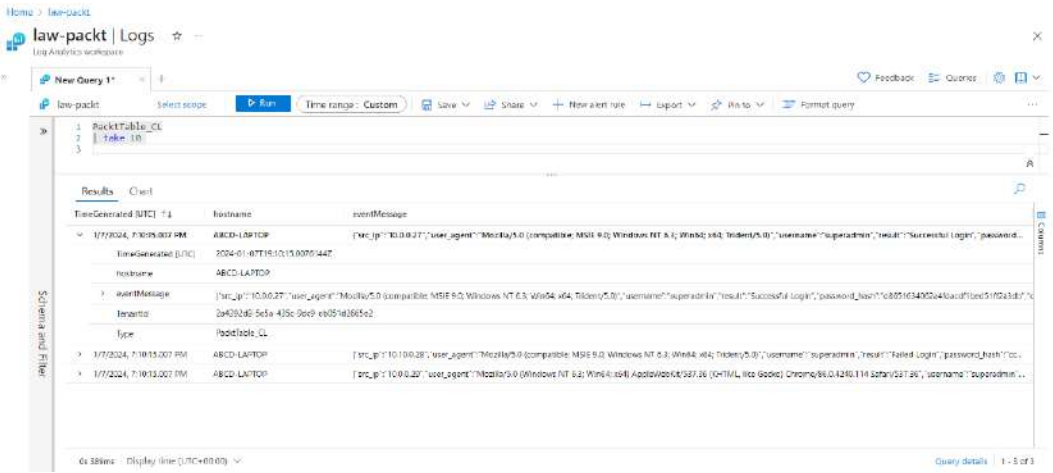

Figure 3.20 – KQL query showing custom logs

Now that we've discussed the main elements of the ingestion pipeline for creating a custom log and metric in Azure Monitor, let's take a closer look at each component of the pipeline in the next section and how to customize it further.

Understanding the data ingestion pipeline in Azure Monitor

Azure Monitor offers a unified data platform that aggregates data from various sources, as previously discussed. To address the complexity of differing data delivery methods and separate configurations, Azure Monitor is evolving its original ingestion architecture into a new **extract, transform, and load** (**ETL**)-like data collection pipeline. This latest development unifies data source destinations, allows initial transformation before ingestion, simplifies the configuration process across data sources, and eases integration with infrastructure management solutions built on the infrastructure-as-code paradigm.

In the previous section, we introduced the main concepts of the new ETL-like platform. In this section, we'll analyze in more detail the fundamental components and their configuration.

Data collection pipeline and DCRs

At the beginning of this chapter, we described how Azure Monitor consolidates data from different sources and provides a set of common tools for analysis. For each individual case, the collection method may differ. For example, when we described the collection of resource logs or platform metrics, the configuration was applied through the diagnostic settings of each resource type. On the other hand, for the Logs Ingestion API or VM metrics/logs, we indicated the need for DCRs.

Although we didn't define explicitly what DCRs were, we could infer that the operation that a DCR does is very similar to an ETL pipeline. It provides data ingestion, transformation, and loading pipelines into Azure Monitor.

Indeed, DCRs define how a particular scenario collects data, performs a transformation on it, and stores it in a destination. One of the benefits of DCRs is that they can be reused among multiple resources if the resources share the same ingestion and processing steps. For more information about DCRs, refer to [3] in the *Further reading* section.

As of today, there are four collection scenarios that require the use of a specific type of DCR:

- **Logs Ingestion API**: We already analyzed this scenario in detail in the *External telemetry – integrating insights from external sources* section, where we saw that data sent through a Python client or a REST API call connects to the DCE and a DCR. The DCR analyzes the structure of the input data, performs a transformation to match the input structure with the schema of the table at the destination, and finally loads the data into the custom table of the workspace or one of the supported Azure tables.
- **AMA**: This agent runs on the VM and has an associated DCR that defines what type of data to collect from the VM. Subsequently, it sends the data to the ingestion pipeline where the specified transformation is performed, and then it is sent to the destination. A DCE can be used to have greater control over the network in the ingestion process and access to the configuration service (associated DCRs).
- **Log Analytics workspace transformation**: In this scenario, the DCR is associated with a workspace and performs transformations for a specific set of tables. Data does not require a DCE since it is sent, for example, through the diagnostic settings of a resource. Subsequently, it is transformed in the ingestion pipeline by the transformation defined in the DCR and then stored in the destination table. It is important to mention that this DCR only applies if there is no DCR of another type applying. That is, if an Azure VM running Windows sends performance counters to the `Perf` table in the workspace and uses AMA and a DCR for that purpose, the DCR associated with AMA causes the transformation associated with the workspace to be ignored.

- **Event hub**: In this type of scenario, a DCR is used to allow data ingestion from an event hub to a Log Analytics workspace. In this case, configuring a DCE is necessary. Like the previous DCRs, the event hub DCR defines the structure of the input data, the transformations, and the destination tables in the workspace. The supported destination tables in the workspace are the same as those in the Logs Ingestion API scenario, namely, custom tables and a set of supported Azure tables. Finally, this scenario is in Azure offers preview features to you for evaluation purposes. A preview may include preview, beta, or other pre-release features, services, software, or regions. Previews are subject to reduced or different service terms, as set forth in your service agreement and the preview supplemental terms -`https://azure.microsoft.com/en-us/support/legal/preview-supplemental-terms/`. Previews are made available to you on the condition that you agree to these terms of use, which supplement your agreement governing the use of Azure. and requires the Log Analytics workspace to be linked to a dedicated cluster or have a commitment tier, and the event hub namespace to have public access enabled.

Having examined the data ingestion pipeline and its connection to DCRs, we proceed to describe the association between DCRs and various resources for gathering and processing ingested data.

DCR associations

In the *Custom metrics – tailoring monitoring to your needs* section, we described how AMA integrated with DCRs to be able to submit the required custom metrics we were configuring inside the VM that we wanted to monitor. This configuration allowed the agent to understand the data it should collect and send to the ingestion pipeline.

This association is necessary so that when AMA connects to the Azure Monitor configuration endpoint, it knows which DCRs are linked to that VM and can read the structure of the DCR to determine what data to collect. Additionally, this association is needed so that when AMA sends the data to the Azure Monitor ingestion endpoint, the service knows what transformations to apply and the destination for the transformed data.

To create the association between the resource and the DCR, the easiest way is through a Bicep template file. It would help you to parametrize the required components and reuse them in the future. Depending on whether the VM is in Azure or outside the association rule requires different information. The following is the Bicep template code for each scenario:

- For Azure, it is as follows:

```
param name string = 'PacktVMAssociation'
param vmName string = 'PacktVM'
param dataCollectionRuleId string = '/
subscriptions/<subscription_id>/resourceGroups/rg-packt/
providers/Microsoft.Insights/dataCollectionRules/dcr-packt'


resource vm 'Microsoft.Compute/virtualMachines@2023-09-01'
existing = {
```

```
  name: vmName
}

resource association 'Microsoft.Insights/
dataCollectionRuleAssociations@2022-06-01' = {
  name: name
  scope: vm
  properties: {
    description: 'Association of data collection rule.'
    dataCollectionRuleId: dataCollectionRuleId
  }
}
```

- For an on-premises VM or another cloud, it is as follows:

```
param name string = 'PacktVMAssociation'
param vmName string = 'PacktVM'
param dataCollectionRuleId string = '/
subscriptions/<subscription_id>/resourceGroups/rg-packt/
providers/Microsoft.Insights/dataCollectionRules/dcr-packt'

resource vm 'Microsoft.HybridCompute/machines@2023-10-03-
preview' existing = {
  name: vmName
}

resource association 'Microsoft.Insights/
dataCollectionRuleAssociations@2022'06-01' = {
  name: name
  scope: vm
  properties: {
    description: 'Association of data collection rule.'
    dataCollectionRuleId: dataCollectionRuleId
  }
}
```

The primary distinction between the two scenarios revolves around the resource type associated with the VM. VMs outside Azure are managed through Azure Arc, with the corresponding resource type being `Microsoft.HybridCompute`. In contrast, VMs inside Azure are linked to the `Microsoft.Compute` resource type.

If you aim to associate the DCR with a different type of resource, modifying the `scope` property of the association object to direct it toward the appropriate resource is essential.

Now that we've discussed how to link a DCR to a particular resource, let's move on to the next step: defining the transformations for our ingested data. This involves specifying the logic for converting raw data into the desired format, making it suitable for analysis and processing within Azure Monitor.

Transformation types

Transformations in DCRs allow for filtering or modification of input data during the ingestion pipeline before persisting it in the destination. These modifications are crucial to address different needs, such as enriching data or obfuscating sensitive information.

It is important to know that while all scenarios support transformations in their DCRs, not all Azure tables support transformations.

Transformations are carried out using the KQL query language, so each KQL statement is applied to each record individually. The individual application to each record means that operators that apply to multiple records, such as summarize or make-series, cannot be used.

Lastly, as mentioned in the Logs Ingestion API section, all tables in the workspace require the `TimeGenerated` column. Therefore, if a transformation is included, the KQL statements of the transformation must ensure that the output includes the `TimeGenerated` column.

It is worth noting that including transformations in DCRs allows for the creation of output flows to single or multiple destinations with different transformations. In the *Creating a DCR* section, we saw an example of a DCR for the Logs Ingestion API scenario, where the following was the case:

- The structure of the input data has three columns: `timestamp`, `hostname`, and `eventMessage`
- The transformation used the `extend` operator to include the `TimeGenerated` column and the `project-away` operator to exclude the `timestamp` column from the transformation output
- The data was loaded only into the custom table `PacktTable_CL` in the `law-packt` workspace

Now, let us assume that additionally, we want to send the input data to an Azure Log Analytics table, for example, `Syslog`. Note that this Azure table supports both data ingestion through the Logs Ingestion API and transformations. Therefore, we can update the DCR from the previous example as follows:

1. Include a second `DataFlow` to enable the ingestion of the same input data.
2. Add a KQL transformation whose output matches the schema of the `Syslog` table.
3. Send the data to both the `Syslog` and `PacktTable_CL` tables in the `law-packt` workspace.

You can find the full DCR in the `chapter03` folder inside our GitHub repository. The name of the file is `dcrTransf.bicep`. On the contrary, if we want to send data from the same source to two tables in different workspaces, we must create multiple DCRs and configure the application to use both DCRs. Currently, only all DCR tables that belong to the same workspace are supported.

Finally, within a data flow, there can be multiple streams, but if the data flow includes a transformation, then two data flows must be created, each containing only one stream.

We have covered the basics of DCRs and data transformations in Azure Monitor, but as we showed in the *Creating a DCR* section, DCRs have a specific schema depending on the source type. So, to learn the finer points of schema details to tailor the pipeline to various monitoring scenarios, we have added a section called *Exploring customization options for custom monitoring* in the *Appendix* at the end of this book. Just like deciphering the blueprint of a building, understanding the nuances of each schema enables us to customize Azure Monitor observability capabilities for our custom needs. With a solid grasp of schema intricacies, users can optimize their ingestion pipeline to efficiently collect and process monitoring data.

Summary

This chapter has covered the details of the different components that work together inside Azure Monitor to collect, transform, and store data from various sources. We outlined how Azure Monitor collects data from various sources through different supported destinations and that information is not collected only from the resources created by the user but not also by the platform itself, such as Azure subscription activity, or resource operations within subscriptions.

We also explained how Azure Monitor is evolving its original ingestion architecture into a new ETL-like data collection pipeline. You learned how this new pipeline unifies data source destinations, allows for initial transformation before ingestion, simplifies the configuration process across data sources, and eases integration with infrastructure management solutions built on the infrastructure-as-code paradigm.

Finally, we detailed how to use DCRs to define how a particular scenario collects data and performs a transformation to filter or modify input data during the ingestion pipeline before persisting it in the destination.

In the next chapter, we will start taking a deeper look into how the data collected can be analyzed inside Azure Monitor.

Further reading

Here, you can find the links to expand your knowledge about the specific concepts not covered in this book but referenced in this chapter:

- [1] *Diagnostic settings in Azure Monitor*: `https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/diagnostic-settings`
- [2] *Client libraries for Logs Ingestion API*: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/tutorial-logs-ingestion-code?tabs=python#sample-code`
- [3] *Data collection rules (DCRs) in Azure Monitor*: `https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/data-collection-rule-overview`

Part 2:
Working with Azure Monitor

In this part, you will go into more detail about each of the different scenarios that a monitoring strategy requires. It starts with the fundamentals of analyzing the information collected through the different sources for insights extraction. After that, it covers in detail how you can react to those insights through the alerts integrated with Azure Monitor, and how you can create your own dashboards and custom visualizations. Finally, it includes a specific chapter for application performance monitoring through the Application Insights service integrated inside Azure Monitor.

This part includes the following chapters:

- *Chapter 4, Analyzing Your Data Using Logs and Metrics*
- *Chapter 5, Responding to Monitoring Events*
- *Chapter 6, Visualizing Your Logs and Metrics*
- *Chapter 7, Application Observability and Performance Monitoring with Application Insights*

4

Analyzing Your Data Using Logs and Metrics

Part 1 of this book focused on providing a general overview of observability in cloud environments, how Azure Monitor provides the best observability platform for your cloud resources deployed in Azure, and the fundamental knowledge of the elements that the service is built on top of, and the data sources it can ingest and analyze. With all that knowledge, you are prepared now to go deeper in this chapter into the different areas relevant to your monitoring experience with Azure Monitor.

In this chapter, we will explore the critical aspects of analyzing data through logs and metrics within Azure Monitor. Monitoring your cloud environment is essential for ensuring its health, performance, and security. Logs and metrics provide the foundational data needed to diagnose issues, understand trends, and make informed decisions. By the end of this chapter, you will have a comprehensive understanding of how to leverage Azure Monitor's analytics capabilities.

We are going to cover the following topics:

- Understanding Basic Logs and Analytics Logs
- Querying in Log Analytics with KQL
- Parsing log capabilities to simplify querying
- Harnessing metrics for in-depth analysis

Technical requirements

The technical requirement for this chapter is having access to Azure Monitor. We will use the Azure virtual machine created in *Chapter 2* when discussing Azure metrics.

Understanding Basic Logs and Analytics Logs

In Azure Monitor, logs play a crucial role in monitoring and diagnosing issues within your applications and infrastructure. Until now, we have described Azure Monitor logs as a single type of log; however, Azure Monitor provides two types of logs to accommodate different use cases and performance needs – **Basic Logs** and **Analytics Logs**. Understanding the differences between these two types of logs is essential for optimizing your monitoring strategy and making informed decisions about data collection and analysis. Let's explore the differences between them.

Basic Logs

Basic Logs are designed for high-volume, low-cost log ingestion. They are suitable for scenarios where you need to collect a large amount of log data but only require limited querying capabilities for auditing, debugging, or troubleshooting. They don't support analytics or alerts. Their main characteristics are as follows:

- **Cost-Effectiveness**: Basic Logs are less expensive to ingest and store compared to Analytics Logs, making them ideal for high-volume data. You pay per the amount of GB ingested and per data volume scanned when querying the information.
- **Limited Querying**: While you can still query Basic Logs, the query capabilities are limited compared to Analytics Logs. They are intended for scenarios where detailed analysis and complex queries are not required. As described in the next section, a reduced set of **Kusto Query Language** (**KQL**) operators is available.
- **Retention**: Basic Logs typically have a shorter retention period than Analytics Logs. They are suitable for data that does not need to be retained for long-term analysis. Retention is fixed at 8 days.

Their main use cases are collecting large volumes of telemetry data, such as diagnostics logs or performance counters, or storing logs for short-term investigations or troubleshooting. For example, imagine you are monitoring a large fleet of IoT devices that generates a high volume of telemetry data. You need to collect this data for health monitoring, but it does not require complex querying. Basic Logs would be a cost-effective solution for this scenario.

Analytics Logs

Analytics Logs are designed for in-depth analysis and complex querying. They offer powerful query capabilities and are suitable for scenarios where detailed insights and long-term analysis are required. They support all analytic and alerting capabilities. Their main characteristics are as follows:

- **Advanced Querying**: Analytics Logs support rich querying capabilities using KQL, allowing complex data analysis and insights.
- **Enhanced Performance**: They are optimized for performance, enabling fast querying even with large datasets.

- **Longer Retention**: Analytics Logs typically offer long retention periods, making them suitable for historical analysis and compliance requirements. Retention costs are included in the price for the first 31 days.
- **Integration**: They integrate seamlessly with other Azure Monitor features, such as alerts, dashboards, and workbooks, providing a comprehensive monitoring solution.

Their main use cases are performing root cause analysis and troubleshooting complex issues, retaining logs for long-term analysis to meet regulatory compliance and audit requirements, or creating detailed dashboards and reports for operational insights and decision-making. For example, consider a scenario where you are managing a critical web application. You need to perform detailed diagnostics, analyze user behavior, and generate comprehensive reports for stakeholders. Analytics Logs provide the advanced querying capabilities and performance needed for such in-depth analysis.

Next, let's show how you can change the log data plan for a specific table.

Selecting the log type for a table

Azure Monitor supports switching from Basic to Analytics Logs or vice versa once per week. Changes can be applied through Azure Portal, the CLI, or PowerShell. Let's see how to do it:

1. First, open your Log Analytics workspaces. In the left menu, you will see the **Tables** option. Click on it to open the list of tables available inside your workspace.

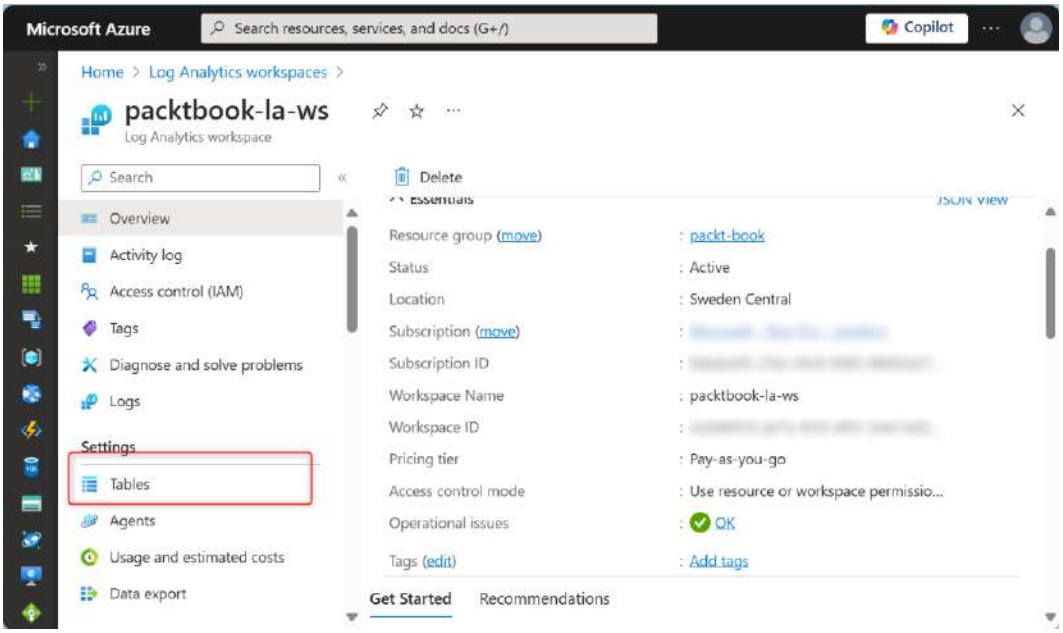

Figure 4.1 – Selection of Log Analytics workspace tables

2. Search for the table you want to modify. In our case, we would like to modify the Key Vault audit logs table called **AZKVAuditLogs**. When you have identified the table, click on the three dots on the right side of the screen to open the contextual menu. There, you will find the **Manage table** option. Click on it.

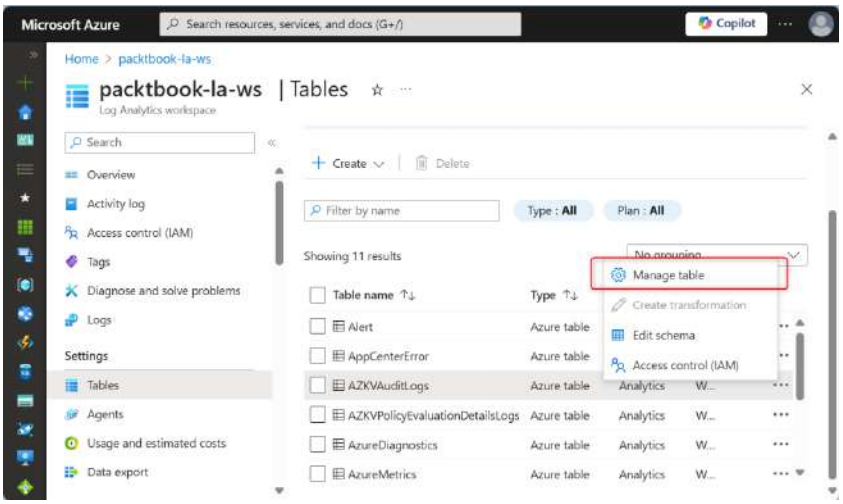

Figure 4.2 – Opening the configuration properties of a table

3. The **Table plan** drop-down menu allows you to change the default **Analytics** value to the **Basic** one we are interested in. If you select a different table and the option is disabled, it means that Azure Monitor does not support Basic Logs for that one. We will discuss this after this example. Select **Save** to close this tab.

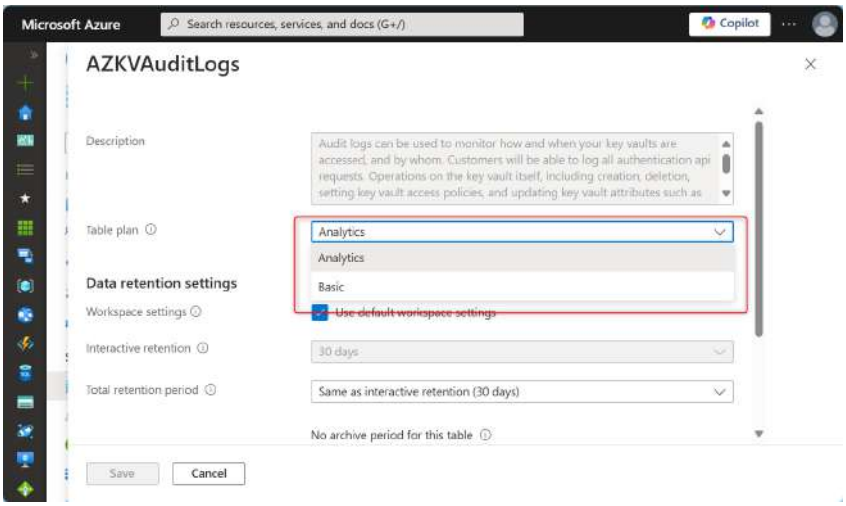

Figure 4.3 – Modifying the plan of a table

After saving the changes to the table, the Azure portal will now show the updated configuration for your table.

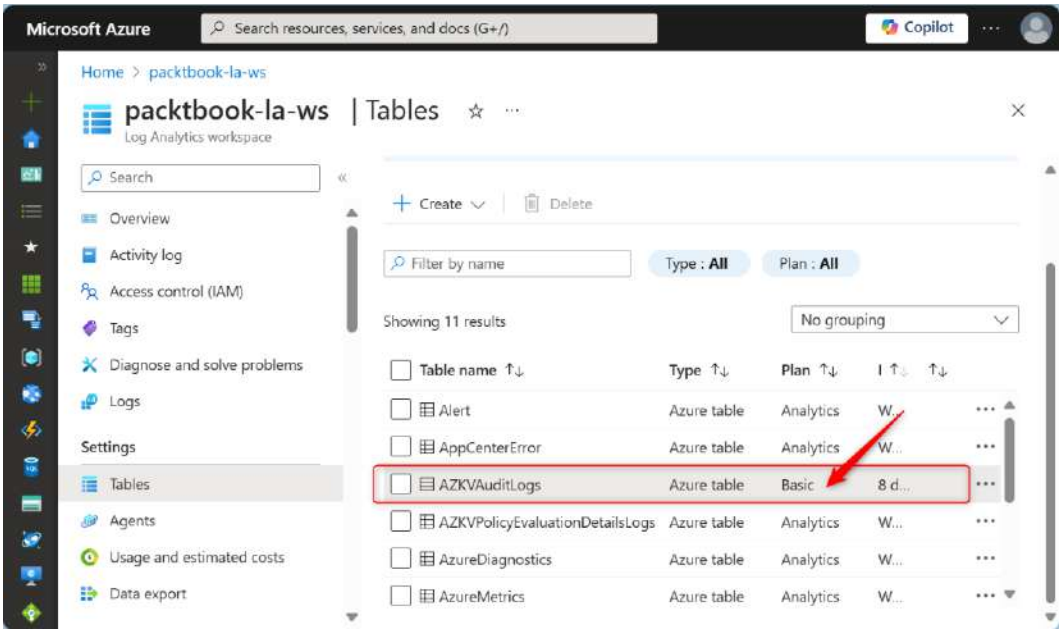

Figure 4.4 – Checking the changes being applied

As mentioned before, you can see in the preceding screenshot that the retention period has been changed to **8 days**, while the original analytics table had a longer retention period. If you change from Analytics to Basic, the service will automatically move the data older than 8 days into the archive. This change is immediate; you need to be aware of this because your queries will return fewer results than the Analytics configuration.

It's important to mention that not all tables support Basic Logs yet. If you are using custom tables, which will be described in *Chapter 5*, all of them are supported. However, not all native Azure Monitor tables are supported. We recommend reviewing the official documentation for an up-to-date list [1].

Your choice of Basic Logs or Analytics Logs in Azure Monitor depends on your specific use case and requirements. Understanding these differences will help you optimize your monitoring strategy. Let's move on now to the querying capabilities included inside Azure Monitor.

Querying in Log Analytics with KQL

Understanding how to analyze your Azure Monitor's data efficiently is crucial. One of the powerful tools at your disposal is the KQL. KQL is designed to help you query and manipulate data not only inside Azure Monitor but also other Microsoft Products, such as Azure Data Explorer and Microsoft Fabric. Its simplicity and effectiveness make it invaluable for extracting insights from large datasets. We introduced KQL with a simple example in the *Log Analytics and data insights* section of *Chapter 2*. In this section, we will cover it in more detail to ensure you can build your own queries efficiently. Azure Monitor supports a subset of the features that KQL provides. It's important to have it in mind when reviewing other sources beyond this book to make sure that specific functionality is included on Azure Monitor, or it's only supported on Azure Data Explorer or Fabric.

KQL is a read-only query language, which means that it is only used to retrieve data and perform data analytics on large datasets. It is specifically designed to query structured, semi-structured, and unstructured data stored in Azure. KQL's syntax is intuitive and easy to learn, especially for those who are familiar with SQL. Its main characteristics are as follows:

- **Simplicity**: KQL is designed to be user-friendly and easy to read, even for those who are not experienced with query languages.
- **Powerful data retrieval**: It allows you to perform complex queries that can filter, sort, and summarize large datasets efficiently.
- **Real-time analysis**: KQL is optimized for speed, enabling real-time data analysis and visualization.
- **Built-in functions**: A wide array of functions for data transformation, aggregation, and visualization are available.

Kusto queries are stated in plain text, using a data-flow model that is easy to read, author, and automate. Before building our queries, let's first understand how we can access the log data ingested inside the Log Analytics workspace through the Azure portal.

Understanding the Log Analytics interface

To access Azure Monitor's querying capabilities, you need to open the Log Analytics workspace user interface. As described in *Chapter 1*, the workspace is the heart of Azure Monitor, where you can store, query, and analyze log data from various sources. Before going into more detail about how to build your queries, this introduction will guide you through the steps to access the Log Analytics workspace via the Azure portal and start executing them:

1. The first step is navigating to the Log Analytics workspaces. You can find them using the search box at the top of the Azure portal; type `Log Analytics Workspace`. The service should appear in the results, as shown in the following screenshot:

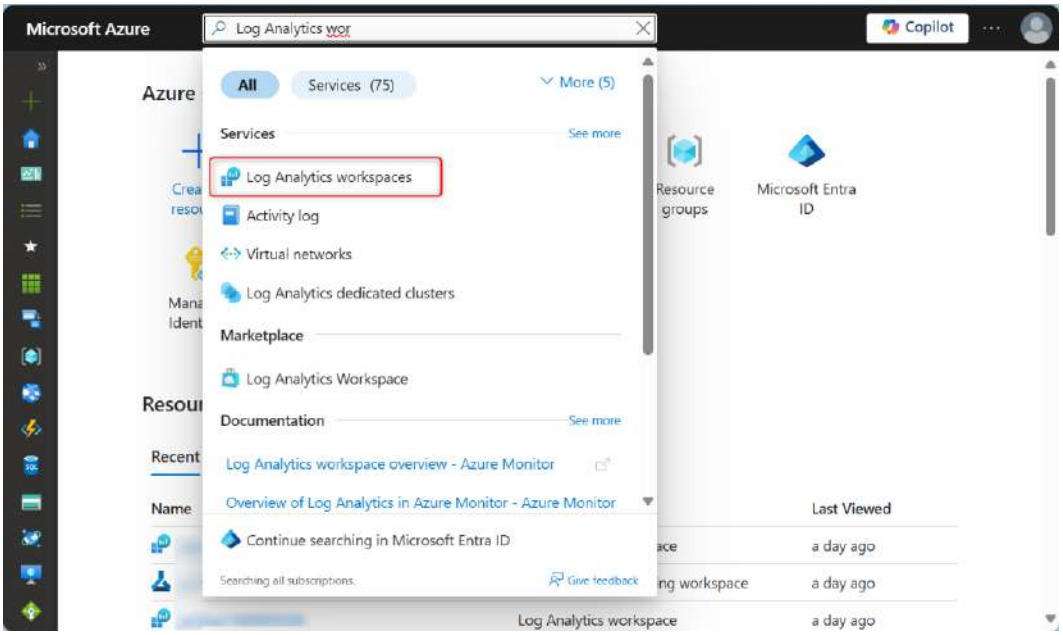

Figure 4.5 – Opening your Log Analytics workspace using the search box

On the **Log Analytics workspaces Overview** page, find and select **Logs** in the left menu.

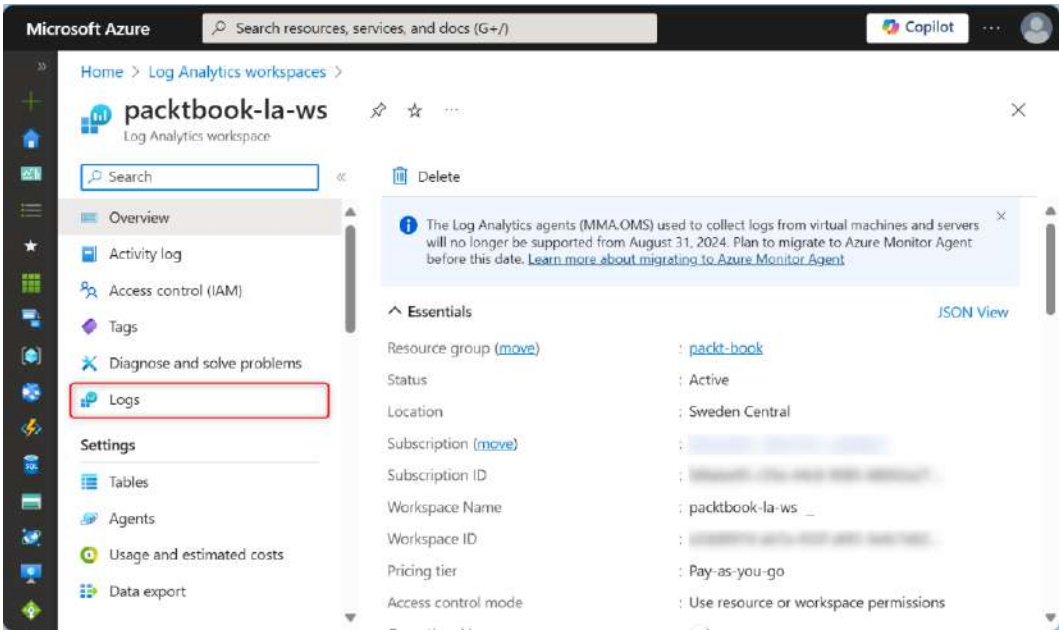

Figure 4.6 – Opening Azure Monitor Logs inside your workspace

2. The new **Simple mode** is opened. It quickly visualizes the information inside Azure Monitor tables. Click on the **Select a table** button to open the list of available tables.

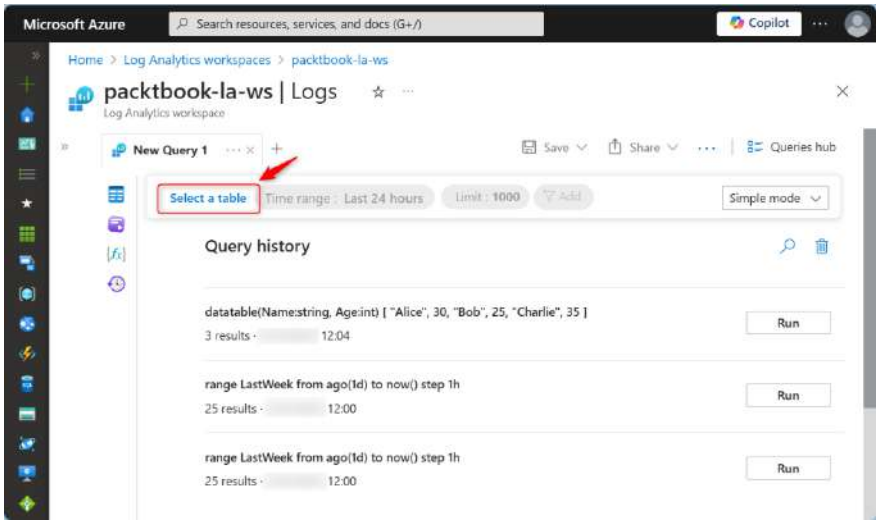

Figure 4.7 – Selecting a Log Analytics table in Simple mode

3. We will select **AzureDiagnostics**, as shown in *Figure 4.8*. You can select any table by following this process. To customize our queries, we will need to change **Simple mode** to **KQL mode** using the drop-down menu on the right side of the interface.

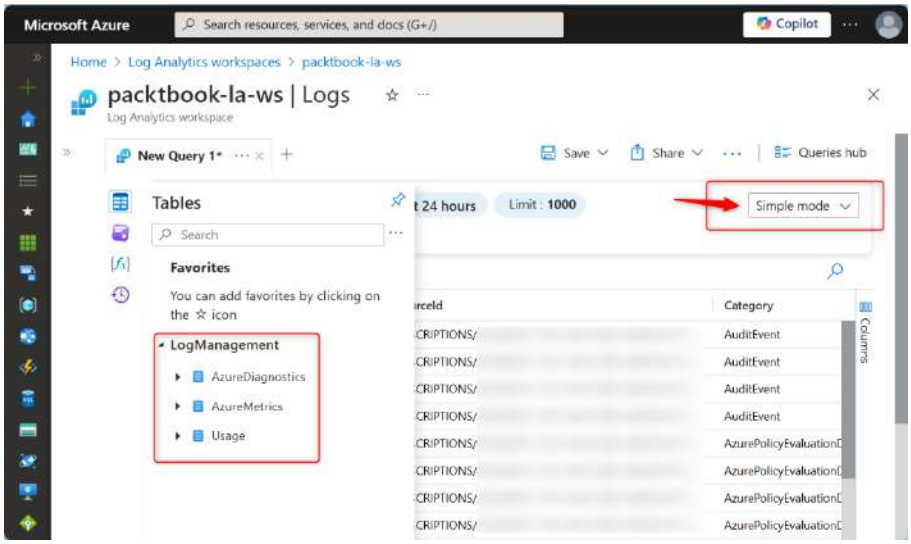

Figure 4.8 – Modifying the log view from Simple mode to KQL mode

4. Once the text editor is open, write a simple query to retrieve the last five results in the table:

```
AzureDiagnostics
| take 5
```

They will appear in the **Results** tab at the bottom of the portal. By default, they are shown as a table; however, Azure Monitor supports different rendering options, as discussed later in this chapter.

> **Tip**
> It is possible to include comments in your KQL queries using // (two slashes) followed by any sequence of characters. A comment can be on its own line, at the end of a query, or in the middle of a KQL query or command. Text inside comments is ignored.

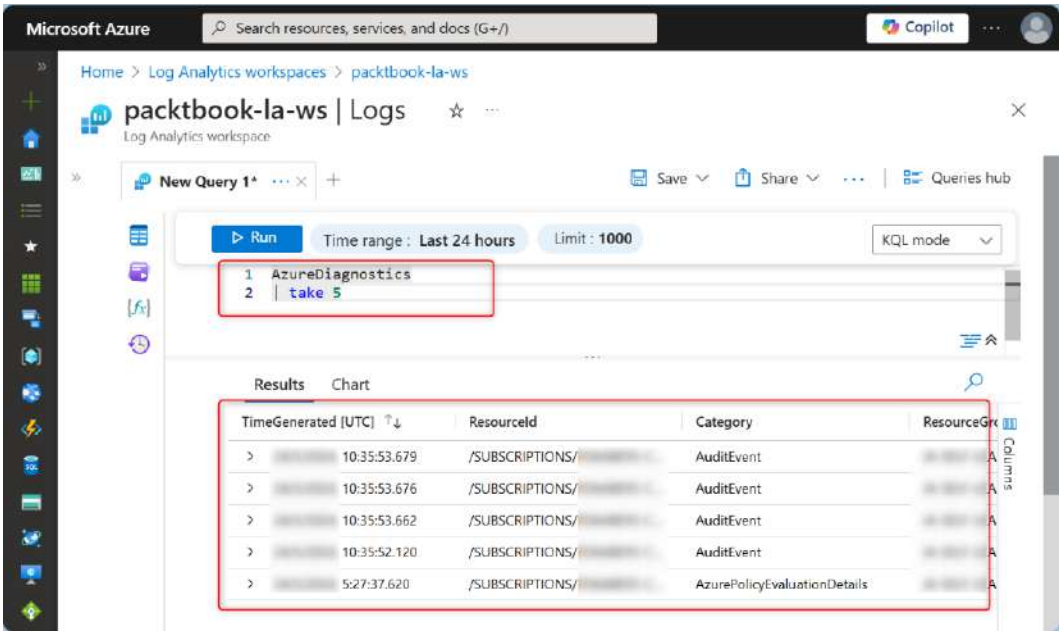

Figure 4.9 – Running a simple query in KQL mode

Accessing and using the Log Analytics workspace in Azure Monitor is a straightforward process, as you have seen. Now, it is time to learn more about KQL and how queries are built to analyze our logs.

Structure of KQL queries

A query is structured using query statements. The are three types of statements: **let**, **set**, and **tabular expressions**. Queries can be built using one or more of those query statements, although at least one of them should be a **tabular expression statement** (**TES**). In this section, we will cover each of these statement types in more detail, starting with tabular expressions.

TESs

At its core, a TES is a series of steps or operations that you perform on a table of data to filter, sort, summarize, and transform it. If you have experience with SQL, it's like a standard SQL query. Each step of the query takes a table as input and produces another table as output, which can then be further processed by subsequent steps. While the term tabular expression statement might sound complex, it refers to a straightforward concept – operations performed on tables of data to derive meaningful insights.

A TES is constructed by chaining together a series of commands or **operators** using the *pipe* (|) symbol. Each command transforms the data in some way, and the pipe symbol indicates the flow of data from one command to the next. This chaining mechanism makes it easy to read and understand the progression of data transformations.

As a simple analogy, imagine you have a conveyor belt in a factory where each station performs a specific task. The data on the conveyor belt (table) moves through different stations (commands), each modifying the data in some way until the final product (the desired result) is achieved. Each station can only see the data that has been processed by the previous station, ensuring a clear and logical flow of operations.

The basic components of this type of statement are as follows:

- **Source Table**: The initial table of data that you start with.
- **Operators**: Commands that perform specific operations on the data, such as filtering, projecting columns, summarizing, and sorting. They are optional if no transformation is required.
- **Pipes** (|): Symbols that connect the operators, indicating the flow of data from one operation to the next.
- **Render Instruction**: Transforms query results into visualizations to help interpret data. It is optional.

The structure of a TES query is as follows:

```
Source | Operator1 | Operator2 | [RenderInstruction]
```

It is also common to have one component per line to make it easier to read:

```
Source
| Operator1
| Operator2
| RenderInstruction
```

Before continuing with more details of each component, let's illustrate these concepts with a simple example. Suppose you have a table called `LogsTable` and you want to find the number of error logs generated in the last 24 hours. This is how you would construct the TES:

1. Start with the source table:

   ```
   LogsTable
   ```

2. Filter the data to include only logs from the last 24 hours using the `where` operator:

   ```
   LogsTable
   | where TimeGenerated > ago(24h)
   ```

3. Further filter to include only error logs:

   ```
   LogsTable
   | where TimeGenerated > ago(24h)
   | where Level == "Error"
   ```

4. Count the number of error logs using the `summarize` operator:

   ```
   LogsTable
   | where TimeGenerated > ago(24h)
   | where Level == "Error"
   | summarize ErrorCount = count()
   ```

Each step in this TES takes the table from the previous step, applies a transformation, and passes the resulting table to the next step. This chaining of commands allows a clear, logical flow of data processing.

To build your own queries, it's important to know which different sources of data are available, the operators to manipulate them, and the rendering alternatives for visualizing the results. Each of the following sections covers these aspects of building queries.

Sources

TES operates on tabular data sources to perform various data manipulations. Understanding the different types of sources that a TES can use is essential for effectively querying and analyzing data. It supports five types of sources:

- **Table references**: A table reference is the simplest and most common source for a TES. It directly refers to a table stored inside the Log Analytics workspace. In the previous example, we used this type of reference when accessing the `LogsTable`.
- **The tabular range operator**: The tabular `range` operator generates a table of numbers within a specified range. It is useful for creating sequences of values for analysis or joining with other data. For example, this query generates a table with a single column, `x`, containing values from `1` to `10`:

  ```
  range x from 1 to 10 step 1
  ```

  This query creates a table with entries for the last 24 hours:

  ```
  range Last24h from ago(1d) to now() step 1h
  ```

- **The print operator**: The `print` operator creates a single-row, single-column table from a specified value or expression. It is useful for testing or returning simple results. This query produces a table with one row and one column, where the `pi` column has the value `3.14159`:

  ```
  print pi = 3.14159
  ```

- **An invocation of a function that returns a table**: Functions in KQL can return tables. These functions can be predefined or user-defined and are invoked to return a set of results that can be further processed by a TES. In this example, `GetErrorLogs` is a function that returns a table of error logs. The function is then invoked, and the resulting table is summarized:

  ```
  let GetErrorLogs = () {
              LogsTable
              | where Level == "Error"
  };
  GetErrorLogs()
  | summarize Count = count()
  ```

- **A table literal**: The `datatable` operator allows you to define a table in line with specific columns and rows. It is useful for creating small, static datasets for testing or as lookup tables. This query creates a table with two columns, `Name` and `Age`, and three rows of data:

  ```
  datatable(Name:string, Age:int)
  [
      "Alice", 30,
      "Bob", 25,
      "Charlie", 35
  ]
  ```

Understanding these sources allows you to use KQL for data analysis. Whether you're querying large datasets, creating test data, or invoking custom functions, these sources provide the flexibility to handle the common scenarios you will encounter when using Azure Monitor. Next in our learning path are operators.

Operators

KQL uses operators to manipulate and analyze data. They help you to filter, sort, aggregate, and shape your data to extract meaningful insights. More than 40 different tabular operators are available inside KQL; let's explore some of the most widely used operators with examples to illustrate their use. We recommend checking the official documentation for further details [2]:

- `where`: The `where` operator is used to filter rows in a table based on specified conditions. It helps you narrow down your data to the rows that are relevant to your query. A condition evaluates to a Boolean for each row passed to the operator. Only rows in which the expression is evaluated to `true` are returned.
- `project`: The `project` operator selects specific columns to include in the result set and can also rename them. It's useful for focusing on relevant data fields. It is like the `SELECT` SQL command.
- `summarize`: The `summarize` operator aggregates data based on specified columns and functions (e.g., count and average). It helps with performing calculations and summarizations.
- `extend`: The `extend` operator adds new columns or modifies existing ones based on expressions. It's useful for creating new data fields derived from existing data.
- `order by`: The `order by` operator sorts the rows in a table based on one or more columns. It allows you to organize data in a specific order.
- `take`: The `take` operator limits the number of rows returned in the result set. It's useful for sampling data.
- `mv-expand`: The `mv-expand` operator expands multivalued fields into multiple rows, making it easier to work with arrays or lists stored in a single column.
- `join`: The `join` operator combines rows from two tables based on a related column, allowing more complex queries involving multiple data sources.

Let's build an example using the previous operators. Imagine that you want to analyze logs from the last 24 hours, focusing on error logs generated by a particular application, count the occurrences of specific keywords in the messages, add a column to indicate the length of each error message, enrich the results with additional application details, sort the results by error count, limit the number of rows, and expand any multivalued fields. Does it seem complex? Not at all.

Let's do it step by step:

1. Filter first the logs to include only those generated in the last 24 hours, with a log level of `Error`, from the `MyApp` application, and with specific words:

```
LogsTable
| where TimeGenerated > ago(24h)
| where Level == "Error"
| where Application == "MyApp"
| where Message contains "failure" or Message contains "timeout"
```

2. Extend the table to add a new column for message length:

```
| extend MessageLength = strlen(Message)
```

3. Summarize the data with the columns we are interested in:

```
| summarize
    ErrorCount = count(),
    FailureCount = countif(Message contains "failure"),
    TimeoutCount = countif(Message contains "timeout"),
    AvgMessageLength = avg(MessageLength)
    by Application
```

4. Join the summarized error data with `ApplicationDetailsTable` based on the `Application` column to enrich the results with additional details, such as application names and owners:

```
| join kind=inner (ApplicationDetailsTable) on Application
```

5. Sort the results by the `ErrorCount` column in descending order to prioritize applications with the highest number of errors:

```
| order by ErrorCount desc
```

6. Limit the output to the top five applications with the highest error counts:

```
| take 5
```

7. Expand the `Tags` field (assuming it is a multivalued field) so that each tag appears as a separate row:

```
| mv-expand Tags
```

8. Select the relevant columns from the joined tables to display in the final result:

```
| project Application, ErrorCount, FailureCount,
TimeoutCount,  AvgMessageLength, ApplicationName, Owner, Tags
```

After selecting the relevant information for us, the last step is to decide how to present it.

Rendering instructions

As mentioned earlier, render instructions are used to visualize the results of your queries. They transform raw data into graphical representations, making it easier to understand and interpret. These instructions are particularly useful in dashboards and reports within Azure Monitor, where visual insights are crucial for quick analysis. They can produce various types of visualizations, such as charts, time series graphs, and pie charts.

To add a render instruction to your query, simply append the `render` keyword followed by the type of visualization you want. This instruction must be at the end of the query. If you introduce it before the end of the query, the engine that processes your query will generate an error. Here are some of the common render types you can use in KQL:

- `table`: This query displays the data in tabular format, showing specific columns. If no rendering instruction is added, results are displayed as a table by default:

```
LogsTable
| project TimeGenerated, Level, Message
| render table
```

- `timechart`: This query creates a time chart showing the count of logs per hour for the last 24 hours:

```
LogsTable
| where TimeGenerated > ago(24h)
| summarize count() by bin(TimeGenerated, 1h)
| render timechart
```

- `barchart`: This query produces a bar chart showing the number of logs for each level (e.g., `Error`, `Warning`, and `Information`):

```
LogsTable
| summarize count() by Level
| render barchart
```

- `piechart`: This query generates a pie chart showing the proportion of logs for each level:

```
LogsTable
| summarize count() by Level
| render piechart
```

The following table summarizes all the rendering options inside KQL in alphabetical order:

Rendering command	Description	Visualization
`areachart`	Displays data over time or categories using filled areas, emphasizing the magnitude of values	
`barchart`	Represents categorical data with rectangular bars; useful for comparing different categories	
`columnchart`	Like the bar chart but uses vertical bars to represent data; useful for comparing categories or time periods	
`piechart`	Displays data as slices of a pie; useful for showing proportions of a whole	
`scatterchart`	Plots individual data points on a two-dimensional plane; useful for showing relationships between variables	
`table`	Displays data in tabular format; useful for detailed data inspection	
`treemap`	Displays hierarchical data as nested rectangles; useful for visualizing proportions within a hierarchy	

Table 4.1 – List of rendering commands in KQL

Render instructions can often be customized with additional parameters to fine-tune the visualization. For instance, you can specify titles, labels, and other settings to make your visual more informative and tailored to your needs. Each render type has its own optional parameters; we recommend checking out the official documentation for further details [3].

let expression statements

In KQL, `let` expressions allow you to define and reuse variables or subqueries within your queries. This can make your queries more organized, readable, and efficient by breaking down complex operations into manageable parts. It acts like a variable declaration in programming, allowing you to store intermediate results or calculations.

Its main benefits are as follows:

- **Readability**: This includes breaking down complex queries into named subqueries and making them easier to read and understand.
- **Reusability**: You can reuse the same subquery or value multiple times within a larger query.
- **Maintainability**: If you need to make changes, you can update the `let` expression in one place, rather than modifying multiple instances throughout your query.

The basic syntax of a `let` statement is as follows:

```
let VariableName = Subquery_or_Value;
```

Suppose you want to analyze error logs and perform multiple operations on them. Using `let` expressions, you can define intermediate steps and reuse them. For example, we can rewrite the example from the *Operators* section like this:

```
let SummarizedErrors = LogsTable
| where TimeGenerated > ago(24h)
| where Level == "Error"
| where Application == "MyApp"
| where Message contains "failure" or "timeout"
| extend MessageLength = strlen(Message)
| summarize ErrorCount = count(), FailureCount = countif(Message
contains "failure"), TimeoutCount = countif(Message contains
"timeout"), AvgMessageLength = avg(MessageLength) by Application;

SummarizedErrors
| join kind=inner (ApplicationDetailsTable) on Application
| order by ErrorCount desc
| take 5
| mv-expand Tags
| project Application, ErrorCount, FailureCount, TimeoutCount,
AvgMessageLength, ApplicationName, Owner, Tags
```

In this case, we have divided the query into two parts. The first one obtains the information about the logs we are interested in, while the second part uses the `let` statement to enrich the results with additional application details through the `join` operator. Structuring your queries in this way helps you to reuse them, so you won't need to write them multiple times.

set expression statements

In KQL, `set` expressions are used to define constants or global settings that can be applied throughout the query. Unlike `let` statements, which define subqueries or variables that are scoped to the current query, `set` expressions can affect the entire session or query environment. Azure Monitor does not support `set` expressions, but they have been included here for reference due to their relevance in KQL.

The `set` statement affects the behavior of the query. These settings can control various things, such as formatting options, performance optimization, and session-specific configurations.

In the first section of this chapter, we discussed the differences between Basic and Analytics Logs. Let's now take a look at the set of KQL operators available in Basic Logs and their limitations compared to the ones just introduced in this section.

Querying limitations for Basic Logs

While Basic Logs in Azure Monitor provide a cost-effective solution for high-volume data ingestion, they come with certain querying limitations. Understanding these limitations is crucial for using Basic Logs effectively and optimizing your log analysis strategy. Here's an overview of the key restrictions when querying Basic Logs.

Basic Logs support a simplified subset of KQL, focusing on essential data retrieval operations. This limitation ensures that queries remain efficient and cost-effective. Supported operators are:

- `where`: Filters records based on specified conditions
- `extend`: Adds new columns or modifies existing ones
- `project`: Selects specific columns to include in the result set

  The following four operators are different variants of this one:

  - `project-away`: Excludes specified columns from the result set
  - `project-keep`: Keeps only the specified columns in the result set
  - `project-rename`: Renames columns in the result set
  - `project-reorder`: Changes the order of columns in the result set

It also supports two extra operators. The first is `parse`, which extracts and creates columns based on patterns in text data. The other is its variation, `parse-where`, which combines parsing with a filtering condition. In the next section, we will look at these two operators in more detail.

Parsing log capabilities to simplify querying

Azure Monitor provides powerful parsing capabilities that allow you to extract and manipulate data from log entries. These parsing capabilities are essential for transforming unstructured log data into a structured format that can be more easily analyzed. Key operators for parsing logs in Azure Monitor include `parse`, `parse-where`, and `parse-kv`. Here's how you can use these operators to effectively parse and analyze your log data:

- `parse`: The `parse` operator is used to extract data from text columns by applying a specified pattern. This operator creates new columns from the extracted values. The structure of the operator is as follows:

  ```
  LogsTable
  | parse TextColumn with "pattern" ExtractedValue
  ```

  For example, imagine you have a log message such as `Error: File not found in directory /user/docs`. You want to extract the error type and the directory path:

  ```
  LogsTable
  | parse Message with "Error: " ErrorType " in directory "
  DirectoryPath
  ```

  This query will create two new columns, `ErrorType` and `DirectoryPath`, with values extracted based on the pattern.

- `parse-where`: The `parse-where` operator combines the functionality of parsing and filtering. It applies a pattern to a text column and filters rows where the pattern matches. The structure of the operator is as follows:

  ```
  LogsTable
  | parse-where TextColumn with "pattern" ExtractedValue
  ```

  For example, suppose you want to filter logs that specifically mention a `timeout` error and extract details from those logs:

  ```
  LogsTable
  | parse-where Message with "Error: timeout in " Component
  ```

  This query will keep only logs where the pattern matches and create a new column, `Component`, with the extracted values.

- `parse-kv`: The `parse-kv` operator is used to extract key-value pairs from a text column and create new columns for each key-value pair found. This operator is especially useful for logs that contain structured data in a key-value format. The structure of the operator is as follows:

  ```
  LogsTable
  | parse-kv TextColumn [options]
  ```

For example, consider a log entry that contains key-value pairs, such as `user=john;action=login;status=success`. You want to extract these pairs into separate columns:

```
LogsTable
| parse-kv Message
```

This query will create new columns user, action, and status, each containing the corresponding values from the key-value pairs.

KQL also provides specialized functions to parse common file formats. These functions are designed to handle predefined structures, making it easier to work with data in standard formats. The list of available functions is as follows:

- **Command line**: The `parse_command_line()` function parses a command line to extract the arguments passed to the command.
- **CSV**: The `parse_csv()` function parses a single string with comma-separated values. It only supports a single record. If multiple records are passed, only the first one is returned.
- **File path**: The `parse_path()` function parses a file path into its components, such as folder, filename, and extension. On top of simple paths, it also supports shared paths, long paths, and schemas.
- **IPv4**: The `parse_ipv4()` function parses an IPv4 address and provides a standardized representation using a signed 64-bit wide long number. It is possible to parse only the network mask with the `parse_ipv4_mask()` function or use `parse_ipv6()` and `parse_ipv6_mask()` for their IPv6 alternatives.
- **JSON**: The `parse_json()` function parses a JSON string into a dynamic object and extracts specific properties. It tries to convert the input data into Azure Monitor data types to enable native capabilities on top of those properties.
- **URL**: The `parse_url()` function parses a URL into its components, such as scheme, host, port, path, and query string.
- **URL query**: The `parse_urlquery()` function is similar to the previous one, but it only parses a URL query string into its key-value pairs.
- **User agent**: The `parse_user_agent()` function parses a user agent string into its components, such as browser, operating system, and device.
- **Version string**: The `parse_version()` function parses a version string into a comparable decimal number.
- **XML**: The `parse_xml()` function parses an XML string into JSON. After that, it returns a dynamic object from where the specific properties can be obtained.

All the options described so far are focused on parsing data at query time. Parsing at query time means transforming log data into a structured format when running a query in Azure Monitor. It allows

you to adjust parsing logic dynamically based on the specific needs of each query, simplifies the data ingestion process by deferring parsing until the data is needed for analysis, and potentially reduces the processing required during data ingestion. However, parsing data at query time can increase query execution time, especially for large datasets, and queries may become more complex as they include parsing logic, which can make them harder to read and maintain.

An alternative is parsing data at collection time. This means transforming log data into a structured format as it is being ingested into Azure Monitor. It is described in more detail in the next chapter when we discuss **Data Collection Rules** (**DCRs**). It allows you to improve query performance since the data is already structured and ready for analysis; it also ensures that data is uniformly parsed and structured, reducing variability and potential errors in data formatting, and structured data is immediately available for analysis without requiring additional processing during queries.

On the other hand, once data is parsed and ingested, changing the parsing rules requires re-ingesting the data or modifying the data processing pipeline, and setting up parsing rules at collection time can add complexity to the data ingestion process.

Understanding the differences between parsing data at collection time and query time allows you to choose the approach that best fits your performance, flexibility, and cost requirements. Choosing one over the other depends on your needs of immediate availability of structured data or the flexibility of dynamic parsing on the fly.

Having explored the comprehensive capabilities of Azure Monitor Log Analytics, including querying logs with KQL, parsing log data, and accessing the Log Analytics workspace, we now have a solid understanding of how to analyze and interpret log data effectively. However, to get a complete picture of your system's health and performance, it's essential to complement log analysis with metrics monitoring. In the next section, we will look into Azure Monitor Metrics.

Harnessing metrics for in-depth analysis

While logs provide detailed, event-driven insights, metrics offer a complementary perspective by delivering quantitative data points that track the performance and health of your resources over time. To effectively collect, visualize, and analyze these metrics, Azure Monitor provides a powerful tool known as Azure Metrics Explorer (aka Metrics Explorer).

Azure Metrics Explorer is a feature within Azure Monitor. It allows you to interactively explore, chart, and gain insights from a vast array of metrics, helping you understand the state and performance of your applications and infrastructure. Before going into further details, let's see how we can get access to it through the Azure portal.

Accessing Azure Metrics Explorer

Metrics Explorer supports multiple accessing paths. If you are interested in a specific resource, you can open it through the **Metrics** menu entry inside the resource. However, if you want to explore across

multiple resources, it's common to go directly to the Metric Explorer interface inside Azure Monitor. Let's see how to open Metrics Explorer inside the Azure portal:

1. You can find Metrics Explorer using the search box at the top of the Azure portal; type the text `Metrics`. The service will appear in the results, as shown in the following screenshot:

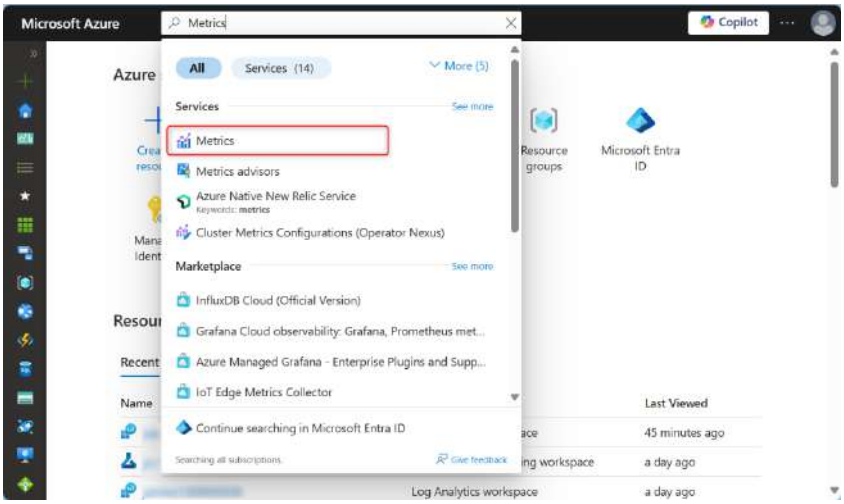

Figure 4.10 – Opening the Metrics blade through the search box

2. A wizard will open to allow you to find the resource or resources you are interested in. In this case, let's select the same virtual machine that we used in the previous chapter called **chapter2-vm**, as shown in the next screenshot:

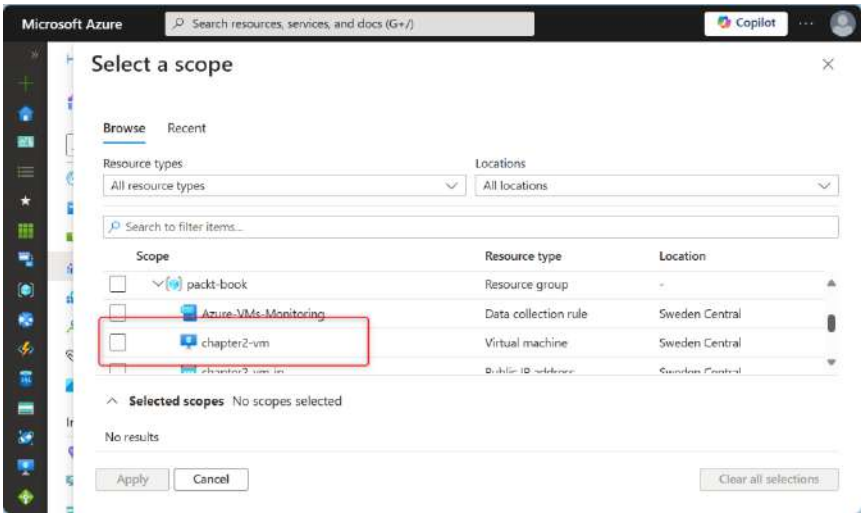

Figure 4.11 – Configuring our scope to a specific virtual machine

3. Once the resource is selected, Azure Monitor will show you the Metrics Explorer so you can start selecting the right metrics to analyze inside it, as you can see in *Figure 4.12*.
4. The next step is to configure and customize metric charts. Select the metric you want to visualize, choose the appropriate aggregation type, and apply any necessary filters or splits. Let's use the **Percentage CPU** usage metric for the virtual machine as an example.

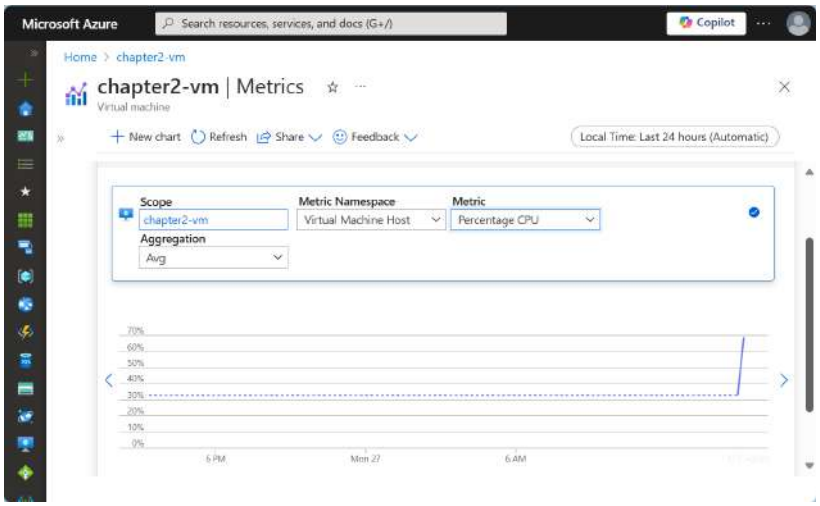

Figure 4.12 – Visualizing the Percentage CPU usage of the selected virtual machine

As shown in the previous screenshot, the data displayed is not particularly relevant. Since my virtual machine has just started, there isn't enough information in the chart to provide meaningful insights for the last 24 hours.

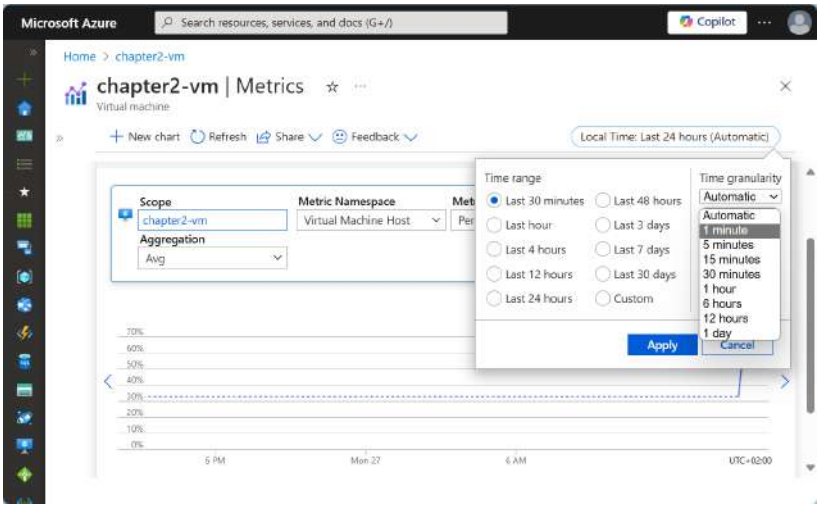

Figure 4.13 – Adjust the time range and granularity for the metric chart

To address this, the first step is to select the appropriate time granularity for the specific scenario. In this case, selecting **Last 30 minutes** for the time range and setting the granularity to **1 minute** will provide more relevant details.

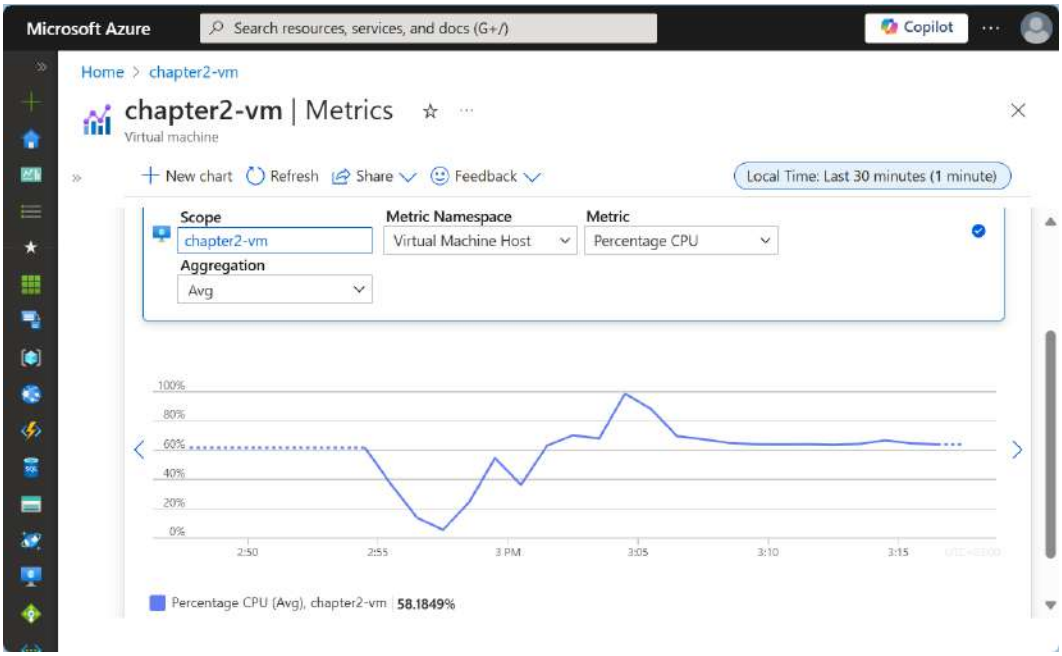

Figure 4.14 – Visualizing the metric after the time adjustment

This is how you can work inside Metrics Explorer to customize your experience to analyze and represent specific metrics for your resources. If this is not enough, Metrics Explorer allows you to overlay multiple metrics on a single chart, enabling you to compare and correlate different performance indicators directly. This feature enhances the ability to compare and correlate different metrics, providing a comprehensive view of resource performance and health. This is useful for comparative analysis scenarios and event correlation.

It also allows you to create multiple charts, each visualizing different metrics, and organize them on a single dashboard. This provides a holistic view of various performance indicators. This is useful for creating a small dashboard that includes multiple charts for a comprehensive view of different metrics, such as CPU usage, memory usage, disk I/O, and network latency, or for segmented analysis, separating different types of metrics into individual charts for focused analysis while still being visible together on the same dashboard.

Both options are available in the top menu, as shown in the following screenshot:

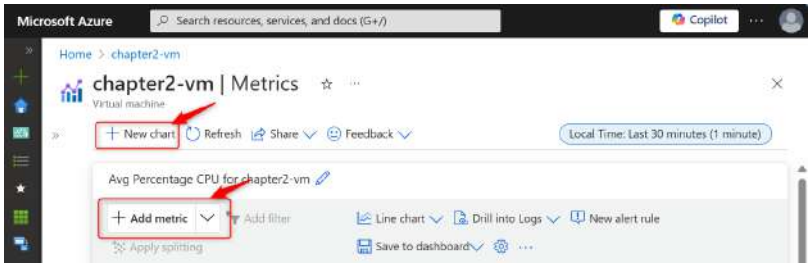

Figure 4.15 – Adding a new chart inside for a more complex analysis

For example, let's create a new chart to visualize the **Available Memory Bytes** inside the virtual machine and the **Disk Read Bytes/sec** to check if there is any strange behavior regarding those peaks. As you can see in the following screenshot, it seems that neither the memory nor the disk experienced any peaks during the same time period.

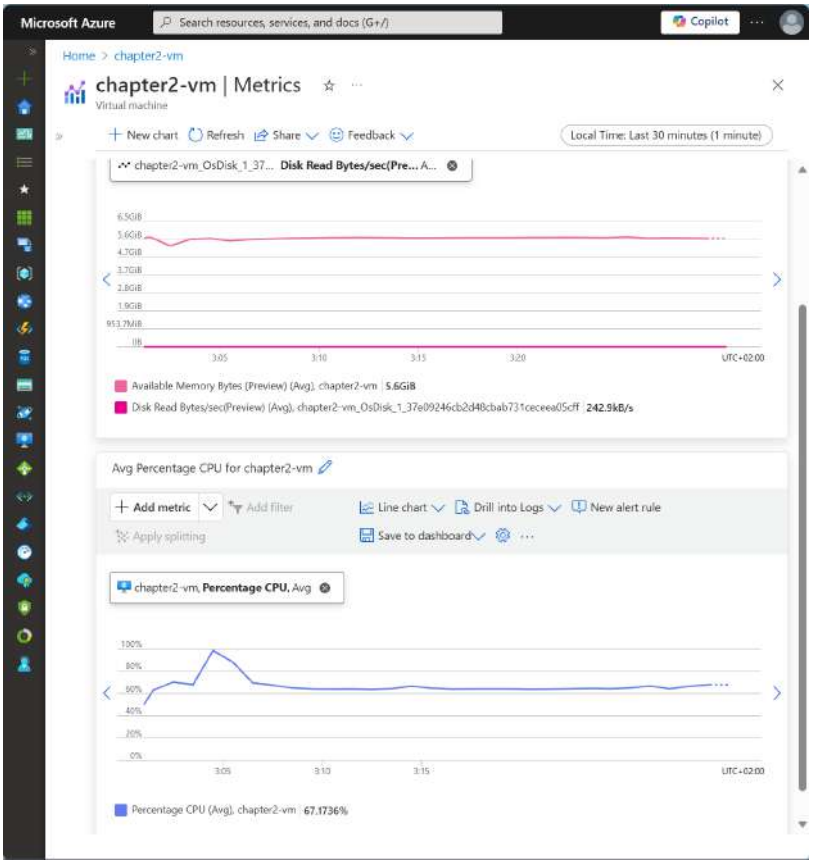

Figure 4.16 – Multiple charts visible at the same time on the Metrics blade

After learning how to add multiple metrics to a chart and multiple charts to a dashboard, let's explore how to customize the experience. By selecting the gear icon in your chart menu, as shown in the following screenshot, you can access the configuration options. Here, you can choose the chart type, set the title, customize the axis ranges, adjust the legend, and modify other details of your query. Additionally, if you've added multiple metrics to the same chart and wish to customize their colors for easier analysis, you can do so by clicking the color square next to the metric name. This allows you to change the color to your preference.

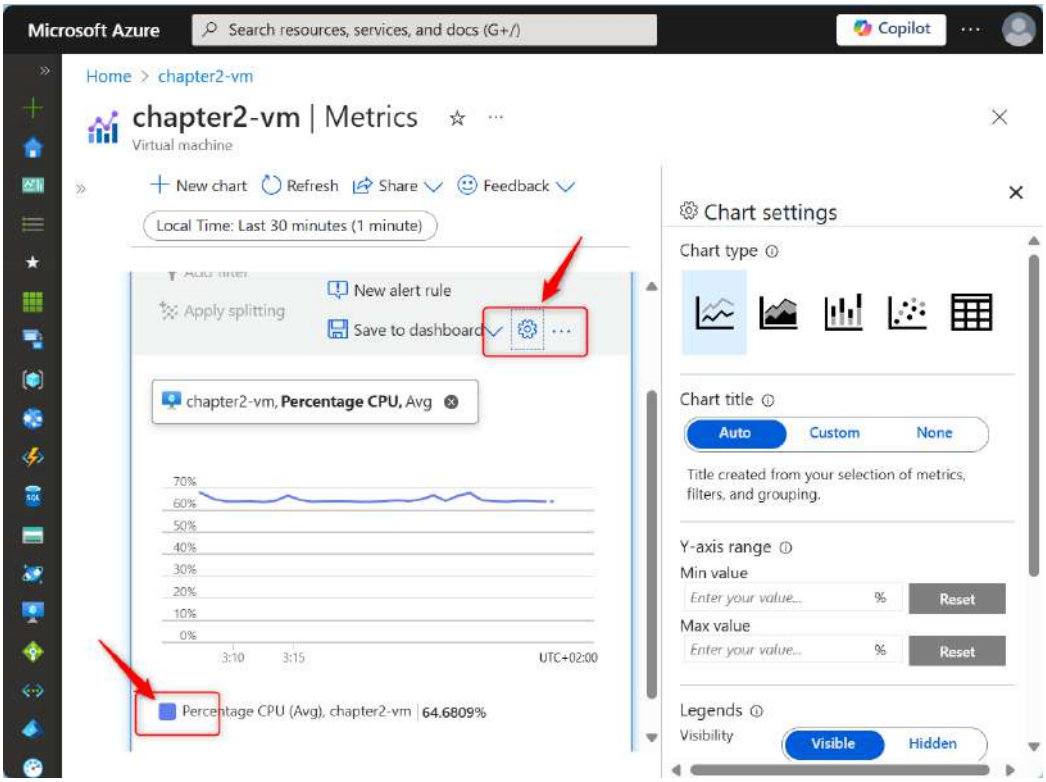

Figure 4.17 – Customizing your visualizations

Understanding how to use and interpret metrics aggregations is essential for gaining accurate and meaningful insights into the health and performance of your Azure resources. The next section explains them in more detail.

Understanding metric aggregations

One of the key features of Metrics Explorer is its ability to perform **metric aggregations**. Aggregations help you summarize and make sense of large volumes of metric data by combining individual data points into meaningful statistics. This section will explain the different types of metric aggregations available in Azure Metrics Explorer and how they can be used to gain insights into your resource performance.

Metric aggregations are statistical methods used to combine multiple data points over a specified time interval or across different dimensions. Aggregations provide a summarized view of the data, making it easier to analyze trends and patterns. Metrics Explorer supports five different statistics:

- **Average** (`avg`): The average aggregation calculates the mean value of all data points within a specified time interval or dimension. It is useful for understanding the typical value of a metric, such as the average CPU utilization of a virtual machine.
- **Count**: The `count` aggregation counts the number of data points within a specified time interval or dimension. It is useful for metrics that represent discrete events, such as the number of failed login attempts or the number of API calls.
- **Maximum** (`max`): The `max` aggregation finds the largest value among all data points within a specified time interval or dimension. It's useful for identifying peak values, such as the maximum CPU utilization or the highest number of requests per second.
- **Minimum** (`min`): The `min` aggregation finds the smallest value among all data points within a specified time interval or dimension. It's useful for identifying the lowest observed value of a metric, such as the minimum memory usage of a resource.
- **Sum**: The `sum` aggregation totals all data points within a specified time interval or dimension. It's ideal for metrics that represent cumulative values, such as total transactions processed, or total bytes transferred.

Metrics aggregations are closely related to time granularity. It refers to the time interval at which metrics data points are aggregated to be displayed on a chart. It differs from the frequency at which the data is collected and ingested into Azure Monitor.

While the minimum granularity supported by Metric Explorer is 1 minute, your collection frequency can be lower. Azure Monitor can aggregate sub-minute metrics and display them in your charts. If you are interested in learning more about how the process of aggregating works, we recommend checking out the official documentation [4].

When using Azure Monitor to track and analyze the performance of your Azure resources, selecting the appropriate combination of time granularity and frequency of metrics is crucial. This decision impacts the effectiveness of your monitoring strategy and the accuracy of your insights.

Time granularity lets you control how detailed a chart is. For example, if you are monitoring your CPU usage and your granularity is too big, for example, the minimum of 1 minute supported by Azure Monitor, you will see lots of tiny changes and peaks in usage. This will not help you identify trends correctly. On the other hand, if your granularity is high, such as more than 30 minutes, the smoothness produced in the graph could give you the details about the usage trend, but you may miss peaks where your virtual machine is not able to handle the workload correctly.

It's crucial to understand what's what you are looking for when monitoring your system to choose the best relationship between frequency and time granularity. It's important to know about metrics dimensions with aggregations as well, as discussed in the following section.

Understanding metric dimensions

In *Chapter 2*, we introduced the concept of dimensions when talking about Azure metrics. Although the most common scenario is having a single-dimension metric represented as a name-value pair, sometimes this is not enough.

Metric dimensions are additional attributes associated with a metric that allow you to break down, filter, and analyze the metric data more granularly. They provide context to the metric values and enable detailed analysis by segmenting data based on specific characteristics. Imagine you are monitoring the sales performance of a store. Instead of just showing the total sales, dimensions allow you to break down the sales data by product category, region, or sales channel, providing a deeper understanding of the sales performance.

While a single dimension provides a single aggregate value for the metric, without any breakdown by other attributes, multidimensional metrics include one or more dimensions, allowing you to segment and filter the metric data based on various attributes. Each combination of dimension values provides a distinct metric series.

Single-dimensional metrics are useful for high-level monitoring and gaining an overall understanding of resource performance, while multidimensional metrics are useful for detailed analysis, allowing you to identify patterns, trends, and anomalies within specific segments of the data.

Azure Monitor fully supports both single and multidimensional metrics within Metrics Explorer. It's essential to understand the type of metric you are analyzing. For instance, imagine you add two data disks to the virtual machine monitored in this chapter to support a new workload. This workload is designed to use both disks in parallel for better throughput. However, after deployment, the performance is not as expected. You decide to use Metrics Explorer to investigate the issue. The following screenshot shows the configuration of the **Data Disk Write Operations/Sec** metric. Notice the small icon before the metric name, indicating that it is a multidimensional metric:

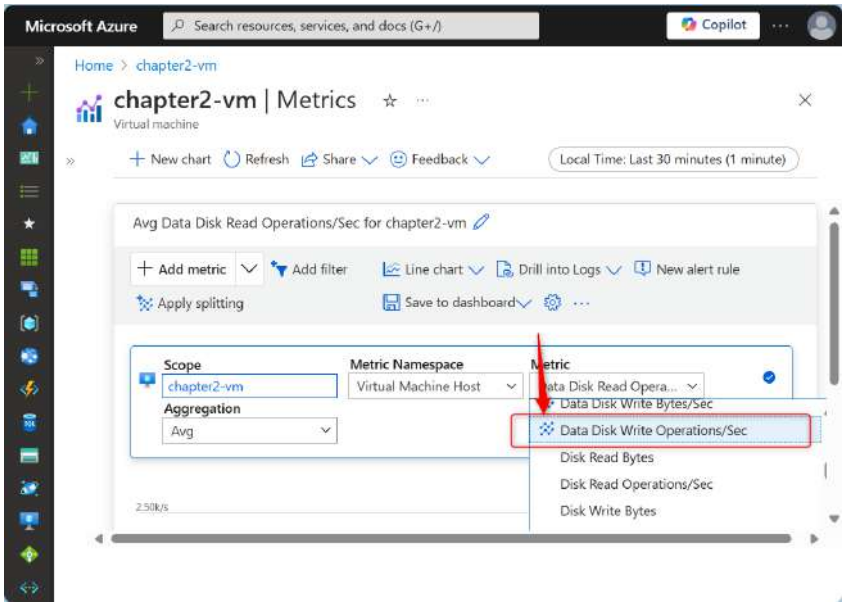

Figure 4.18 – Difference between multidimensional and single-dimension metrics

If you add the metric, you will see a few operations occurring. However, by selecting the **Apply splitting** option, as indicated in the screenshot, you can view the metrics for each disk independently instead of the default aggregated version.

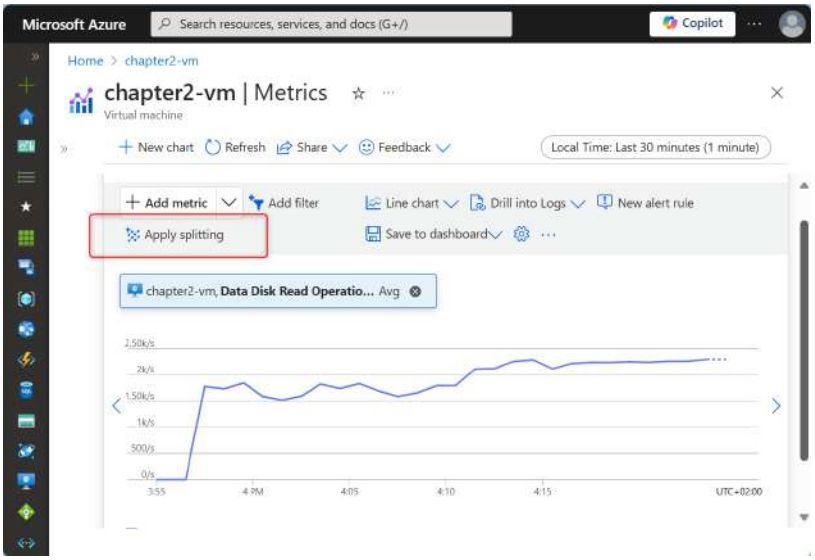

Figure 4.19 – Customizing the visualization of a multidimensional metric

In this case, the metric includes another dimension: the `LUN` number of each disk. This additional dimension allows a more detailed analysis rather than a high-level overview. As shown in the following screenshot, after adding the new dimension to the graph, it becomes clear that `LUN 0` is generating all the usage, while `LUN 1` is doing nothing.

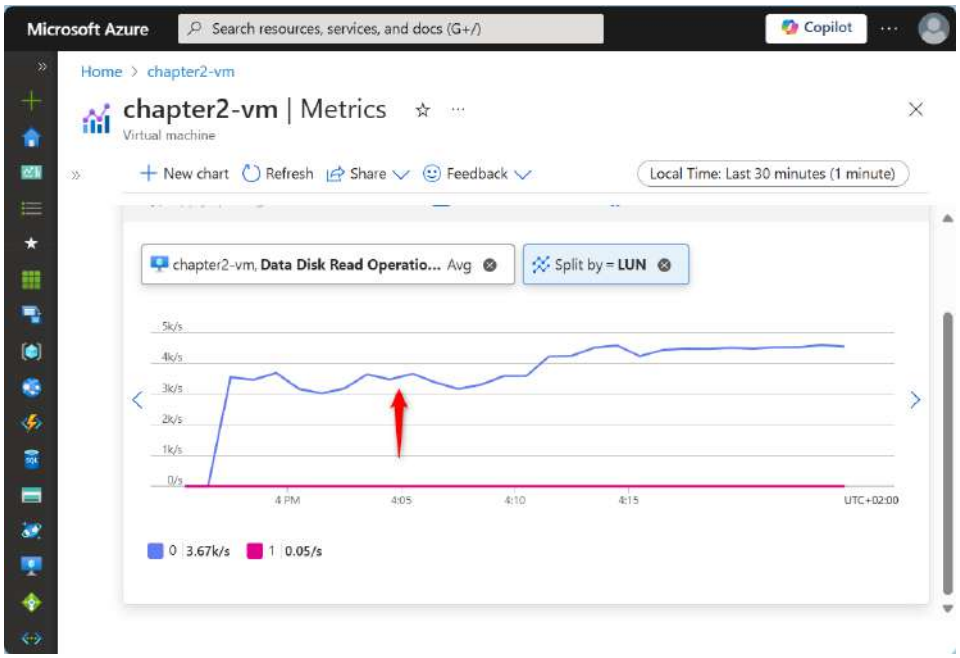

Figure 4.20 – Multidimensional representation of the LUN metric

In this section, we have provided you with the tools and techniques necessary to harness the full power of Metrics Explorer. By understanding how to access and navigate the Metrics Explorer, configure and customize metric charts, and use features such as multiple metrics on a single chart and metric dimensions, you are now equipped to perform in-depth analysis and gain meaningful insights into your Azure resources' performance and health.

The next step is to apply these capabilities to your specific use cases, enabling you to monitor, diagnose, and optimize your environment effectively.

Summary

In this chapter, we described the functionality that Azure Monitor provides for analyzing data using logs and metrics. We began by exploring the critical role that logs play in monitoring and diagnosing issues in cloud environments. We explored the two primary types of logs in Azure Monitor – Basic Logs and Analytics Logs – highlighting their differences, use cases, and characteristics.

We then explained how to query log data using KQL. KQL's powerful yet simple syntax allows efficient data retrieval, transformation, and visualization. We covered key KQL components such as tabular expressions, `let` statements, and parsing capabilities, and demonstrated their practical applications in extracting meaningful insights from large datasets.

Next, we explored the use of Azure Metrics Explorer for collecting, visualizing, and analyzing metrics. We demonstrated how to access Metrics Explorer, configure metric charts, and understand metric aggregations and dimensions. By using these tools, you can gain a comprehensive view of your system's performance and health, facilitating proactive decision-making and optimization.

In the next chapter, we will build upon our foundational knowledge of monitoring by focusing on how to respond to monitoring events using alerts. We'll explore the various types of alerts available in Azure Monitor, how to configure them, and best practices for setting up a responsive and proactive alerting system. This will enable you to not only monitor your systems but also take immediate action when critical issues arise, ensuring minimal downtime and optimal performance.

Further reading

Here, you can find the resources to expand your knowledge about concepts not covered in this book but mentioned in this chapter:

- [1] Azure Monitor Basic logs supported tables: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/basic-logs-configure?tabs=portal-1#supported-tables`.
- [2] Azure Monitor query operator list: `https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query/queries`.
- [3] Azure Monitor rendering operator visualization types: `https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query/visualization-anomalychart?pivots=azuredataexplorer`.
- [4] Azure Monitor metrics aggregation and display explained: `https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/metrics-aggregation-explained#how-aggregation-works`.

5
Responding to Monitoring Events

In this chapter, we will explore how to effectively respond to monitoring events using Azure Monitor. We will begin by exploring the structure of alerts and actions to proactively respond to monitoring events. Here, you will learn how to set up alert conditions, thresholds, and automated responses, ensuring timely intervention to mitigate potential issues. We will also show you how to establish a robust incident response plan, optimizing your organization's ability to maintain the health and reliability of cloud resources. By the end of this chapter, you will be equipped to handle monitoring events with confidence, ensuring your Azure environment operates smoothly.

We are going to cover the following topics:

- Configuring proactive alerts in Azure Monitor
- Automated responses to monitoring events
- Threshold definition for effective alerting
- Establishing an incident response plan

Technical requirements

We must have access to Azure Monitor. We will use an Azure storage account and several virtual machines that were created just to show examples when discussing Azure alert types and other topics throughout the chapter.

Configuring proactive alerts in Azure Monitor

In this section, you will learn how to configure the different types of Azure Monitor alerts. However, before explaining the different configurations of each type of alert, it is important to understand the flow of an alert and the components that make it up.

The flow of an alert in Azure is as follows:

1. Azure resources emit telemetry to Azure Monitor Logs or Azure Metrics.
2. Telemetry is evaluated by the alert rule, which, for a specific scope or resource, captures the signal and checks if it meets a certain condition. If multiple resources are being monitored, the condition is evaluated for each resource.
3. If the conditions of the alert rule are met, an alert is triggered for each resource. Alerts have a retention period of 30 days and can be viewed from the Azure portal. Through alert processing rules, alerts can be modified to add or suppress notifications or actions, filter a subset of alerts to apply them to or specify the time intervals for receiving notifications.
4. The alert triggers action groups that consist of a set of notifications and actions to alert users that an alert has been triggered and include context for it.
5. Depending on the type, triggered alerts may auto-resolve or require manual resolution by the user.

From the previous flow, it is clear that an alert is configured with several components:

- **Alert rule**
- **Action group**
- **Alert processing rules**
- **Alert integrations**

In *Chapter 2*, we explained the different types of alerts that can be created in Azure Monitor, but we did not cover the configuration steps for each alert rule or how to configure action groups, actions, and notifications. Although the Alert Creation Wizard configures all components one after the other, the configuration of action groups, alert processing rules, and alert integrations will be covered in the *Automated responses to monitoring events* section, and in this first section, you will learn how to configure the following types of alert rules during the alert creation flow:

- **Metric alert rules**
- **Log search alert rules**
- **Activity log alert rules**

Smart detection alerts are part of **Application Insights** and will be covered in detail in *Chapter 7*.

Metric alert rules

Azure metric alert rules are a key feature for monitoring the performance and health of your resources. They continuously monitor a resource by evaluating specific conditions in its metrics (such as CPU usage and memory consumption) at regular intervals. Let's learn how to configure a metric alert rule.

In the Azure Monitor portal, select **Alerts** from the left menu. Then, select **Create** and choose **Alert rule**, as shown in the following screenshot:

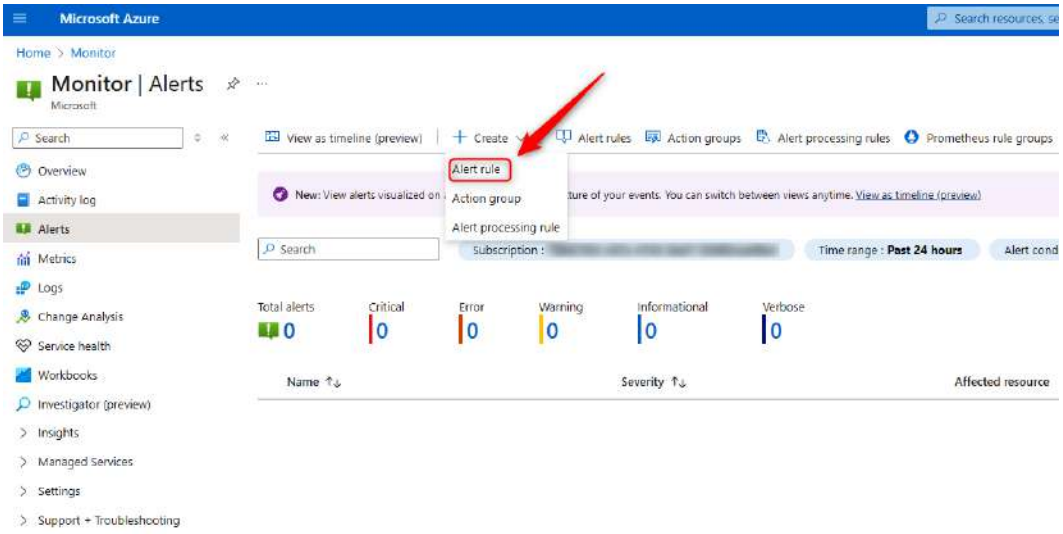

Figure 5.1 – Alert rule creation tab

After selecting the **Alert rule** button, the alert wizard appears showing the **Scope**, **Condition**, **Actions**, and **Details** tabs. Let's see the options in each of them.

Scope

During the creation of the alert rule, it is necessary to configure its scope. You can filter the scope by subscription, resource type, or resource location. The scope can encompass multiple resources of the same type within the same region. In the following screenshot, you can see what the scope selection looks like:

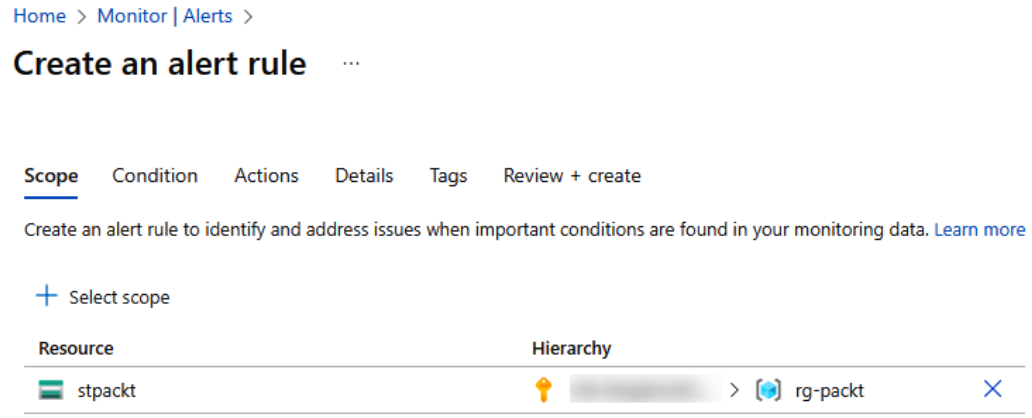

Figure 5.2 – Alert rule scope

Condition

To see all signals grouped by signal type, click on **See all signals**, as shown in the following figure. The list displayed depends on the type of resource selected on the **Scope** tab.

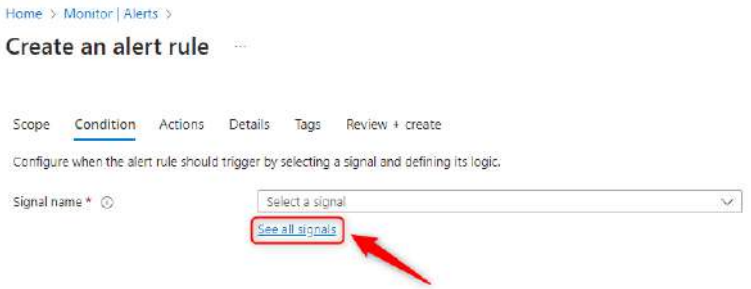

Figure 5.3 – See all signals tab

The signal type for this type of alert rule is Metrics. The signal source is the service that emits the signal. In the case of this type of alert rule, as shown in *Figure 5.4*, the signal source is **Platform metrics**.

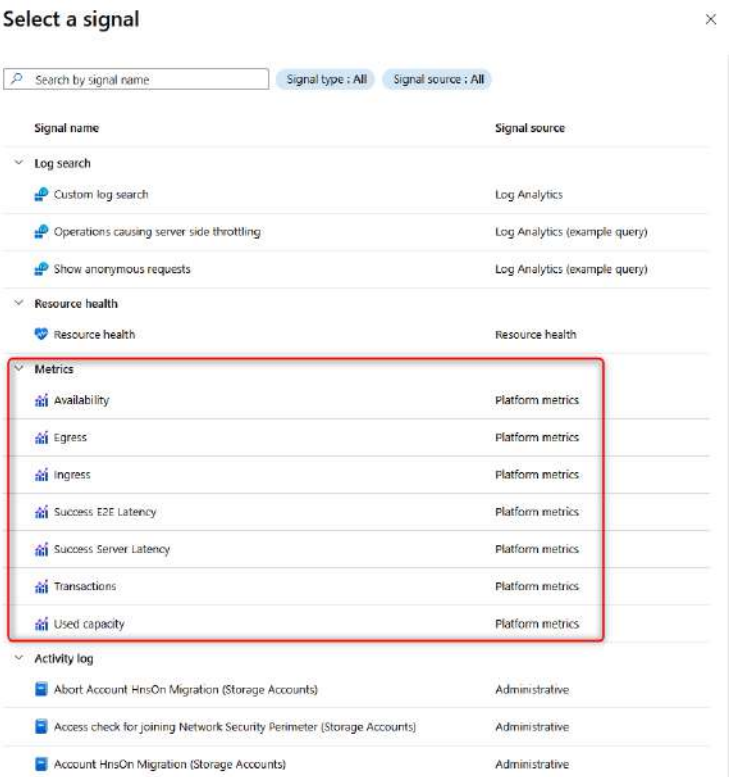

Figure 5.4 – Signals from platform metrics

It is important to note that some metric and log signals might not be available if the scope includes multiple resources. The alert rule can include multiple conditions as long as the scope is a single resource. In our example, we set the signal name to **Transactions**. After selecting the signal, the **Condition** tab opens the following three additional configurations (see *Figure 5.5*):

- **Alert logic**: These fields allow you to fine-tune the alert conditions and determine when an alert should be triggered based on the behavior of the monitored metric. Through these parameters, you can customize the rule to fit the requirements of your application or environment in Azure. The fields and possible values are as follows:
  - **Threshold**: We can choose either **Static** or **Dynamic**. In our example, we select **Static**.
  - **Operator**: The options are **Greater than**, **Greater than or equal to**, **Less than**, and **Less than or equal to**. In our example, we select **Greater than**.
  - **Aggregation type**: The options are **Sum**, **Count**, **Average**, **Min**, and **Max**.
  - **Threshold value**: This is relevant only if you selected a static threshold.
  - **Unit**: The options **Count**, **KB**, **MB**, and **GB** are available only if you select a static threshold and if the metric signal supports units.
  - **Threshold sensitivity**: The options are **High**, **Medium**, or **Low**, only if you select a dynamic threshold.

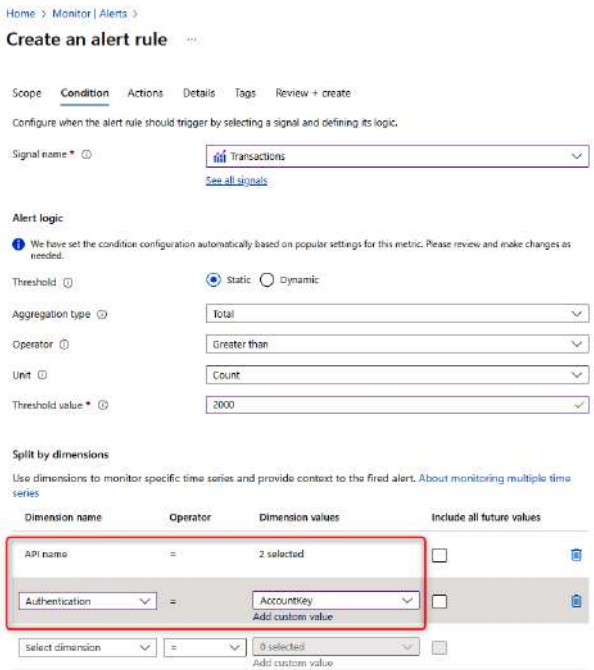

Figure 5.5 – Configuring alert rule logic

- **Split by dimensions**: Dimensions can be either number or string columns. Dimensions allow us to filter the metric and selectively monitor specific time series. This approach allows the metric to be monitored with a focus on individual dimensional values rather than as an aggregate of all dimensional values combined. If you select more than one dimension value, each time series that results from the combination will trigger its own alert.
- In our example, we have five options for the dimension name (**API name**, **Authentication**, **Geo type**, **Response type**, and **Transaction type**) and for each, there are specific dimension values to which you can add your own custom values. To illustrate, as shown in *Figure 5.5*, we select the **API name** dimension with the values **ListContainers** and **GetContainersProperties**, and the **Authentication** dimension with **AccountKey** as the dimension value. Because of this selection, two time series of metrics are being monitored for this alert rule:
  - Transactions where `Resource='stpackt',API name='ListContainers'`, and `Authentication='AccountKey'` are greater than 2,000
  - `Transactions where Resource='stpackt' and APIname='GetContainersProperties'` and `Authentication='AccountKey'` are greater than 2,000

  It is important to mention that in an alert rule with multiple conditions, you can only select one value per dimension. Otherwise, the **Add condition** option will be disabled, as you can see in *Figure 5.6*.

Important note

We have discussed how Azure Monitor allows the use of dimensions to filter metrics, providing a more granular approach to threshold definition. It is advisable to make use of dimensions to enhance accuracy in alerts. By using dimensions, you can conduct an analysis focused on specific aspects of a metric, ensuring that alerts are triggered only for relevant subsets of data. This approach is particularly valuable in scenarios where monitoring the metric as an aggregate could result in less meaningful or actionable alerts. For example, consider the metric of transactions for a storage account, where there's a dimension called API name containing the name of the API called for each transaction (e.g., `DeleteBlob`). If not filtered by API name dimension, an alert might be triggered when there's a high volume of transactions for a specific API in aggregated data, such as `PutBlob` and `GetBlob`. However, using the API name dimension with a custom value of `DeleteBlob` could trigger a precise alert only when the transaction count is high for that specific API.

- **When to evaluate**: The fields and possible values are as follows:
  - **Check every**: Indicate how often the alert rule checks if the condition is met. You can select from 1 minute up to 1 hour.
  - **Lookback period**: Select the time period to look back at each time the data is checked. You can select from 1 minute up to 24 hours.

In our example, as shown in the following figure, we set Check every to **1 minute** and set Lookback period to **5 minutes**.

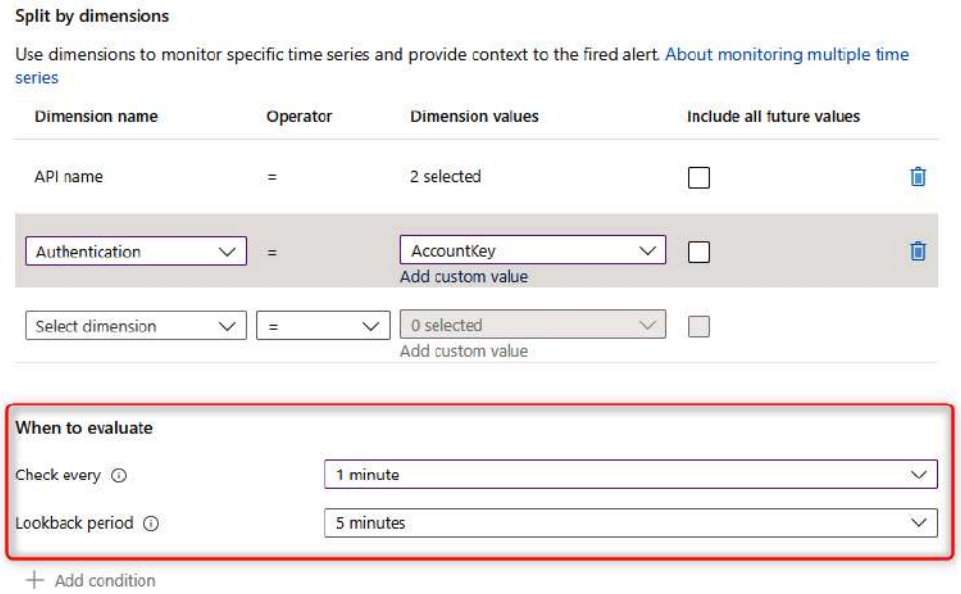

Figure 5.6 – Alert rule frequency and period

In our example, we are configuring a static threshold, but if you configure an alert rule using dynamic thresholds and it is too noisy or triggers too frequently, you may need to lower its sensitivity. As shown in the following figure, consider using the following options:

- Set **Threshold sensitivity** to **Low** to allow greater tolerance for deviations.
- Use the **Advanced options** tab to configure the number of violations in an evaluation period to indicate how many violations will trigger the alert within a specific time period.

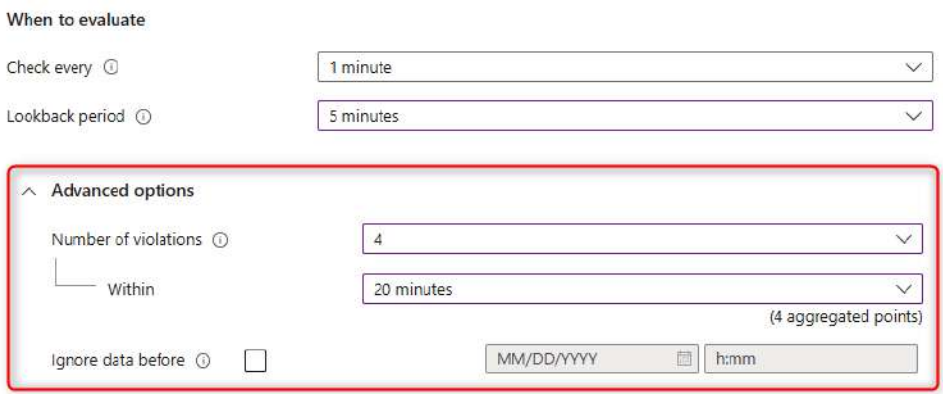

Figure 5.7 – Advanced options to lower alert sensibility

It is also important to mention that if you are configuring an alert with multiple conditions, you can set different values of frequency and period for different conditions; however, the largest one will be used.

Once we have completed the **Condition** tab, we move on to the **Actions** tab of the alert creation wizard.

Actions

Because **Actions** is an extensive topic, we recommend going to the *Automated responses to monitoring events* section to learn how to create notifications, actions, and action groups, and then come back to this section to continue with the **Details** tab. An action group is a collection of actions or notifications triggered when an alert is fired. You can select up to five action groups to attach to the rule. The following figure shows what actions are contained inside the action group we have created for our metric alert rule:

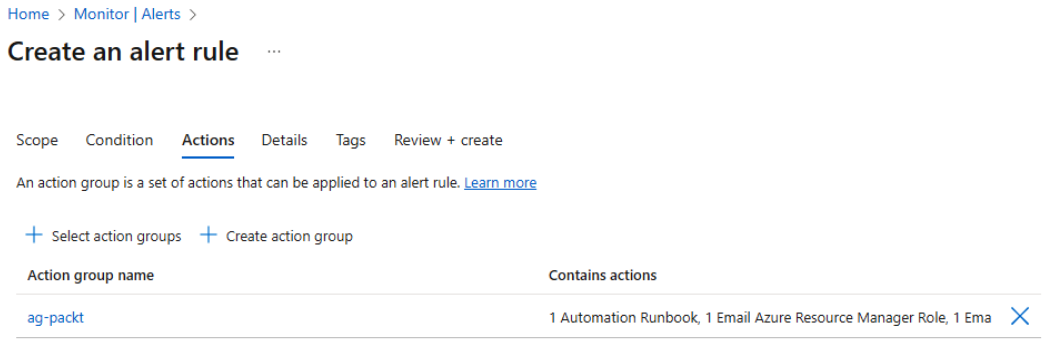

Figure 5.8 – Alert rule actions

Details

As shown in *Figure 5.9*, the fields and possible values are as follows:

- **Subscription**: Select a subscription to save the alert rule.
- **Resource group**: Select a resource group to save the alert rule, in our example, `rg-packt`.
- **Severity**: Select among **Critical**, **Error**, **Warning**, **Informational**, and **Verbose**.
- **Alert rule name**: We will choose the name of the alert rule, for example, `alert-transactions-sapackt`.
- **Alert rule description**: We will describe the alert rule, for example, `Total Transactions is greater than 2000`.

- **Advanced options**: Some advanced options are as follows:
  - **Custom properties**: When action groups are added, there is the option to use custom properties to add your own properties to the alert rule. These properties will be included with the alert payload. The custom properties are specified as key/value pairs and can be static text, or a dynamic value extracted from the alert payload using `${<path to schema field>}`. For example, one custom property name could be `Condition Start Time` with a value of `${data.alertContext.condition.windowStartTime}`, and another custom property could be `Condition End Time` with a value of `${data.alertContext.condition.windowEndTime}`.
  - **Enable upon creation**: Use this if you want to enable the alert rule at the creation time.
  - **Automatically resolve alerts**: Mark this if you want a stateful alert, that is, the alert is resolved when the condition is no longer met. Otherwise, the alert will be stateless and will be triggered each time the condition is met.

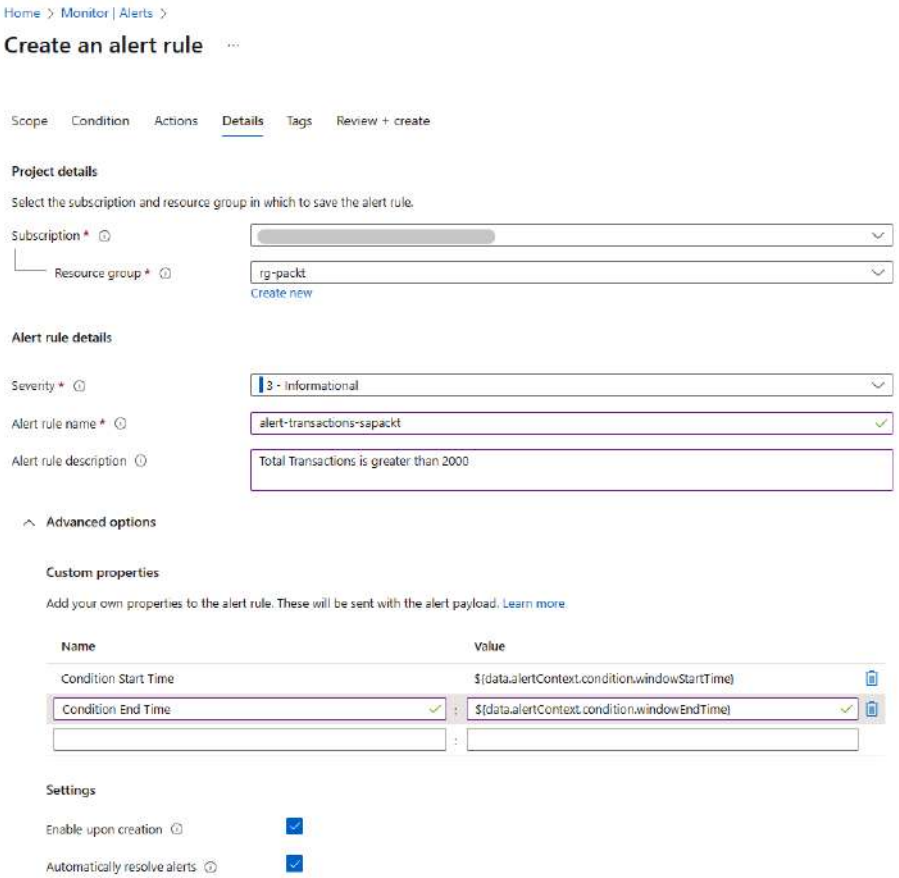

Figure 5.9 – Alert rule details

With this configuration completed, you have learned how to create a metric alert, and we can move on to show you how to create and configure the next type of alert, which is the log search alert.

Log search alert rules

In *Chapter 2* and *Chapter 3*, we introduced resource logs, and in *Chapter 4*, we explained how you can use KQL to analyze them. Log search alert rules allow users to use a KQL query to evaluate resource logs at a predefined frequency and fire an alert if the conditions of the alert rule are met. This type of alert rule is very useful for scenarios where the data you want to monitor is available in logs and you want to use advanced queries. To configure a log search alert rule, we first create a rule, as we did in the *Metric alert rules* section.

After selecting the **Alert rule** button, the alert wizard appears showing the **Scope**, **Condition**, **Actions**, and **Details** tabs to configure the alert. Let's see the options in each of these tabs.

Scope

If we select the **rg-packt** resource group, the alert query scope will include records created by all resources in the resource group. It can include data from multiple Log Analytics workspaces. In the following screenshot, you can see what the scope selection looks like:

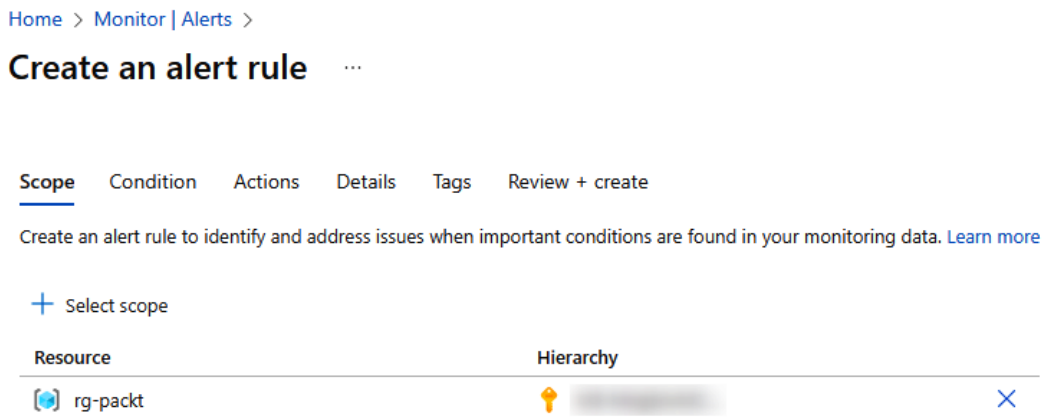

Figure 5.10 – Alert rule scope

Condition

The signal type for this type of alert rule is **Log search**. For this type of alert rule, as shown in *Figure 5.11*, the signal source is **Log Analytics**. To see all signals grouped by signal type , click on **See all signals**. The list displayed depends on the type of resource selected on the **Scope** tab.

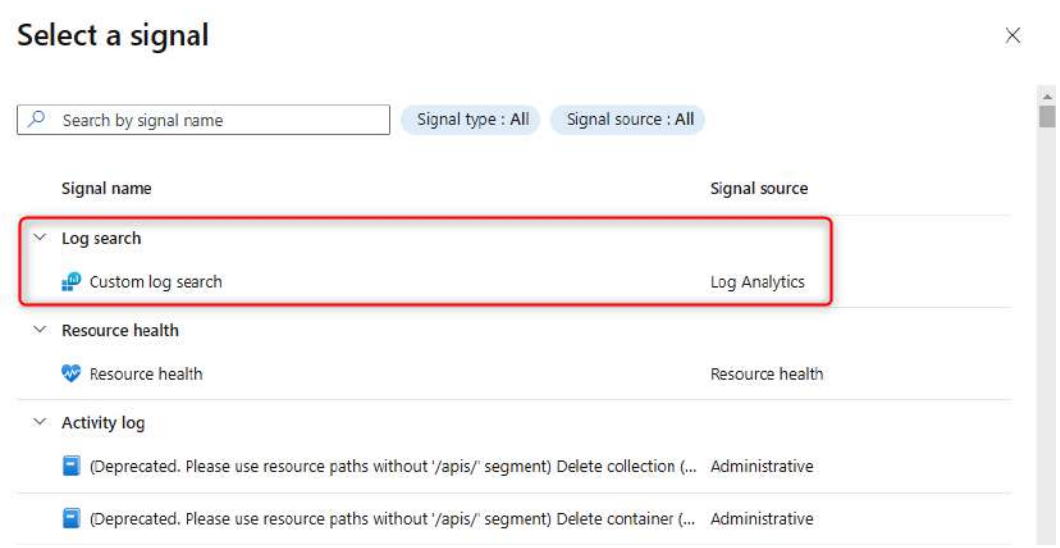

Figure 5.11 – Signals from Log Analytics

In our example, we select **Custom log search** signal name. After selecting the signal, the **Condition** tab opens the query dialog, measurement configuration, and other configurations, as shown in *Figure 5.12*.

It is important to note that the alert query will include records from Log Analytics created by all resources in the scope. However, it is also possible to query an **Azure Data Explorer** (**ADX**) or **Azure Resource Graph** (**ARG**). In this case, it is recommended to add a time range filter to the query. Note that to query data in ARG, you need to use the `arg("")` pattern followed by the ARG table name. For more information about querying ADX or ARG data from Azure Monitor, refer to [1] in the *Further reading* section.

For example, let's see how to alert when there are more than a specific number of virtual machines within the `rg-packt` resource group. First, you should use the following query to get the list of virtual machines within the `rg-packt` resource group:

```
arg("").Resources
| where type =~ 'Microsoft.Compute/virtualMachines'
| where resourceGroup == 'rg-packt'
| project _ResourceId=tolower(id), name, location
```

Paste the above KQL query into the search query dialog box, as shown in the following figure:

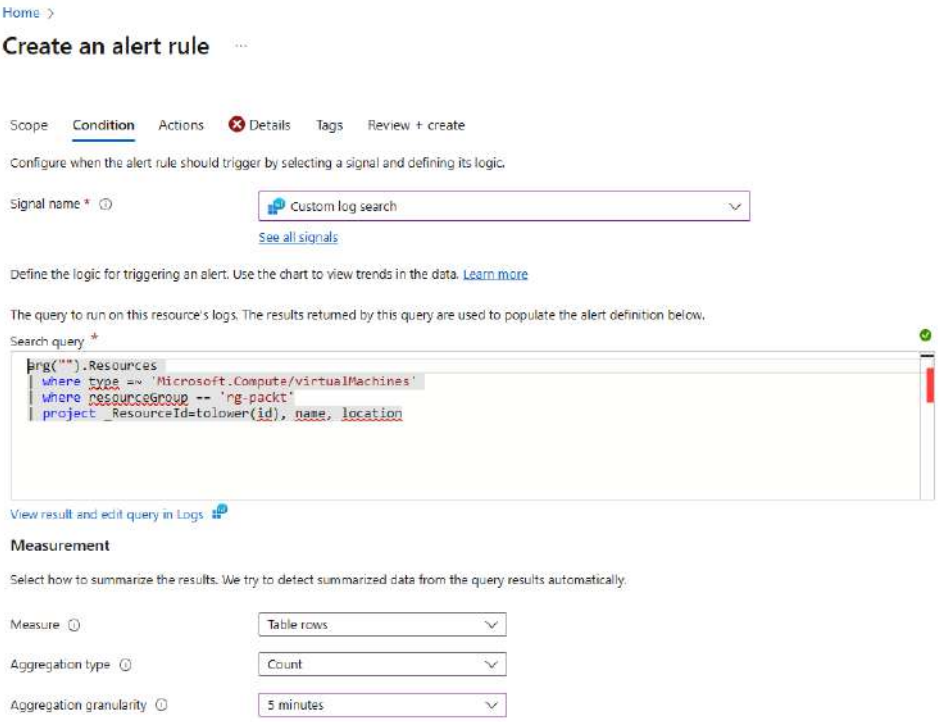

Figure 5.12 – Alert rule condition

Now we configure how to summarize the query results using the measurement section:

- **Measurement**: In this section, select values for the following fields:
  - **Measure**: Select **Table rows** if you want the number of rows returned by the query, or **Calculation of a numeric column** if you want a calculation based on any numeric column.
  - **Aggregation type**: **Count**, **Total**, **Average**, **Minimum**, or **Maximum**. In our example, we use **Count** to get the number of rows returned by the query.
  - **Aggregation granularity**: The range time in which data points are grouped by aggregation type.
- **Split by dimensions**: You can use dimensions to monitor a metric across multiple instances, with each instance monitored separately. In that case, alerts are triggered for each instance where the metric exceeds the set threshold. *Figure 5.13* shows how you can split with **location** as the **Dimension name** and **westeurope** and **northeurope** as **Dimension values**. Because of this selection, you will be able to receive alerts when there are more than four virtual machines within the `rg-packt` resource group in each location.

This is because the following two time series are being monitored for this alert rule:

- Table rows where `location='westeurope'` is greater than 4
- Table rows where `location='northeurope'` is greater than 4

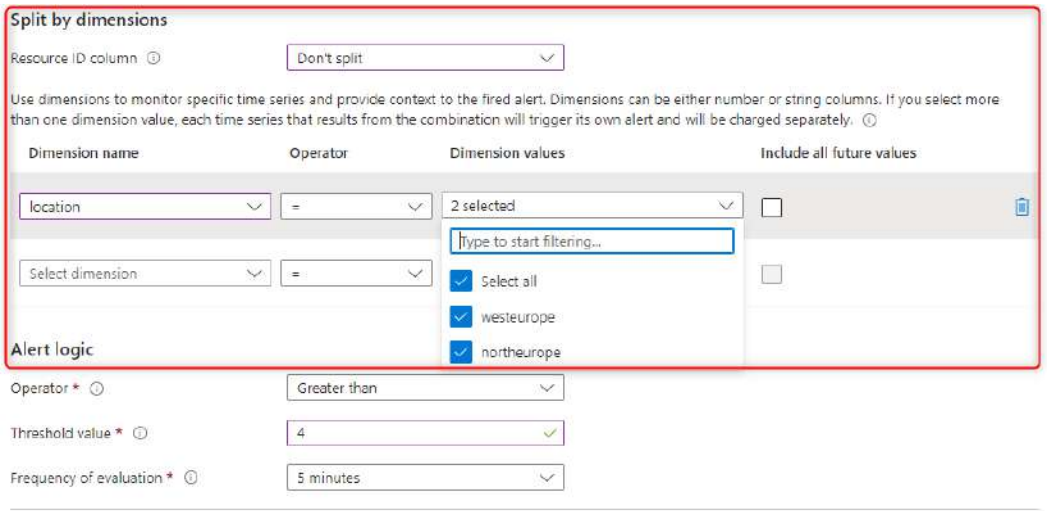

Figure 5.13 – Alert rule splitting by location dimension

However, if you want to apply a condition across multiple resources within a scope, such as a subscription, resource group, or workspace, you may choose not to use dimensions. In our example, we might not split by dimensions if we want to trigger an alert when more than eight machines in a resource group have been deployed, as shown in *Figure 5.14*.

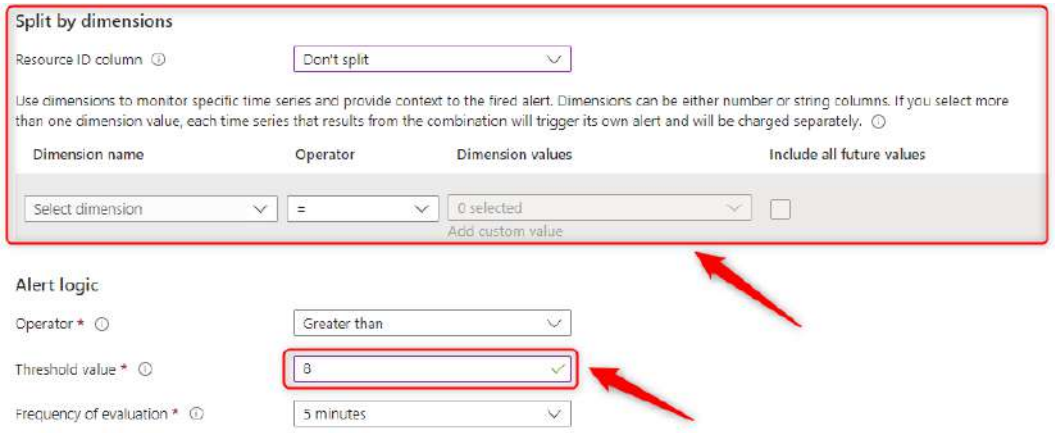

Figure 5.14 – Alert rule without splitting by dimensions

Another important thing to mention is that if your alert rule is set to a scope such as a subscription, resource group, or workspace, alerts are triggered on this scope. In our example, it means that alerts are triggered on the `rg-packt` resource group. Suppose that instead of counting the number of machines within a resource group, you create an alert for monitoring when the memory usage of a group of machines within that resource group is greater than 80. This alert will be triggered in the resource group where the machines are. If you want to receive separate alerts for each affected Azure resource, you should configure the alert with one of the following options:

- Split by **Resource ID column** as a dimension, as shown in *Figure 5.15*, allowing alerts to be triggered on the resource group with this column as a dimension.

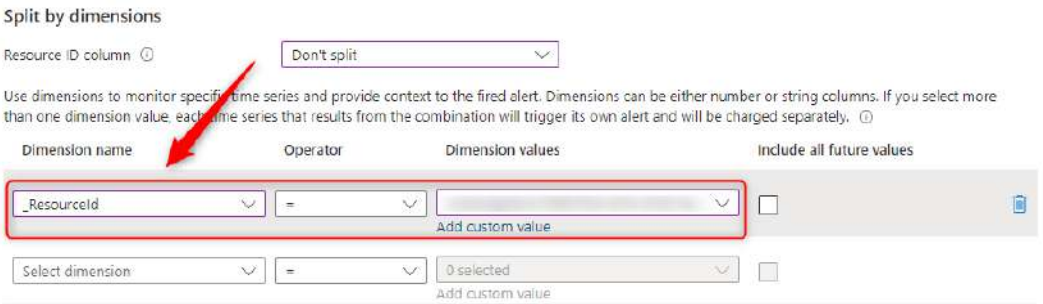

Figure 5.15 – Alert rule splitting by Resource ID column

- Set `_ResourceId` as a dimension in the alert, as shown in *Figure 5.16*. This approach focuses the alert on the specific resource returned by your query, in this case, a virtual machine, rather than the entire resource group.

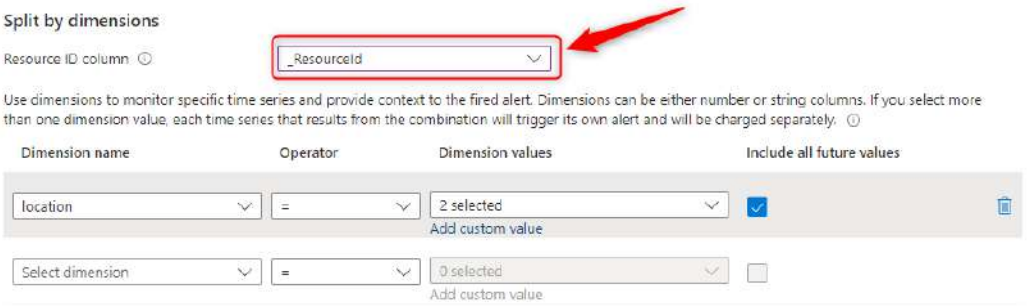

Figure 5.16 – Alert rule _ResourceId as a dimension in the alert

- **Alert logic**: The fields and possible values are as follows:
  - **Operator**: We can choose **Greater than**, **Greater than or equal to**, **Less than**, or **Less than or equal to**. In our example, we select **Greater than**.
  - **Threshold value**: This is a number value. In our case, we select different thresholds depending on the scenario you want to test, as shown in *Figure 5.13* and *Figure 5.14*.
  - **Frequency of evaluation**: Select a value from **1 minute** to **24 hours**.

Actions

As mentioned earlier, the *Automated responses to monitoring events* section provides details about how to create notifications, actions, and action groups.

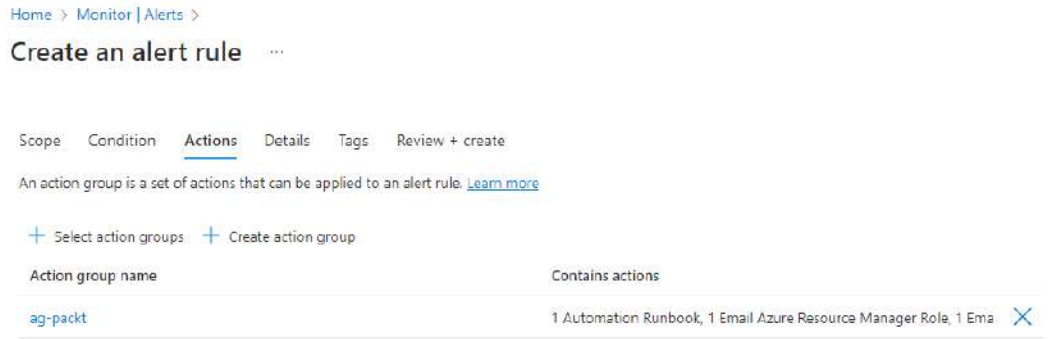

Figure 5.17 – Alert rule actions

Details

As shown in *Figure 5.18,* the fields and possible values are as follows:

- **Subscription**: Select a subscription to save the alert rule.
- **Resource group**: Select a resource group to save the alert rule. In our case, we select `rg-packt`.
- **Severity**: Select among **Critical**, **Error**, **Warning**, **Informational**, and **Verbose**. In our example, we select **Informational**.
- **Alert rule name**: Set the name of the alert rule, for example, `alert-vms-limit`.
- **Alert rule description**: Describe the alert rule, for example, `Total VMs in rg-packt is greater than 8`.
- **Identity**: If you need to query a specific scope, ADX, or ARG, you will need to use a managed identity and assign a role to the identity. If you do not use a managed identity, the alert rule permissions will be based on the permissions of the last principal identity who edited the rule.

The options for this field are **Default**, **System assigned managed identity**, and **User assigned managed identity**. In our example, we used **Default** permissions to query ARG.

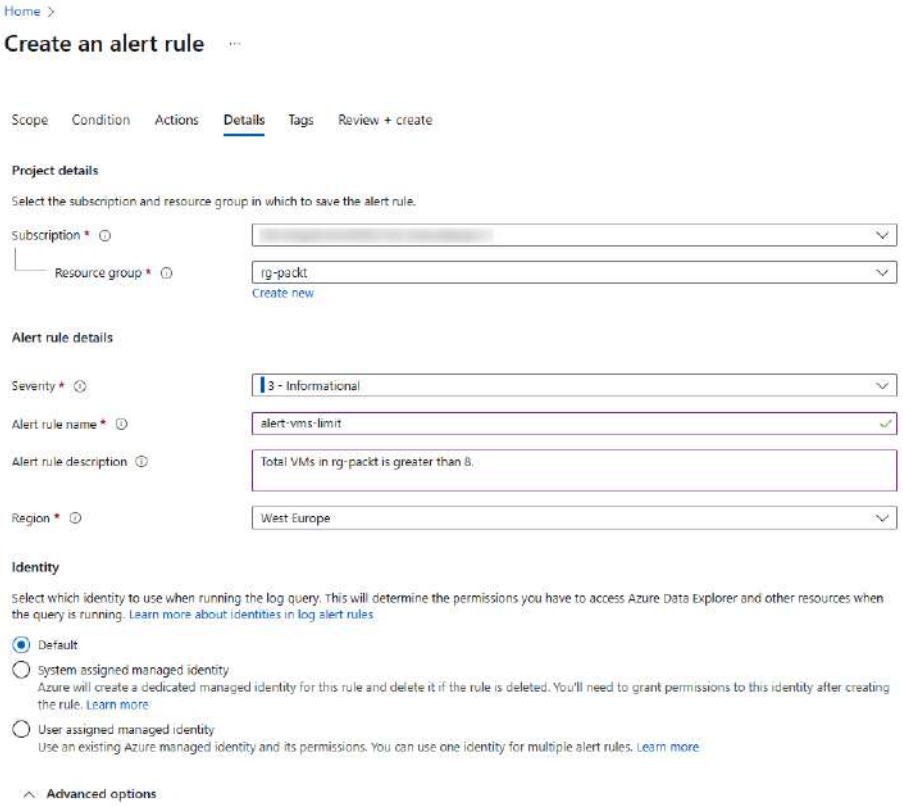

Figure 5.18 – Alert rule details

- **Advanced options**: As shown in *Figure 5.19,* the advanced options are as follows:
  - **Custom properties**: The custom properties are specified as key/value pairs and can be static text, or a dynamic value extracted from the alert payload using `${<path to schema field>}`. For example, the first custom property name could be `VMs limit` with a value of `8`, the second custom property name could be `Condition Start Time` with a value of `${data.alertContext.condition.windowStartTime}`, and the third custom property could be `Condition End Time` with a value of `${data.alertContext.condition.windowEndTime}`.
  - **Enable upon creation**: Use this if you want to enable the alert rule right after you have finished creating it.

- **Automatically resolve alerts**: Mark this if you want a stateful alert, that is, the alert is resolved when the condition is no longer met. Otherwise, the alert will be stateless and will be triggered each time the condition is met.
- **Mute actions**: If you do not enable **Automatically resolve alerts**, then you can configure a range of time to wait before alert actions are triggered again.

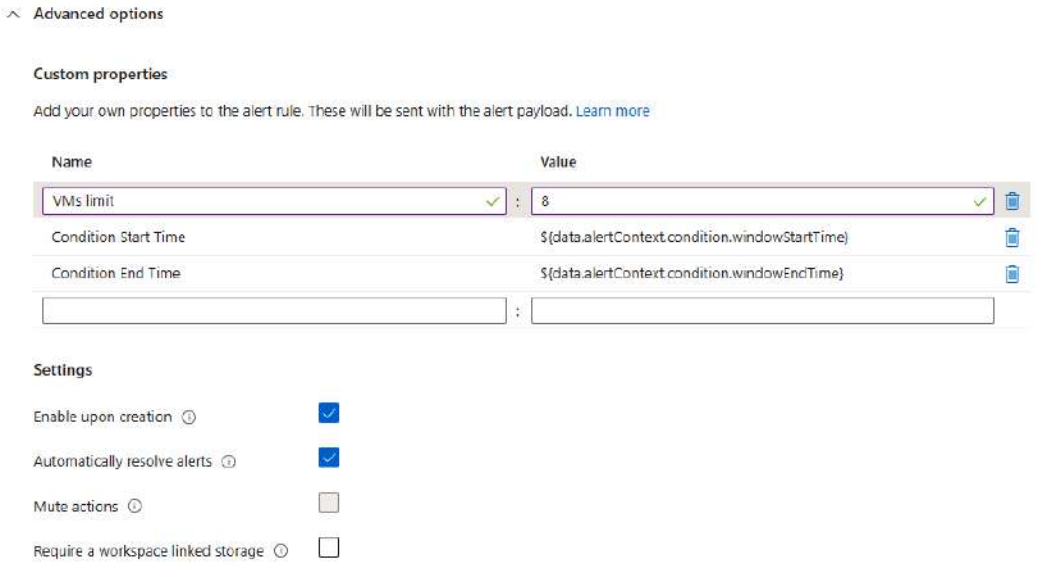

Figure 5.19 – Alert rule details advanced options

With this configuration completed, you have learned how to create a log search alert, and we can move on to show you how to create and configure the next type of alert, which is the activity log alert.

Activity log alert rules

The activity log alert monitors resource activities by checking for new activity log events that match defined conditions. It's useful for scenarios such as monitoring specific operations on resources in a resource group or subscription, such as VM deletions or role assignments. These alerts can be created for any activity log event category (**Administrative Activity log**, **Policy Activity log**, **Autoscaling Activity log**, **Security Activity log**, **Resource Health**, and **Service Health**) except alert events and can be managed as Azure resources. They are specific to the subscription where they are created.

To configure an activity alert rule, we first create a rule, as we did in the *Metric alert rules* section.

After selecting the **Alert rule** button, the alert wizard appears showing the **Scope**, **Condition**, **Actions**, and **Details** tabs to configure the alert. Let's see the options in each of them.

Scope

Select **All storage accounts** in a subscription as the **Scope**. In the following screenshot, you can see what the scope selection looks like.

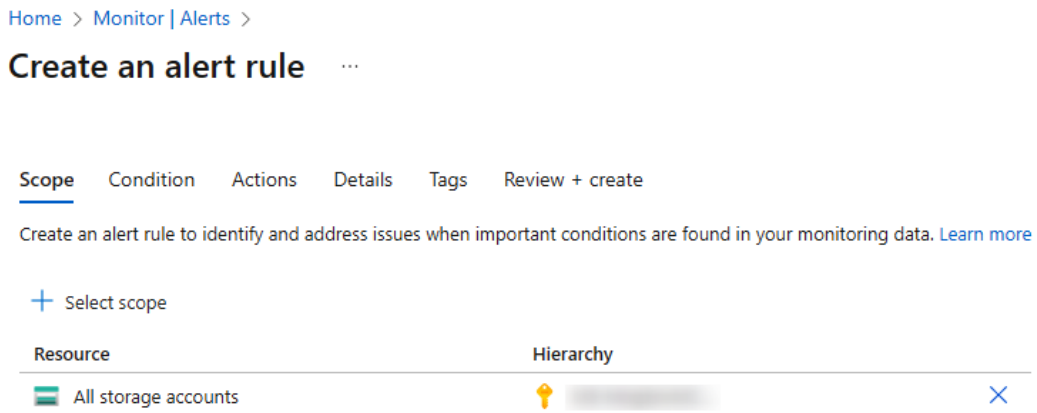

Figure 5.20 – Alert rule scope

Condition

The signal type for this type of alert rule could be **Log alert**, **Service Health**, or **Resource Health**. The signal source is the service that emits the signal. To see all these options, click on **See all signals**.

In the case of this type of alert rule, as shown in *Figure 5.21*, the **Signal source** can be any of the following, depending on the activity log category:

- **Administrative**: Events related to all create, update, delete, and action operations performed through ARM.
- **Policy**: Events related to the effect of Azure Policy.
- **Autoscaling**: Events related to the autoscale operation carried out by any autoscaling configuration in the subscription.
- **Security**: Security activity events generated by Defender for Cloud.
- **Resource health**: Resource health status events to help diagnose and get support for service problems affecting Azure resources. They report on the current and past health of resources, relying on signals from Azure services to assess health and provide actions to address issues.
- **Service health**: Subscription health status events to notify about outages, maintenance, and health issues that occurred in Azure related to used services and resources.

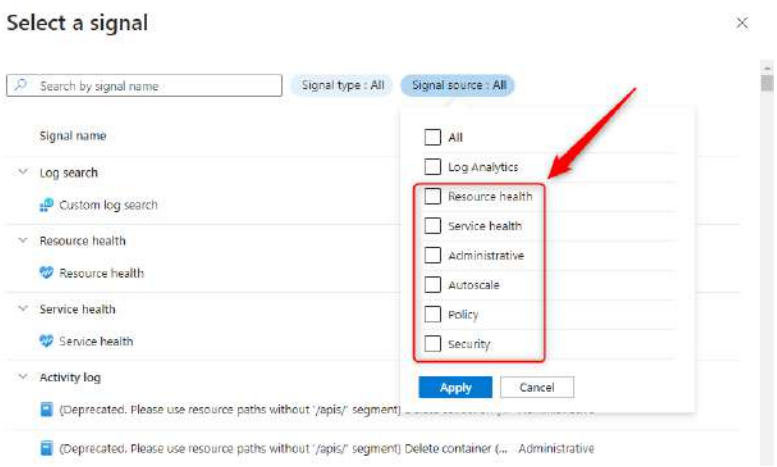

Figure 5.21 – Activity log categories displayed if the subscription scope is selected

Note that the list displayed depends on the type of resource selected on the **Scope** tab. In our case, **Service health** is not displayed, as shown in *Figure 5.22*, but if you change the scope to **Subscription**, it is displayed, as shown in *Figure 5.21*.

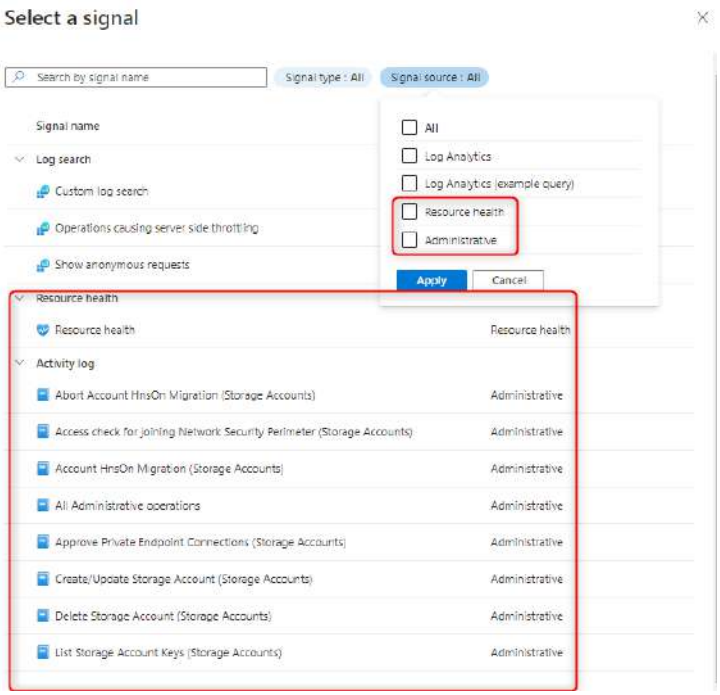

Figure 5.22 – Activity log categories displayed if the All storage accounts scope is selected

Depending on the **Signal type** you select, the **Condition** tab will appear in one form or another. There are the following options depending on the **Signal type** you select:

- **Activity log alert**: An **Alert logic** window appears with fields that allow you to fine-tune the alert conditions and determine when an alert should be triggered based on the behavior of the monitored metric. Through these parameters, you can customize the rule to fit the requirements of your application or environment in Azure. In our example, we select the signal that detects when a storage account is created or updated, specifically whenever the Activity Log contains an event with the **Administrative** category and the signal name **Create/Update Storage Account (Storage Accounts)**. After selecting the signal, the **Condition** tab opens the **Alert logic** window, as shown in *Figure 5.23*.

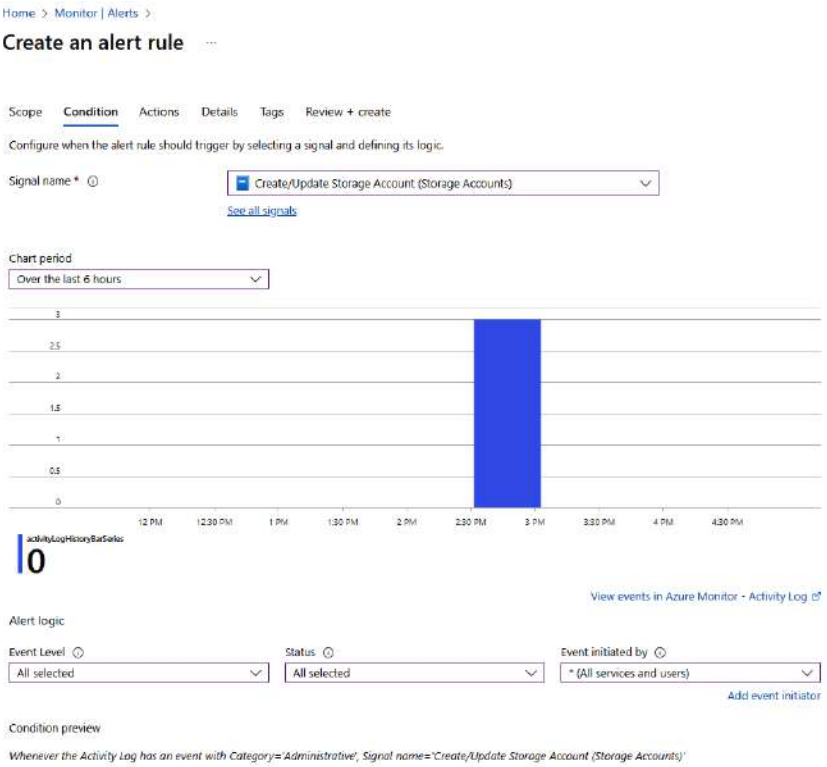

Figure 5.23 – Alert rule condition

As shown in *Figure 5.24*, the field in the **Alert logic** window and their possible values are the following:

- **Event Level**: **Critical**, **Error**, **Warning**, **Informational**, **Verbose**, and **All** are the possible levels of the events for this alert rule.
- **Status**: **Failed**, **Started**, and **Succeeded** are the possible statuses for the alert.

- **Event initiated by**: This selects the user or service principal that initiated the event.

Figure 5.24 – Alert rule logic for an activity log alert

- **Resource health**: As shown in *Figure 5.25*, the **Condition** tab shows you the following options:
  - **Event level**: **Active**, **In Progress**, **Resolved**, and **Updated** are the possible levels of the events for this alert rule.
  - **Current resource status**: **Available**, **Degraded**, and **Unavailable** are the possible values for the current resource status.
  - **Previous resource status**: **Available**, **Degraded**, **Unavailable**, and **Unknown** are the possible values for the previous resource status.
  - **Reason type**: **Platform Initiated**, **Unknown**, and **User Initiated** are the possible values for this type of event.

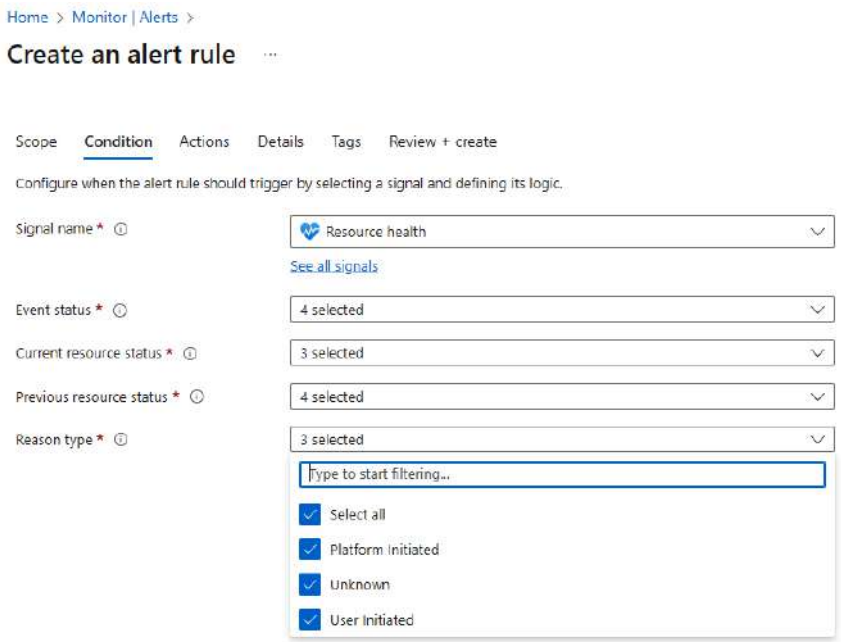

Figure 5.25 – Resource Health alert condition

- **Service health**: As shown in *Figure 5.26*, the **Condition** tab shows you the following options:
  - **Services**: Select the Azure services you are interested in to be alerted.
  - **Regions**: Select the Azure regions.
  - **Event types**: **Service issue**, **Planned maintenance**, **Health advisories**, and **Security advisories** are the possible values for this type of event.

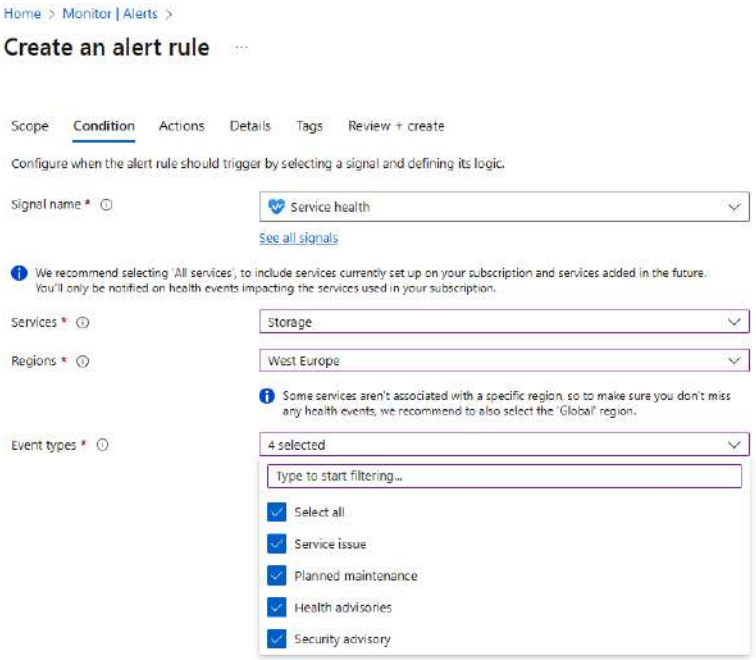

Figure 5.26 – Service health alert condition

Actions

As recommended earlier, the *Automated responses to monitoring events* section provides more details on how to create notifications, actions, and action groups.

Details

As shown in *Figure 5.27,* the fields and possible values are as follows:

- **Subscription**: Select a subscription to save the alert rule.
- **Resource group**: Select a resource group to save the alert rule, in our case, `rg-packt`.
- **Alert rule name**: We will choose the name of the alert rule, for example, `alert-sa-modified`.

- **Alert rule description**: We will describe the alert, for example, `Detect when a storage account has been created or updated`.
- **Advanced options**: some advanced options are as follows:
  - **Custom properties**: When action groups are added, there is the option to use custom properties for adding your own properties to the alert rule as we showed for metric and log search alerts.
  - **Enable alert rule upon creation**: Mark this if you want to enable the alert rule right after you have finished creating it.

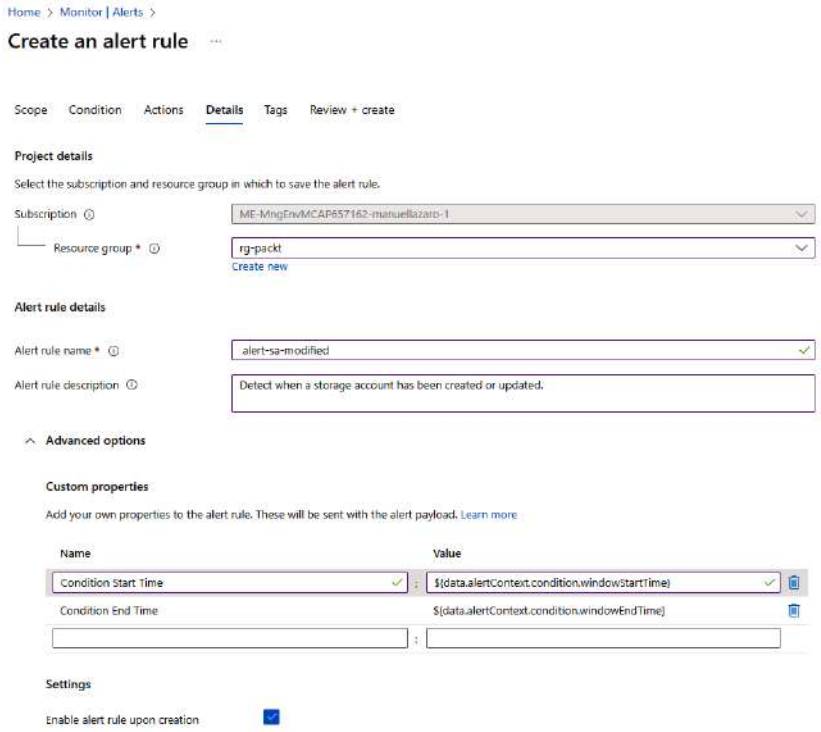

Figure 5.27 – Alert rule details advanced options

So far, you have learned how to create alerts in Azure Monitor. However, in the alert creation wizard, we indicated that you could configure actions for the alert that is triggered by the alert rule, and we are going to explain that in detail in the next section.

Automated responses to monitoring events

In this section, you will learn how to configure action groups, actions, notifications, and alert processing rules to respond automatically and at scale to monitoring events.

Action groups, actions, and notifications

In the previous section, we mentioned that an alert rule can notify or act in response to an alert through action groups. As you have seen, an action group is a collection of actions or notifications that are triggered when an alert is fired. It serves as a container that defines what should happen when an alert condition is met. Action groups can be reused across multiple alert rules, which makes it easy to manage responses to different types of alerts consistently. Each action group can contain multiple actions and can notify different teams or trigger different processes. An alert rule can have up to five associated action groups, which run concurrently. Here, we detail how to configure an action group, actions, and notifications:

1. In the Azure Monitor portal, select **Alerts** from the left-side menu. Then, select **Create** and choose **Action group**, as shown in the following screenshot:

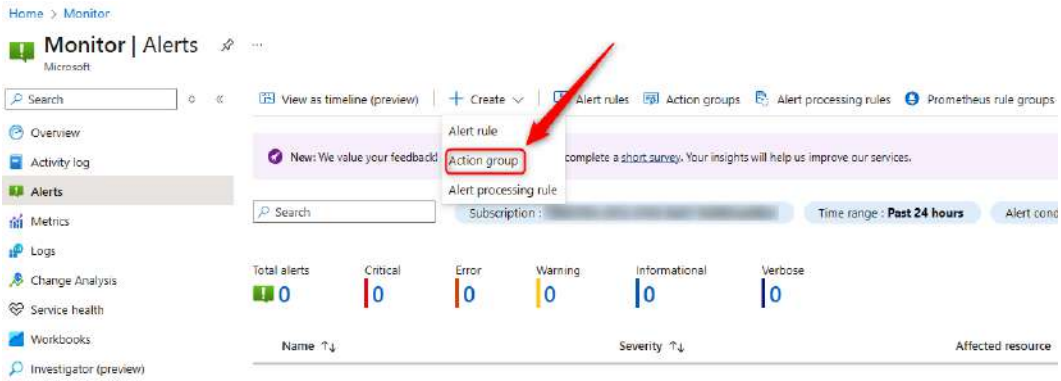

Figure 5.28 – Action group creation button

2. As shown in *Figure 5.29*, the options for the **Basics** tab of the action group wizard are the following:

   - **Subscription**: Select a subscription to manage the resource.
   - **Resource group**: Select a resource group to manage the resource, in our case, `rg-packt`.
   - **Region**: There are two options:
     - **Global**: To ensure regional resilience, the action group persists in at least two regions. The processing of action group actions occurs in any geographic region. In our example, we select this.
     - **Regional**: The action group persists in the selected region, and the processing of actions occurs in this region. Action groups in this model are zone redundant.

- **Action group name**: This is for choosing the name of the resource, for example, `ag-packt`.
- **Display name**: This is the friendly name of the resource, for example, `packtAGroup`.

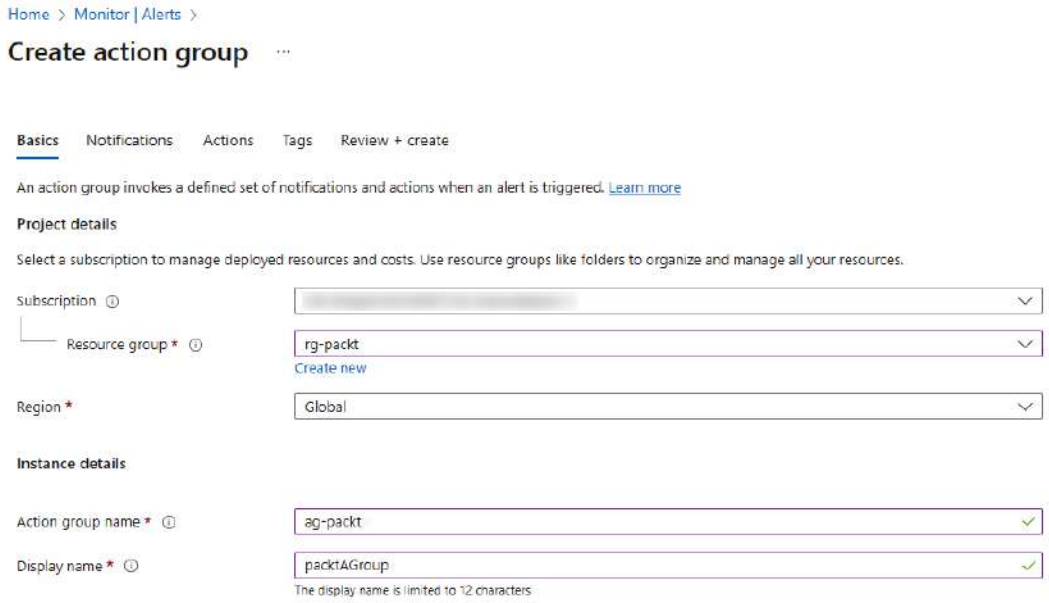

Figure 5.29 – Action group definition

3. Select the **Notifications** tab, we have the following types:

- **Email Azure Resource Manager Role**: Select the subscription role, and an email will be sent to the members with those assigned roles. Email delivery is only supported for users of Microsoft Entra ID. In our example, we called it `notify-owners` and selected the **Owner** role.

Figure 5.30 – Email ARM role notification

- **Email**: Email address where the notification is sent.
- **SMS**: Indicate the **Country code** and the **Phone number** for the SMS recipient.
- **Azure app Push notifications**: Indicate the email that recipients use to sign in to the mobile app.
- **Voice**: Indicate the **Country code** and the **Phone number** for the voice recipient.

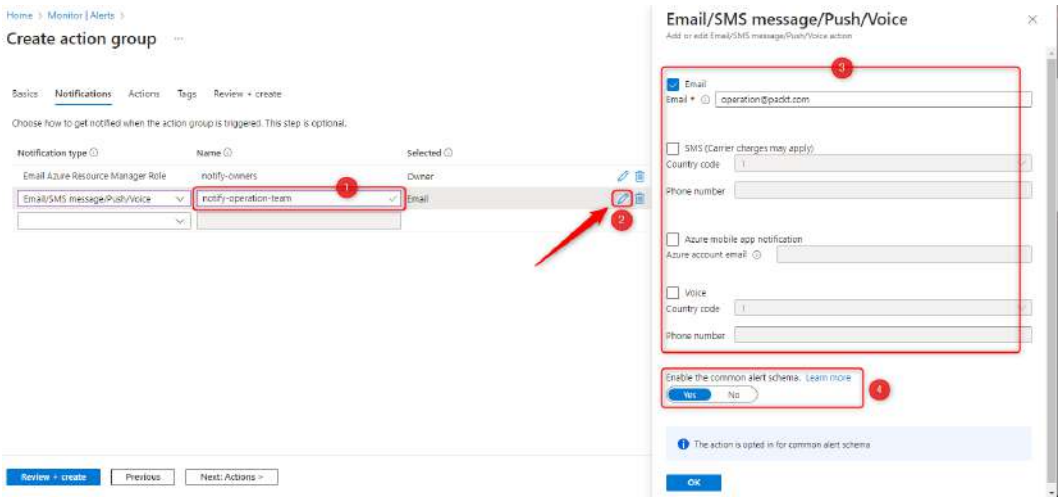

Figure 5.31 – Email/SMS/push/voice notification

4. In each notification type, there is an option to **Enable the common alert schema**, which provides a standardized schema for the consumption of all Azure Monitor alert notifications. If this is not enabled, each alert type has its own schema and notification template for emails, and these should be considered for integrations with webhooks, Azure Logic Apps, Azure Functions, and Azure Automation runbooks, among others. For more information about the **common alert schema**, refer to [2] in the *Further reading* section.

5. Select the **Actions** tab; there are the following types:

   - **Automation Runbook**: Select a built-in or user automation runbook. As shown in *Figure 5.32*, this type of action is particularly useful if, for example, you want to scale up an application running on a virtual machine due to intensive CPU usage during certain periods of time.

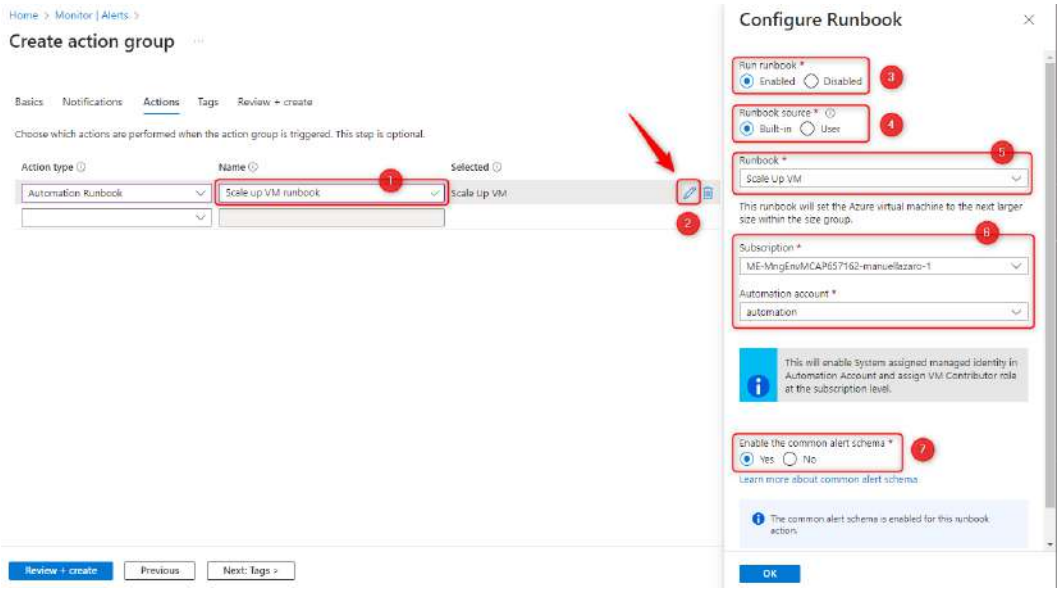

Figure 5.32 – Azure Runbook action

   - **Event Hub**: Indicate the event hub where you want to publish notifications. Then, you can subscribe to the alert notification stream from the event receiver. The following screenshot shows how to configure this action:

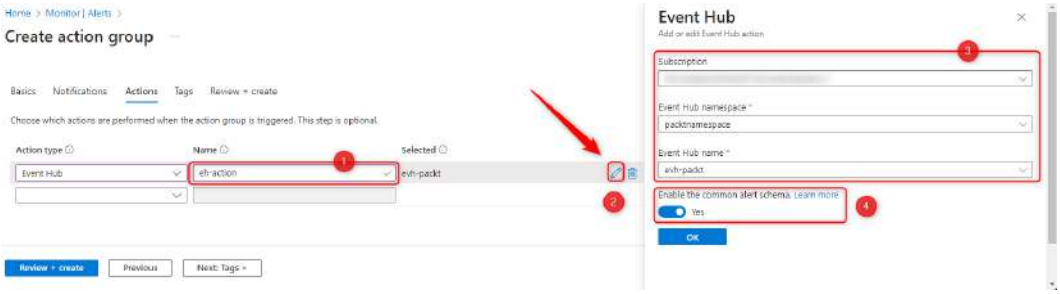

Figure 5.33 – Azure Event Hub action

- **Azure Function**: This action saves the function's HTTP trigger endpoint and access key in the action definition. The action calls this HTTP trigger endpoint in Azure Functions. The following screenshot shows how to configure this action:

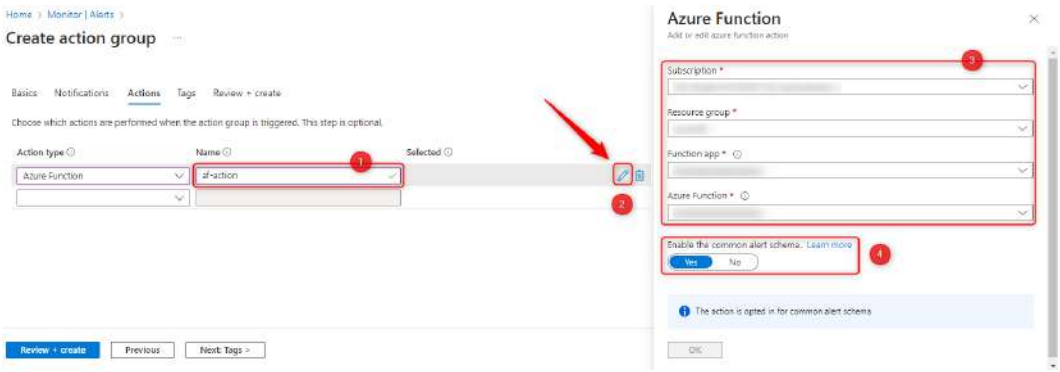

Figure 5.34 – Azure Function action

- **ITSM**: This action sends the event alert to a pre-created ITSM connection. For guidance on setting up an ITSM connection, refer to [3] in the *Further reading* section. The following screenshot shows how to configure this action:

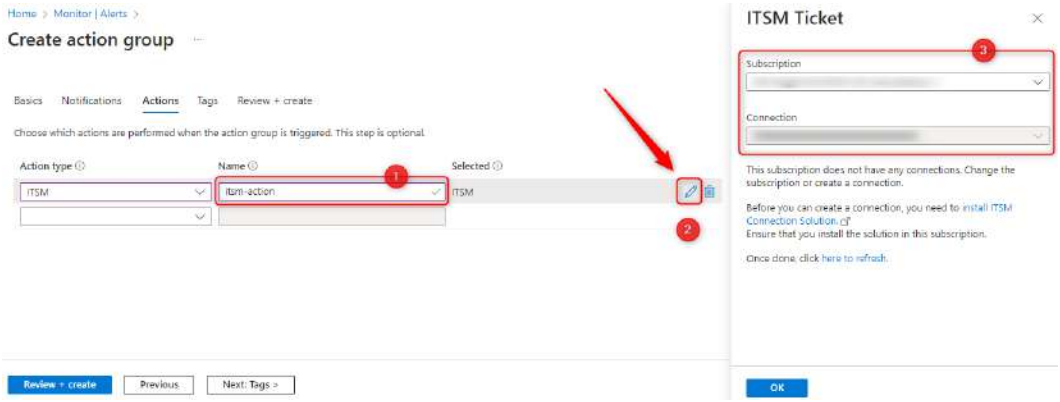

Figure 5.35 – ITSM action

- **Logic App**: You can use Azure Logic Apps to build and customize workflows for integration and to customize your alert notifications. The following screenshot shows how to configure this action:

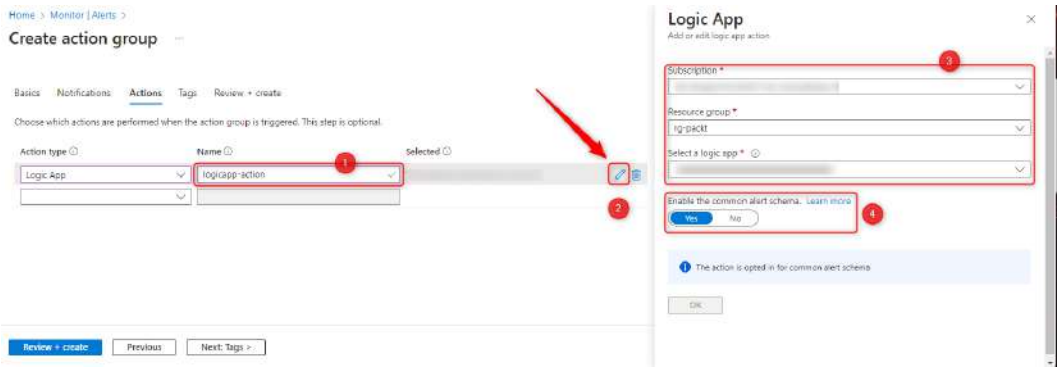

Figure 5.36 – Azure Logic Apps action

- **Secure Webhook**: For this action type, you use Microsoft Entra ID to secure the connection between the action group and the webhook endpoint. The following screenshot shows how to configure this action:

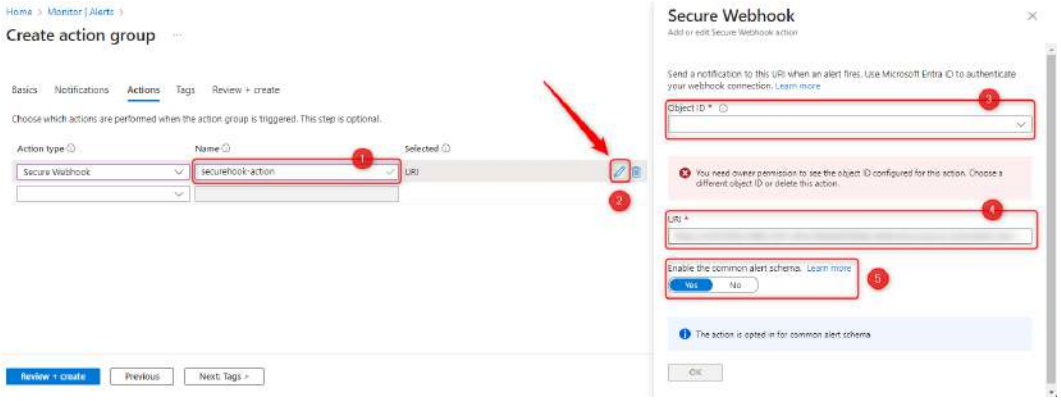

Figure 5.37 – Secure webhook action

- **Webhook**: As with **Secure Webhook**, you can use this type of action if you want to send alert events to a third party, such as Slack or Teams. The only thing to keep in mind is that sometimes the target endpoint expects a specific schema for the JSON payload, so it is necessary to use an Azure Logic App to adapt the schema to the target structure. The following screenshot shows how to configure this action:

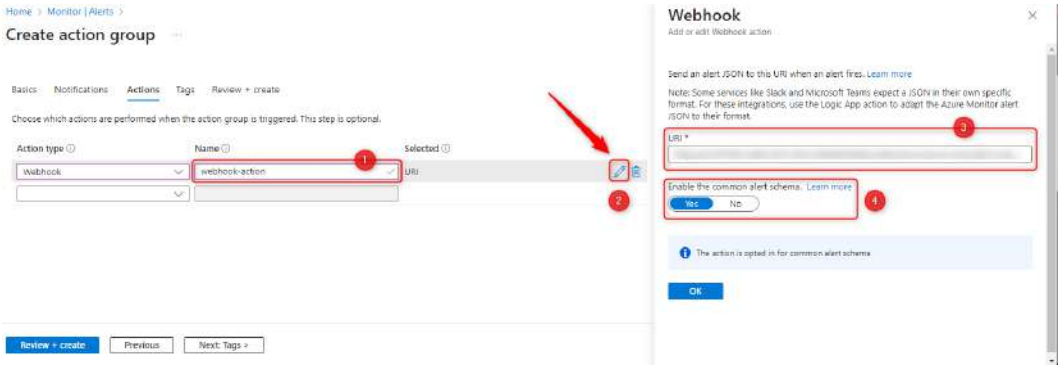

Figure 5.38 – Webhook action

As with each notification type, there is an **Enable the common alert schema** option on each action type, which provides a standardized schema for consuming all Azure Monitor alert actions.

Once we have learned how to configure the action group, actions, and notifications, we will review the alert processing rules to help optimize incident management by ensuring that alerts are handled in a way that aligns with the needs and priorities of your organization.

Alert processing rules

Azure alert processing rules are a feature within Azure Monitor that allow you to modify, filter, or suppress alerts before they trigger the configured actions in an action group. These rules give you more control over how alerts are handled, enabling you to tailor alert behavior based on specific criteria or conditions. For example, an alert processing rule that suppresses alerts can reduce noise by focusing on the most critical alerts or limit notifications and automated changes as consequences of actions in groups of actions during test scenarios. To configure an alert processing rule, we first select the **Alert processing rules** button, as we did in the *Action groups, actions, and notifications* section.

After selecting the **Alert processing rules** button, the alert wizard appears with the **Scope**, **Rule Settings**, **Scheduling**, and **Details** tabs to configure the alert. Let's see the options on each of them.

Scope

It can be a list of several specific Azure resources or resource groups, or an entire subscription. The alert processing rule applies only to alerts triggered by resources within its defined scope. You cannot create an alert processing rule for a resource that belongs to a different subscription. For example, if

we select the resource group **rg-packt**, the alert processing rule will apply to alerts that are triggered on the resource group. In the following screenshot, you can see what the scope selection looks like:

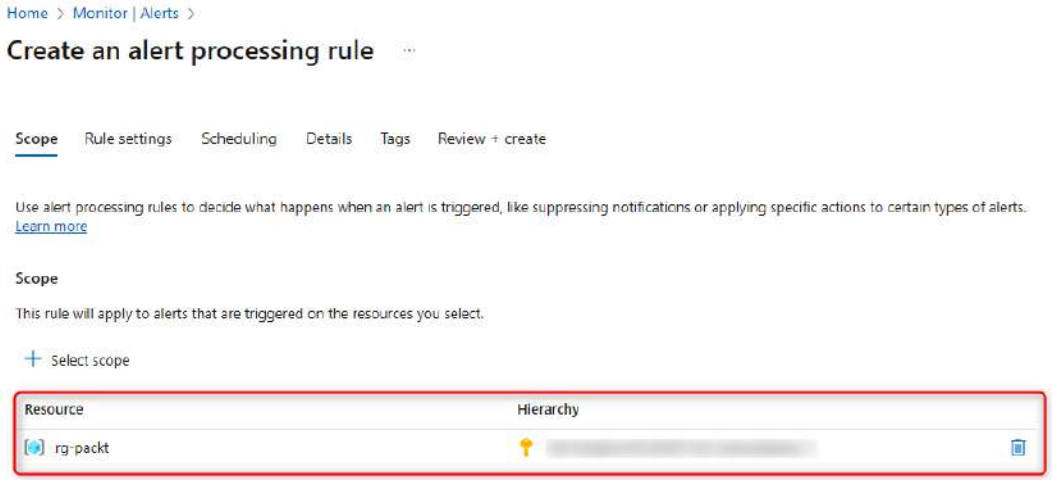

Figure 5.39 – Alert processing rule scope

You can also define filters to specify the subset of alerts that are affected within the scope. The available filters are shown in the following figure. In our example, we will not select any filters so all alerts that are triggered on the resource group **rg-packt** will be affected by the alert processing rule.

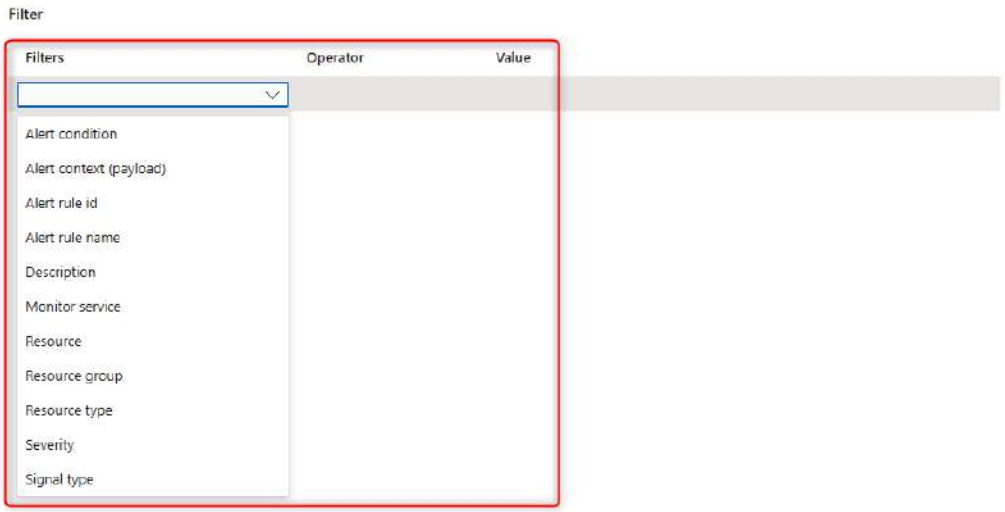

Figure 5.40 – Alert processing rule filter tab

Rule settings

As shown in *Figure 5.41*, you choose between the following options:

- **Suppress notifications**: This option can temporarily suppress alerts during planned maintenance or other known downtimes. This prevents unnecessary notifications and actions during periods when alerts are expected but not critical.
- **Apply action group**: This option allows you to select an existing action group by using the **Select action groups** option. In this way, you can set up actions and notifications for all alerts that are triggered on the **rg-packt** resource group without having to configure each action group in each alert associated with the resource group. The following screenshot shows these options and the selection of **ag-packt** as the action group.

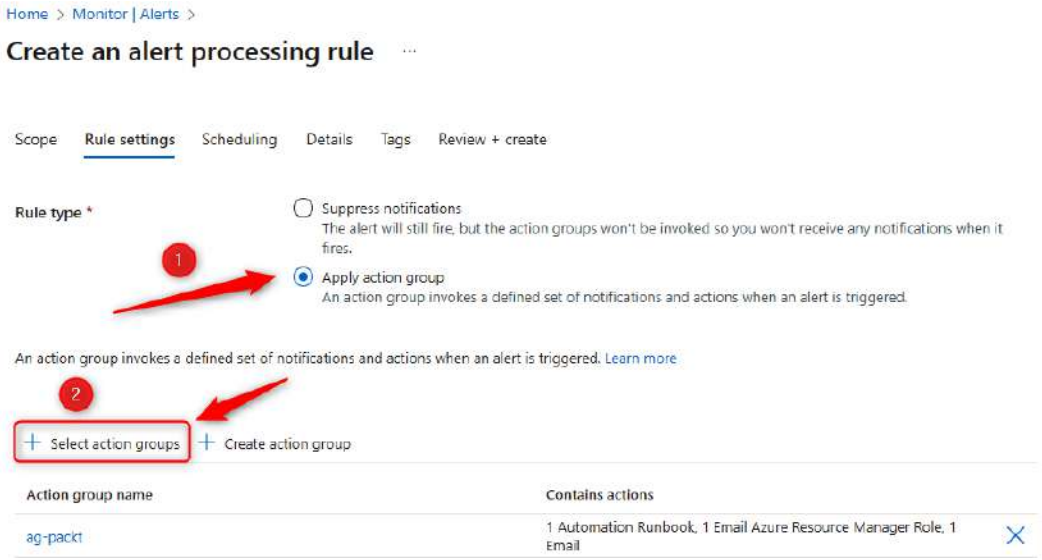

Figure 5.41 – Suppress notifications and Apply action group options

In our example, we select **Suppress notifications** to suppress all the action groups assigned to the alerts that are triggered on the resource group **rg-packt**.

Scheduling

By default, the alert processing rule operates continuously unless disabled. However, as you can see in the following screenshot, you can configure it to run **Always**, **At a specific time**, or set up a **Recurring** schedule.

An example of a schedule for a one-time, all-day planned maintenance in a specific time zone is shown in the following screenshot:

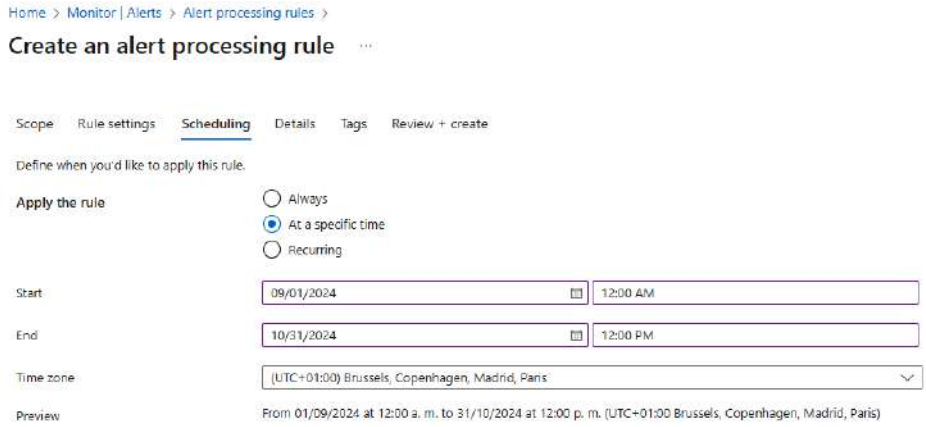

Figure 5.42 – Alert processing rule at a specific time

However, if we are interested in every Monday to Sunday throughout the day from `01/09/2024` to `10/31/2024`, the action groups assigned to the alerts that are triggered on the **rg-packt** resource group are not executed so we must use the **Recurring** option, as shown in *Figure 5.43*:

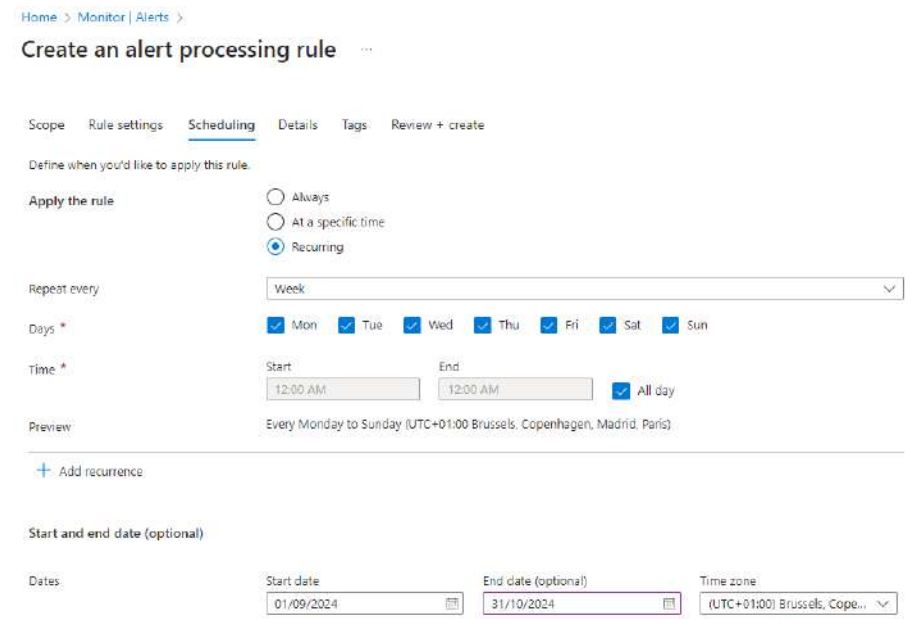

Figure 5.43 – Alert processing rule recurring option

Note that you can click on **Add recurrence** to set up several recurrences to cover complex scenarios such as business hours and non-business hours.

Details

As shown in *Figure 5.44*, the fields and possible values are as follows:

- **Resource group**: Select a resource group to save the alert processing rule, in our case, `rg-packt`.
- **Rule name**: We will choose the name of the alert processing rule, for example, `Suppress alerts on rg-packt`.
- **Description**: We will describe the alert processing rule, for example, `Suppress alerts on the resource group rg-packt during testing period`.
- **Enable alert rule upon creation**: Mark this if you want to enable the alert processing rule right after you have finished creating it.

Figure 5.44 – Alert processing rule Details tab

After creating the alert processing rule, go to the Azure Monitor portal and select **Alert Processing rules**.

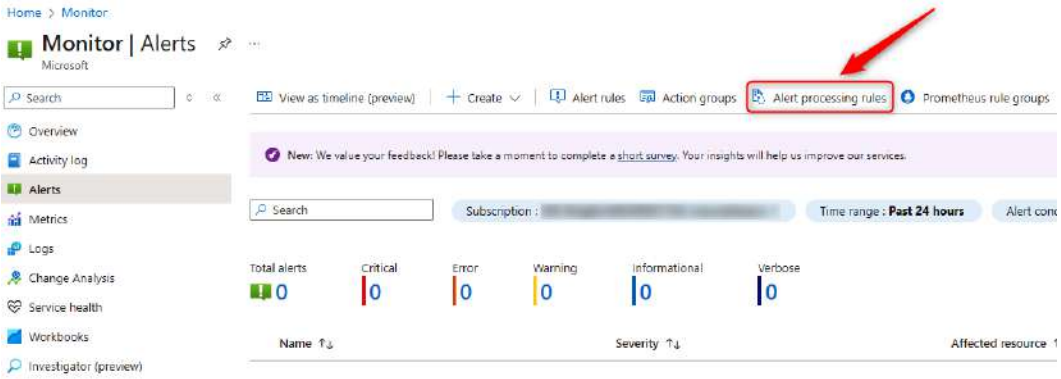

Figure 5.45 – Alert processing rules button

The alert processing rules list appears, and you can verify that the alert processing rule was created successfully and has been enabled, as shown in *Figure 5.46*:

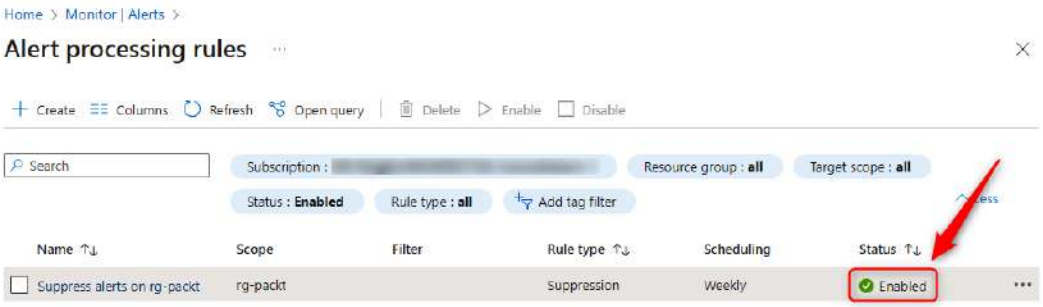

Figure 5.46 – Alert processing rules list

Once we have learned how to create alerts and automated responses, we will move on to describe the best practices in threshold definition.

Threshold definition for effective alerting

As we have seen throughout *Chapters 2, 3, and 4*, Azure Monitor can help to ensure the reliability and performance of applications and services hosted in a multi-cloud ecosystem. A fundamental aspect of Azure Monitor is configuring alert rules, which involves setting thresholds to determine when an alert should be triggered. Achieving precision in defining these thresholds is crucial to ensure that alerts are relevant and actionable by other systems. This section aims to provide guidance on the considerations for establishing effective thresholds for alerts in Azure Monitor when deploying applications in Azure.

Why are thresholds so important?

The importance of effectively defining alert rule thresholds is critical due to the impact it can have on the responsiveness of the monitoring system to promptly address application performance issues. Setting thresholds too low could generate a high volume of false positives, inundating the organization's operations team with unnecessary alerts. Conversely, establishing thresholds that are too high could result in missed error notifications or delayed detection, adversely affecting the performance and availability of applications with potential impacts on business outcomes. Therefore, an effective establishment of thresholds in alert rules ensures a balance between sensitivity and specificity necessary for monitoring application performance.

Factors influencing threshold definitions

When determining the appropriate threshold for a metric, several fundamental factors come into play. These factors are as follows:

- **The nature of the monitored metric**: Understanding the characteristics of the monitored metric is crucial because different metrics may exhibit distinct behavior patterns, such as periodic peaks during peak usage or gradual increases over time. These aspects need to be taken into account when setting thresholds to alert for abnormal behavior and prevent false alarms during expected variations.
- **Characteristics of the application**: Each application or service has specific behavior and performance requirements. Properly defining thresholds involves collaborating with application owners and related groups to gather insights into normal operational conditions and identify thresholds aligned with **service-level objectives** (**SLOs**) and **key performance indicators** (**KPIs**).
- **The dynamics of the environment**: The application is deployed in a cloud or multi-cloud environment, both of which have dynamic natures that demand adaptability when defining thresholds. In this context, setting thresholds that accommodate the variability of these environments involves considering factors such as scalability, resource allocation, and fluctuations in usage demand.

Best practices in threshold definition

Gaining expertise in defining effective thresholds means adhering to best practices tailored to your Azure Monitor environment. Here are some guidelines:

- **Baseline analysis**: Conducting a baseline analysis to establish a starting point for the normal behavior of metrics is essential. Analyzing historical data to identify patterns, trends, and anomalies forms the foundation for defining thresholds that deviate from the expected norms.
- **Collaborative approach**: A crucial aspect of success in threshold definition is adopting a collaborative approach with cross-functional teams, including developers, operations, and business stakeholders connected to the application or system. They possess a comprehensive

understanding of application behavior. Defining thresholds based on the input from each knowledge area involved contributes to establishing thresholds aligned with overall business objectives.

- **Iterative refinement**: One common mistake when defining the threshold for an alert rule is treating it as a singular task rather than an iterative process. However, as the application evolves, so do its performance characteristics. Therefore, regularly reviewing and adjusting thresholds based on changing requirements and dynamics is a key aspect of maintaining an effective alert system.
- **Automation**: Automation is very important for dynamically adjusting thresholds based on real-time data and changing conditions. Azure Monitor and other Azure services provide multiple automation tools for automatically adapting thresholds in response to workload fluctuations, ensuring a proactive and responsive alert mechanism.

Once we have learned how to define thresholds for effective alerts, we will move on to see how this together with the creation of alerts and automatic response to events can help us establish an incident response plan.

Establishing an incident response plan

One of the most important aspects of a multi-cloud or cloud ecosystem, where there is a dependency on multiple services and processes, is an incident response in order to maintain business continuity and mitigate interruptions in application service. In an environment such as Azure, we have seen that Azure Monitor can be used as a tool to monitor the status and performance of applications but also to respond at scale to events generated by applications and services. In this section, we will explain the importance of establishing a comprehensive incident response plan for an Azure environment, using the capabilities of Azure Monitor to address and resolve incidents efficiently at scale.

Identifying the critical resources and services in the Azure Environment

Before crafting an incident response plan, it's essential to gain a comprehensive understanding of the Azure environment. This includes the following:

- **Identification of critical Azure resources**: Clearly determine which Azure resources are critical for the operation of applications and services. These resources are essential components that, if compromised, could significantly impact your operations. Start by evaluating the role each resource plays in your infrastructure, such as virtual machines, databases, storage accounts, and network components. Consider its contribution to your organization's business processes, customer-facing applications, or internal systems. By clearly determining which Azure resources are critical, you can prioritize monitoring with Azure Monitor, apply stronger security measures, and allocate resources more effectively to maintain business continuity and minimize downtime.

- **Application dependency mapping**: Understand the interdependencies between different Azure resources and how they contribute to the overall application architecture. This process provides a holistic view of how different components, such as network components, databases, and web servers, interact with each other to provide services to your applications. By mapping these dependencies, you can identify potential single points of failure, optimize performance, and plan for scaling or redundancy. For this mapping, you can rely on the techniques provided by Azure Monitor and Application Insights that we saw in *Chapters 3, 4, and 7*. Knowledge of dependencies is crucial for solving problems, as it helps to quickly identify the root cause of the problems. Additionally, understanding these dependencies leads to better impact analysis when making changes to the infrastructure, ensuring that updates do not inadvertently disrupt other parts of the application.
- **Definition of SLOs**: Define SLOs to establish expectations for availability, performance, and reliability. SLOs are measurable targets that define the acceptable level of service quality, such as uptime percentage, response time, and error rate. These objectives are generally driven by business requirements and user expectations. Establishing SLOs provides a benchmark to measure the performance of your Azure resources, and you can use them not only as a framework for capacity planning and monitoring but also for incident management. We recommend you regularly review and adjust SLOs to ensure they stay aligned with changing business goals and user needs, leading to enhanced service quality and greater customer satisfaction.

Once we have identified the critical resources and services in your Azure environment, we will explain how Azure Monitor capabilities can help in incident detection.

Leveraging Azure Monitor for incident detection

As we have seen in *Chapters 2, 3, and 4*, Azure Monitor is a powerful service for monitoring and analyzing Azure resources. One of the capabilities of Azure Monitor that we have seen is that it offers a set of tools and capabilities that enable proactive incident detection. To ensure timely detection of incidents, we should consider the following:

- Configure Azure Monitor alerts for critical resources and services based on predefined thresholds, as we showed in the *Configuring proactive alerts in Azure Monitor* and *Threshold definition for effective alerting* sections. Consider the configuration of alerts for metrics such as CPU usage, memory consumption, network traffic, and other parameters that indicate abnormal behavior or performance issues.
- Establish actions and notifications in response to the triggering of these alerts to proactively react, in real-time and at scale, to potential incidents. You can look at the techniques you have learned in the *Automated responses to monitoring events* section.

Now, we can move on to explain how to organize the procedures for incident response.

Organizing incident response actions

The incident response should follow well-defined procedures and have clear roles and responsibilities. Key steps include the following:

1. **Incident categorization**: Incidents should be classified according to their severity, impact, and urgency to prioritize response efforts effectively. The definition and configuration of alerts, as well as notifications and actions, must be aligned with the categorization of potential incidents.
2. **Forming an incident response team**: Once critical resources and services are identified, it is important to establish teams responsible for managing and resolving incidents. These teams should include members from various departments, such as IT, security, and communications. Each team member should have clearly defined roles and responsibilities to prevent confusion during an incident. It is also important that these teams have participated in the definition and configuration of alerts, as well as notifications and actions, so they are aligned during the incident process.
3. **Establishment of escalation paths**: Establish clear escalation paths to ensure that incidents are routed to the appropriate stakeholders in a timely manner. Additionally, the execution of notifications and actions from Azure Monitor must be aligned with the escalation paths defined for each type of incident.
4. **Train and test incident response teams**: Members of the incident response team should receive regular training on new technologies, threats, procedures, notifications, actions, and how they apply to business applications. Simulation exercises can help identify gaps in the plan and improve response times.
5. **Communication and collaboration**: Establish continuous communication with applications stakeholders and incident response team members and collaborate closely with internal teams and external partners throughout the incident response process.

Once you have identified critical resources and services in your Azure environment and organized incident response, you can proceed to execute an incident response plan by following the steps in the next section.

Executing incident response actions

The key actions to take when an incident occurs are as follows:

1. **Incident classification**: This step involves assessing the nature and scope of the incident to determine its impact on the applications, considering the incident categorization previously conducted.
2. **Containment and mitigation**: Following the defined incident plan and the expected behavior from Azure Monitor alerts, notifications, and actions, complementary measures should be taken to contain the incident and mitigate its impact on Azure resources and services.

3. **Root cause analysis**: A detailed investigation of the incident should be conducted to identify its root cause.
4. **Communication and collaboration**: Maintain constant communication during the incident following the defined communication and collaboration plan.

Once the incident response plan has been executed, it is important to carry out a process of improvement and lessons learned, as discussed in the next section.

Enhancing the incident response plan through continuous improvement and learning

Continuous improvement is important for refining the incident response plan and enhancing incident response capabilities in real time. Key practices are as follows:

- **Post-incident review**: Conducting post-incident reviews to assess the effectiveness of response actions and notifications and identify areas for improvement
- **Documentation and knowledge sharing**: Documenting lessons learned from incidents and sharing best practices and knowledge with the incident response team
- **Training and drills**: Conducting regular training sessions and drills to ensure that the incident response team is well-prepared to handle various incidents effectively

Throughout this section, you have learned how to create a comprehensive incident response plan for an Azure environment. You have also learned how, by leveraging Azure Monitor alerts and automation response capabilities, you can efficiently address and resolve incidents at scale, ensuring business continuity and minimizing disruptions to your application services.

Summary

In this chapter, we provide detailed guidance on managing incidents and monitoring events with Azure Monitor, including configuring alerts based on telemetry issued by Azure resources and evaluating specific conditions to trigger notifications and automated actions. We also examine how to configure the techniques offered by Azure Monitor for automated, at-scale response to monitoring events.

We outlined the steps to define effective alert thresholds to prevent false positives and ensure alerts are relevant, considering factors such as the nature of the monitored metric and application characteristics.

Finally, we detailed the steps to establish an incident response plan from identifying critical resources and using Azure Monitor for incident detection, to organizing the response plan and key actions for the successful execution of the plan.

This chapter has prepared you to confidently manage monitoring events, helping to ensure your Azure environment runs smoothly. In the next chapter, we will explore the visual aspects of monitoring in Azure, where you will learn how to create impactful dashboards and reports that provide insights into your logs and metrics.

Further reading

Here, you can find the links to expand your knowledge about the specific concepts that were not covered in this book but were mentioned in this chapter:

- [1] Correlate data in Azure Data Explorer and Azure Resource Graph with data in a Log Analytics workspace: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/azure-monitor-data-explorer-proxy#query-data-in-azure-resource-graph-by-using-arg-preview`.
- [2] Azure Common Alert Schema: `https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-common-schema`.
- [3] Connect Azure to ITSM tools: `https://learn.microsoft.com/en-us/azure/azure-monitor/alerts/itsmc-overview#itsm-integration-workflow`.

6
Visualizing Your Logs and Metrics

In the previous chapters, we saw how data is continuously generated from a multitude of sources such as applications, services, and infrastructure. This data, often in the form of logs and metrics, holds the key to understanding the health, performance, and security of your systems. However, raw data alone can be overwhelming and difficult to interpret. The true value of this data emerges when it is transformed into meaningful insights through effective visualization.

This chapter introduces the visual side of monitoring with Azure. We'll explore how Azure's robust set of tools and services, such as Azure Workbooks, Azure Dashboards, Azure Managed Grafana, and Microsoft Power BI on Azure, empower you to create user-friendly and actionable visual representations of your data.

Upon completing this chapter, you'll get a deep understanding of how to create impactful dashboards and reports that provide intuitive insights into your logs and metrics. You'll know the power of visualization in communicating complex monitoring data to stakeholders, facilitating informed decision-making. You'll also develop skills in customizing and optimizing visualizations for various monitoring scenarios, enhancing your ability to convey the health and performance of your cloud resources effectively.

In this chapter, we will cover the aforementioned in the following sections:

- Exploring Azure visualization tools
- Choosing the right visualization tool
- Building a custom monitoring workbook – a hands-on example

Technical requirements

To follow along with all the examples in this chapter, you will need an Azure account with an active subscription, access to Azure Monitor, familiarity with the Azure portal, and basic knowledge of log management and query languages.

To follow along with all the examples in the *Building a custom monitoring workbook – a hands-on example* section, we recommend downloading the associated files available in the GitHub repository of the book. You will find the relevant files in the `chapter06` folder (`https://github.com/PacktPublishing/Cloud-Observability-with-Azure-Monitor/tree/main/chapter06`).

Exploring Azure visualization tools

This section explores the built-in tools in Azure to visualize collected data. Utilizing visual tools such as charts and graphs enables you to go deeper into your monitoring data, uncover issues, and identify patterns. Additionally, you can create custom visualizations tailored to the specific needs of different users within your organization.

Azure Monitor Insights

Azure provides insights into certain services through visualizations called Insights, which are built on the analysis of a subset of available telemetry collected from these services. In some cases, the collection of this telemetry may require agents, as we will see in *Chapter 3*. It is designed to use prebuilt Azure Workbooks to provide deep insights into the performance, health, and operation of your applications and infrastructure. It consolidates data from various sources to offer a holistic view, facilitating proactive management and optimization of your Azure environment. These Insights are classified as follows:

- **Compute**: Azure VM Insights and Azure Container Insights
- **Networking**: Azure Network Insights
- **Storage**: Azure Storage Insights and Azure Backup Insights
- **Databases**: Azure Cosmos DB Insights and Azure Monitor for Azure Cache Redis
- **Analytics**: Azure Data Explorer Analytics and Azure Monitor Log Analytics workspaces
- **Security**: Azure Key Vault Insights
- **Monitor**: Azure Monitor Application Insights, Azure activity log insights, and Azure Monitor for Resource Groups

- **Integration**: Azure Service Bus Insights and Azure IoT Edge
- **Workloads**: Azure SQL Insights and Azure Monitor for SAP solutions
- **Other services**: Azure Virtual Desktop Insights, Azure Stack HCI Insights, and Windows Update for Business reports insights

You can access the Insights view from the Azure Monitor left menu inside the **Insights Hub** option, as shown in the following screenshot. A list of all the pre-defined workbooks included with Azure Monitor will appear. Currently, a total of 18 workbooks are available.

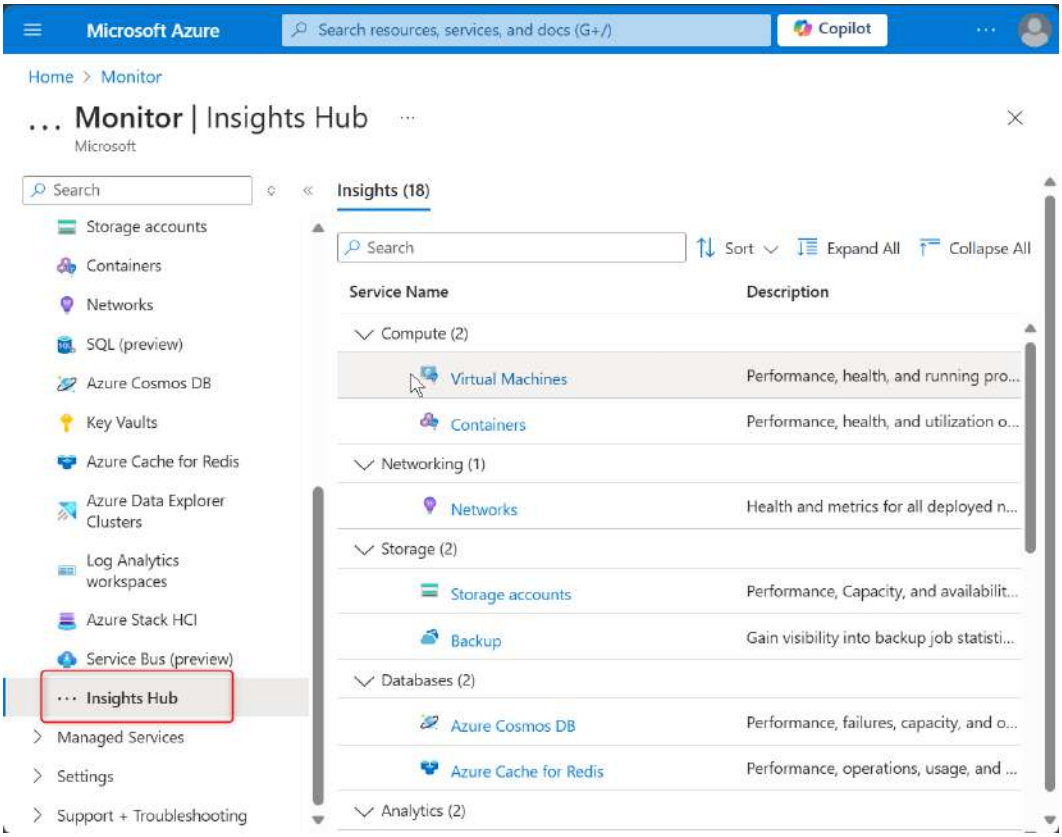

Figure 6.1 – The Insights Hub services

If you select the **Storage accounts** workbook from the list, a report like the following screenshot will appear. You can find a list of all your storage accounts in the current scope, with information about their current transactions. If you navigate around the report, other information is available, such as the accounts' currently used capacity.

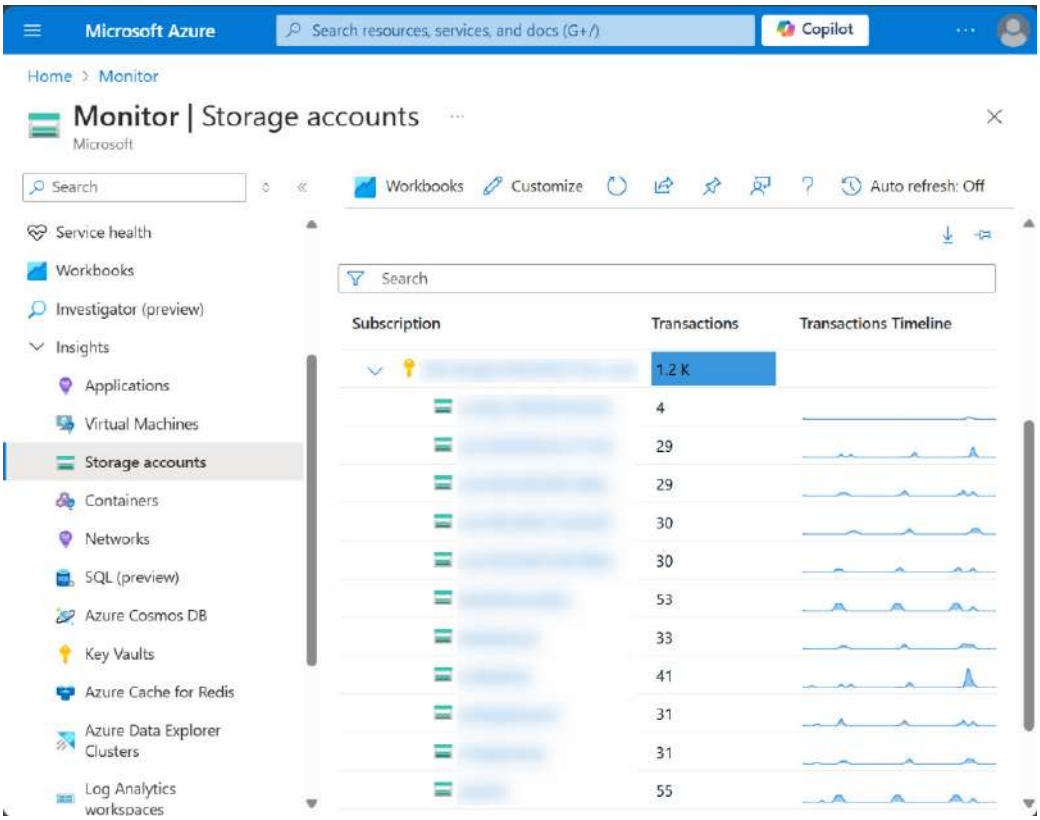

Figure 6.2 – Storage Insights

We highly recommend exploring all the different workbooks included by default inside the *Insights Hub* because they provide a good overview of what's happening with your resources, and they also provide a great summary of all the visualization options that Azure Workbooks offers you as a user to build custom visualizations. In the next section, we will introduce Azure Workbooks in more detail.

Azure Workbooks

Azure Workbooks are interactive reports that combine text, query results, and visualizations to present data in a comprehensive and user-friendly manner. They are part of Azure Monitor, designed to help users analyze telemetry data collected from various Azure services. Workbooks can be used for a wide range of scenarios, including troubleshooting, monitoring applications, diagnosing issues, creating

operational dashboards, and generating reports for various stakeholders making them an essential tool for IT administrators, developers, and business analysts. In the following screenshot, you can see a graphical representation of what can be achieved using Azure Workbooks – in this case, a complex and colorful visualization of the key metrics associated with the virtual machines that are available inside your subscription.

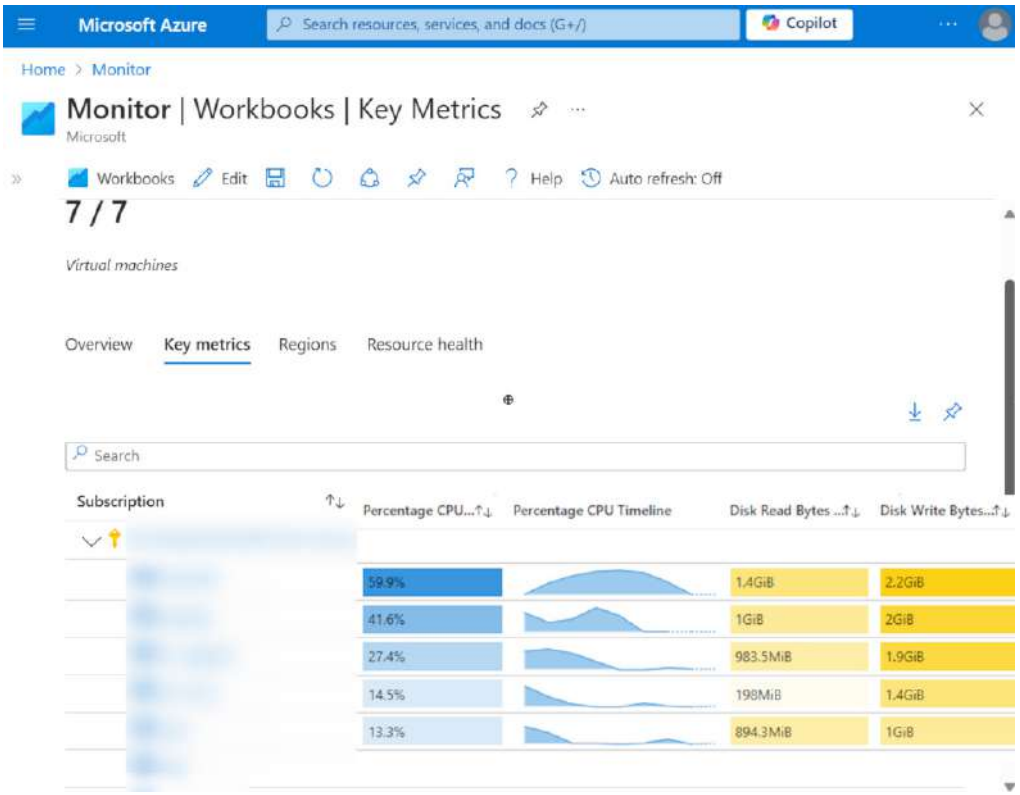

Figure 6.3 – Azure Workbooks – key metrics for an Azure virtual machine

As previously mentioned, Azure Workbooks are a powerful tool for visualizing and analyzing telemetry data from your Azure environment. The main features they provide are as follows:

- **Interactive visualizations**: Workbooks support various visualizations, such as charts, graphs, grids, and maps, enabling users to present data in the most effective format. These visualizations are interactive, allowing users to drill down into details and uncover insights that might be hidden in raw data.
- **Custom queries with KQL**: Users can write custom queries to extract specific data from Azure Monitor Logs and Metrics. This flexibility allows for specific analysis and reporting, meeting the unique needs of different stakeholders.

- **Pre-built templates**: Azure Monitor provides a collection of pre-built workbook templates for common monitoring scenarios. These templates can be customized further to fit specific requirements, saving time and effort in creating workbooks from scratch.
- **Comprehensive data integration**: Workbooks can pull data from multiple Azure sources, including Azure Monitor Logs, Azure Resource Graph, and Application Insights. This integration capability ensures that users have a comprehensive view of their environment, combining metrics and logs in a single, cohesive report.
- **Collaboration and sharing**: Workbooks can be shared with other users within an organization, facilitating collaboration and ensuring that everyone has access to the same insights. This collaboration enhances troubleshooting and analysis tasks, ensuring that everyone works with the same data and visualizations.
- **Customizable layouts**: The layout of a workbook can be adjusted to suit specific needs. With sections for Markdown text, parameters, queries, and visualizations, users can create a logical and user-friendly flow, effectively telling the story of their data. In the last section of this chapter, we will see how to create an Azure Workbook, but before doing so, it is important to follow best practices in the design and use of workbooks to get the most out of the capabilities they offer us, which are as follows:
    - **Consistent layout and design**: Maintain a consistent layout and design across your workbooks to ensure that they are easy to navigate and understand. Use clear headings, consistent color schemes, and logical data grouping.
    - **Optimizing KQL queries**: Write efficient KQL queries to minimize the impact on performance. Avoid overly complex queries that can slow down a workbook and make it less responsive. By writing concise and well-optimized KQL queries, you can strike a balance between extracting valuable insights and maintaining a seamless user experience.
    - **Leverage pre-built templates**: Use pre-built templates as a starting point to speed up the creation process. Instead of starting from scratch, users can leverage these pre-built templates and customize them to fit their unique monitoring requirements. By utilizing pre-built templates, organizations save time, streamline the creation process, and benefit from a consistent structure for their workbooks.
    - **Regular updates**: Keep your workbooks up to date with the latest monitoring requirements and data sources. Regularly review and update the queries and visualizations to ensure that they remain relevant and accurate.
    - **Collaborative development**: Involve team members in the creation and review process to gather diverse insights and ensure that the workbooks meet the needs of all stakeholders.

Azure Workbooks are ideal when you need to perform deep data analysis and exploration, you require advanced querying capabilities and custom data sources, and interactive data visualization and the ability to drill down into data is important. However, sometimes, you need something simpler to get

a high-level, real-time overview of resource status and key metrics and a centralized view of metrics from various Azure resources, without complex configuration. In those scenarios, Azure dashboards provide a better alternative.

Azure dashboards

Azure dashboards offer a powerful, customizable, and integrated way to visualize and monitor your Azure resources in the Azure portal. In this section, we will explore Azure dashboards' features and best practices.

Dashboards are designed to provide a consolidated view of the health, performance, and usage of your cloud infrastructure, allowing you to make informed decisions and quickly respond to issues. For example, the following figure shows a simple dashboard that provides a global view of the resources available in your subscription, such as an inventory.

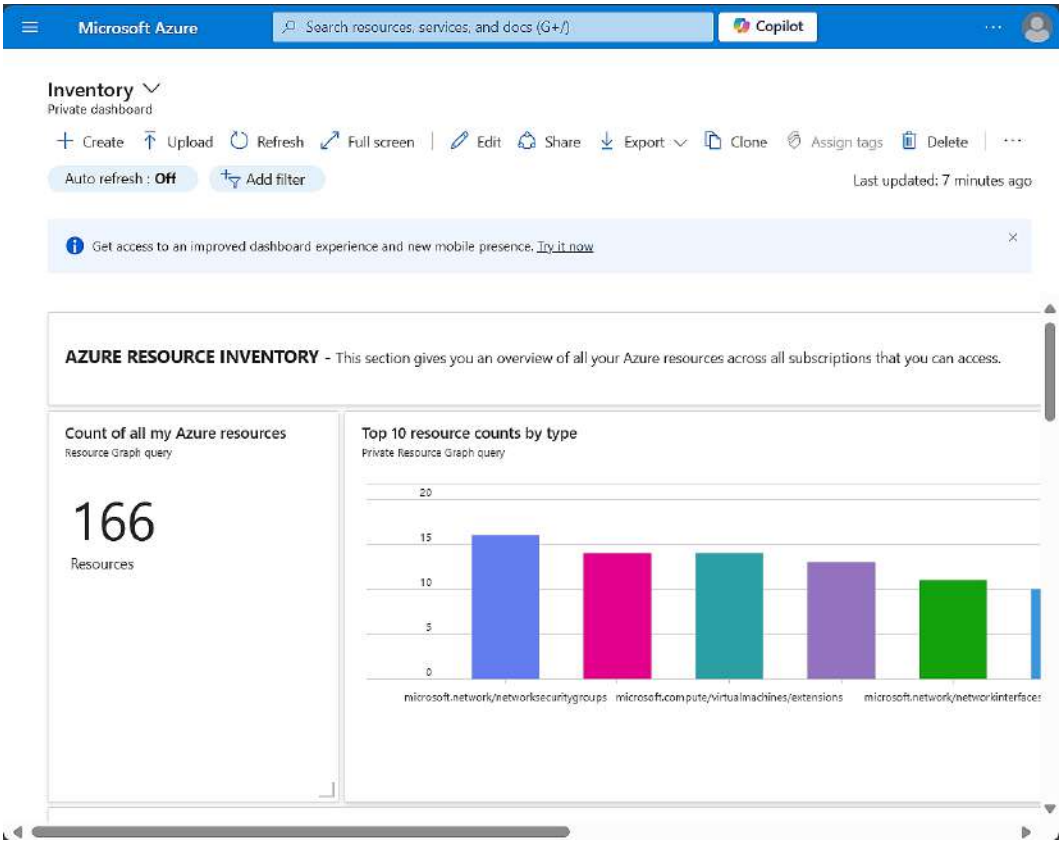

Figure 6.4 – An Azure Monitor dashboard

An Azure dashboard's main features are as follows:

- **Customizable layouts**: Azure dashboards allow you to arrange tiles in any layout that best suits your needs. You can resize, move, and group tiles to create a personalized view of your Azure environment. The following screenshot shows how tiles can be distributed across the gray grid, shown in the background, when editing your dashboards.

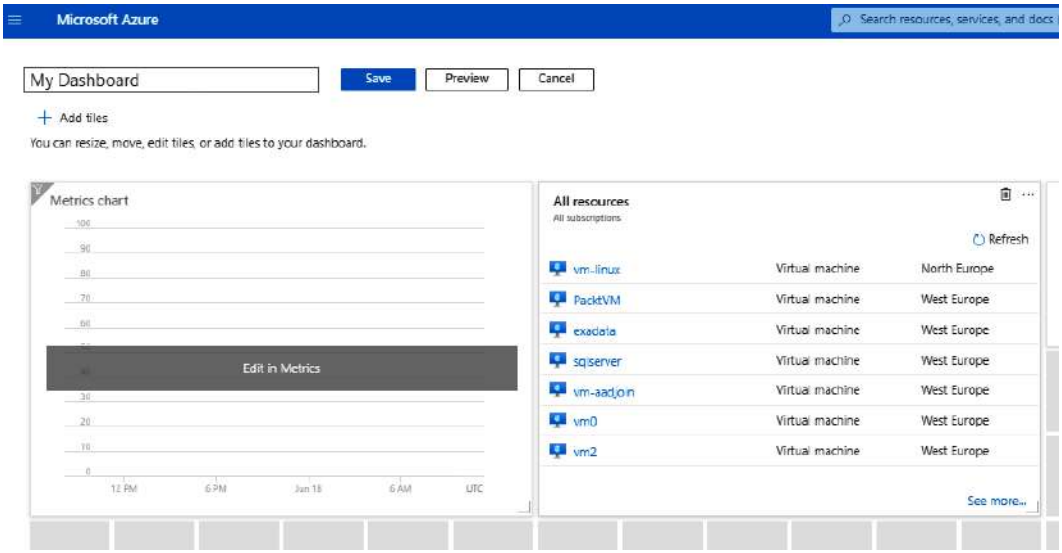

Figure 6.5 – An Azure Monitor dashboard layout

- **A wide range of tile types**: Azure dashboards support various types of tiles, including metric charts, resource graphs, service health, alerts, and logs. This flexibility allows you to incorporate different data visualizations into a single dashboard. In the following figure, you can see some of the options available inside the **Tile Gallery** window to customize your visualizations.

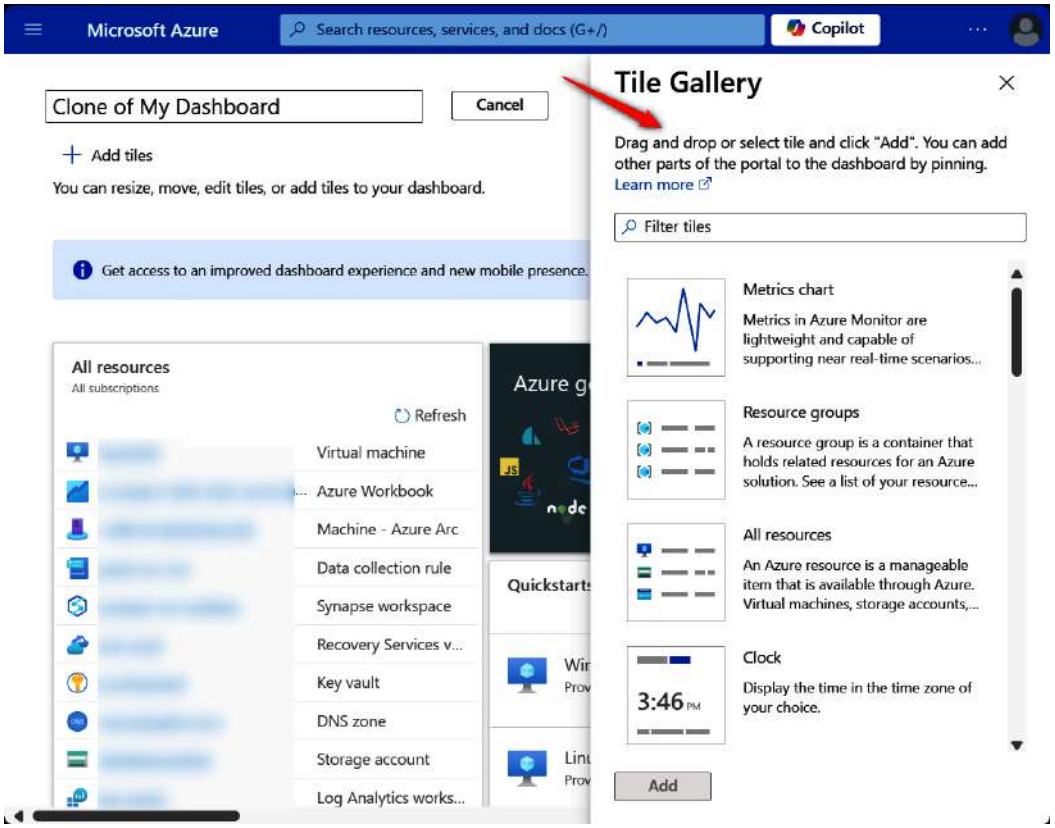

Figure 6.6 – Azure Monitor dashboard tiles

- **Integration with Azure Services**: Azure dashboards offer direct integration with Azure Monitor, Application Insights, and other Azure enabling comprehensive monitoring and diagnostics.
- **Interactive and real-time data**: Azure dashboards offer interactive tiles that allow you to drill down into specific metrics or logs for deeper analysis. Dashboards update in real time if you refresh the full dashboard or specific tiles, providing up-to-date information on your resources. You can also opt for the auto-refresh feature at the dashboard level.
- **Role-Based Access Control (RBAC)**: Administrators can control who has access to your dashboards using Azure's built-in RBAC. This ensures that the right people have the right level of access to the data they need.
- **Sharing and collaboration**: Azure Dashboards can be shared with team members, making it easy to collaborate and ensure that everyone works with the same information. Shared dashboards can be set to read-only or editable, depending on the needs of your team.

Next, we will explore the best practices in the design and use of Azure dashboards, and in the *Further reading* section of this chapter, you can expand your knowledge about the Azure dashboard creation process [1]:

- **Keep it simple**: Avoid overcrowding your dashboard with excessive tiles. Prioritize the most essential metrics and visualizations that deliver actionable insights.
- **Use grouping**: Organize related tiles together to form logical sections. This enhances a dashboard's navigability and helps you locate information quickly.
- **Leverage RBAC**: Implement RBAC to ensure that only authorized users can view or modify your dashboards. This practice helps maintain data security and integrity.
- **Regular updates**: Regularly review and update your dashboards to keep them relevant. Add new metrics or tiles as your monitoring needs evolve.
- **Interactive features**: Utilize the interactive capabilities of Azure dashboards. Drill down into specific metrics or logs for more in-depth analysis directly from the dashboard.

Before introducing the open source solutions available inside Azure to visualize your logs and metrics, we will take a look at Microsoft Power BI on Azure. It is another tool within the Azure ecosystem for data visualization and analysis, and it has its own unique set of features and use cases.

Microsoft Power BI on Azure

Microsoft Power BI on Azure is mostly seen as a comprehensive business intelligence tool, designed for in-depth data analysis, reporting, and dashboarding, often involving complex datasets and business metrics. However, it is also an alternative visualization tool, thanks to the integration capabilities between Azure Monitor Logs and Power BI, via Log Analytics, to create detailed reports and dashboards. In the following steps, we will explain briefly how to configure this integration, and in the *Further reading* section of this chapter, you can expand your knowledge beyond the complete integration steps [2]:

1. The first step is to create the Power BI datasets and reports from Log Analytics queries. There are two options to do this:

    I. **Export to Power BI (as an M query)**: This exports the query and connection string to a `.txt` file for use in Power BI Desktop. This way, you can model or transform data in ways not supported by the Power BI service.

    II. **Export to Power BI (new Dataset)**: This creates a new dataset directly in the Power BI service for further analysis and reporting.

    To be able to export your information from Log Analytics in any of the two previous options, follow this configuration (*Figure 6.7*):

    A. Open the options settings inside your Log Analytics workspace, represented as ellipses (**…**).

B. You will then see the contextual menu where you can select the **Share** option. After that, the two options will be visible – to export the results as an M query or a dataset.

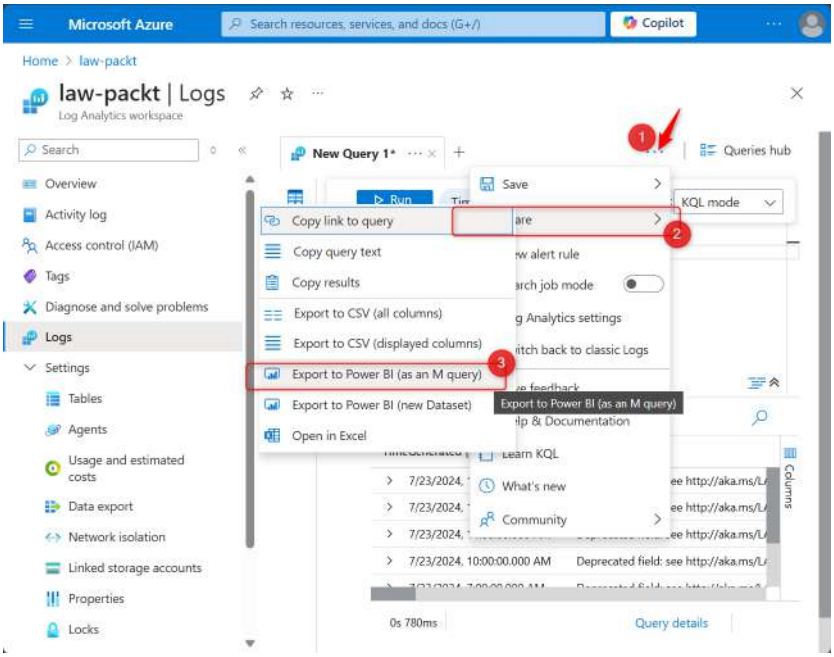

Figure 6.7 – Power BI datasets and reports from Log Analytics queries

2. After the Power BI dataset has been created, you should follow the standard configuration process on the Power BI side. It is possible to enable the incremental refresh of both Power BI datasets and Power BI dataflows to update only new data increments. Remember to include a datetime field in the query results.
3. Finally, once your data is sent to Power BI, you can create comprehensive reports and dashboards to monitor and analyze Azure log data effectively.

All previous services are features and products developed and managed by Microsoft. However, the ecosystem of tools for cloud monitoring visualization has grown in recent years at the same rate as cloud adoption. Grafana has become one of the key open source solutions to monitor cloud workloads, so we will briefly cover the managed services provided by Azure in Azure Managed Grafana next.

Azure Managed Grafana

Azure Managed Grafana is a powerful solution within the Azure ecosystem, combining the flexibility of Grafana's open source platform with the reliability and integration capabilities of Azure's managed services. We will explore the main Azure Managed Grafana features in this section, and in the *Further*

reading section, you can expand your knowledge on how to set up Azure Managed Grafana and build a Grafana dashboard [3]:

- **Multi-platform and multi-cloud visualizations**: Azure Managed Grafana offers a unified view of metrics and logs across multiple platforms and multi-cloud environments. This capability allows cloud teams to consolidate monitoring efforts in a single pane of glass.
- **Integration with Azure Services**: As a managed service in Azure, Azure Managed Grafana integrates seamlessly with Azure Monitor, Azure Log Analytics, Azure Data Explorer, and other Azure services. This integration facilitates comprehensive monitoring and analysis, leveraging Azure's data storage and management capabilities.
- **Advanced data visualization**: Grafana is renowned for its rich set of visualization options, including graphs, charts, and dashboards that support time-series data. Managed Grafana extends these capabilities with Azure-specific enhancements, enabling users to create dynamic and interactive dashboards tailored to specific monitoring needs.
- **Scalability and performance**: Azure Managed Grafana scales to handle large volumes of data and supports high-performance requirements. Data caching and optimization features further enhance performance, ensuring responsiveness even with extensive data queries.
- **Prometheus and Cloud Native Computing Foundation (CNCF) compliance**: Azure Managed Grafana supports Prometheus metrics natively and is compliant with CNCF standards. This makes it an ideal choice for environments that adhere to these industry standards, ensuring compatibility and interoperability across platforms.
- **Integration with third-party tools**: Azure Managed Grafana's compatibility with third-party monitoring tools and plugins enhances its versatility, allowing organizations to extend its capabilities and integrate additional data sources as needed. This flexibility supports custom monitoring setups and complex operational requirements.

So far, we have explored five different visualization tools accessible inside Azure to create your own dashboards and reports. However, which one should you choose based on your needs and requirements? In the next section, we will provide guidance on how to simplify your selection process.

Choosing the right visualization tool

By the end of this section, you will have a clear understanding of the strengths and appropriate use cases for each visualization tool, enabling you to select the one that best fits your data visualization needs and enhances your ability to derive actionable insights from your data:

- **Azure Monitor Insights**: This provides deep, built-in insights into various Azure services, offering tailored monitoring and diagnostic capabilities. It is an ideal option if you want deep insights into the performance, health, and operation of your applications and infrastructure, based on a subset of available telemetry collected from these Azure services.

Insights are published directly by Microsoft, leveraging various data sources, including metrics, logs, and events. This provides a unified approach to monitoring Azure resources, which helps in troubleshooting, capacity planning, and ensuring optimal performance and availability of applications and services. It provides a reference or baseline about the key metrics per service that allows a user to customize it further. Azure Monitor Insights is based on Azure Workbooks technology. In this chapter, you will learn how to build custom workbooks to expose the relevant insights for your application or infrastructure.

- **Azure Workbooks**: This is a native Azure dashboarding platform, designed specifically for engineering and technical teams to visualize and investigate various scenarios. One of its significant strengths is the ability to auto-refresh, making it a valuable reporting tool for app developers, cloud engineers, and other technical personnel who require up-to-date information. Azure Workbooks come with a range of out-of-the-box templates and public GitHub reports, facilitating quick setup and deployment.

  The platform's ability to use parameters for dynamic, real-time updates ensures that users can interactively explore data, providing high-level summaries and allowing for deeper dives into specific items, using selected query values. This feature makes Azure Workbooks a good option for teams that want highly effective monitoring and diagnostics. Additionally, it supports querying more sources than other visualization tools, making it a comprehensive solution to integrate diverse data streams.

  Finally, Azure Workbooks enable users to create tailored dashboards that meet specific organizational needs at no additional cost. They are designed for collaboration and troubleshooting, allowing multiple users to work together seamlessly, sharing insights and resolving issues efficiently. These attributes make Azure Workbooks a good option for technical teams seeking to enhance their monitoring and reporting capabilities within the Azure ecosystem, as well as applications running in a multi-cloud environment.

- **Azure dashboards**: These serve as a powerful and cost-effective native Azure dashboarding platform, ideal for environments that exclusively use Azure or Azure Arc. Similarly to Azure Workbooks, one of the key strengths of dashboards is that they incur no additional cost, making them an economical choice for organizations. Azure dashboards support at-scale deployments, which means they can efficiently handle large and complex environments. These dashboards enable users to combine metrics graphs with log query results and operational data for related services, providing a comprehensive view of a system's health and performance. This capability allows for a more integrated and holistic approach to monitoring. Additionally, Azure dashboards facilitate collaboration through seamless integration with Azure RBAC, allowing service owners and other stakeholders to share dashboards easily. This makes them an excellent tool for cross-functional teams who need to monitor and manage their Azure resources collectively.

- **Microsoft Power BI on Azure**: This is a premier business analytics tool known for its rich visualizations and extensive capabilities in slicing and dicing data. It is particularly effective at creating external visualizations aimed at management and executives, offering clear and compelling insights through interactive dashboards and reports. Power BI supports comprehensive BI analytics, enabling users to design business-centric KPI dashboards that track long-term trends and performance metrics. One of Power BI's significant strengths is its ability to integrate data from multiple sources, providing a unified view of diverse datasets. The platform caches results in a cube for better performance, ensuring fast and responsive data analysis. Power BI is also designed for easy sharing across organizations, facilitating collaboration and ensuring that stakeholders at all levels have access to the same insights. These attributes make Power BI an ideal choice for organizations seeking to leverage data for strategic planning, performance monitoring, and operational optimization
- **Azure Managed Grafana**: This is an advanced visualization platform that provides multi-platform, multi-cloud, single-pane-of-glass visualizations. One of its key strengths is its ability to seamlessly integrate with Azure while also supporting data from various other cloud providers and on-premises environments. This makes it an ideal choice for users without direct Azure access who need to monitor and visualize diverse data sources in one unified interface. Its capability to combine time-series and event data within a single visualization panel enhances its use for real-time monitoring and historical analysis. The platform supports dynamic dashboards, allowing users to filter and view data based on dynamic variables, which is especially useful in multi-cloud environments.

  Azure Managed Grafana has built-in support for Prometheus and other third-party monitoring tools, and it comes with out-of-the-box plugins for most monitoring tools and platforms, along with a rich repository of community-created and community-supported templates that focus on operations. These features enable quick and efficient dashboard creation, tailored to specific monitoring needs.

  Grafana's dashboards are perfect for displaying overall statuses, up/down indicators, and high-level trend reports for management and executive-level users. Furthermore, its vendor-agnostic approach allows for the creation of business continuity and disaster recovery scenarios that can operate across any cloud provider or on-premises setup, ensuring robust and flexible monitoring solutions for diverse operational needs.

After all this theory about the options available to visualize your information inside Azure Monitor, let's move on to a practical example, using an Azure Workbook to create your own custom monitoring solution. Of all the options described, workbooks are the most powerful and, conversely, the most difficult ones to start with if you don't have a basic knowledge of how they work. The following section provides some basic knowledge that will help you to build more complex reports as needed.

Building a custom monitoring workbook – a hands-on example

Some of the most common problems that operations teams face when the number of Azure resources in an organization grows are reducing the cost of unused resources, avoiding misconfigurations, and simplifying operations. To help in such situations, we will show you how to create a new workbook and how to add elements to it in this section. Specifically, we will create a workbook that helps us identify orphaned network resources.

This lab does not cover all the workbook configuration options, so we recommend reviewing the *Further reading* section of this chapter, where you can expand your knowledge on how to create more advanced Azure Workbooks.

To create a new Azure Workbook, follow these steps:

1. Go to the Azure **Workbooks** page and select an empty template or **New** from the top toolbar. This will open the editing interface inside Azure Workbooks.

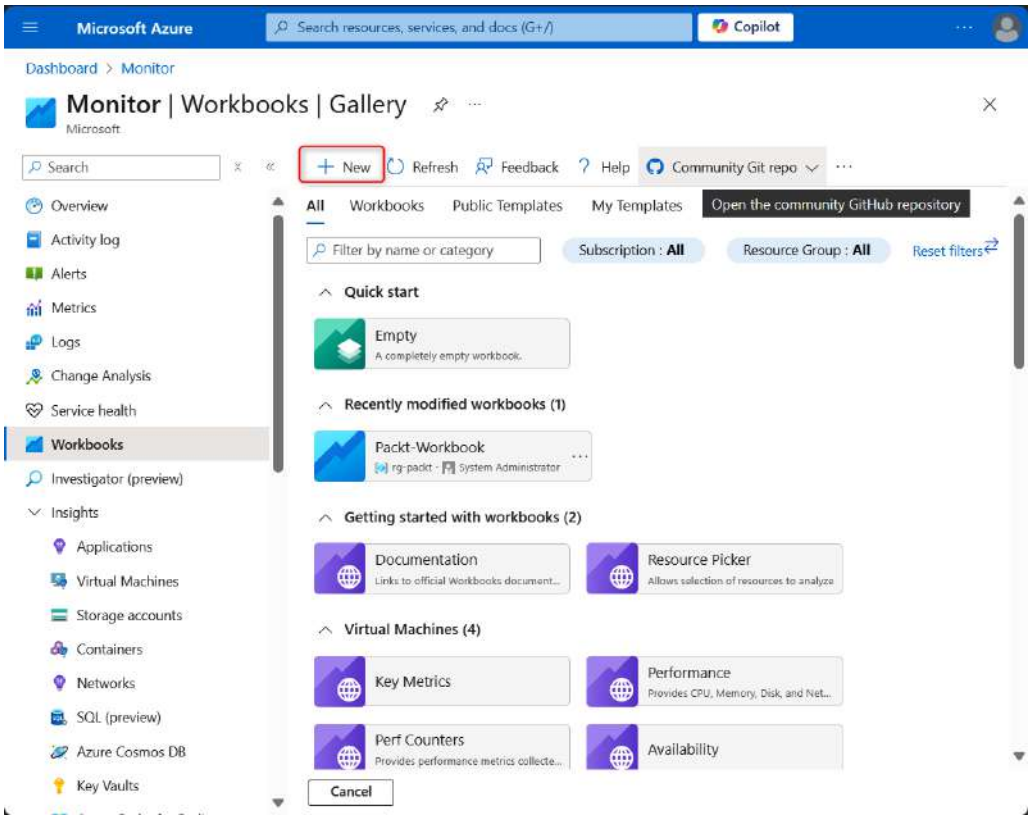

Figure 6.8 – Creating a new Azure Workbook

2. Azure Workbooks allow you to combine any of the available elements (**Text**, **Parameters**, **Queries**, **Metric charts**, **Links**, and **Groups**) inside your workbook. Click the **+ Add** button to start adding items when the workbook has no content.

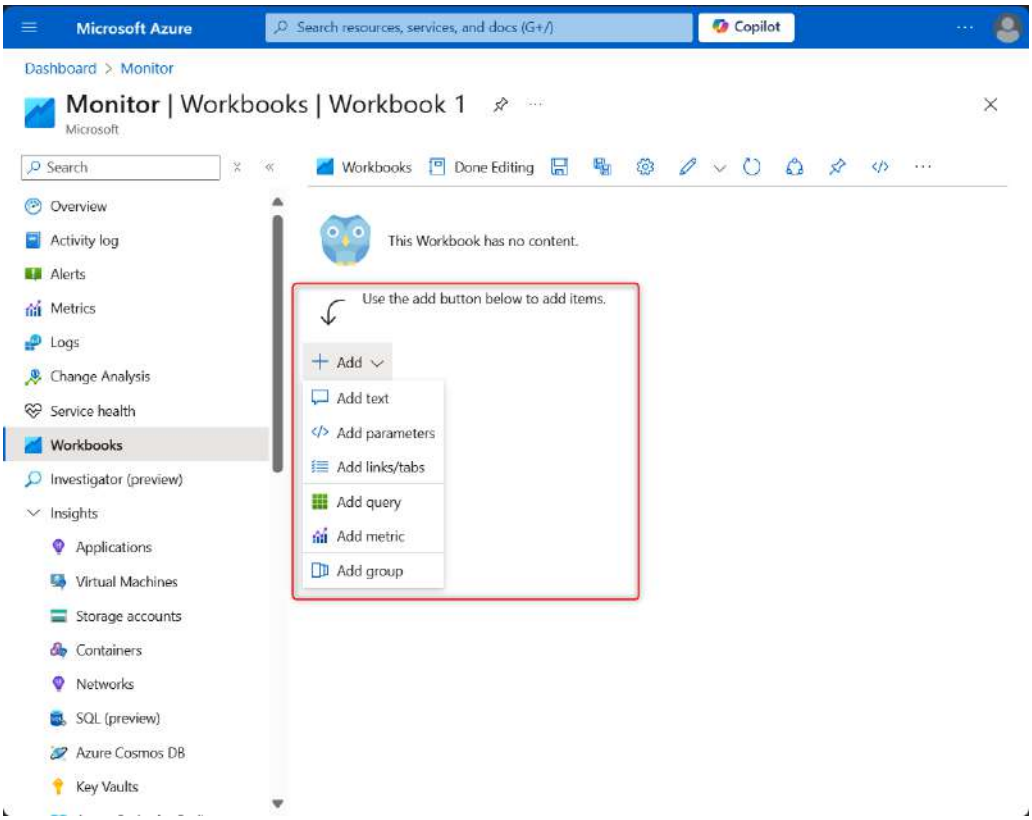

Figure 6.9 – Creating a new workbook

3. Select the **Add Text** option to create a new *text type* item. We will call it `text - 0`. From the **Text style** options, we will choose **Plain**, and in the **Markdown text** cell, we will add a description of the purpose of our workbook.

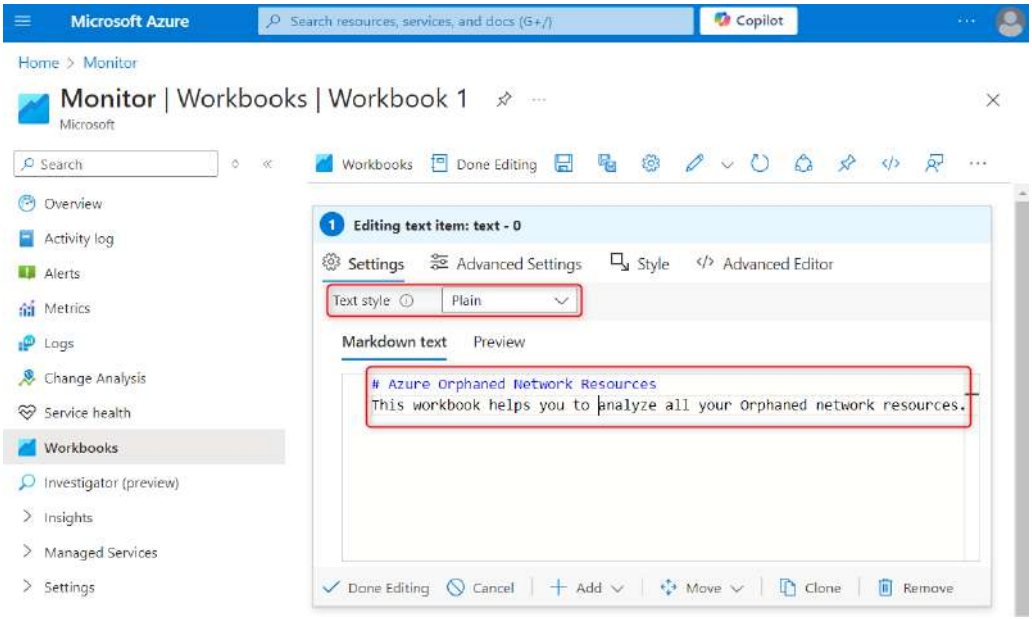

Figure 6.10 – Creating a text item

4. If you want to preview the results, just click on the **Preview** button, as shown in the following screenshot. Markdown text will be rendered as it would appear in your workbook.

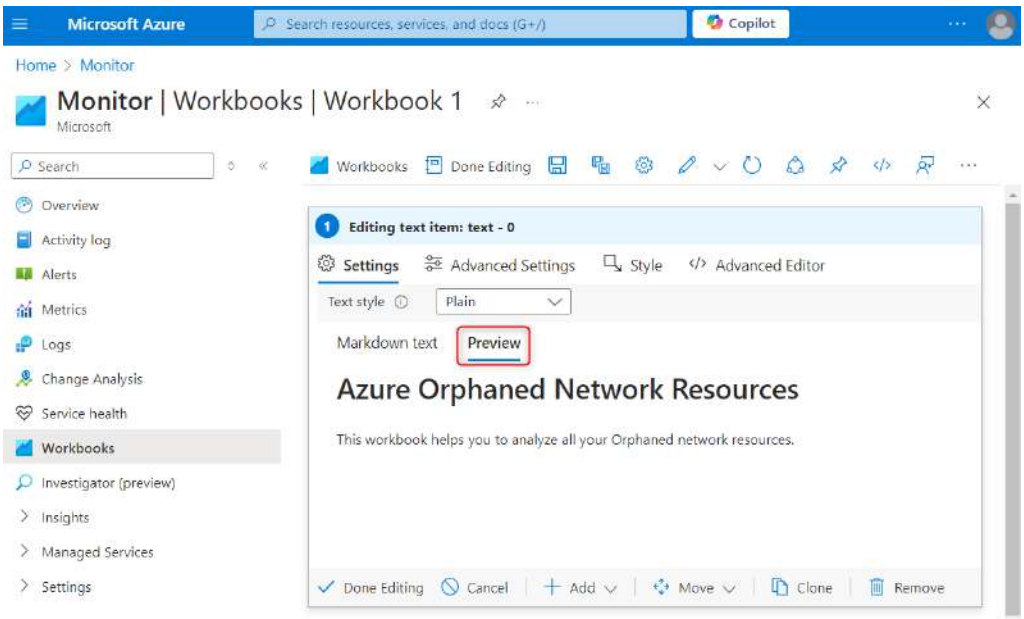

Figure 6.11 – A preview of the text item content

5. After the new text item has been added, the **Add Parameter** button will appear. Click on it to create an item of type `parameter`, which we will need to reference throughout the workbook.

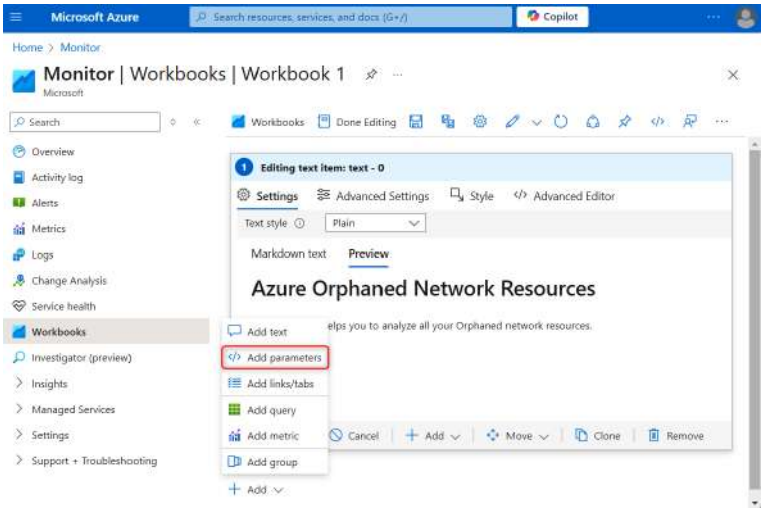

Figure 6.12 – Add parameters button

After clicking the **Add parameters** button, the Parameter setting tab will appear. We will create this parameter in the **Pills** style and then click on the **Add Parameter** button to configure the parameter settings.

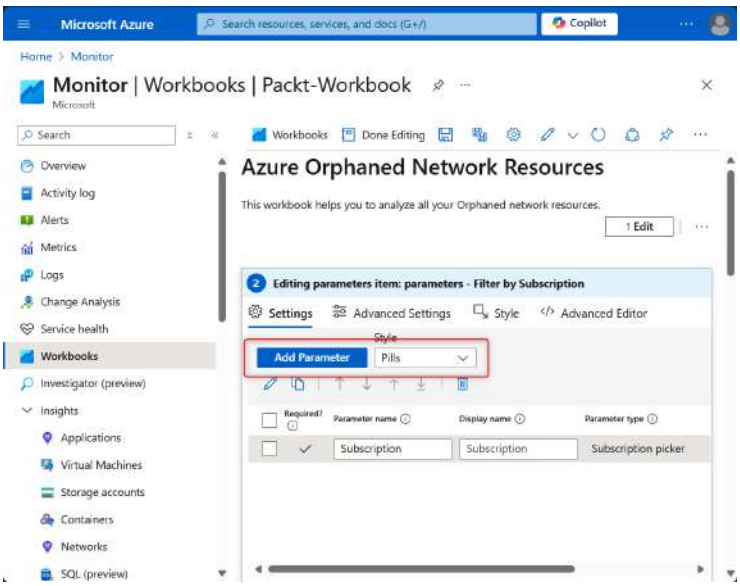

Figure 6.13 – Creating a parameter item

In the parameter settings, we will call it `Subscription`, select **Subscription picker** as parameter type, and tick the **Required** and **Allow multiple selections** checkboxes. In **Get data from**, we will select **All Subscriptions** and check the **All** box so that these options are added to the drop-down options.

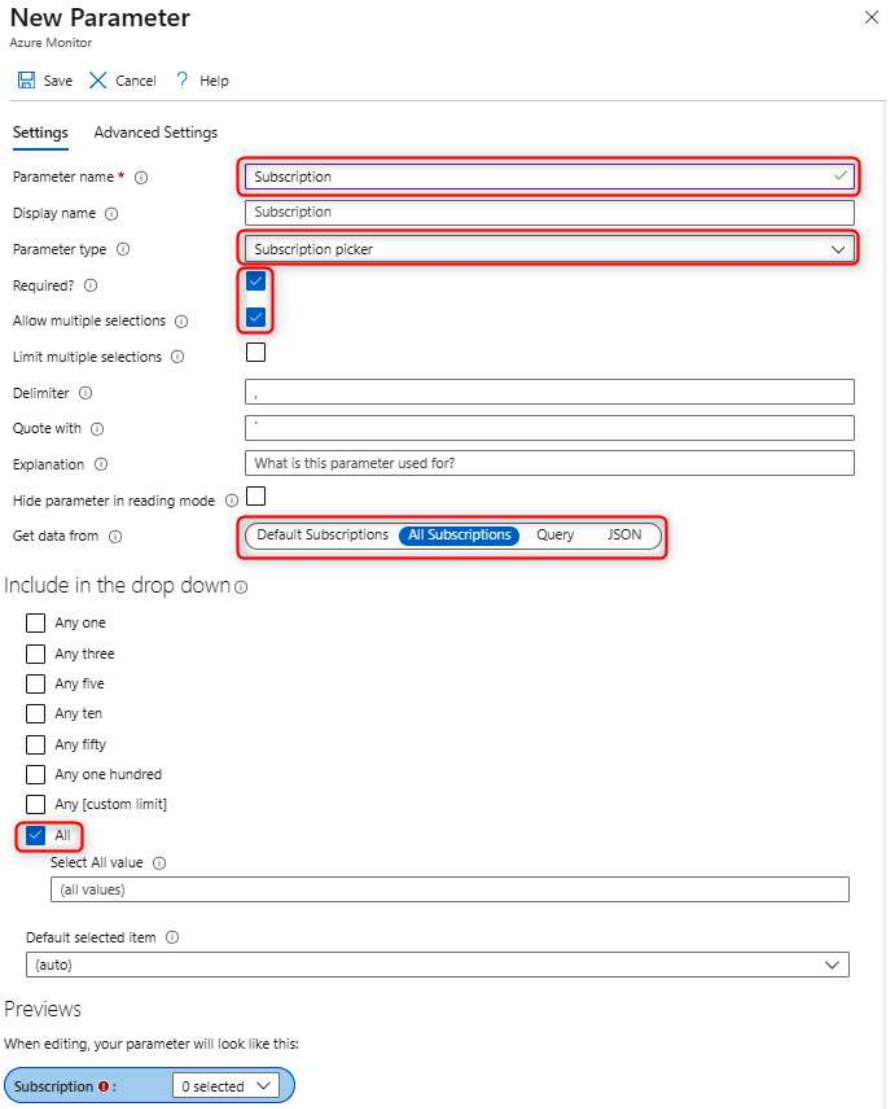

Figure 6.14 – Parameter item settings

6. Now, we will create six *tabs* (**Overview**, **Public IPs**, **NICs**, **NSGs**, **vNets**, and **snets**). To do that, we will use the **+ Add** button, available after creating the previous item. Select the **Add links** option to create a new *links type* item. We will call it `links - Tabs`. You need to change the **Style** drop-down list value to **Tabs** and define the action associated with each tab as **Set a parameter value**. We will use the value defined by each tab to decide which information to show and hide later in our graph. To do that, define the value of the parameter as **tab** and the settings as the tab name, as shown in the following figure.

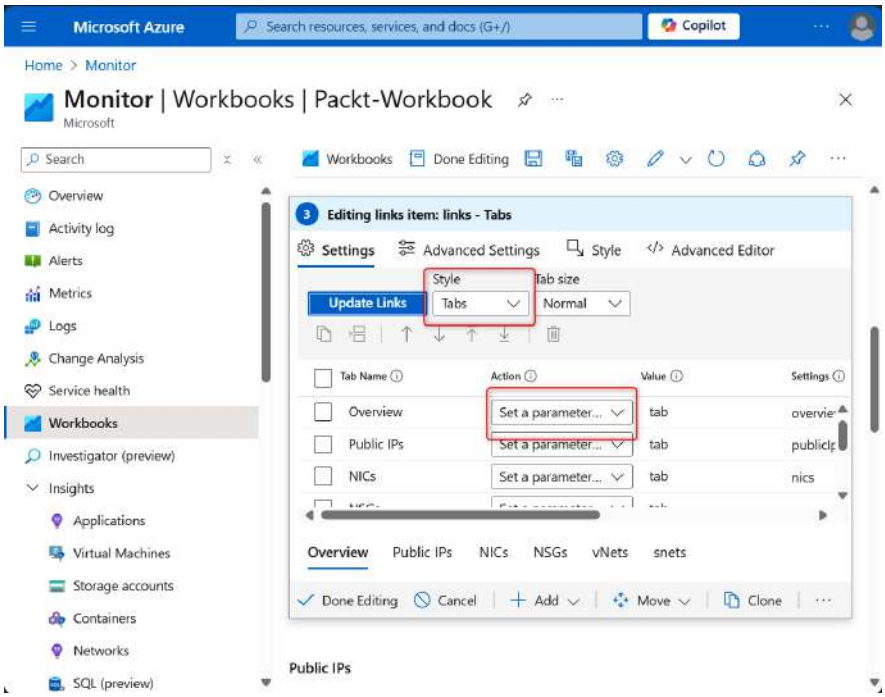

Figure 6.15 – The linked Tabs item

7. Again, we will use the **+ Add** button, available after creating the previous item. You can create a group item to include six different visualizations by selecting the **Add group** option. The visualizations that we are interested in are as follows:

   - The number of orphaned public IPs
   - The number of orphaned NICs
   - The number of orphaned NSGs
   - The number of orphaned vNets
   - The number of orphaned snets

We will set **Load type** as **Lazy** so that a group is only loaded when the item is visible. If the tab is never selected, the group remains hidden, and as a result, the content is not loaded.

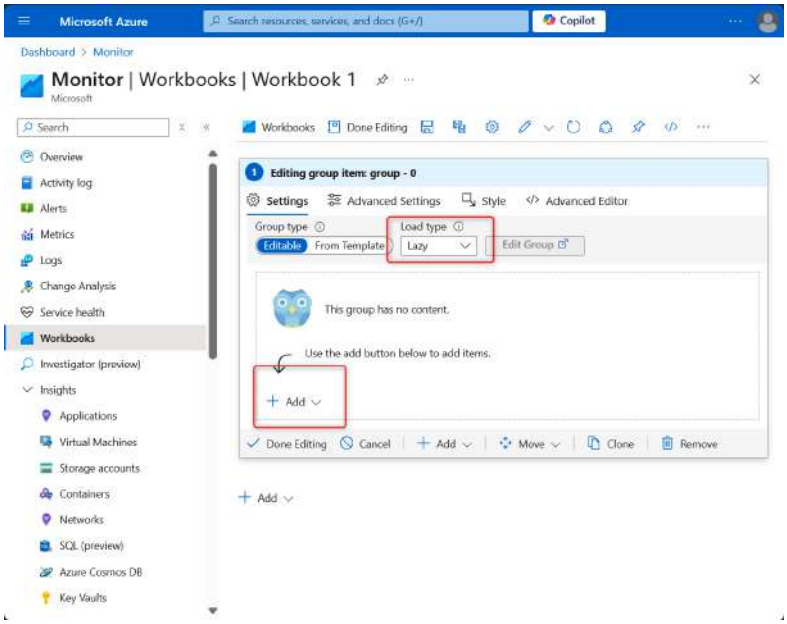

Figure 6.16 – Adding a group item

8. As we mentioned earlier, we want this group to be visible only in the **Overview** tab, so we will make the group conditional visible by adding, via **Advanced Settings**, the **tab | equals | overview** visibility condition.

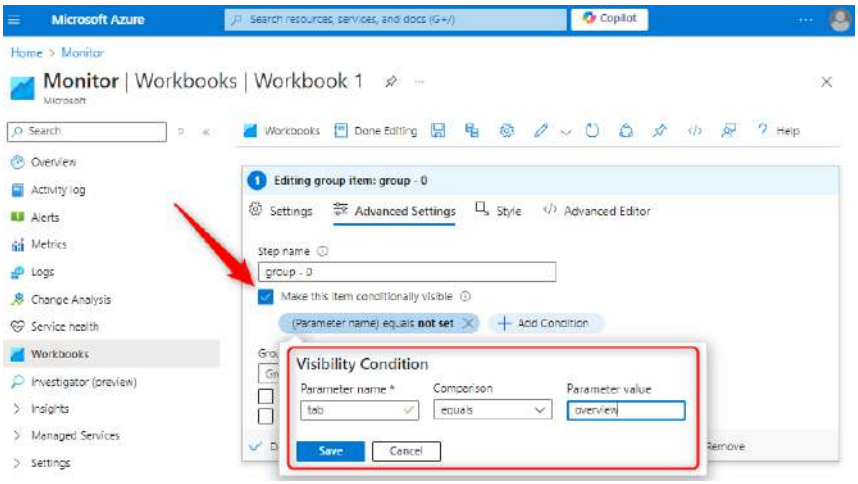

Figure 6.17 – The group item conditionally visible

9. After the basic visualization structure of our report is configured, now it is time to show some information inside it. Using the same **+ Add** button as before, select the **Add query** option with the **Azure Resource Graph** data source, as shown in the following screenshot.

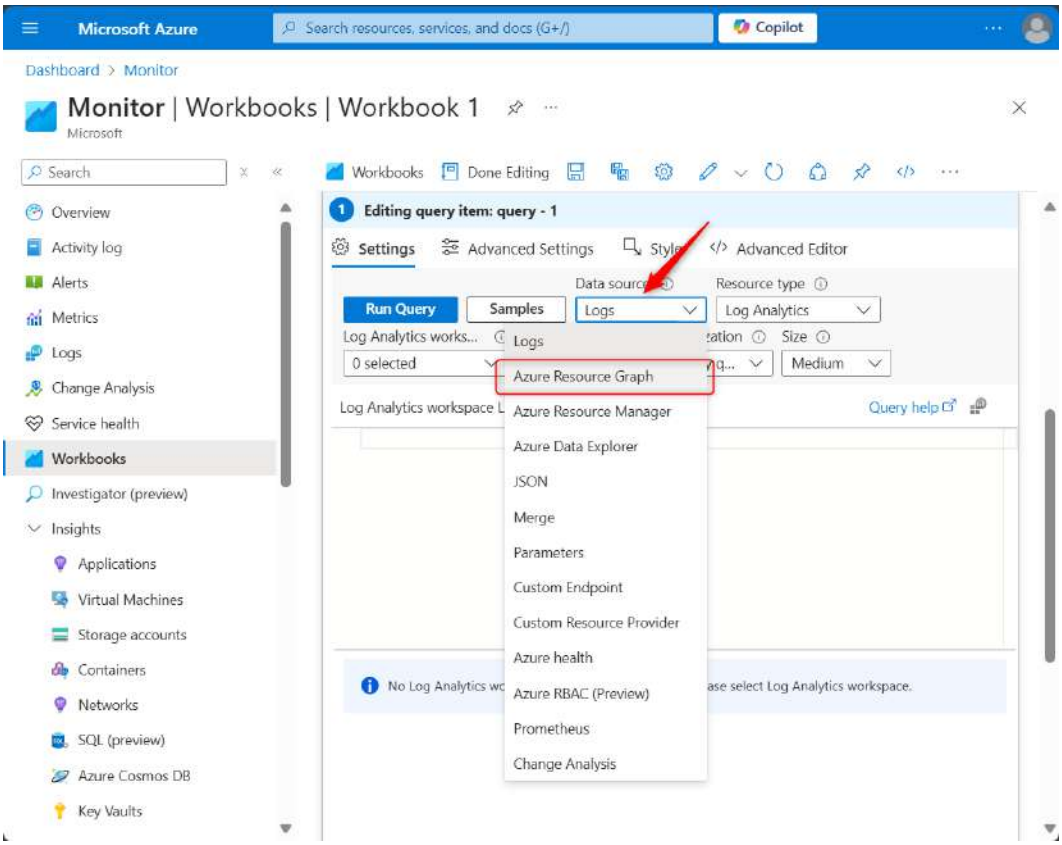

Figure 6.18 – The query item

10. Configure the **Subscriptions** parameter to use all the subscriptions you have access to and include the query in the textbox:

```
resources
| where type == "microsoft.network/publicipaddresses"
| where properties.ipConfiguration == "" and properties.
natGateway == "" and properties.publicIPPrefix == ""
| summarize count(type)
```

Under **Visualization**, select **Tiles** instead of **Set by query**.

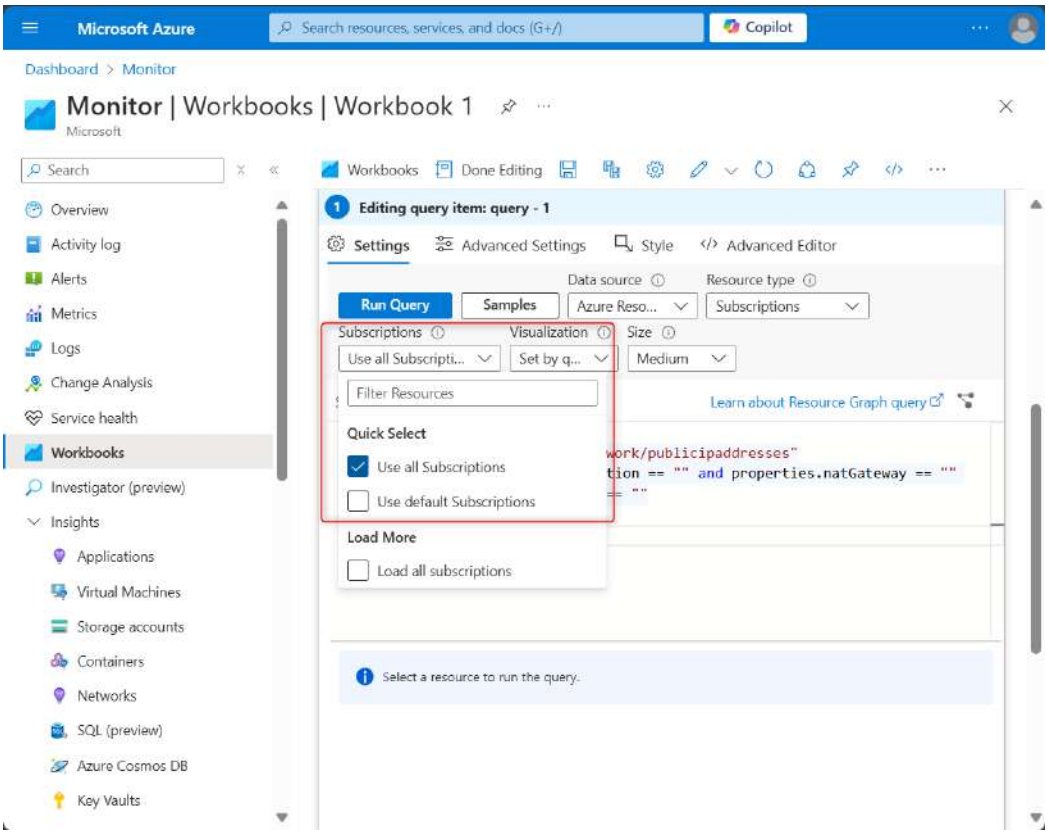

Figure 6.19 – The query item settings

11. Then, we will replicate the same steps five more times to configure the other services we are interested in visualizing. A fully detailed workbook can be found at `https://raw.githubusercontent.com/dolevshor/azure-orphan-resources/main/Workbook/Azure%20Orphaned%20Resources%20v2.0.workbook`.

12. After you have completed the previous steps, select the **Advanced Settings** tab to define a name for this cell inside the workbook and a chart title.

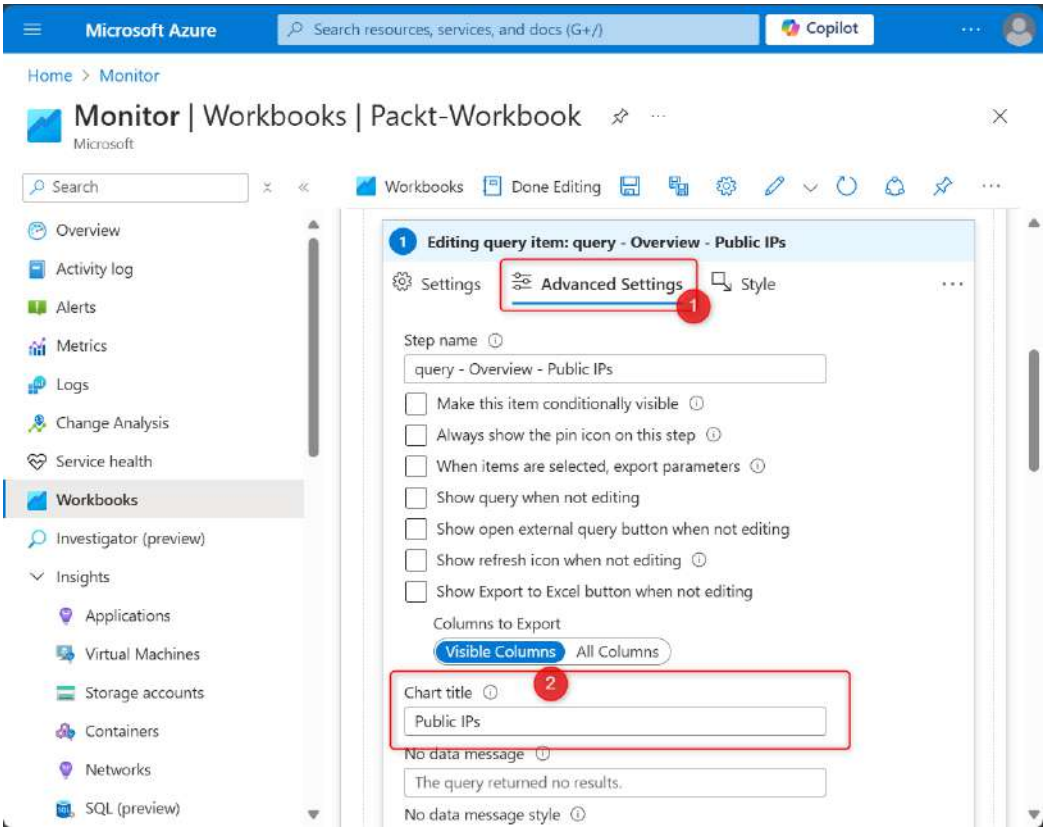

Figure 6.20 – The query item – Advanced Settings

13. It is possible to configure how the tiles defined in *Step 10* are rendered; if you click the **Tile Settings** button while editing your query, the configuration options will appear.

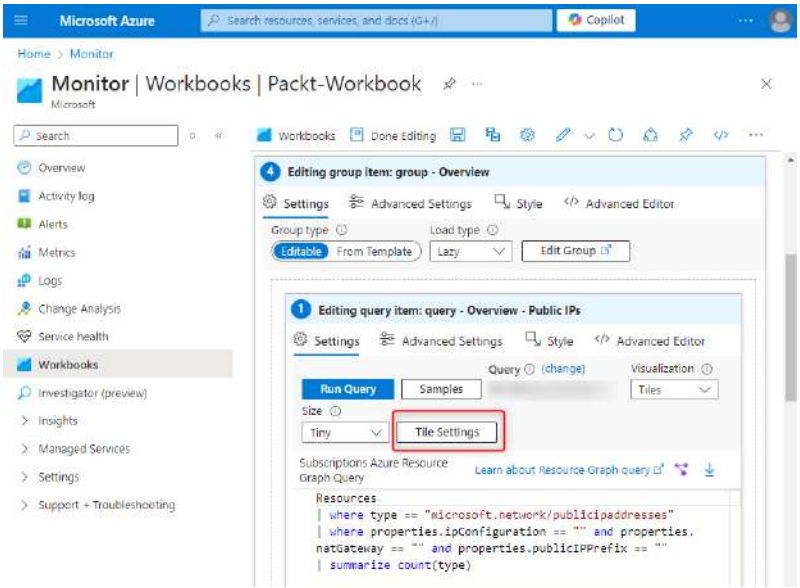

Figure 6.21 – Query item – Tile Settings

14. **Tile Settings** allows you to configure all the visual elements associated with the tile, from the way data is represented to the color palette used. We recommend reviewing all the different options to customize your report as needed.

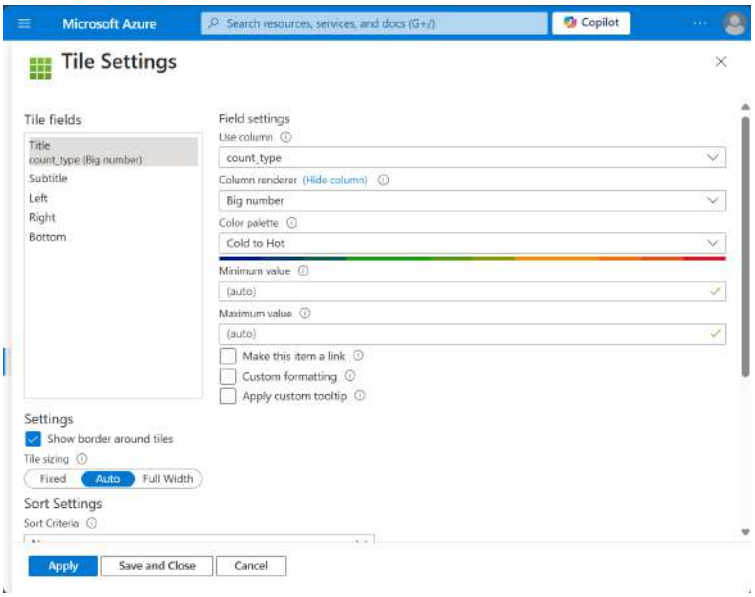

Figure 6.22 – Query item – Tile Settings

15. By default, the width of items when they are created is set to **auto**, and the maximum width field will be applied. To be able to include multiple items on the same horizontal, set the width of each item in a percentage value. Select **Style**, tick the **Make this item a custom width** checkbox, and set a percentage value that adjusts the item horizontally to be as narrow as possible.

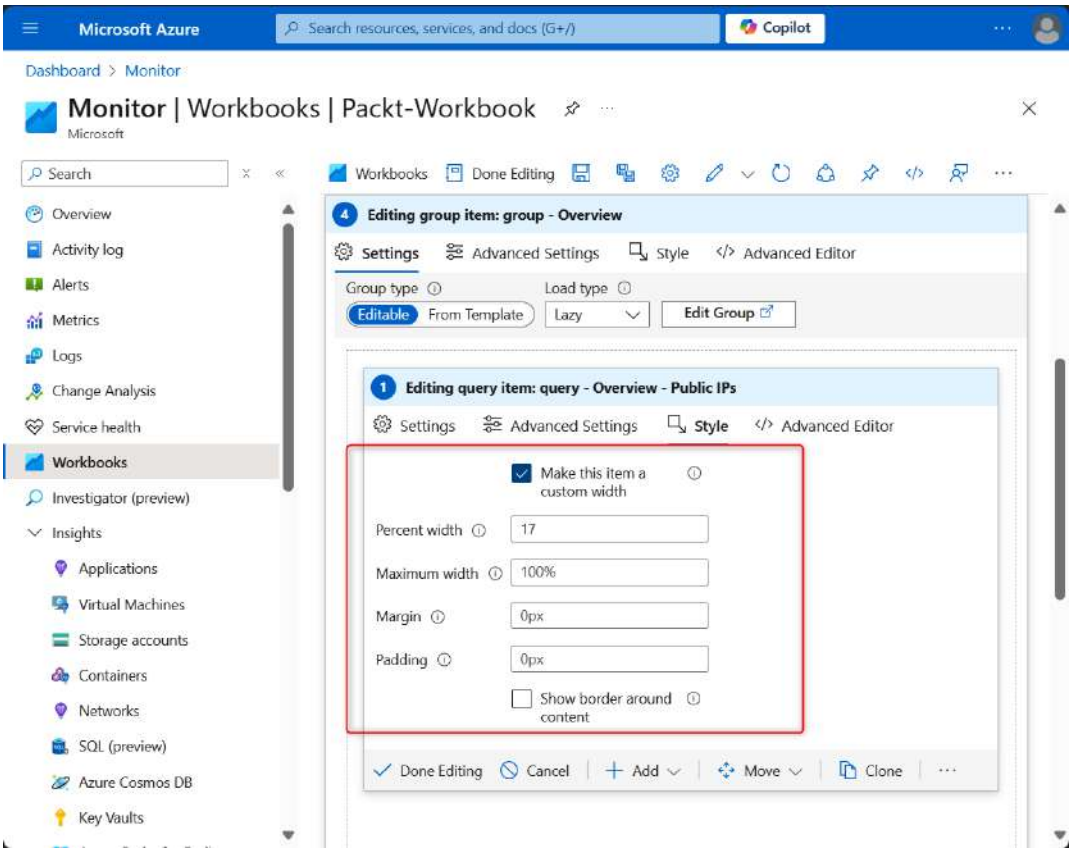

Figure 6.23 – Query items – custom width

After all these steps, we have configured the **Overview** tab of our workbook to show a group of tiles with the identified resources not in use, classified by their resource type, as shown in the following screenshot.

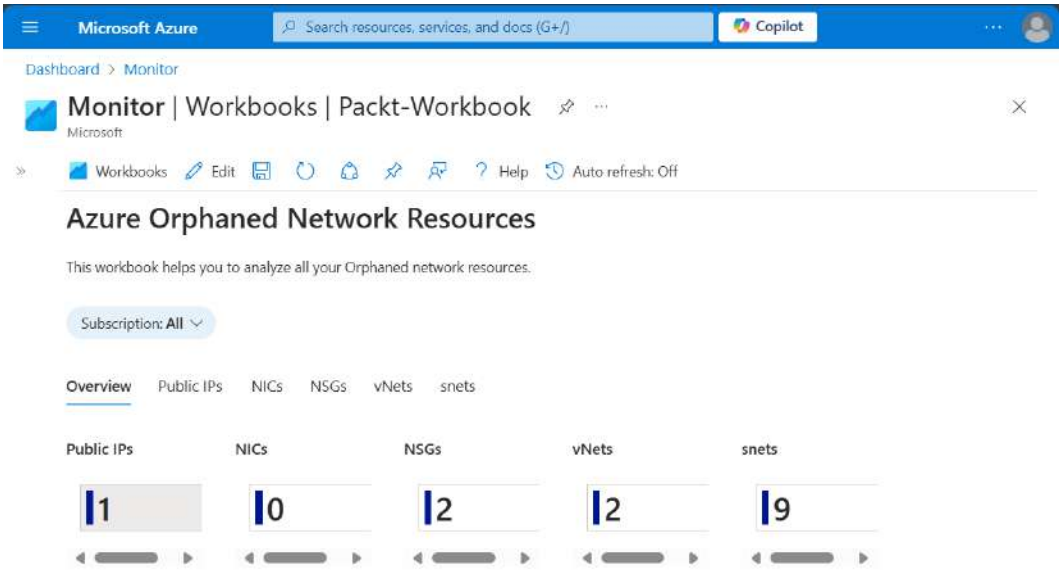

Figure 6.24 – Overview tab

16. However, this is not useful at all; we need to have a list of the specific resources to decide what to do with them. So, once the **Overview** tab is configured, let's complete the second tab with the list of public IPs.

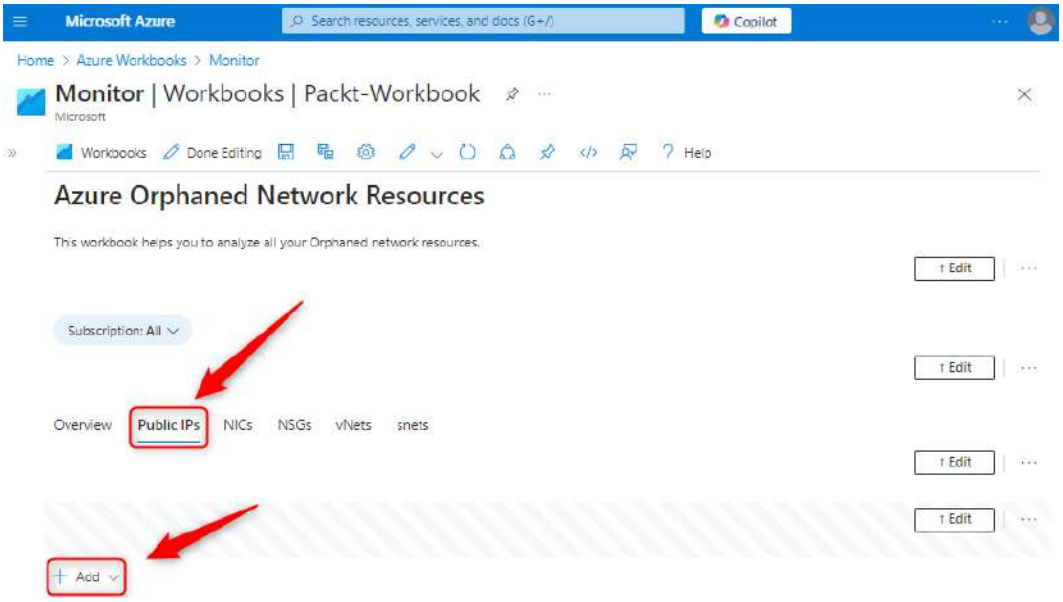

Figure 6.25 – Public IPs tab

17. Using the **+ Add** button, like in the second step, we will create a new group using the **Add group** button. Call this item `group - Public IPs`, which will contain two resources of type query to render the required information. The first query item can be reused from the one we already made before for the **Overview** tab, which only collected the number of orphaned public IPs. The second item that we include here is to show us the details of the orphaned public IPs by projecting the `ResourceName`, `ResourceGroup`, `Location`, `Subscription`, `Type`, `AllocationMethod`, `Tags`, and `Details` fields.

    To do that, click the **+ Add** button again and select the **Add query** type. In the query cell, include the following query:

```
resources
| where type == "microsoft.network/publicipaddresses"
| where properties.ipConfiguration == "" and properties.
natGateway == "" and properties.publicIPPrefix == ""
| extend Details = pack_all()
| project ResourceName=id, ResourceGroup=resourceGroup,
Location=location, Subscription=subscriptionId,
Type=tostring(sku.name), AllocationMethod=tostring(properties.
publicIPAllocationMethod), Tags=tags, Details
```

    You can see the details in the following figure:

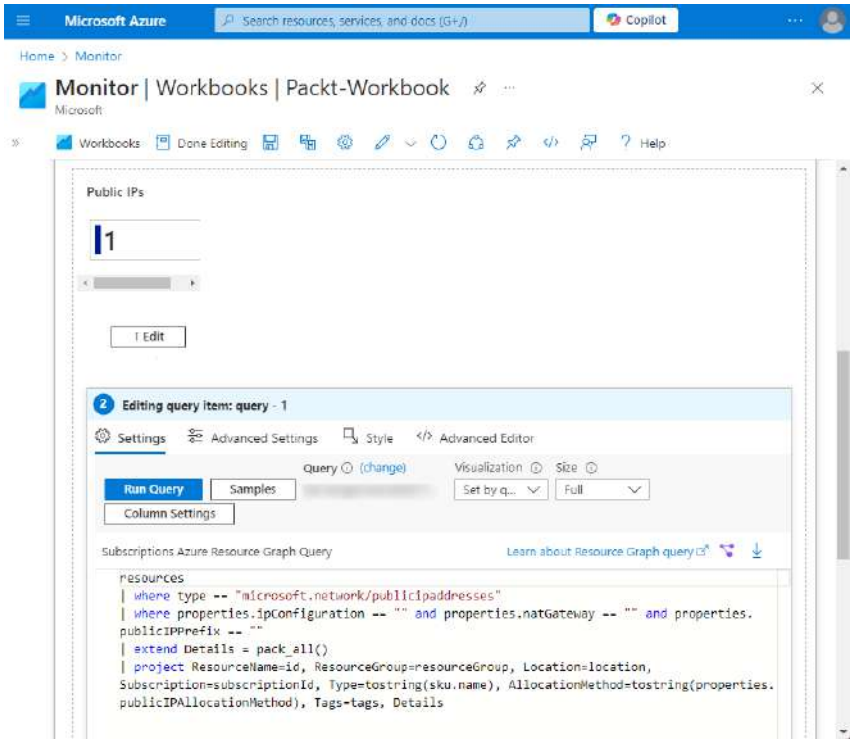

Figure 6.26 – The second query item in the Public IPs tab link

18. Under **Visualization**, set **Set by query** and **Full** as **Size**. In the **Edit column settings**, use the settings shown in the following screenshot.

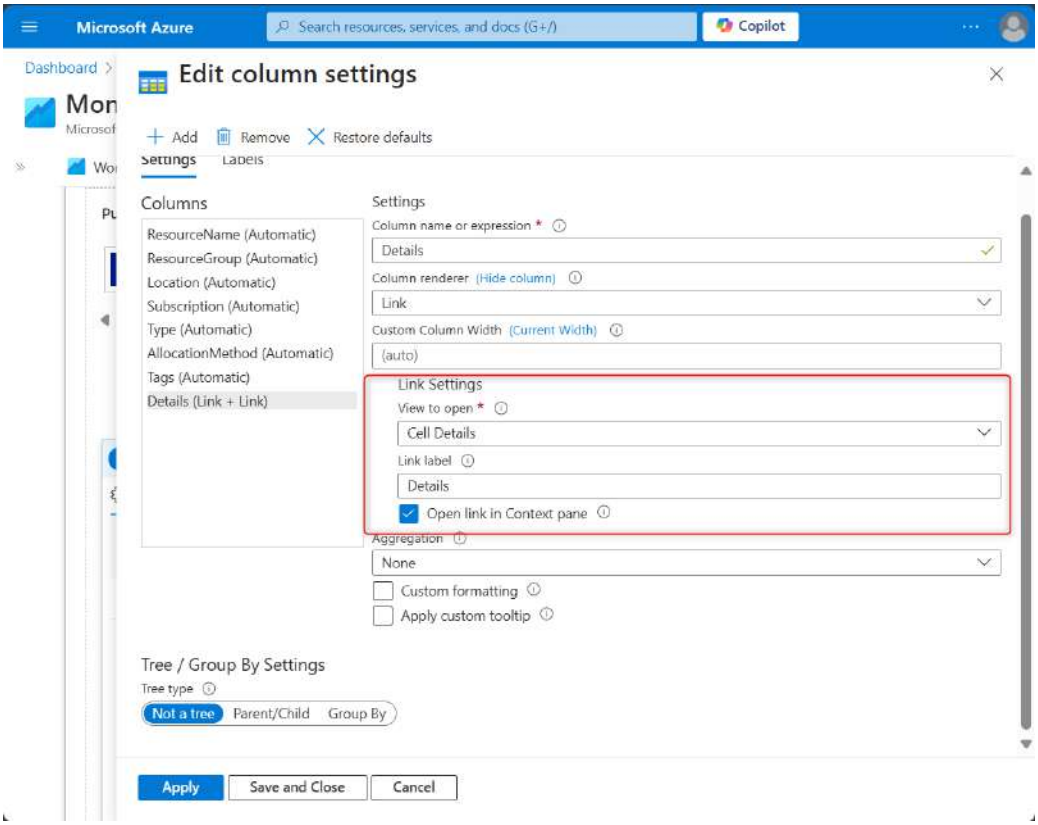

Figure 6.27 – The query item Resource Graph column settings

By using **Edit column settings**, we can display the query result in a friendlier and more customized format, including links to the *resource overview* and its details. Note that the **Details** pane is the result of applying the `pack_all()` operator, which creates a `dynamic property bag` object from all the columns of the tabular expression, and setting **View to Open** with the **Cell Details** value will open a view of details that shows all the values in a cell. Finally, checking the **Open link in context panel** box will only open a new view in a pop-up context panel, on the side, rather than a full view, as shown in the following screenshot.

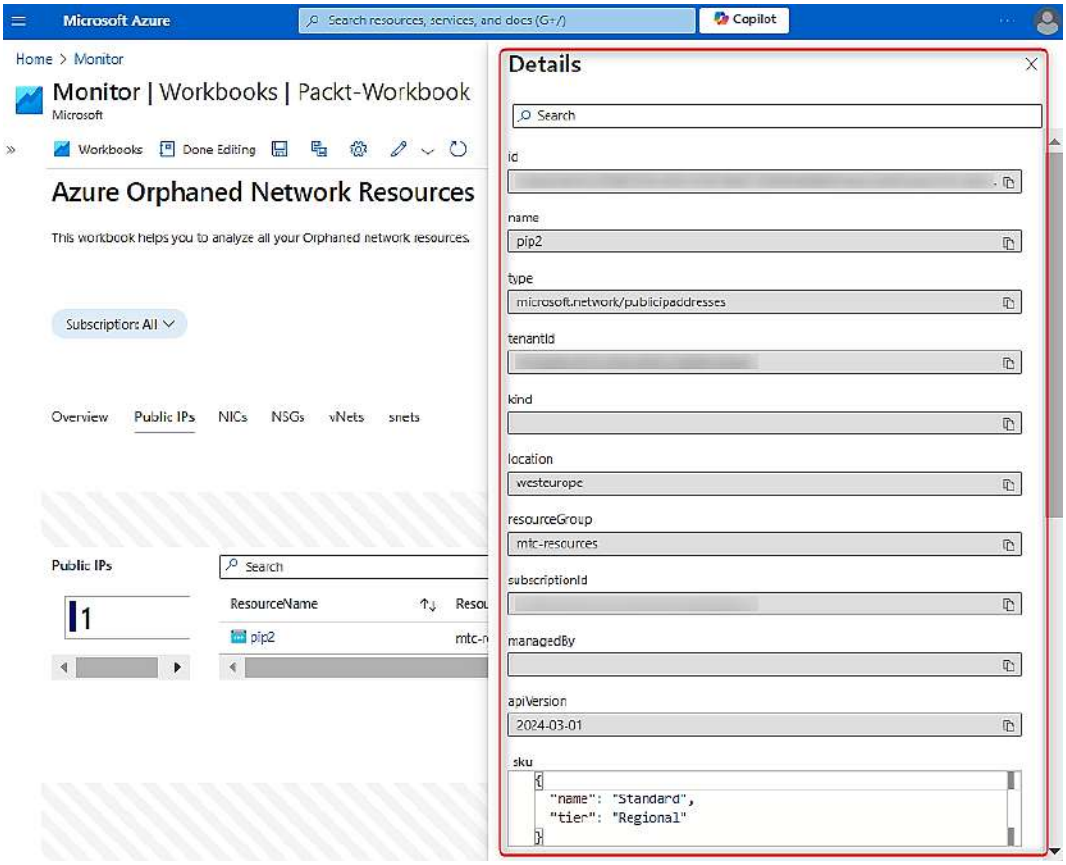

Figure 6.28 – The Public IPs tab visualization

You can replicate the same configuration steps on the rest of the tab links, for NICs, NSGs, vNets, and snets, to see your full report.

19. Finally, save all your changes using the **Save** option, and fill in all the required parameters – **Title**, **Subscription**, **Resource group**, and **Location**.

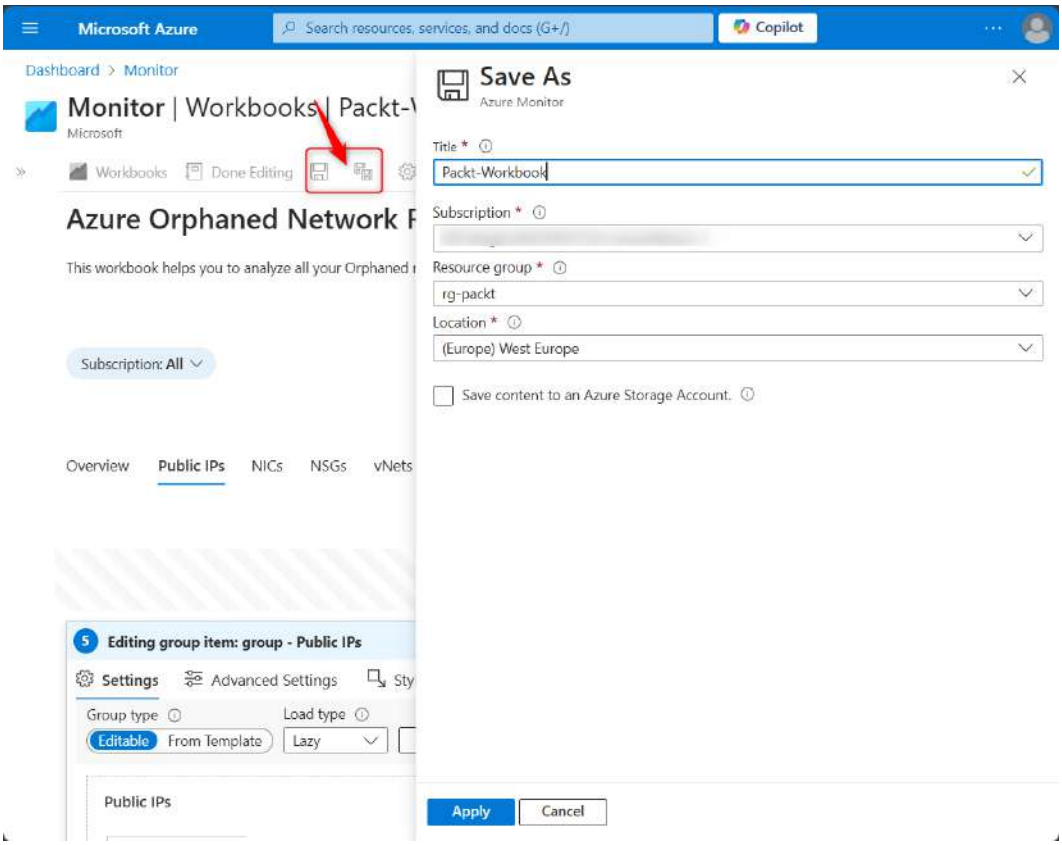

Figure 6.29 – Saving an Azure Workbook

With the previous steps, we explained the process to create an Azure Workbook, but there may be cases in which you simply want to import another user's workbook to modify it or apply it without changes within Azure Monitor. If so, as shown in the following figure, workbooks can be exported to an external repository by clicking on the button shown in the following figure.

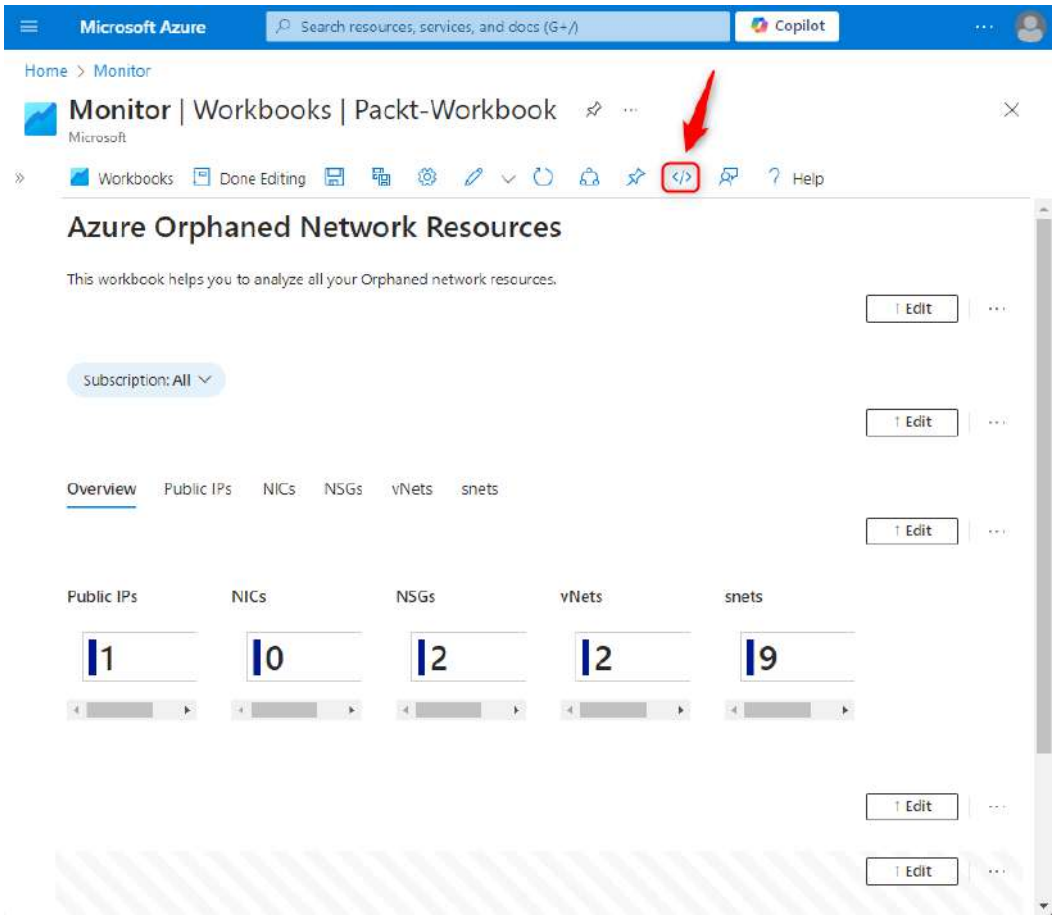

Figure 6.30 – Exporting an Azure Workbook

You can then copy the code from the workbook you just created, paste it into a Visual Studio Code file, save it in the gallery template format (`.workbook`) or ARM template format (`.json`), and then upload it to a GitHub repository.

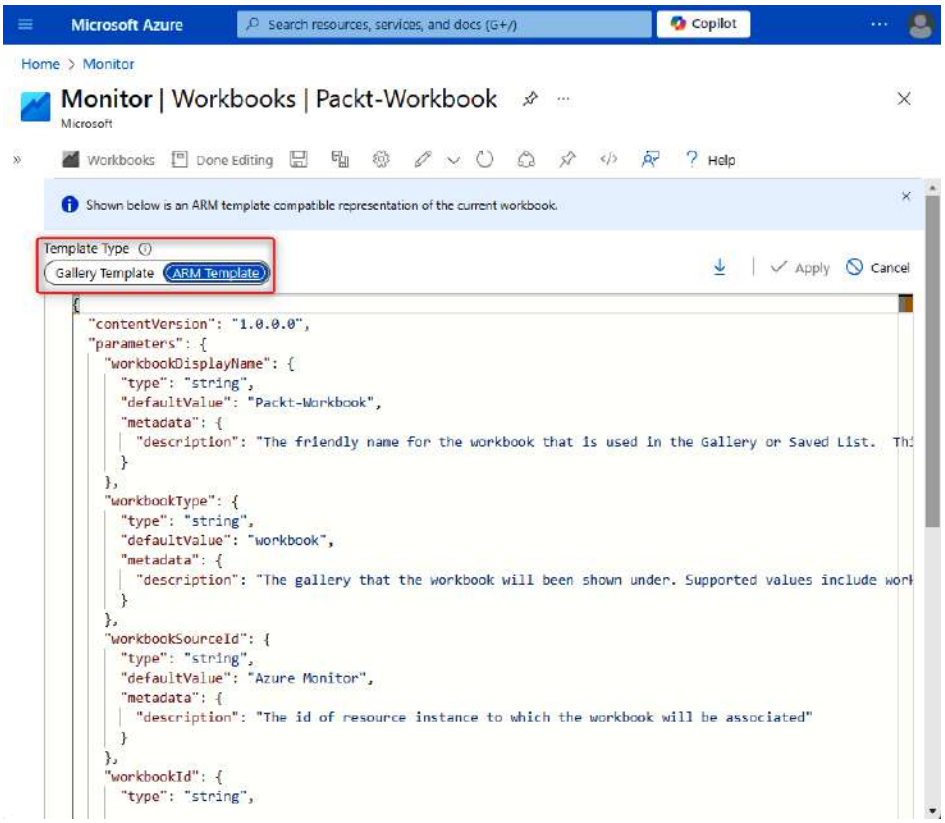

Figure 6.31 – Exporting an Azure Workbook in the JSON format

We have exported this workbook in both formats, and you can find it in the repository (`https://github.com/PacktPublishing/Cloud-Observability-with-Azure-Monitor/tree/main/chapter06`) for further customization. However, if you want to import a more detailed workbook about identifying orphaned resources, we recommend that you import and customize the following workbook: `https://raw.githubusercontent.com/dolevshor/azure-orphan-resources/main/Workbook/Azure%20Orphaned%20Resources%20v2.0.workbook`.

This hands-on tutorial has guided you through the process of creating a custom monitoring workbook in Azure to manage resource usage, detect misconfigurations, and simplify operations by identifying orphaned network resources. The process involved starting with an empty workbook template, adding text to describe its purpose, and creating all the different blocks to display different types of orphaned resources.

By completing this lab, you have created a detailed and functional Azure monitoring workbook that will help you learn the basics to continue creating more complex workbooks, aligned with your requirements and business needs.

Summary

In this chapter, we covered the visual side of monitoring with Azure, exploring tools and services that allow you to create intuitive and actionable visual representations of log and metric data.

We then moved on to the Azure visualization tools you can use to visualize collected data, such as Azure Monitor Insights, Azure Workbooks, Azure dashboards, Azure Managed Grafana, and Microsoft Power BI on Azure. Each of these tools has specific features and use cases that cater to different monitoring needs and scenarios.

The chapter also highlighted the importance of criteria when choosing the right visualization tool. The choice of the most suitable visualization tool depends on various factors, such as the type and source of the data, the format and level of detail of the visualization, the target audience, and the purpose of monitoring. The chapter provided a guide to compare and select the optimal tool based on these criteria.

Finally, we provided a lab practice that demonstrates how to create a custom Azure Workbook to visualize monitoring data from Azure resources. The workbook uses KQL queries, parameters, and different types of visualization to create a comprehensive and interactive dashboard.

As we conclude our introduction to Azure Monitor visualization tools, we will now turn our attention in the next chapter to Application Insights, part of Azure Monitor, a powerful tool that provides extensive observability and performance monitoring capabilities for your applications.

Further reading

Here, you can find links to expand your knowledge about the specific concepts not fully covered in this book but referenced in this chapter:

- [1] Azure Monitor – creating a dashboard: `https://learn.microsoft.com/en-us/azure/azure-portal/azure-portal-dashboards`.
- [2] Integrating Log Analytics with Power BI: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/log-powerbi`.
- [3] Monitoring Azure services and applications by using Grafana: `https://learn.microsoft.com/en-us/azure/azure-monitor/visualize/grafana-plugin`.

7

Application Observability and Performance Monitoring with Application Insights

In the previous chapters, we explored the features that Azure Monitor offers as a robust framework for monitoring, diagnosing, and optimizing cloud environments. With this foundational knowledge in place, we will now shift our focus to the specialized capabilities of Azure Monitor for **Application Performance Monitoring** (**APM**). **Application Insights**, part of Azure Monitor, is a powerful tool that provides extensive observability and performance monitoring capabilities for your applications. This chapter covers the details of instrumenting your code, implementing diagnostics, applying tracing techniques, and leveraging logging best practices. By the end of this chapter, you will have the skills to gain deep insights into your applications, ensuring optimal user experiences and maintaining the health of your applications.

In this chapter, we're going to cover the following main topics:

- Understanding Application Insights fundamentals
- Instrumenting code for monitoring
- Application Health monitoring through out-of-the-box experiences
- Diagnostics implementation for application health
- Logging and tracing for deep insights
- Ensuring optimal user experiences through observability

Technical requirements

To follow along with all the examples in this chapter, we recommend downloading the associated files available in the GitHub repository of the book. You will find the relevant files in the `chapter07` folder (`https://github.com/PacktPublishing/Cloud-Observability-with-Azure-Monitor/tree/main/chapter07`). The Azure CLI will be used to deploy and configure the resources. An integrated development environment such as Visual Studio Code is recommended to compile and execute the examples.

You will also need an Azure subscription with enough permissions to deploy a new Application Insights instance.

Understanding Application Insights fundamentals

Azure Application Insights, now a component of Azure Monitor, started as a standalone APM service in Azure. APM is a critical practice focused on ensuring that software applications meet performance standards and provide a high-quality user experience. This involves monitoring and managing their performance and availability.

In the context of Application Insights, APM involves several key activities:

- **Performance monitoring**: Continuously tracking the performance of applications, including response times, throughput, and resource utilization, to ensure they meet predefined performance criteria
- **Anomaly detection**: Identifying unusual patterns or anomalies in application behavior that could indicate potential performance issues or failures
- **Diagnostics and troubleshooting**: Providing detailed insights and diagnostic data to help developers and IT professionals troubleshoot and resolve performance problems and errors
- **User experience monitoring**: Understanding how end users interact with the application and identifying areas where the user experience can be improved
- **Optimization**: Using the insights gained from monitoring and diagnostics to optimize application performance, enhance reliability, and improve resource efficiency

Application Insights is designed to automatically collect a wide range of **telemetry data** from your application. This data includes performance metrics, request rates, response times, dependency data, exceptions, and custom events.

One of the main attractions of Application Insights is its ability to automatically instrument various application platforms and languages, such as .NET, Java, and Node.js. This means that without significant code changes, you can start collecting valuable telemetry data from your applications.

It also offers extensive manual instrumentation capabilities that allow developers to collect custom telemetry data for more detailed insights into their applications. This ensures that the monitoring and

diagnostic data is aligned with your specific needs, providing actionable insights that help maintain and improve application performance and reliability.

Application Insights also excels in its analytical capabilities. It provides pre-built and customizable dashboards to help visualize this data effectively. For diagnostics, Application Insights integrates seamlessly with Visual Studio, streamlining the process of debugging and performance analysis, although it is not required. It is designed to be scalable and flexible, accommodating applications of any size, from small web apps to large distributed systems, with configuration options that can be adjusted to meet your specific monitoring needs.

Why use Application Insights?

Application Insights offers a holistic view of your application's performance, helping you quickly identify and resolve issues. By integrating Application Insights with your applications, you can significantly improve their performance by identifying and optimizing performance bottlenecks and monitoring resource usage to ensure efficient scaling. This leads to an enhanced user experience, as you can track user interactions and behavior to better understand how your application is being used, ensuring it meets user expectations for response times and reliability. Moreover, Application Insights helps ensure the reliability of your application by detecting and diagnosing errors and failures promptly, allowing you to set up proactive alerts and notifications to address issues before they impact users.

Let's continue now by going into more detail about the information that Application Insights can collect, and the instrumentation options it provides.

Instrumenting code for monitoring

Instrumentation is a crucial process for monitoring and understanding the performance and behavior of your application in a production environment. By adding specific code to your application, you can collect detailed diagnostic data, including performance metrics, error logs, and trace information.

The Application Insights data model is aligned with the three pillars of observability discussed in *Chapter 1*. It supports **distributed tracing**, **metrics**, and **logs** as telemetry data types. Let's explain them in more detail within the context of Application Insights.

Distributed tracing is a method used to track the flow of requests through various services and components in a distributed system. It provides a detailed view of how different services interact with each other and helps in understanding the end-to-end execution flow of a request.

In Application Insights, distributed tracing works by capturing and correlating telemetry data across different components and services. Each request is assigned a unique identifier, known as an operation ID, which is propagated through all the components involved in handling the request. This enables Application Insights to stitch together a comprehensive trace that shows the journey of the request from start to finish.

For example, if a user request involves a web frontend, a backend API, and a database, distributed tracing will capture telemetry data at each stage. It will show how long the request spent in each component, highlight any errors or performance bottlenecks and provide insights into the overall latency of the request. This is invaluable for diagnosing complex issues in microservices architectures and ensuring that each part of the system is performing optimally.

Metrics are quantitative measures that provide insights into various aspects of your application's performance and health. Like Azure Monitor, Application Insights collects a range of built-in metrics, and developers can also define custom metrics to track specific performance indicators.

It is possible to also define custom metrics to track application-specific performance indicators, such as the time taken to process orders, the number of user registrations, or the average transaction amount. These metrics can be collected using the telemetry client provided by the Application Insights SDK and can be analyzed and visualized using the Application Insights portal.

Regarding metrics, the service defines two types:

- **Standard metrics**, which refers to information aggregated during the collection phase at one-minute intervals. These metrics are stored as time series and are recommended for building dashboards and generating real-time alerts. They provided the minimum, maximum, and standard deviation values.
- **Log-based metrics**, where all events are stored without previous aggregation, providing a detailed repository for analytics and diagnosis. These metrics are translated into Kusto queries from stored events behind the scenes to provide the results.

Lastly, we have logs. This refers to a detailed record of events that occur within your application. They provide a narrative of the application's activities, capturing information about errors, warnings, informational messages, and debugging details.

In Application Insights, logs are collected through various logging frameworks integrated with the Application Insights SDK, such as Log4j for Java applications, and NLog or ILogger for .NET applications. Developers can log custom messages that include contextual information about the state of the application at specific points in time.

Logs are essential for troubleshooting and diagnosing issues. They provide a granular view of the application's operations, enabling developers to pinpoint the root cause of problems and understand the context in which they occur. Logs can be searched, filtered, and analyzed using the Application Insights portal, providing powerful tools for identifying and resolving issues.

Application Insights supports two options to collect all this information, as previously mentioned: **automatic instrumentation** and **manual instrumentation**. Let's understand the main differences between them.

Differences between automatic instrumentation and manual instrumentation

Automatic instrumentation refers to the process where Application Insights automatically collects telemetry data from your application without requiring code changes. This is achieved through the integration of the Application Insights agent into the environment where your application is running. Once installed, it collects and sends a variety of telemetry data to the Application Insights resource. It auto-collects the same signals out of the box as the ones provided by the SDK.

Automatic instrumentation typically includes the following:

- **Request tracking**: Automatically logs HTTP requests made to your application, including response times, status codes, and URLs
- **Dependency tracking**: Automatically tracks calls to external services and resources, such as databases, REST APIs, and third-party services
- **Exception tracking**: Automatically captures unhandled exceptions and logs detailed information, including stack traces and error messages
- **Performance counters**: Collects data on CPU usage, memory consumption, and other performance metrics from the host environment
- **Live metrics**: Provides real-time metrics on request rates, response times, and failure rates

The primary advantage of automatic instrumentation is its simplicity and ease of use. It provides a comprehensive set of default telemetry data with minimal configuration, enabling developers to quickly start monitoring their applications without extensive setup. Access to application source code is not needed, and special configuration changes are not required either.

However, not everything is perfect. Automatic instrumentation supports a limited set of environments, resource providers, and programming languages. The list of supported scenarios is growing with the passage of time; for the most updated list, we recommend reviewing the official documentation [1] linked in the *Further reading* section.

Manual instrumentation, on the other hand, involves explicitly adding code to your application to collect custom telemetry data. This approach gives developers fine-grained control over what data is collected and how it is logged. Using the Application Insights SDK, developers can log custom events, metrics, traces, and exceptions that are specific to their application's business logic and performance requirements.

Key aspects of manual instrumentation include the following:

- **Custom events**: Logging specific user actions or business operations, such as user logins, purchases, or feature usage
- **Custom metrics**: Tracking performance indicators that are relevant to the application's context, such as processing times for specific tasks or resource usage for particular operations

- **Custom traces**: Adding trace statements to follow the execution flow of the application, which is useful for debugging and understanding complex interactions
- **Custom exceptions**: Logging handled exceptions with additional context, providing more detailed information for diagnosing issues
- **Telemetry processors**: Creating custom telemetry processors to modify or enrich telemetry data before it is sent to Application Insights. This can include adding custom properties or tags to telemetry items for better filtering and analysis

Both scenarios are not mutually exclusive, they can be used in tandem to provide a robust monitoring solution. Automatic instrumentation ensures that critical telemetry data is always collected with minimal effort, while manual instrumentation allows for detailed and specific insights tailored to the application's unique requirements. This combination enables comprehensive monitoring and diagnostics, helping developers maintain high performance and reliability in their applications.

Let's focus now on the two alternatives provided by Application Insights to instrument our applications and collect the required information.

Alternatives for collecting telemetry data in Application Insights

Initially, Application Insights started providing a set of proprietary native service SDKs designed to seamlessly integrate with various application platforms and languages. Those SDKs provided a simplified way to automatically collect telemetry data, such as performance metrics, request and dependency tracking, exceptions, and log data, with minimal configuration from the developer's perspective. The native SDKs were tailored to specific environments and frameworks, offering deep integration and optimized performance.

For example, the Application Insights SDK for .NET can be easily integrated into ASP.NET applications. Once integrated, it automatically tracks HTTP requests, dependencies, exceptions, and performance metrics. Similarly, there are SDKs available for Java, Node.js, and Python, each providing similar out-of-the-box capabilities.

However, this does not always properly align with the requirements of developers for an open standard implementation that can interoperate with other services and tools. Using native SDK agents can tie you to services provided only by Azure, but open standards such as **OpenTelemetry** provide a way to collect and export data from your applications so it can be integrated into different APM platforms as required.

OpenTelemetry is an open source observability framework that provides a unified set of APIs, libraries, agents, and instrumentation to enable the collection of distributed traces and metrics. It is a vendor-neutral standard, supported by many observability platforms, including Application Insights. OpenTelemetry aims to provide a consistent and flexible approach to instrumentation across different programming languages and platforms and it is sponsored by the **Cloud Native Computing Foundation** (**CNCF**), part of the Linux Foundation.

Microsoft is investing in OpenTelemetry through their **Azure Monitor OpenTelemetry distro**. It is a custom distribution of the OpenTelemetry project provided by Microsoft to help developers monitor and observe their applications running on Azure more easily. It simplifies its integration with Azure Monitor. It provides several benefits:

- **Vendor neutrality**: It is supported by multiple observability platforms, allowing for flexibility and avoiding vendor lock-in. This is particularly useful if you plan to use multiple monitoring tools or switch providers in the future.
- **Consistency across platforms**: It provides a consistent API for instrumentation, making it easier to implement and maintain telemetry collection across different services and environments.
- **Customization and extensibility**: It offers extensive customization options, allowing developers to tailor telemetry collection to their specific needs. You can define custom spans, metrics, and attributes to capture detailed insights into your application's performance and behavior.

While Azure Monitor OpenTelemetry is the alternative to Application Insights, there is still a gap between the features it supports and the classic Application Insights SDKs. The main reasons to choose one or another are:

- **Ease of use**: Native service SDKs are generally easier to set up and require less configuration, making them ideal for quickly getting started with Application Insights. OpenTelemetry may require more initial setup and configuration but offers greater flexibility and standardization.
- **Flexibility**: OpenTelemetry provides a vendor-neutral approach and allows for more customization, making it suitable for complex or multi-cloud environments. Native SDKs are optimized for specific platforms and offer seamless integration with Application Insights.
- **Standardization**: OpenTelemetry promotes a standardized approach to observability, which can simplify instrumentation across diverse environments. Native SDKs, while comprehensive, are specific to Application Insights and may not provide the same level of cross-platform consistency.

In the end, the final decision between the two depends on your specific needs, your existing infrastructure, and your plans for observability.

Application Health monitoring through out-of-the-box experiences

In this section, we will explore the various out-of-the-box experiences provided by Application Insights, focusing on how they can be leveraged to monitor and maintain the health of your applications. Azure's Application Insights provides a set of out-of-the-box features designed to offer comprehensive monitoring and diagnostic capabilities without requiring extensive configuration or customization.

These integrated features automatically collect and display telemetry data from your application. As an overview, they are the application dashboard, **Application map**, **Failures** view, **Performance** view, **Availability** view, **Live metrics**, and user behavior analytics. Each of these features serves a specific purpose in the monitoring and diagnostic process. To provide a practical understanding, we will demonstrate their implementation and usage through a custom .NET Core application built from scratch. This approach will help you see how Application Insights can be integrated into a real-world application and how its powerful features can be used to monitor and improve your application's health.

Let's understand each of these features at a high level:

- **Application dashboard**: The application dashboard provides a high-level overview of your application's **Key Performance Indicators** (**KPIs**), such as request rates, response times, and failure rates. This dashboard is designed to give you a quick snapshot of your application's health, allowing you to monitor its overall performance immediately.

  We will start by setting up a new .NET Core web application and integrating Application Insights. By running the application and generating some traffic, we will observe how the application dashboard automatically starts collecting and displaying key metrics such as request rates, response times, and failure rates. This hands-on example will show you how to monitor the overall performance of your application using the dashboard.

- **Application map**: **Application map** offers a visual representation of your application's architecture, highlighting the interactions between various components and services. This map helps you understand dependencies, identify bottlenecks, and visualize the flow of requests through your application.

  We will explore **Application map** by configuring our .NET Core application to interact with multiple services and dependencies. We will visualize these interactions in **Application map**, demonstrating how it helps to understand the architecture, dependencies, and potential bottlenecks within the application. This will provide a clear picture of how different components interact and how to use the map for troubleshooting.

- **Failures view**: The **Failures** view is essential for diagnosing and resolving issues. It aggregates data on exceptions, failed requests, and other errors, providing detailed information about the root causes of failures. This view enables you to quickly identify problem areas and take corrective actions.

  To highlight the **Failures** view, we will introduce some controlled exceptions and failures in our application. We will then use the **Failures** view to diagnose these issues, showing how to access detailed error information, trace the root causes, and resolve the problems. This practical example will illustrate the importance of proactive error monitoring and diagnostics.

- **Performance view**: The **Performance** view focuses on the response times and performance metrics of your application. It helps you identify slow endpoints, analyze request performance, and optimize your application to improve its efficiency and user experience.

In the *The Performance view* section, we will focus on optimizing the application's performance. By analyzing response times and request performance, we will identify slow endpoints and demonstrate how to optimize them for better efficiency. This will include implementing custom metrics to monitor specific performance indicators and using the insights gained to improve the application's responsiveness.

- **Availability view**: The **Availability** view monitors your application's uptime and responsiveness through synthetic tests. These tests simulate user interactions from multiple locations worldwide, providing insights into the global availability of your application and helping you ensure consistent performance for all users.

  We will set up synthetic availability tests to monitor the application's uptime and responsiveness from different geographic locations. This will involve configuring the tests to run at regular intervals and analyzing the results in the **Availability** view. This example will show how to ensure your application remains available and performant for users worldwide.

- **Live metrics**: **Live metrics** provides real-time monitoring of your application's telemetry data. This feature is particularly useful during deployments or high-traffic periods, offering immediate feedback on the current state of your application and allowing for rapid issue detection and resolution.

  During a simulated deployment or high-traffic event, we will use **Live metrics** to monitor the application in real time. This will provide immediate feedback on the application's state, allowing us to detect and address any emerging issues promptly. This live demonstration will highlight the value of real-time monitoring for maintaining application stability.

- **User behavior analytics**: User behavior analytics offers insights into how users interact with your application. It tracks user sessions, page views, navigation paths, and engagement metrics, helping you understand user behavior and improve the overall user experience.

  Finally, we will enable user behavior analytics to track user sessions, page views, and navigation paths within the application. By analyzing this data, we will gain insights into user interactions and behaviors, helping us to make informed decisions to enhance the user experience. This practical example will illustrate how to use user analytics to improve application usability and engagement.

Let's start then building our application from scratch.

Preparing our environment and an example .NET Core application

In this section, we will explore how to implement Application Insights in a custom .NET Core application built from scratch and running on AlmaLinux. This practical approach will help you understand the concepts discussed earlier by providing a step-by-step guide to instrumenting your application for monitoring, diagnostics, and performance optimization. By the end of this section,

you will have hands-on experience in setting up Application Insights, collecting telemetry data, and leveraging the service's powerful out-of-the-box features to maintain and enhance your application's health and performance.

Let's start configuring our environment:

1. Before starting to write our application, we need to install .NET Core SDK. Open your terminal and execute the following commands to install the .NET SDK:

```
curl -sSL -O https://packages.microsoft.com/config/alma/8/
packages-microsoft-prod.rpm
sudo rpm -i packages-microsoft-prod.rpm
sudo dnf update
sudo dnf install dotnet-sdk-8.0
```

   For other Linux distributions, follow the installation instructions on the Microsoft .NET website (`https://dotnet.microsoft.com/en-us/download/dotnet/5.0`).

2. Verify the installation by running the following:

```
dotnet --version
```

3. Install Visual Studio Code from `https://code.visualstudio.com/Download` and open a terminal in Visual Studio Code by selecting **View | Terminal**.

4. Create a new .NET Core application:

```
dotnet new webapp -n AppInsightsExample
```

   You should see something like the following:

```
[                book]$ dotnet new webapp -n AppInsightsExample

Welcome to .NET 8.0!
---------------------
SDK Version: 8.0.105

----------------
Installed an ASP.NET Core HTTPS development certificate.
To trust the certificate, view the instructions: https://aka.ms/dotnet-https-linux

----------------
Write your first app: https://aka.ms/dotnet-hello-world
Find out what's new: https://aka.ms/dotnet-whats-new
Explore documentation: https://aka.ms/dotnet-docs
Report issues and find source on GitHub: https://github.com/dotnet/core
Use 'dotnet --help' to see available commands or visit: https://aka.ms/dotnet-cli
--------------------------------------------------------------------------------
The template "ASP.NET Core Web App (Razor Pages)" was created successfully.
This template contains technologies from parties other than Microsoft, see https://aka.ms/aspnetcore/8.0-third-party-notices for details.

Processing post-creation actions...
Restoring                    /AppInsightsExample/AppInsightsExample.csproj:
  Determining projects to restore...
  Restored                   /AppInsightsExample/AppInsightsExample.csproj (in 481 ms).
Restore succeeded.
```

Figure 7.1 – Creation of a new web application using the dotnet CLI

5. Open the main `Program.cs`.

> **Important note**
> If you are prompted to install the recommended C# Dev Kit, we suggest accepting it.

6. Test that everything has been set up properly by running your application and pressing *F5*. A browser will appear with a **Welcome** message.

Once our application is ready, it is time to set up an Application Insights resource in Azure and obtain the connection string. This connection string is used to configure your .NET Core application to send telemetry data to Azure Monitor. Let's create the required Application Insights resource:

1. Go to the Azure portal and sign in with your Azure account. After that, type `Application Insights` on the top search bar and select **Application Insights** from the resulting list.
2. The Azure Marketplace details web page will open. Click the **Create** button to begin setting up a new Application Insights resource. You will need to provide the following details:
    I. **Subscription**: Select the Azure subscription you want to use.
    II. **Resource Group**: Choose an existing resource group or create a new one.
    III. **Name**: Enter a unique name for your Application Insights resource.
    IV. **Region**: Select the Azure region where you want to host the Application Insights resource.
    V. **WORKSPACE DETAILS**: Choose your existing Log Analytics workspace created in previous chapters.
3. After providing all the necessary details, click **Review + create** to review your configuration. Then, click **Create** to create the Application Insights resource.
4. Once the resource is created, navigate to your Application Insights resource in the Azure portal. On the Application Insights resource overview page, you will find various options and information about your resource. In the **Overview** section, look for the **Connection String** option. This string is used to connect your web application to Application Insights. Copy the connection string to your clipboard. You will use this string to configure your .NET Core application.

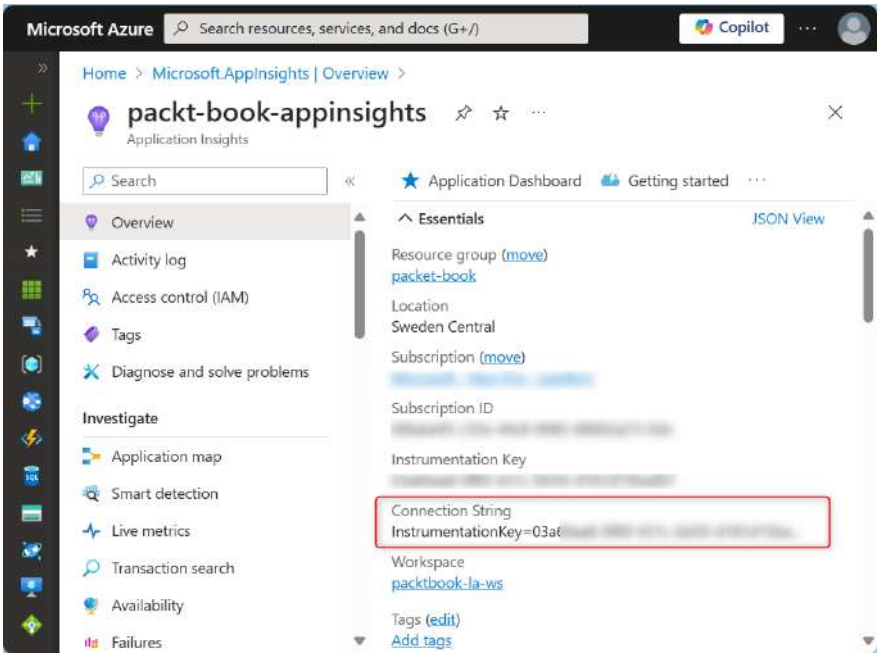

Figure 7.2 – Retrieving the connection string for your Application Insights instance

After both our application and our Application Insights instance are ready, let's start monitoring.

Diagnostics implementation for application health

This section will cover how to implement diagnostic tools and techniques using Application Insights. Before you can leverage Application Insights to monitor your .NET Core application, you need to install the necessary packages and configure your application to send telemetry data to Azure Monitor using the following steps:

1. In your terminal inside Visual Studio Code, run the following command in your project directory to add the `Microsoft.ApplicationInsights.AspNetCore` package. This SDK provides the necessary tools to collect telemetry data and send it to Azure Monitor:

```
dotnet add package Microsoft.ApplicationInsights.AspNetCore
```

2. After installing the SDK, you need to configure Application Insights in your application. Open the `Program.cs` file and add the Application Insights services after the `builder` variable:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddApplicationInsightsTelemetry();
builder.Services.AddRazorPages();
```

3. Configure the connection string to be able to submit data into Azure Monitor. It is recommended to specify the connection string in the `appsettings.json` configuration file.
4. With the SDK installed and configured, Application Insights will automatically start collecting basic telemetry data, such as request rates, response times, and dependency tracking. Run your application to generate some traffic. Navigate through your **Home** and **Privacy** pages to ensure that telemetry data is being collected. Try to open non-existing pages.
5. After that, open the **Live metrics** option on the left menu of an Application Insights resource in the Azure portal to verify that data is being received, as you can see in the following screenshot.

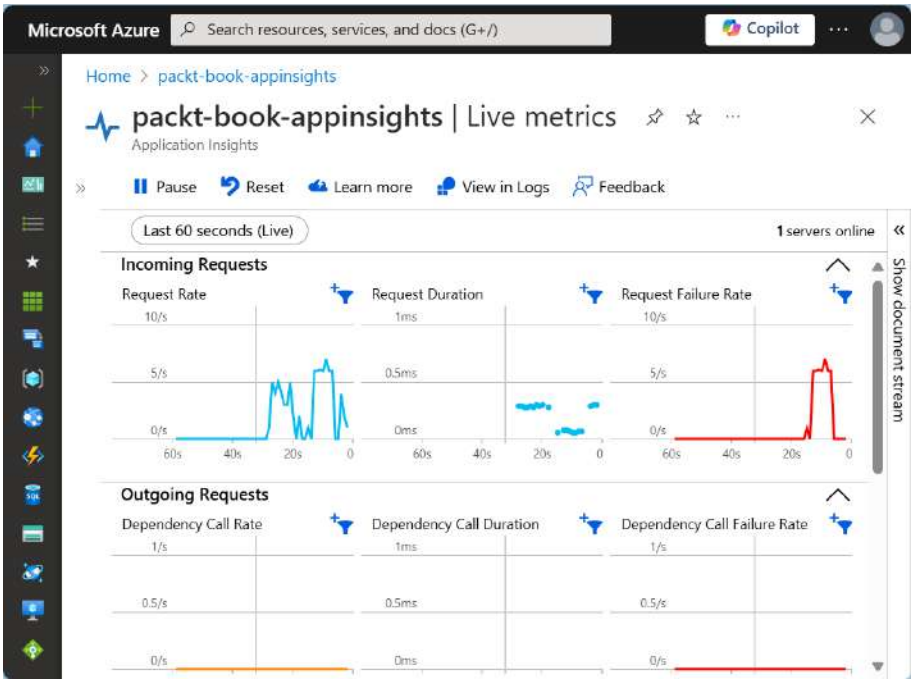

Figure 7.3 – View of the Live metrics dashboard with the real-time information

You can see information from **Incoming Requests** rendered inside the **Live metrics** interface. In these five steps, y you have downloaded the required SDK, configured it, and started collecting real-time metrics of your initial demo application.

The Live metrics view

As observed in *Figure 7.3*, the **Live metrics** view in Application Insights is a powerful feature that provides real-time monitoring of your application's performance and health. This out-of-the-box tool enables you to see live telemetry data for the last 60 seconds as it flows from your application, allowing for immediate insights and quick responses to emerging issues.

The view allows you to customize which metrics are displayed. You can add or remove metrics, adjust the display order, and set thresholds for alerts, tailoring the view to provide the most useful information.

It supports filtering, allowing you to drill down into specific aspects of your application's performance. In the following screenshot, you can see how clicking the filtering button opens the different options supported to customize the queries shown in this specific view.

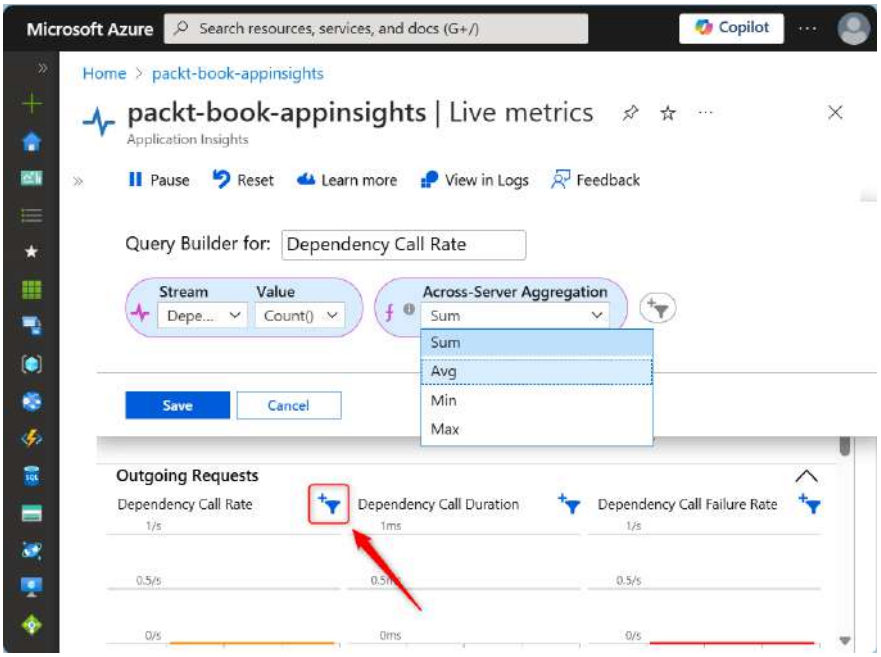

Figure 7.4 – Customizing your visualizations through filters

Live metrics is particularly valuable during application deployments or updates. It allows you to monitor the impact of changes in real time, ensuring that any new issues introduced by the deployment are quickly identified and addressed.

However, relying solely on **Live metrics** for a global monitoring strategy has significant limitations, particularly in terms of customization and time range. In contrast, the application dashboard offers a more comprehensive and customizable solution, making it essential for effective global monitoring.

The application dashboard

The application dashboard is a default feature. It aggregates various types of telemetry data into a single, cohesive view. It provides a high-level summary of the application's performance, reliability, and usage patterns. The dashboard is divided into several key sections, each focusing on a different aspect of the application's health: **Usage**, **Reliability**, **Responsiveness**, and **Browser** (referring to browser connectivity), as shown in the following screenshot.

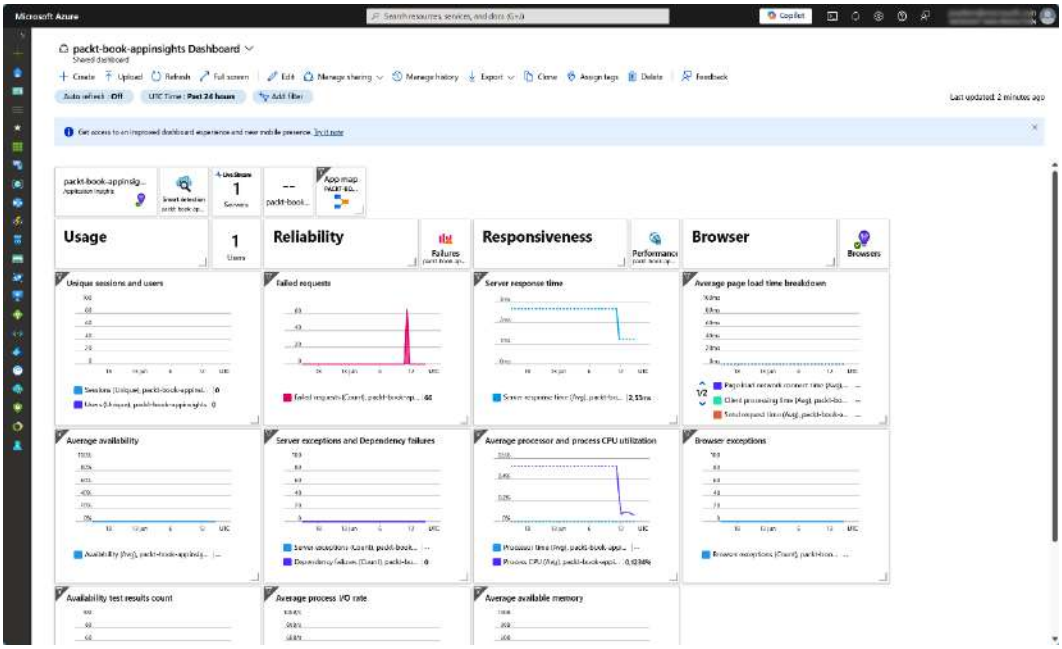

Figure 7.5 – A global overview of your dashboard

The application dashboard is built on top of the Azure dashboards feature described in *Chapter 6*. It offers extensive customization capabilities. You can create custom dashboards specific to your monitoring needs by selecting the exact metrics and visualizations you want to display. This includes combining different types of data, such as performance metrics, error rates, and user behavior analytics. Customizable dashboards allow you to focus on the most critical aspects of your application's health and performance, providing a more targeted and relevant monitoring experience.

Unlike **Live metrics**, the application dashboard supports historical data and long-term trend analysis. You can configure dashboards to display data over extended periods, such as days, weeks, or months. This historical perspective is essential for understanding performance trends, identifying recurring issues, and making data-driven decisions for capacity planning and optimization. Trend analysis helps in proactively addressing potential problems before they escalate, ensuring the long-term health and stability of your application.

Having explored the capabilities of **Live metrics** and the application dashboard, which provide a general and comprehensive view of your application's real-time telemetry and long-term performance trends, it's important to understand other aspects of your application's health. While these tools offer valuable insights at a high level, detailed analysis of performance and failures is important for diagnosing issues and optimizing your application's efficiency. This is where the **Performance** and **Failures** views come into play. These specialized views allow you to drill down into the specifics of how your application performs and help identify and resolve errors that could impact user experience and reliability.

The Failures view

The **Failures** view in Application Insights helps developers to identify, diagnose, and resolve issues within their applications.

It aggregates data related to all types of application failures, including unhandled exceptions, failed HTTP requests, and dependency failures. It presents this data in an organized and accessible manner, enabling you to quickly identify and address issues.

The application we deployed in the previous section contains a web page for the privacy policy at `http://localhost:5175/Privacy`. If you try to access `http://localhost:5175/Privacy2` or `http://localhost:5175/Privacy3`, as shown in the following screenshot, a failed request will be registered by Application Insights. In this case, we have tried to access this missing page multiple times and the HTTP `404` errors are properly registered.

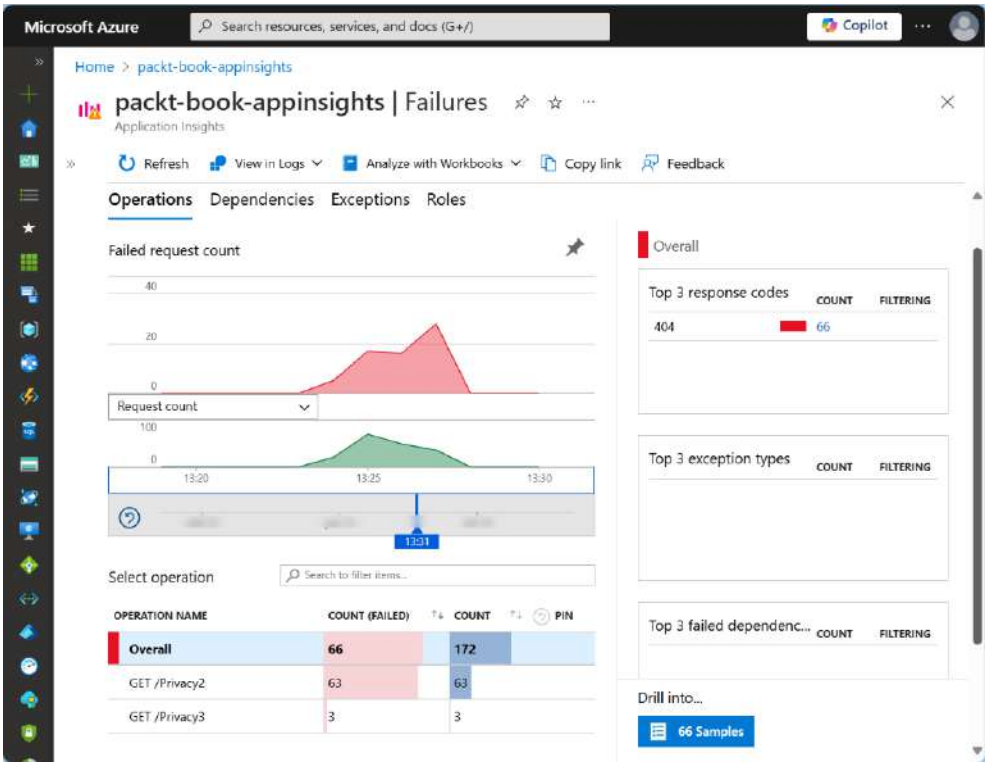

Figure 7.6 – General overview of the failures detected in your web application

The view provides insights into failed HTTP requests, including information about the request URLs, response codes, and failure rates. By analyzing this data, you can identify patterns in request failures, such as specific endpoints that are more prone to errors or high failure rates during peak usage times. Not only that, but it also provides tools for root cause analysis, allowing you to drill down into specific failures to uncover the underlying issues. You can trace the sequence of events leading up to a failure, examine related telemetry data, and identify potential causes and solutions.

If you click on the number of errors associated with our `404` failures, a list of all the samples collected by Application Insights will be shown on the right side. Select the suggested event and a details window will open to allow you to continue your root cause analysis.

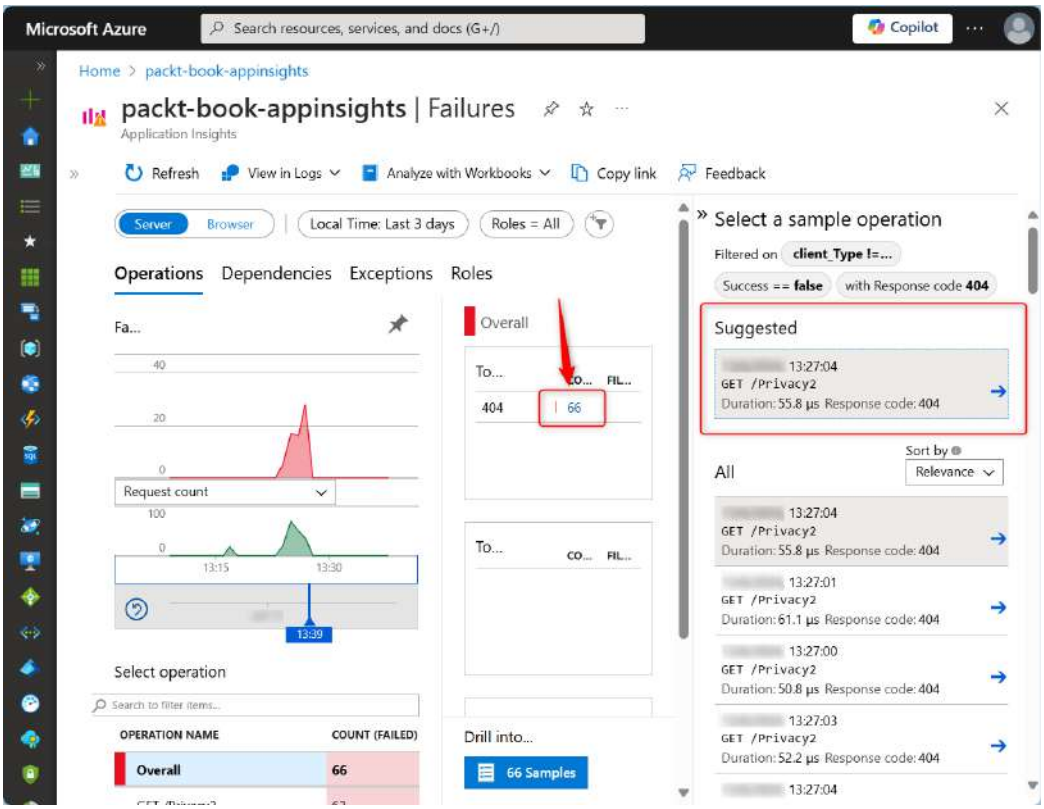

Figure 7.7 – List of the failures detected in your web application

This new window provides information about the specific event, such as the URL or the request time, as shown in the following screenshot.

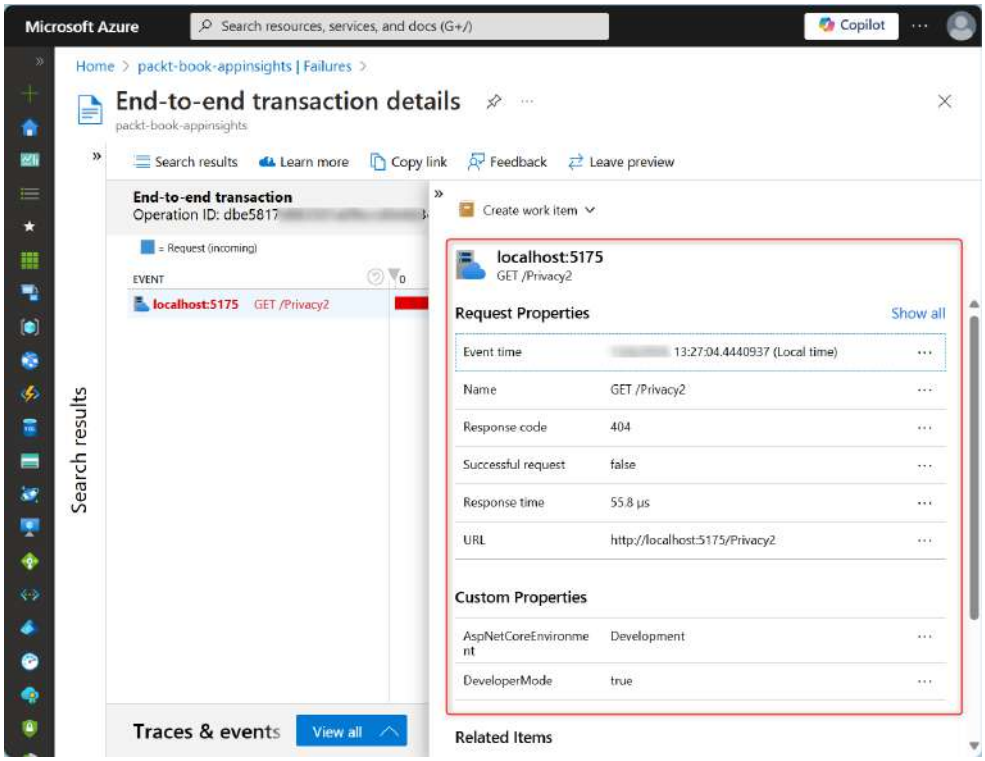

Figure 7.8 – Details of a failure detected in your web application

Not only that, but if you scroll down to the **Related Items** section, Application Insights suggests some relevant information for your analysis, such as the following:

- **Show what happened before and after this request in User Flows**
- **Show trend of this request over time**
- **Show trend of this request with this response code over time**
- **All available telemetry 5 minutes before and after this event**

However, the **Failures** view provides more information and details on top of that. It supports tracking exceptions. Application Insights captures detailed information about unhandled exceptions occurring in your application with their logs stack traces, error messages, and the context in which the exceptions occurred. This detailed information helps in understanding the root cause of the errors and provides a clear path for debugging and fixing the issues.

To test it, let's modify the **Privacy** web page to generate an exception when it is loaded. To do that, open the `Privacy.cshtml.cs` file and modify the `OnGet` method:

```
    public void OnGet(){
        throw new Exception("This is a test exception for Application
Insights");
    }
```

After that, execute your application again and access the **Privacy** page. An error page with all the details will be rendered because we are executing it inside a development environment. In a production environment, the exception will be handled appropriately with an error message.

After a few minutes, the information will already be populated into the **Exceptions** tab inside the **Failures** view, as shown in the following screenshot. A single exception in our **Privacy** page has been detected. If you select the sample available in the **Drill into…** section, a detailed view of our exception will be shown.

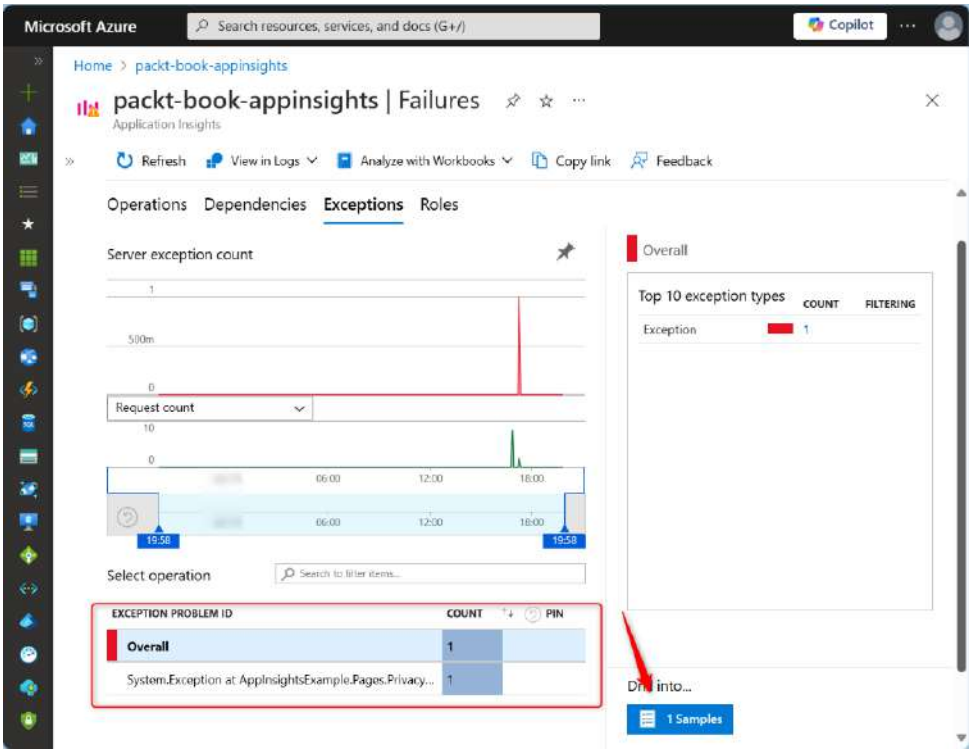

Figure 7.9 – General overview of the exceptions in your web application

It is possible to see the message we defined when introducing the exception inside our code, plus other detailed information about it, such as the exception type, where the exception was produced, or a stack trace with all the details, as shown in the following screenshot.

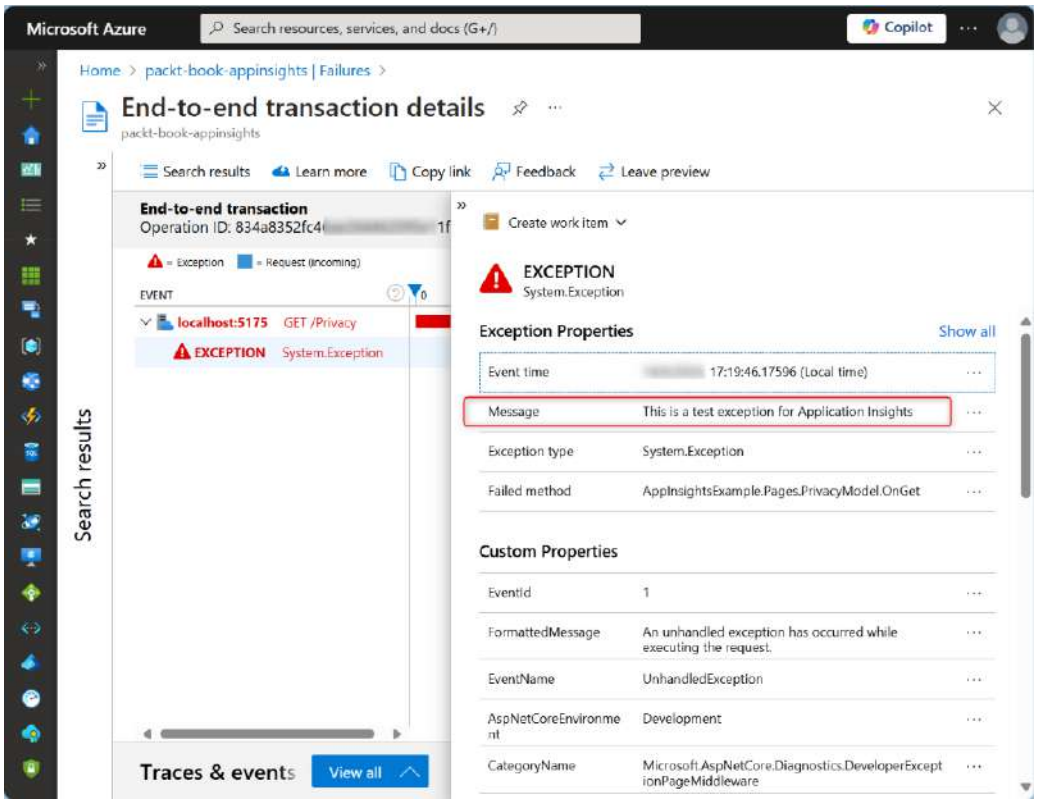

Figure 7.10 – Details of the exception in your web application

The **Failures** view also tracks dependency failures, such as external APIs, databases, and other services related to the application functionality. The **Failures** view tracks failures in these dependencies, offering details about the type of dependency, the failure rate, and the error messages returned. This helps in diagnosing issues related not only to your code but also to external services.

With this proactive approach in mind, let's transition from examining application failures to analyzing application performance. The **Performance** view in Application Insights provides detailed metrics and insights that allow you to optimize response times, resource usage, and overall application efficiency, ensuring a seamless and responsive user experience.

The Performance view

The **Performance** view in Application Insights is designed to provide detailed insights into the performance of your application. This out-of-the-box tool helps you monitor key performance metrics, identify bottlenecks, and optimize the overall efficiency of your application.

It aggregates performance-related telemetry data and presents it in a user-friendly format. It includes detailed metrics and visualizations that highlight the performance of various components and interactions within your application.

As shown in the following screenshot, the user experience is like the **Failures** view. You can track the response times of your application's requests. It provides detailed metrics on the average, median, and percentile response times, helping you understand how quickly your application is responding to user requests.

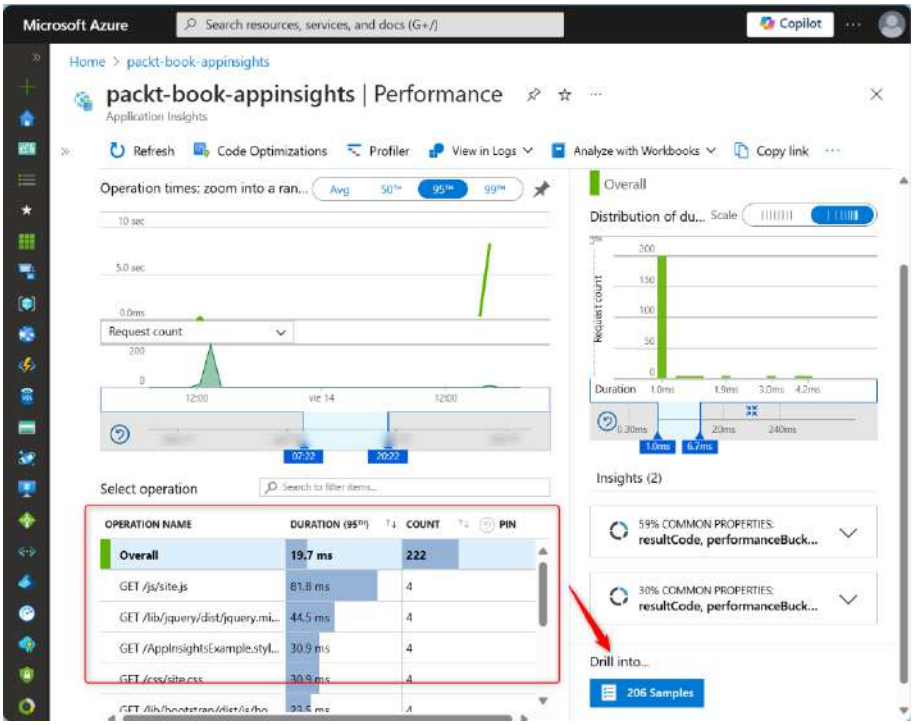

Figure 7.11 – General overview of the performance of your web application

As highlighted in the screenshot, this feature breaks down the performance of individual requests, showing the duration of each request and how it compares to others. By analyzing request performance, you can pinpoint specific operations that are slower and may need optimization through the **Drill into…** option, as before.

In this section, we discussed how capturing and analyzing diagnostic data, such as exceptions and performance metrics, can help identify and resolve issues that may impact your application's reliability and performance. While robust diagnostics are essential for understanding the overall health of your application, they primarily focus on reactive measures—identifying and addressing issues after they occur.

To complement these diagnostics, logging and tracing techniques offer a more proactive approach to application monitoring. Both allow you to follow the execution flow of your application, providing insights into how requests are processed and how services interact within your system. By integrating them with your diagnostics strategy, you can not only detect and diagnose issues more effectively but also gain a comprehensive view of your application's behavior and performance.

Let's now move on to look at the details of the logging and tracing techniques available in Application Insights and explore how they can enhance your application's visibility and troubleshooting capabilities.

Logging and tracing for deep insights

Logging is a fundamental aspect of observability, as discussed in *Chapter 4*. Application Insights offers logging capabilities that enable you to capture detailed application logs, track custom events, and analyze log data to gain actionable insights.

Application Insights provides several options for capturing and analyzing application logs. These options include the following:

- **Integration with popular logging frameworks**: For example, Application Insights integrates seamlessly with ASP.NET Core, allowing you to capture logs generated by the application. By adding the Application Insights SDK to our ASP.NET Core project, you can automatically track logs. For applications using popular logging frameworks such as log4net and NLog, Application Insights offers integration capabilities that enable you to send log data directly to Azure Monitor. This integration allows you to leverage your existing logging setup while benefiting from the advanced analytics and visualization tools provided by Application Insights.
- **Custom logging**: Application Insights allows you to create and send custom telemetry data. You can define and track custom events, metrics, and traces that are specific to your application's logic. This flexibility ensures that you can capture the exact data you need to gain deep insights into your application's behavior. The `TelemetryClient` class provided by Application Insights enables you to log custom events programmatically. This API can be used to capture detailed contextual information, such as user actions, application states, and performance metrics.

Application Insights also uses KQL to query and analyze log data, as explained in *Chapter 4*. This powerful query language allows you to perform complex searches, aggregations, and analyses on your log data.

Application Insights' information is stored inside an Azure Log Analytics workspace, providing a centralized location for your log data from multiple sources. This integration enables you to correlate data across different services and applications, offering a global view of your environment. It is now time to apply logging to our example.

Configuring log collection in our application

Continuing with our example ASP.NET Core application, the logs Application Insights collects have the level **Warning** or above by default. There are six different log levels that can be used in your application, as defined in the following table:

Level	Value	Description
Trace	0	This provides the highest level of detail about what is happening in your application. You need to be careful when enabling this level due to the impact on your performance and the sensitive app information it could contain.
Debug	1	As its name suggests, this provides a useful level of detail while debugging and development. As with the previous log level, it generates a great amount of information.
Information	2	This is a standard log level with general information about what is happening inside your app.
Warning	3	This is useful for monitoring errors that don't provide a failure in your app but are considered abnormal or unexpected.
Error	4	As the name implies, this is useful for monitoring errors and exceptions not handled by the application that produced a failure in the ongoing operation.
Critical	5	This indicated failures at the application level that require direct consideration.

Table 7.1 – List of log levels available

Let's modify the default level to `Information` to show how you can customize log collection. The first step is modifying the `appsetings.json` file to overwrite the default behavior of Application Insights. As you can see in the following code, we have a new `ApplicationInsights` object inside the `Logging` object to specify that we are interested in `Information` and above log levels:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    },
    "ApplicationInsights": {
      "LogLevel": {
        "Default": "Information"
[…]
```

After that, the second step is to modify the `OnGet` method in the `Privacy.cshtml.cs` file by replacing the previous code that was generating an exception with the next code, which includes two types of traces – one generated by the `LogWarning` method with a log level of type `Warning` and the other one with the `LogInformation` method with a log level of type `Information`:

```
    public void OnGet()
    {
        _logger.LogWarning("This is a log with Warning level for
Application Insights");
        _logger.LogInformation("This is a log with Information level
for Application Insights");
    }
}
```

If you execute your application again and open the **Privacy** page, Application Insights captures those logs and sends them back to Azure. If you want to check it, you need to go to the **Logs** option inside Application Insights and execute a new query using the **Run** button to retrieve the information from the `traces` table, as shown in the following screenshot, on the **Results** tab.

Among all the information collected by Application Insights automatically, you will see the entries for the modification we just made inside our file.

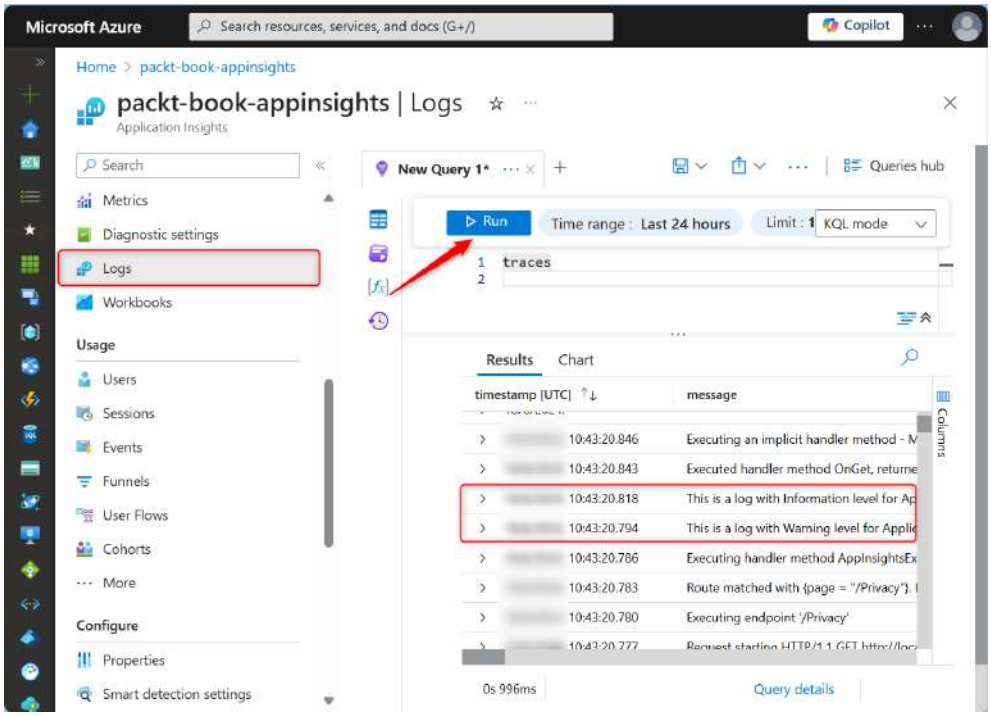

Figure 7.12 – Details of the entries collected inside your traces table

If you are using another logging framework, such as log4net or NLog, you can call its own methods and Application Insights will track those calls following the log level established in the application settings file.

With these integrations, you can monitor application health, diagnose issues, and audit activities without significant changes to your existing logging infrastructure. However, while these integrations provide a robust foundation for capturing logs, there are scenarios where you need more granular control over the logging process to capture specific events, metrics, or traces.

For custom logging scenarios, Application Insights offers the flexibility to define and track custom telemetry using the trace API. This API enables you to log custom events programmatically, allowing you to capture detailed contextual information and application-specific data. By leveraging the trace API, you can customize your logging strategy, ensuring that you gather the precise information required. Let's explore how to implement custom logging using it.

Customizing the information collected through the trace API

The trace API in Application Insights enables developers to capture detailed and application-specific telemetry data. It is the core API that standard telemetry uses to submit all the information collected automatically.

The trace API is part of the Application Insights SDK, specifically the Telemetry Client. This API provides methods for logging various types of telemetry, including traces, events, metrics, and exceptions. The trace API allows you to log custom trace messages. Each trace can include custom properties and metrics, providing a rich context for analysis.

Using our `Privacy.cshtml.cs` page as a baseline, we need to make a few changes to use this API. In the following piece of code, you can find the changes in bold. The first thing is to add a new `TelemetryClient` object that will handle all the connectivity back to Azure. By default, the Application Insights SDK injects a singleton into the dependency injection container so we can reuse it to avoid multiple instances inside our application:

```
using Microsoft.ApplicationInsights;
using Microsoft.ApplicationInsights.DataContracts;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace AppInsightsExample.Pages;

public class PrivacyModel: PageModel
{
    private readonly ILogger<PrivacyModel> _logger;
    private readonly TelemetryClient _telemetryClient;

    public PrivacyModel(ILogger<PrivacyModel> logger, TelemetryClient
telemetryClient)
    {
```

```
        _logger = logger;
        _telemetryClient = telemetryClient;
    }

    public void OnGet()
    {
        _telemetryClient.TrackTrace("This is a custom log collected by
the Trace API", SeverityLevel.Warning, new Dictionary<string, string>
        {
            { "CustomProperty1", "CustomValue1" },
            { "CustomProperty2", "CustomValue2" },
            { "CustomProperty3", "CustomValue3" },
        });
    }
}
```

After that, the `TrackTrace` method will allow us to pass the required custom information for our log trace. We need to specify the log message, the log level, and, if we want, some extra custom properties to attach to the entry. As you can see in the following screenshot, this information will appear in the **Logs** section in the Azure portal.

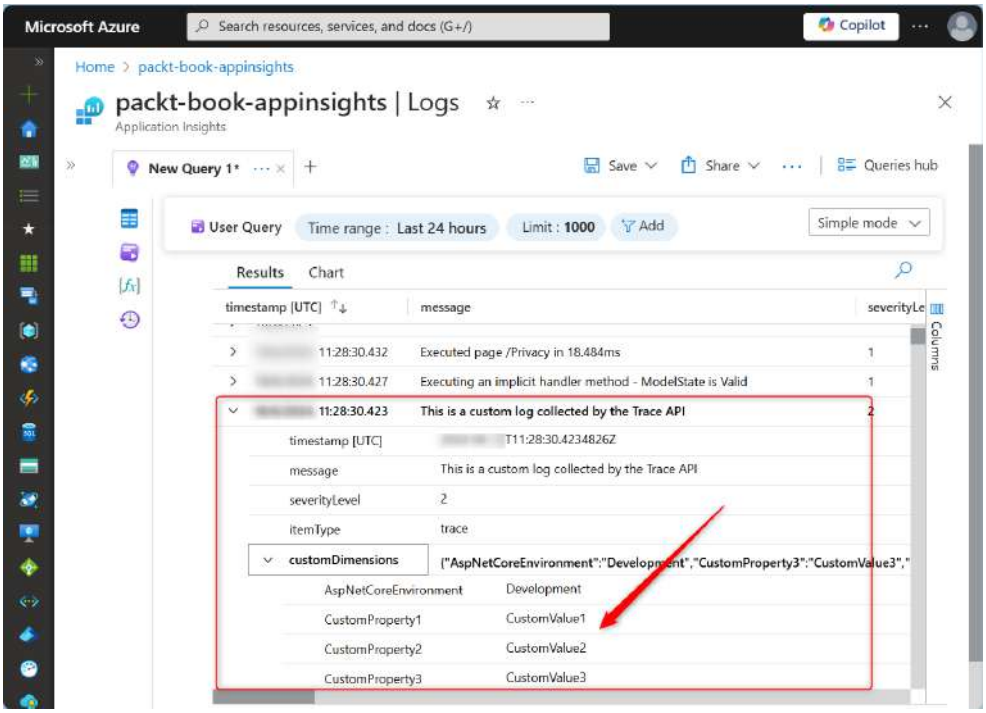

Figure 7.13 – Details of the custom properties included in your logs

This API doesn't just support traces; it provides methods for all the different types of entries supported by Application Insights:

- **Custom events**: In addition to traces, the trace API allows you to log custom events through the `TrackEvent` method. Custom events are useful for tracking specific actions or milestones within your application, such as user logins, purchases, or feature usage. Events can include custom properties to provide additional context.
- **Custom metrics**: The trace API also supports the logging of custom metrics through the `TrackMetric` method. Custom metrics allow you to track performance indicators and other quantitative data that are specific to your application's needs. Metrics can be aggregated and analyzed over time to identify trends and performance issues.
- **Exception tracking**: The trace API enables detailed exception tracking by logging custom exceptions through the `TrackException` method. This includes capturing exception details, stack traces, and any additional context that may help in diagnosing the root cause of issues.
- **Request tracking**: The `TrackRequest` method logs details about incoming HTTP requests to your application. This is essential for understanding how your application handles requests, monitoring performance, and diagnosing issues.
- **Dependency tracking**: The `TrackDependency` method logs information about dependencies your application relies on, such as database calls, external services, or APIs. This method is helpful for understanding the performance and reliability of these dependencies.
- **Page view tracking**: The `TrackPageView` method is used to log information about page views within your application. This is particularly useful for web applications where tracking user navigation and page interactions is crucial for understanding user behavior and optimizing the user experience.

The trace API is your go-to tool to get granular control over what data is logged, providing a rich context through the integration with custom properties included with each trace or event.

Tracing allows you to follow the execution flow of your application, making it easier to debug issues and understand application behavior. Application Insights directly provides distributed tracing capabilities. It captures the entire life cycle of a request as it travels through various services and components. This end-to-end tracking provides a complete picture of the request's journey, including the time spent in each service and any errors encountered along the way.

To link telemetry data across different services, Application Insights uses correlation IDs. These unique identifiers are propagated with each request, ensuring that all related telemetry (such as traces, exceptions, and dependency calls) can be correlated back to the original request. This correlation is essential for understanding how different parts of your system interact and affect each other.

Application Insights defines a data model for distributed telemetry correlation. This model uses specific identifiers to associate telemetry with logical operations and track the flow of requests through different layers of an application. There are two main identifiers:

- **Operation ID** (`operation_Id`): This property is present in every telemetry item. It represents the identifier for a logical operation, which is a higher-level process that might involve multiple smaller operations across various services. All telemetry items (such as traces, requests, dependencies, and exceptions) involved in this logical operation share the same `operation_Id`. This shared identifier allows you to correlate telemetry data across different services and components, providing a unified view of the operation.
- **Parent ID** (`operation_parentId`): Within a distributed logical operation, there are smaller operations processed by individual components. Each of these operations is represented by request telemetry, which has its own unique identifier, or ID. `operation_parentId` is used to link telemetry items to their immediate parent operation.

Operation IDs and parent IDs work in the following way:

1. **Request telemetry**: When a component processes a request, it generates a request telemetry item with a unique ID called `itemId`. This ID identifies the specific operation globally. All related telemetry items, such as traces and exceptions, set their `operation_parentId` to this request ID, linking them to the specific request.
2. **Dependency telemetry**: Dependency telemetry represents outgoing operations, such as HTTP calls to other components or services. Each dependency call generates a unique ID, also called `itemId`. When a dependency call initiates a request in another component, the resulting request telemetry in that component uses this dependency ID as its `operation_parentId`. This linkage creates a traceable path from the initial request, through the dependency call, to the subsequent request in the downstream component.
3. **Building the distributed trace**: Using the operation ID, operation parent ID, request ID, and dependency ID, you can construct a complete view of the distributed logical operation. These identifiers define the causality order of telemetry calls, allowing you to trace the flow of operations through your entire application architecture.

Coming back to the previous example of the usage of the trace API call, you can see in the following screenshot how both `operation_Id` and `operation_Parent_Id` are defined inside our event automatically.

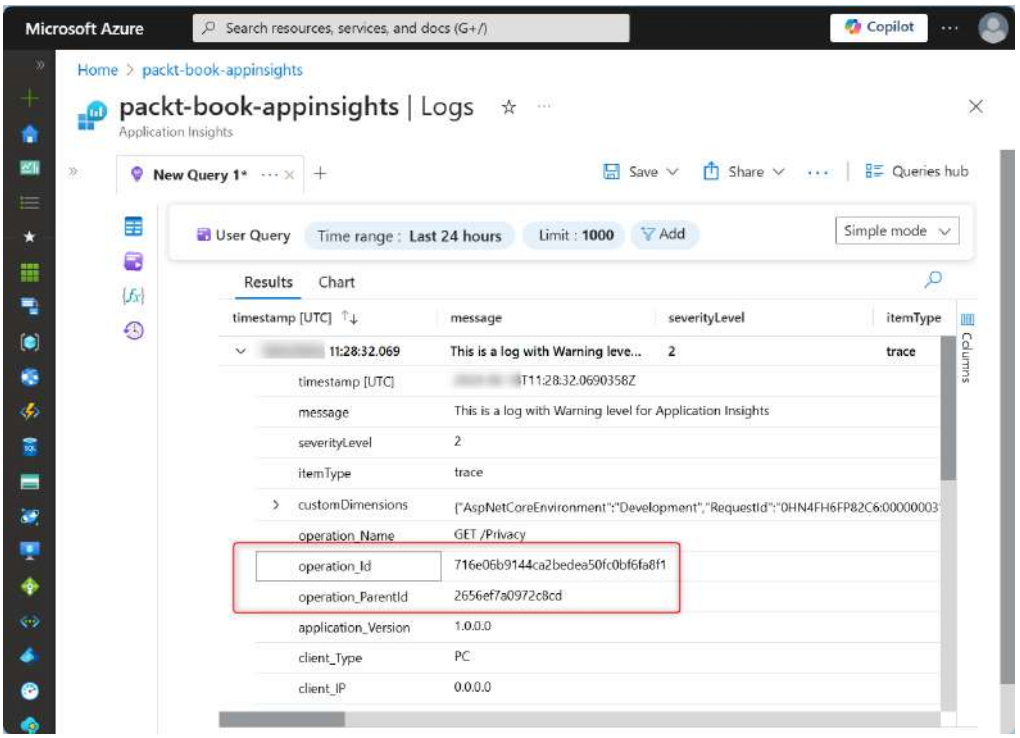

Figure 7.14 – Details of the operation entries inside a log event

Using the value of `operation_Parent_Id` to filter all the available traces inside our repository, we can easily see the whole path our request followed from when the server received our request to when it completed, as shown in the following screenshot.

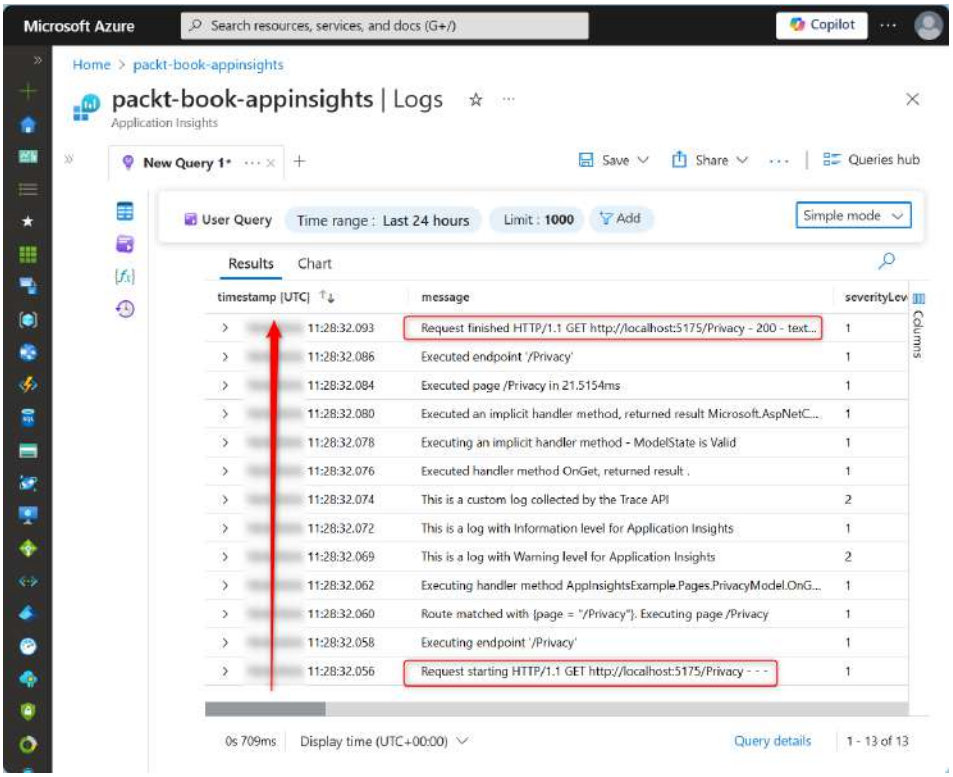

Figure 7.15 – End-to-end logs of a specific request

Equally important to understanding our internal dependencies is tracing interactions with external dependencies. Our application often relies on external services such as databases, third-party APIs, and other microservices. Tracing these dependencies allows us to monitor their performance and understand their impact on our application.

Application Insights automatically tracks calls to external dependencies, such as databases, APIs, and third-party services. This tracking includes details such as response times, success rates, and failure reasons.

Expanding your tracing to external dependencies

Application map is one of the standout features of Application Insights, providing a visual representation of your application's architecture and its dependencies. It is designed to give you a clear, graphical overview of how various components of your application interact with each other. It automatically detects the different parts of your application, such as web services, databases, and external dependencies, and displays them as nodes on the map. Lines connecting these nodes represent the interactions between them, including HTTP requests, database queries, and calls to external services.

Let's add an external dependency to our simple demo application. In this case, we want to retrieve the weather information from an external website called **Open-Meteo** (`https://open-meteo.com/`).

This website provides an endpoint that can be called for free without any previous registration. We are not interested in the information at all, in this case, so we will request it without rendering it inside our website to simplify our example.

We will add our code to the `Privacy.cshtml.cs` page. In the following source list, you have the contents of the file with the relevant parts in bold:

```
using Microsoft.AspNetCore.Mvc.RazorPages;
namespace AppInsightsExample.Pages;
public class PrivacyModel : PageModel
{
    private readonly ILogger<PrivacyModel> _logger;
    private readonly HttpClient _httpClient;

    public PrivacyModel(ILogger<PrivacyModel> logger,
IHttpClientFactory httpClientFactory)
    {
        _logger = logger;
        _httpClient = httpClientFactory.CreateClient();
    }
    public async void OnGet()
    {
        string url = "https://api.open-meteo.com/v1/
forecast?latitude=52.52&longitude=13.41&current=temperature_2m,wind_
speed_10m&hourly=temperature_2m,relative_humidity_2m,wind_speed_10m";
        var response = await _httpClient.GetAsync(url);
        if (response.IsSuccessStatusCode)
        {
            var content = await response.Content.ReadAsStringAsync();
            _logger.LogInformation(content);
        }
    }

}
```

Compile and run your application and connect several times to the **Privacy** page. After a few minutes, Application Insights will show you the details of this new dependency in their **Application map** graph, as shown in the following screenshot.

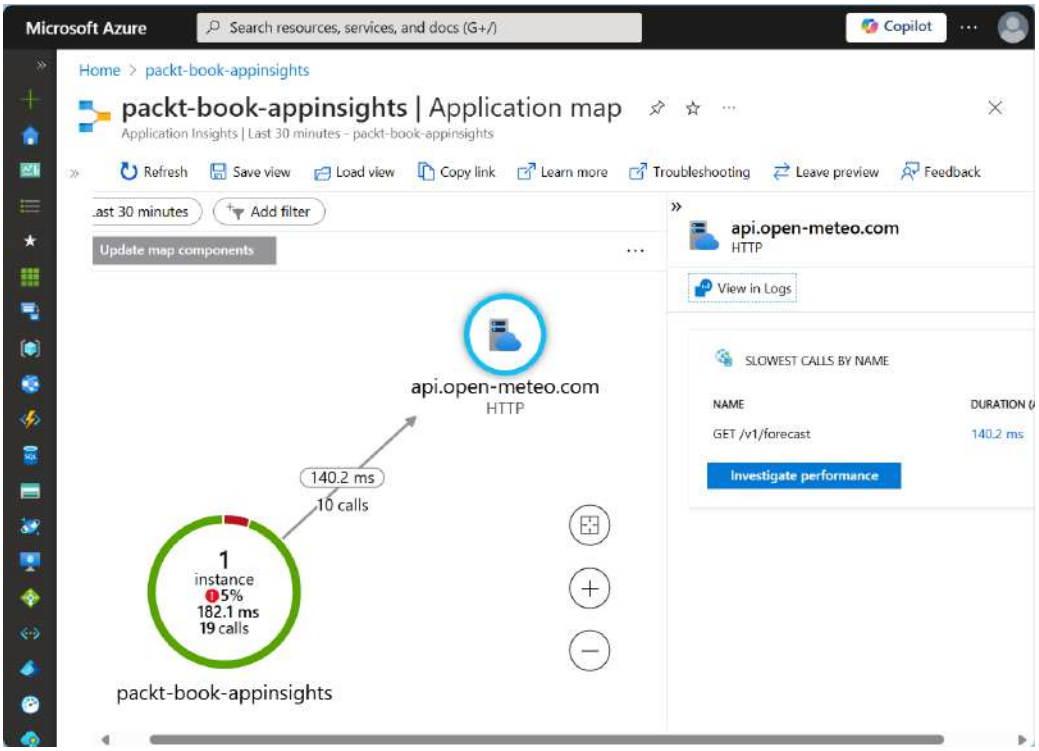

Figure 7.16 – A general overview of the external dependencies of your web application

You can see that, in this case, we made **10** successful calls to the `/v1/forecast` endpoint with an average of **140.2** milliseconds. If you want to analyze those results further, the **Investigate performance** button allows you to open the detailed **Performance** view already described.

In this section, we covered the powerful logging and distributed tracing capabilities provided by Application Insights. We explored how detailed logging and the use of correlation IDs and parent IDs enable you to gain deep insights into your application's behavior, track the flow of requests across distributed systems, and diagnose complex issues effectively.

As we transition to the next section, our focus will shift to the availability and user experience features of Application Insights. While logging and tracing provide the technical insights needed to keep your application running smoothly, understanding and enhancing user experience is equally crucial. By leveraging Application Insights' capabilities for monitoring application availability and analyzing user behavior, you can ensure that your application not only performs well technically but also meets and exceeds user expectations.

Ensuring optimal user experiences through observability

In today's digital landscape, delivering a responsive user experience is one of the key points to the success of any application. Users expect fast, reliable, and intuitive interactions, and any deviation from these expectations can lead to dissatisfaction and loss of engagement.

Observability extends beyond traditional monitoring by providing deep insights into application behavior, user journeys, and performance metrics. One of the aspects to ensure an optimal user experience is maintaining the availability of your application. Users expect your application to always be accessible, regardless of their location or the time of day. This is where the **Availability** view in Application Insights becomes invaluable.

The Availability view

The **Availability** view in Application Insights is a feature designed to monitor and ensure the continuous availability of your application. This out-of-the-box tool helps you track the uptime and responsiveness of your application through regular tests and provides insights into any incidents of downtime or failures.

It focuses on tracking the availability of your application through synthetic transactions, which are predefined tests that simulate user interactions. These tests run at regular intervals from various locations around the globe to verify that your application is accessible and performing as expected. This ensures that you can monitor the accessibility and performance of your application from different geographical regions, providing a comprehensive view of its global availability.

Application Insights calls those synthetic transactions **standard tests**. These tests can be configured to run at regular intervals and from multiple locations worldwide, providing a comprehensive view of your application's availability and performance. They are also called URL ping tests because they ping the configured URL to ensure that your web application or API endpoint is reachable and performing well.

To create a new one, you need to open the **Availability** view inside your Application Insights instance and select **Add Standard test**, as indicated in the following screenshot.

> **Retirement of classic tests**
>
> As you can see in the screenshot, there is an option to create a **classic test**. Although they are currently still available, Microsoft has stated that they will be retired, and customers should transition to the new standard test type. In this section, we will focus only on the supported test types.

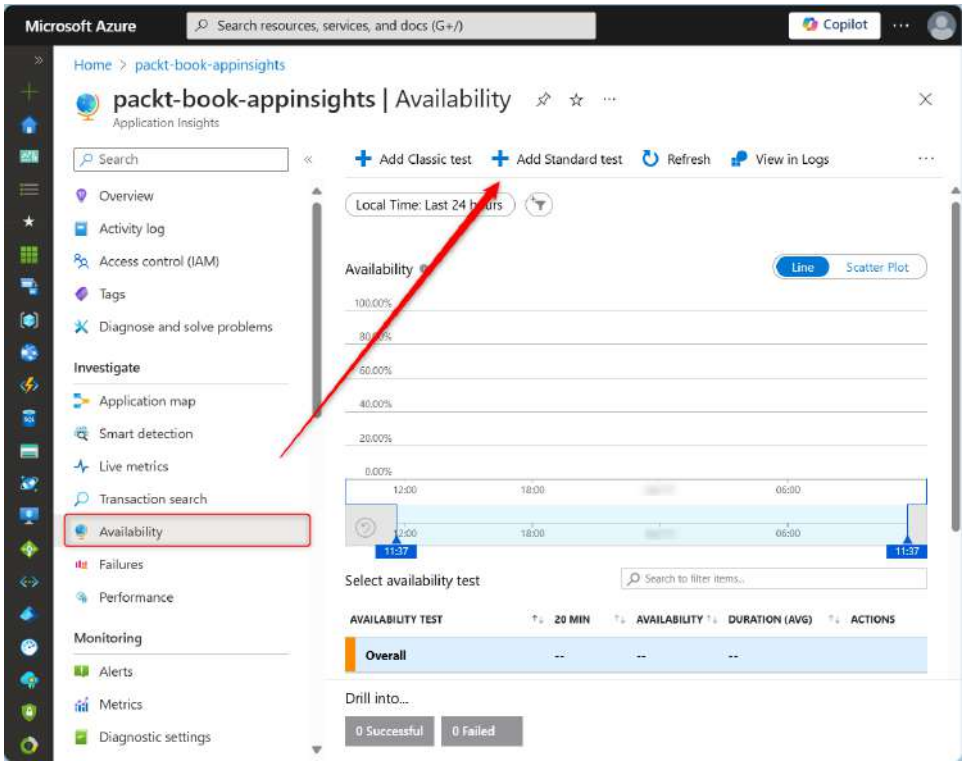

Figure 7.17 – Adding a new availability test

The minimum information that you need to provide to create a new test is a name, to identify it, and a URL to monitor. As you can see in the following screenshot, it's possible to further customize your availability test with extra configurations regarding the frequency of the test and the location, with a choice across the worldwide Azure data centers available.

It's possible to customize the request through specific HTTP request verbs and parameters and validate not only the HTTP response code but also the content inside the response.

After the test is created, the **Availability** view will provide you with an updated dashboard of your website status. It provides insights into any incidents of downtime or failures detected by the synthetic tests. It logs the time and duration of the failures, the locations from which the failures were detected, and any error messages or status codes returned. This information helps in diagnosing the root cause of availability issues and addressing them promptly.

The **Availability** view maintains historical data of all the availability tests, allowing you to analyze trends over time. You can review past incidents of downtime, compare performance across different periods, and identify patterns that may indicate underlying issues. This historical perspective is valuable for long-term planning and reliability improvements.

It also includes metrics on the success rates of the availability tests, showing the percentage of tests that passed versus those that failed. It also tracks the response times of the tests, providing insights into the performance of your application during the tests. These metrics help in understanding the reliability and performance of your application over time. In the following screenshot, you can see that our site has an availability of 100%.

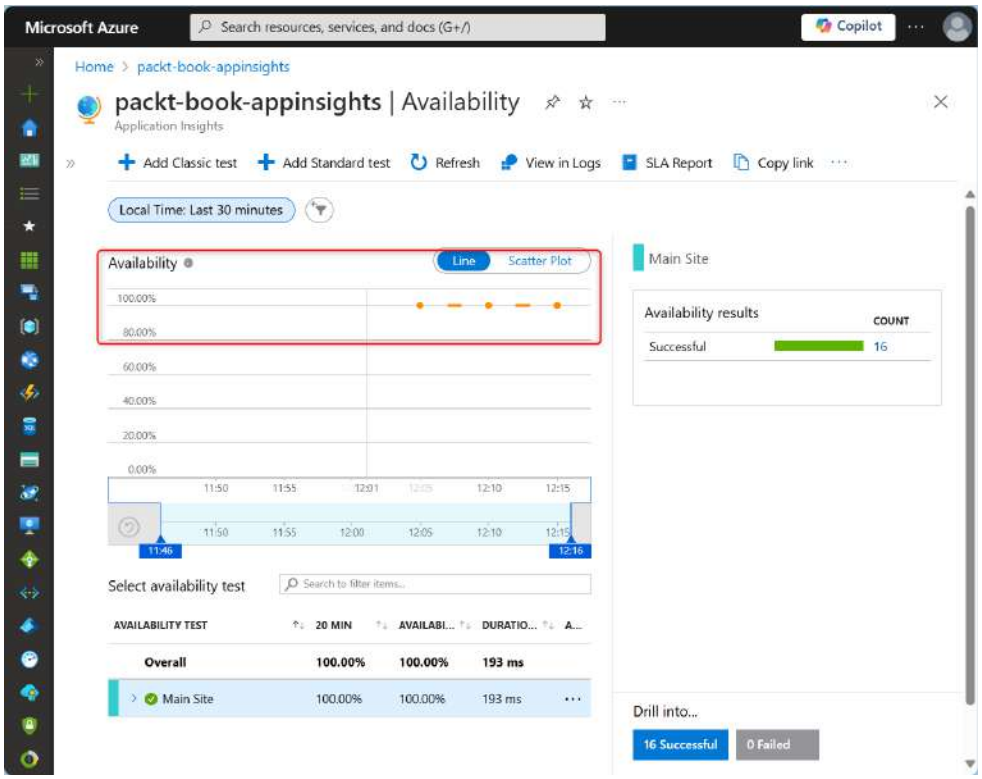

Figure 7.18 – Details of the availability of our web application

Finally, it also supports setting up alerts to notify you whenever an availability test fails or when the success rate drops below a certain threshold. These alerts can be configured to send notifications via email or SMS, or be integrated with other monitoring tools as described in *Chapter 5*. Proactive alerts ensure that you are immediately aware of any issues impacting your application's availability, allowing for quick remediation.

To create a new alert, select **AVAILABILITY TEST** in the **Availability** view and then click on **Open Rules (Alerts) page**, as shown in the following screenshot. It will launch the standard Azure Monitor wizard to create custom alert rules. By default, a single rule is available that triggers when two or more locations return an error when testing your URL endpoint.

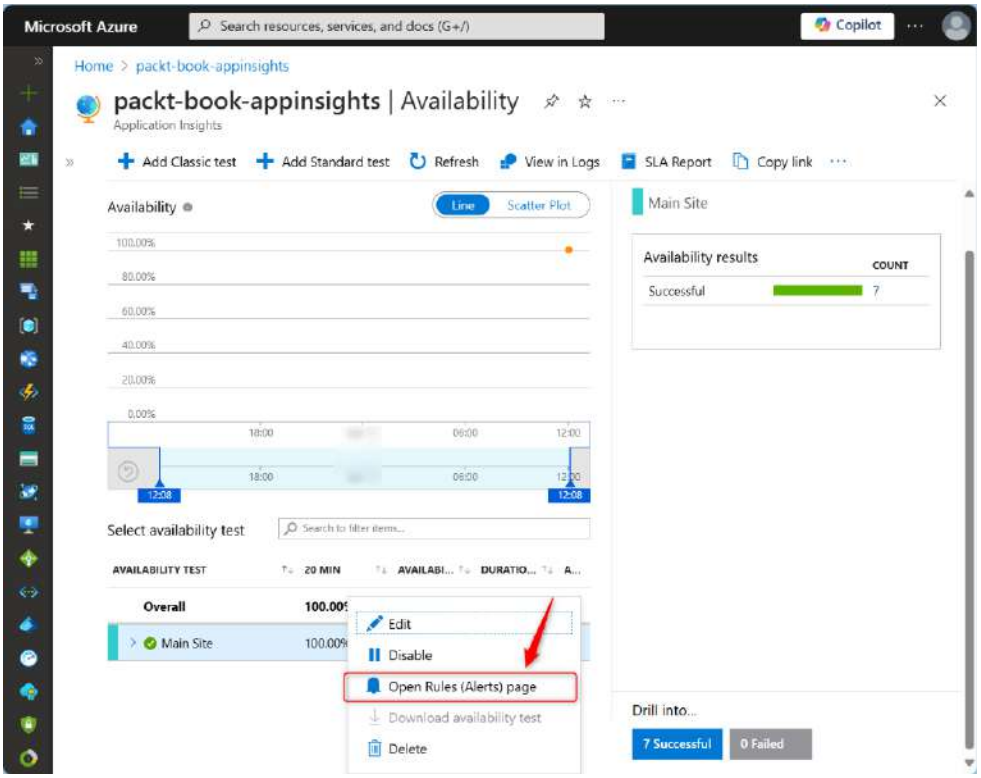

Figure 7.19 – Creating an alert rule from our availability test

Ensuring that your application is always available and responsive is fundamental to delivering a reliable user experience. However, maintaining high availability is just one aspect of ensuring an optimal user experience. To truly understand and enhance how users interact with your application, you need to know about user behavior and engagement. This is where user behavior analytics comes into play. By tracking and analyzing user interactions, you can gain valuable insights into user preferences, identify friction points, and optimize the overall user journey. Let's explore how user behavior analytics in Application Insights can provide the detailed insights necessary to elevate your application's user experience.

User behavior analytics view

User behavior analytics in Application Insights, identified as **Usage** inside the Azure portal, provides insights into user actions, preferences, and engagement patterns. This feature allows you to track and analyze various aspects of user behavior, helping you to optimize your application's design and functionality to better meet user needs. By leveraging user behavior analytics, you can make data-driven decisions that enhance usability, increase user satisfaction, and drive higher engagement.

It aggregates data on user interactions and presents it in a way that allows you to analyze and optimize the user experience. This includes tracking user sessions, page views, user flows, and engagement metrics. Let's describe them in more detail:

- **User sessions and session metrics**: The service tracks individual user sessions, providing detailed metrics on session duration, number of interactions, and pages viewed per session. This helps in understanding how long users stay on your application and how they navigate through it. Users are counted through the usage of IDs stored as cookies inside the user's browser. It helps to identify multiple accesses by the same user through the same browser. Sessions are restarted after 30 minutes of inactivity or after 24 hours when there is continuous usage.

  If you are interested in specific user demographics, such as geographic location, device type, and browser usage, the usage of cohorts helps in understanding the diversity of your user base and improving the application experience to meet the needs of different user segments. A cohort is a set of users, operations, or events that share something between them, such as geographic location, device type, and browser usage.

- **User flows**: This feature visualizes the paths users take through your application. By analyzing user flows, you can identify common navigation patterns, popular entry and exit points, and potential bottlenecks or drop-off points. This information is crucial for optimizing user journeys and ensuring a smooth navigation experience.

  The **User Flows** tool starts with a specified initial event, such as a custom event, exception, or request. From this initial point, **User Flows** displays the events that occurred before and after user sessions. The thickness of the lines represents the number of times users followed each path. **Special Session Started** nodes indicate where sessions began, while **Session Ended** nodes show how many users stopped sending page views or custom events after the previous node, highlighting likely exit points from your site.

- **Custom events and funnels**: In addition to standard metrics, you can define and track custom events that are specific to your application's functionality. Creating funnels for these events helps in analyzing the steps users take to complete key actions, identifying drop-off points, and optimizing the process to improve completion rates.

- **Retention analysis**: This feature tracks how well your application retains users over time. By analyzing retention rates, you can determine the effectiveness of your application in keeping users engaged and coming back. High retention rates often indicate a positive user experience, while low retention rates may signal the need for improvements.

In conclusion, understanding and leveraging the **User Flows** tool is crucial for gaining insights into user behavior on your site. However, it's important to note that demonstrating all these details within our demo application is challenging. This environment doesn't capture the full range of user behaviors and interactions that would be found in a production environment.

Deploying Application Insights in a real production environment will enable you to gather comprehensive data, offering a clearer picture of user interactions and highlighting areas for improvement more effectively. This real-world deployment will provide the depth of data needed to fully utilize the capabilities of the **User Flows** tool and enhance your application's performance and user experience.

Summary

In this chapter, we covered the comprehensive capabilities of Application Insights, a component of Azure Monitor. We began exploring its automatic and manual instrumentation capabilities, which allow developers to collect detailed telemetry data without significant code changes. We then moved on to the practical aspects of instrumenting code for monitoring, covering the setup process and the benefits of integrating Application Insights into a .NET Core application.

The chapter also highlighted the importance of diagnostics implementation for maintaining application health. We examined how to implement effective diagnostics to quickly identify and resolve issues impacting reliability and performance.

Tracing techniques were another critical focus, as we discussed the use of correlation IDs and parent IDs to track the flow of requests across distributed systems. These techniques help in understanding the execution flow and interactions within the application.

The logging capabilities of Application Insights were introduced, including integration with popular logging frameworks and the use of the trace API for custom telemetry. We demonstrated how to capture detailed application logs, custom events, metrics, and traces to gain deep insights into application behavior.

Finally, we explored ensuring optimal user experiences through observability. We discussed how to use features such as the **Availability** view and user behavior analytics to monitor application availability, and track user interactions.

As we conclude our introduction to Application Insights, we now turn our attention to the monitoring strategies required for hybrid and multi-cloud environments. In the next chapter, we will explore how Azure Monitor seamlessly integrates with Azure Arc and **System Center Operations Manager** (**SCOM**) managed instances. This chapter will guide you through best practices for extending your monitoring capabilities beyond Azure, ensuring a unified and comprehensive approach to observability across diverse environments.

Further reading

Here, you can find the links to expand your knowledge about specific concepts not covered in this book but referenced in this chapter:

- [1] Auto instrumentation capabilities for Azure Monitor applications: `https://learn.microsoft.com/en-us/azure/azure-monitor/app/codeless-overview#supported-environments-languages-and-resource-providers`.

Part 3: The Road Ahead with Azure Observability

The last part of this book covers relevant topics that, although they are not a core part of the Azure Monitor service, are required to design and deploy a successful monitoring strategy in the cloud. It expands our observability conversation to hybrid and multi-cloud scenarios where Azure Monitor can contribute to providing a global vision of your current platform. It also introduces the different integration features that Azure Monitor supports to enable third-party applications to ingest Azure Monitor data and expand its features and capabilities further. We also dedicated a specific section to best practices to deploy Azure Monitor and control its costs. Finally, we explore future trends that we have observed in this space.

This part includes the following chapters:

- *Chapter 8, Hybrid and Multi-Cloud Monitoring*
- *Chapter 9, Integrating with Third-Party Tools*
- *Chapter 10, Building Your Monitoring Strategy*
- *Chapter 11, Cost Management and Optimization*
- *Chapter 12, Future Trends and Looking Ahead*
- *Appendix*

8

Hybrid and Multi-Cloud Monitoring

In the previous chapters, we have mentioned how most organizations run hybrid and multi-cloud setups as it gives them the flexibility to take advantage of the best features of multiple cloud providers while maintaining on-premises infrastructure. However, this complexity poses several challenges, including ensuring consistent performance, security, and compliance across platform environments. One of the fundamental aspects of overcoming these challenges is effective and consistent monitoring across all these platforms to have visibility and control over the entire IT ecosystem.

This chapter introduces the specific tools and techniques available in Azure for monitoring resources in hybrid and multi-cloud environments. We'll examine how Azure Monitor integrates with **Azure Arc** and **System Center Operations Manager** (**SCOM**) managed instances to enable detailed monitoring of other cloud platforms and on-premises systems.

Additionally, we will discuss best practices for extending your monitoring capabilities to hybrid and multi-cloud environments, ensuring a unified and comprehensive approach to observability.

Upon completing this chapter, you'll have a deep understanding of extending observability by seamlessly integrating and expanding your previous knowledge of Azure Monitor with Azure Arc to gain insights into hybrid cloud environments. You will also be able to develop strategies for using Azure Monitor in multi-cloud environments, ensuring a unified monitoring approach across different cloud providers, and understand how to integrate and leverage Azure Monitor with SCOM managed instances, creating a cohesive monitoring strategy for on-premises and hybrid environments based on System Center.

In this chapter, we're going to cover the following main topics:

- Extending observability with Azure Arc
- Harnessing the power of Azure Monitor in multi-cloud environments
- Integrating Azure Monitor with SCOM Managed Instance
- Lab: Configuring Azure Monitor with Arc for AWS

Technical requirements

To follow all the examples available in this chapter, you need an Azure account with an active subscription, access to Azure Monitor, familiarity with the Azure portal, and basic knowledge of log management and query languages.

For the lab section, you will need the following additional prerequisites:

- An AWS account (`https://repost.aws/es/knowledge-center/create-and-activate-aws-account`).
- Linux (root) and Windows administrator permissions.
- The following resource providers must be registered with your subscription:
  - `Microsoft.HybridCompute`
  - `Microsoft.GuestConfiguration`
  - `Microsoft.HybridConnectivity`
  - `Microsoft.AzureArcData` (only if you plan to Arc-enable SQL Server instances)

  Follow the steps at `https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/resource-providers-and-types#register-resource-provider` to register the preceding resource providers with your Azure subscription.
- Your AWS virtual machine, running a supported operating system. For more details, see the *Further reading* section [1].

Extending Observability with Azure Arc

Azure Arc aims to connect on-premises and cloud environments by offering a unified operational approach, combining the automation and scalability of the Azure platform and its services. This way, cloud operations teams can manage and monitor their Windows and Linux servers, as well as virtual machines, regardless of whether they are hosted outside of Azure, on their corporate network, or through a third-party cloud provider.

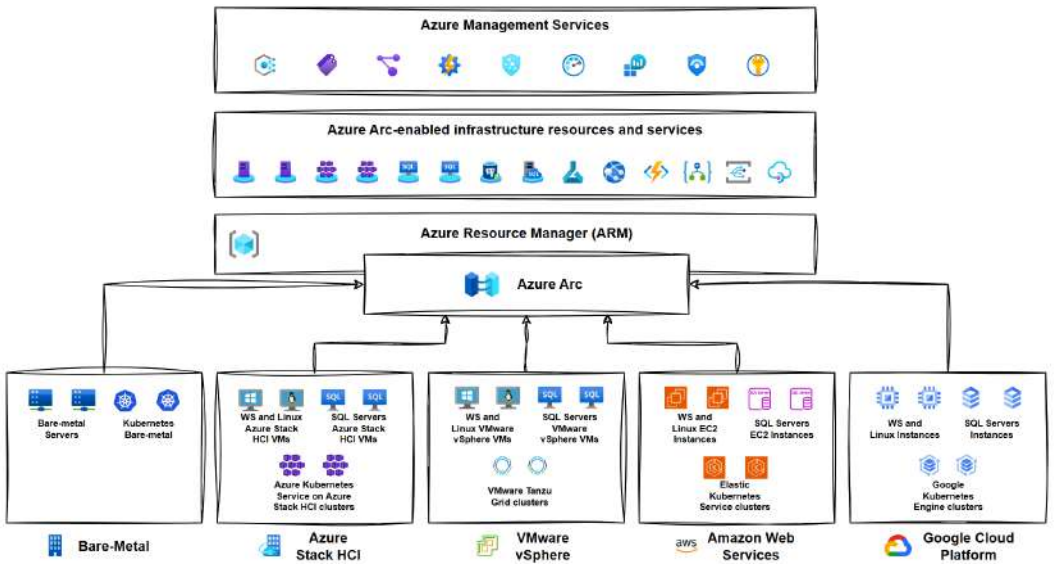

Figure 8.1 – Azure Arc architecture

In this section, we'll explain how, through Azure Arc, you can centrally monitor Azure Arc-enabled servers within your Azure business environment. We will not go into the details of how Azure Arc works or the different deployment options, but in the *Further reading* section of this chapter, you can expand your knowledge about this and other aspects of Azure Arc [2].

Let's say that as a monitoring team leader for an organization with a multi-cloud ecosystem that includes industrial environments, you decide to integrate your non-Azure machines into Azure Arc-enabled servers. Your challenge here is to achieve observability across your broader IT infrastructure just as you do with your virtual machines in Azure, which requires intelligent, scalable alerts along with comprehensive metrics, logs, and visualizations.

As we showed in previous chapters, you can take advantage of Azure monitoring tools, such as **Log Analytics**, **VM insights**, and **Azure dashboards**, to improve observability in your Azure environment. However, if you want to extend the same level of visibility you have for Azure resources to those outside of Azure, ensuring consistent and comprehensive monitoring across your entire IT landscape, you first need to onboard your servers into Azure Arc.

Once your servers have been onboarded to Azure Arc, you can manage them like any other Azure virtual machine. As we mentioned in previous chapters, to be able to monitor the operating system and any workload running on the machine or server using VM insights, analyze and alert using Azure Monitor, and perform security monitoring in Azure using Microsoft Defender for Cloud or Microsoft Sentinel, you need to install the **Azure Monitor Agent** (**AMA**). So, in order to monitor your machine or server registered with Azure Arc-enabled servers, you would need to install the AMA just like you would an Azure virtual machine. In the next section, we'll explain the different methods to deploy AMA for Azure Arc-enabled servers.

Harnessing the power of Azure Monitor in multi-cloud environments

Azure Arc-enabled servers are compatible with the Azure VM extension architecture, allowing for post-deployment configuration and automation tasks, including the installation of the AMA extension. The following are the different methods to deploy the AMA VM extension to Azure Arc-enabled servers based on the operating system:

- **Azure Policy**: Automate the deployment of the AMA using Azure Policy. We recommend this method for deploying at scale. This method ensures consistency and compliance across your environment by automatically applying the agent to newly added or existing servers based on defined policies. From Azure Policy, you can search for the policy definitions to assign based on the Arc-enabled machine operating system:
    - **For Linux**: Search for **Configure Linux Arc-enabled machines to run Azure Monitor Agent**:

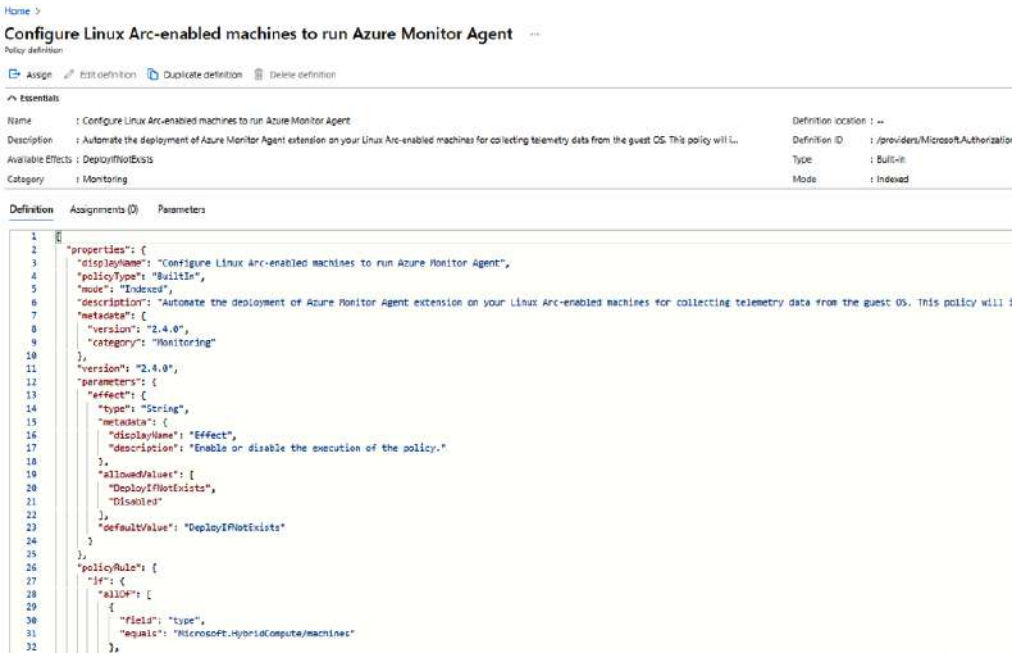

Figure 8.2 – Policy Definition: Configure Linux Arc-enabled machines to run Azure Monitor Agent

The preceding figure shows the contents of the policy definition for configuring Linux Arc-enabled machines to run the AMA.

- **For Windows**: Search for **Configure Windows Arc-enabled machines to run Azure Monitor Agent**:

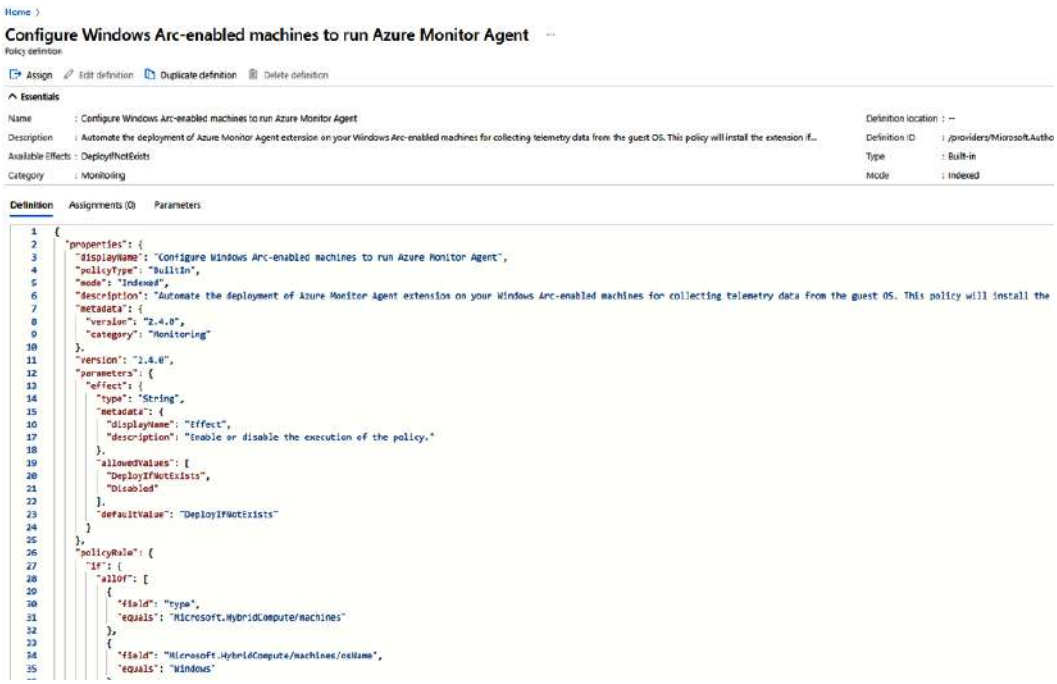

Figure 8.3 – Policy Definition: Configure Windows Arc-enabled machines to run Azure Monitor Agent

The preceding figure shows the contents of the policy definition for configuring Windows Arc-enabled machines to run the AMA.

- **Azure portal**: This is particularly advantageous for testing purposes and managing a small number of machines, due to its intuitive graphical interface. However, there are limitations to using the portal, such as limited automation capabilities and the inability to manage multiple Arc-enabled servers simultaneously. Additionally, the portal does not support deploying the Dependency agent, so you will need to use another listed method to do so. Finally, the Azure portal only supports specifying a single workspace for reporting. To learn how to install the AMA from the Azure portal, see the *External telemetry – integrating insights from external sources* section of *Chapter 3*, where we explained the process of creating a data collection rule.

- **Azure CLI**: Utilize the Azure **Command-Line Interface** (**CLI**) to manually install the AMA on individual servers or automate the deployment through scripts. Just like the Azure portal, the Azure CLI is particularly advantageous for testing purposes and managing a small number of machines, but through scripting, you can use it for more complex and large-scale environments that require advanced configurations and automation. Here are the CLI commands you can use to install the AMA on Azure Arc-enabled servers.
    - For Windows:

```
az connectedmachine extension create --name
"AzureMonitorWindowsAgent" --publisher "Microsoft.Azure.
Monitor" --type "AzureMonitorWindowsAgent" --machine-name
"arcServerName" --resource-group "resourceGroupName" --location
"location" --auto-upgrade-minor-version true --enable-auto-
upgrade true
```

    - For Linux:

```
az connectedmachine extension create --name
"AzureMonitorLinuxAgent" --publisher "Microsoft.Azure.Monitor"
--type "AzureMonitorLinuxAgent" --machine-name "arcServerName"
--resource-group "resourceGroupName" --location "location"
--auto-upgrade-minor-version true --enable-auto-upgrade true
```

Important note

We are using the authentication method based on system-managed identity because it is the only supported type for Azure Arc-enabled servers.

- **PowerShell**: Similar to the Azure CLI, PowerShell scripts can be used to deploy the AMA, offering another option for automation and integration with existing management workflows. Here are the PowerShell commands you can use to install the AMA on Azure Arc-enabled servers.
    - For Windows:

```
New-AzConnectedMachineExtension -Name
"AzureMonitorWindowsAgent" -ExtensionType
"AzureMonitorWindowsAgent" -Publisher "Microsoft.Azure.
Monitor" -ResourceGroupName "resourceGroupName" -MachineName
"arcServerName" -Location "location" -EnableAutomaticUpgrade
$true -AutoUpgradeMinorVersion $true
```

    - For Linux:

```
New-AzConnectedMachineExtension -Name "AzureMonitorLinuxAgent"
-ExtensionType "AzureMonitorLinuxAgent" -Publisher "Microsoft.
Azure.Monitor" -ResourceGroupName "resourceGroupName"
-MachineName "arcServerName" -Location "location"
-EnableAutomaticUpgrade $true -AutoUpgradeMinorVersion $true
```

- **Azure Resource Manager** (**ARM**) **Templates**: Use ARM templates for scalable and repeatable deployments. ARM templates allow you to define your infrastructure and configurations as code, enabling you to deploy the AMA across multiple servers efficiently. However, we recommend this method for small or medium deployments.

  - For Windows

```
param vmName string
param location string
resource vm 'Microsoft.Compute/virtualMachines@2021-04-01'
existing = {
  name: vmName
}
resource windowsAzureMonitorAgent 'Microsoft.HybridCompute/
machines/extensions@2023-10-03-preview' = {
  name: 'AzureMonitorWindowsAgent'
  parent: vm
  location: location
  properties: {
    publisher: 'Microsoft.Azure.Monitor'
    type: 'AzureMonitorWindowsAgent'
    autoUpgradeMinorVersion: true
    enableAutomaticUpgrade: true
  }
}
```

  - For Linux:

```
param vmName string
param location string
resource vm 'Microsoft.Compute/virtualMachines@2021-04-01'
existing = {
  name: vmName
}
resource linuxAzureMonitorAgent 'Microsoft.HybridCompute/
machines/extensions@2023-10-03-preview' = {
  name: 'AzureMonitorLinuxAgent'
  parent: vm
  location: location
  properties: {
    publisher: 'Microsoft.Azure.Monitor'
    type: 'AzureMonitorLinuxAgent'
    autoUpgradeMinorVersion: true
    enableAutomaticUpgrade: true
  }
}
```

- **Azure Automation**: Azure Automation is a recommended option for automating the deployment of the AMA VM extension at scale across your environment. By leveraging PowerShell and Python runbooks, it allows you to automate the deployment and configuration process. This approach enables you to define and control the execution schedule. Moreover, Azure Automation supports secure authentication to Arc-enabled servers through managed identities, enhancing the overall security posture of your automation workflows. However, setting up Azure Automation requires an Azure Automation account and proficiency in managing runbooks within the service. In the *Further reading* section, you can see how to create a Python 3.8 runbook in Azure Automation for automating the deployment of the AMA VM extension at scale [3].

Integrating Azure Monitor with SCOM Managed Instance

In this section, we'll introduce **Azure Monitor SCOM Managed Instance** (**SCOM MI**) as an alternative to monitoring on-premise scenarios. This platform offers more features than just monitoring on-premises environments. For organizations looking to build upon their investment in **Configuration Manager** or those with specific on-premises management needs, SCOM presents a compelling option. However, there are scenarios where SCOM and Azure Arc complement each other, and in such scenarios using both tools can provide a balanced approach that combines the best of both solutions.

> **Important note**
>
> We recommend that the decision to use Azure Monitor SCOM Managed Instance, Azure Arc, or both be based on an assessment of the specific needs of the organization and how each of the tools fits into the future IT strategy.

Azure Monitor SCOM Managed Instance is an Azure cloud-based alternative to the traditional **System Center Operations Manager** platform that allows customers to continuously monitor their on-premise workloads with simpler infrastructure management.

Integrating Azure Monitor with SCOM Managed Instance offers a powerful and comprehensive monitoring solution that leverages the strengths of both platforms. Through this integration, an organization can obtain the following benefits:

- **Unified monitoring**: Combining Azure Monitor and SCOM Managed Instance provides a holistic view of the entire IT infrastructure, encompassing both cloud and on-premises resources. This unified monitoring approach helps in identifying and resolving issues more efficiently.
- **Advanced analytics**: Azure Monitor's advanced analytics capabilities, such as Log Analytics and Application Insights, enhance SCOM Managed Instance's monitoring data. This integration allows for deeper insights and better data-driven decision-making.
- **Enhanced alerting and reporting**: Azure Monitor integration enables the use of Azure Monitor's alerting and reporting features we have explained in previous chapters, improving the responsiveness and accuracy of notifications and reports.

Now that you know the benefits of Azure Monitor SCOM Managed Instance, we'll detail how to configure Azure Monitor SCOM Managed Instance with Azure Log Analytics. To carry out this integration, the following are the prerequisites:

- You need to have an Azure Monitor SCOM Managed Instance deployment. For more information on how to create an Azure Monitor SCOM Managed Instance deployment, you can refer to the *Further reading* section [4].
- We recommend that the location of the **Log Analytics workspace** and Azure Monitor SCOM Managed Instance are the same.
- To link the Azure Log Analytics workspace to Azure Monitor SCOM Managed Instance, at least the **Log Analytics Contributor** role is required. This role must be assigned in the workspace resource group to **Microsoft.SCOM** resource provider.

To integrate Azure Monitor with SCOM Managed Instance, follow these steps:

1. Ensure that SCOM Managed Instance is updated to the latest version to support integration capabilities.
2. Verify that the SCOM Managed Instance environment is properly configured and operational.
3. In the Azure portal, navigate to the SCOM Managed Instance page and select the desired SCOM managed instance.
4. On the left pane, select **Log Analytics workspace** and then select the **Link Log Analytics workspace** tab.
5. Fill in the destination details with the subscription and the desired Log Analytics workspace. In the **Log data types** tab, select the data types (**EVENT**, **STATE**, **PERFORMANCE**, and **AUDIT**) you want to synchronize to the Log Analytics workspace.
6. After configuring the integration, verify that the data from SCOM Managed Instance is being successfully ingested into Azure Monitor. From the **Logs** tab, check under **Custom Logs** that the previously selected custom tables (**State_CL**, **Performance_CL**, **Event_CL**, and **Management_CL**) have been created.
7. Test the integration by querying the data and creating alerts from this Log Analytics workspace. Try out simulating alerts in SCOM Managed Instance and ensuring they are accurately reflected in Azure Monitor.

Up to this point, we have integrated Azure Monitor with SCOM Managed Instance and detailed the benefits of this integration. However, what are the best practices to follow to use both solutions? In the next section, we will address the main practices that organizations can implement to use both solutions together.

Best practices for using Azure Monitor with SCOM Managed Instance

Some of the best practices to consider in the use of Azure Monitor and SCOM Managed Instance within an organization are as follows:

- **Data ingestion optimization**: Ensure that only relevant and necessary data is sent from SCOM Managed Instance to Azure Monitor to avoid unnecessary costs and data overload.
- **Custom dashboards**: Create custom dashboards in Azure Monitor to visualize the integrated data from SCOM Managed Instance. These dashboards can provide a centralized view of critical metrics and alerts, facilitating better monitoring and management. You can follow the guidelines indicated in *Chapter 6* related to this topic.
- **Automation and alerts**: Leverage Azure Monitor's automation capabilities to respond to specific alerts or conditions automatically. Use the guidelines indicated in *Chapter 5* related to this topic to create automated workflows that address common issues detected by SCOM Managed Instance.
- **Regular audits and updates**: Regularly audit the integration setup to ensure it remains efficient and effective. Keep both SCOM Managed Instance and Azure Monitor components updated to leverage the latest features and improvements.
- **Training and documentation**: Ensure that the IT team is well-trained in using both SCOM Managed Instance and Azure Monitor. Provide comprehensive documentation on the integration setup, including troubleshooting steps and best practices.

With the topics explained in this section, you are now able to understand the benefits of integrating Azure Monitor with SCOM Managed Instance, such as unified monitoring, advanced analytics, and enhanced alerting. Additionally, you now know the prerequisites and steps for configuring Azure Monitor SCOM Managed Instance with Azure Log Analytics, and what the best practices are for optimizing data ingestion, creating custom dashboards, leveraging automation, conducting regular audits, and ensuring IT team training.

We'll conclude this chapter with the next section, which shows a lab on how to use Azure Monitor to monitor hybrid infrastructure servers, specifically Amazon EC2 instances.

Lab – Configuring Azure Monitor with Arc for AWS

In this lab, we will show how to connect Azure Arc with Amazon EC2 instances and how to deploy the AMA to achieve a completely unified multi-cloud monitoring experience. This connection benefits organizations that want to leverage the unique capabilities of Azure and AWS while maintaining a unified management and monitoring layer.

Before we move on to the steps for the lab, let's ensure that we have deployed an Amazon EC2 instance. For this, use the AWS Management Console to deploy an EC2 instance with OS Image AMI Linux 2023 and call it `packt-arc-vm`. You can follow the steps at `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EC2_GetStarted.html` to deploy the Amazon EC2 instance.

Figure 8.4 – Amazon EC2 instance packt-arc-vm

Once the Amazon EC2 instance has been deployed, you can follow the following steps to register the Amazon EC2 instance in Azure Arc. This can be accomplished through different methods classified as interactive or at scale [5]. Since our Arc onboarding process consists of a single instance, we will use the deployment script method. From the Azure portal, you can generate a script and execute it on the machine to automate the installation and configuration steps of the agent.

1. First, navigate to the **Azure Arc** portal and select **Machines**. Then, click on **Add a machine**.

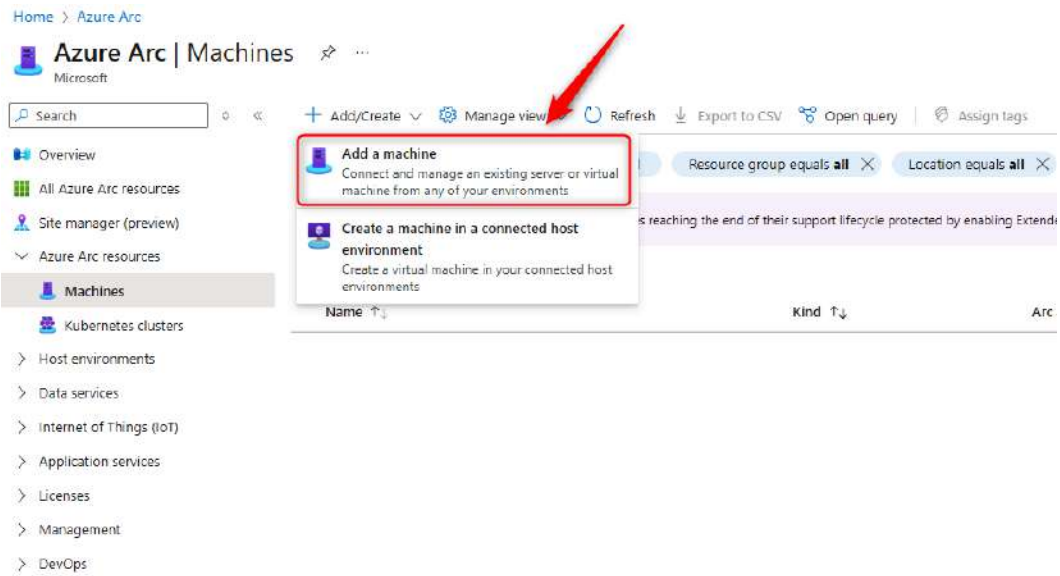

Figure 8.5 – Azure Arc portal

2. Next, select **Generate script** under the **Add a single server** option.

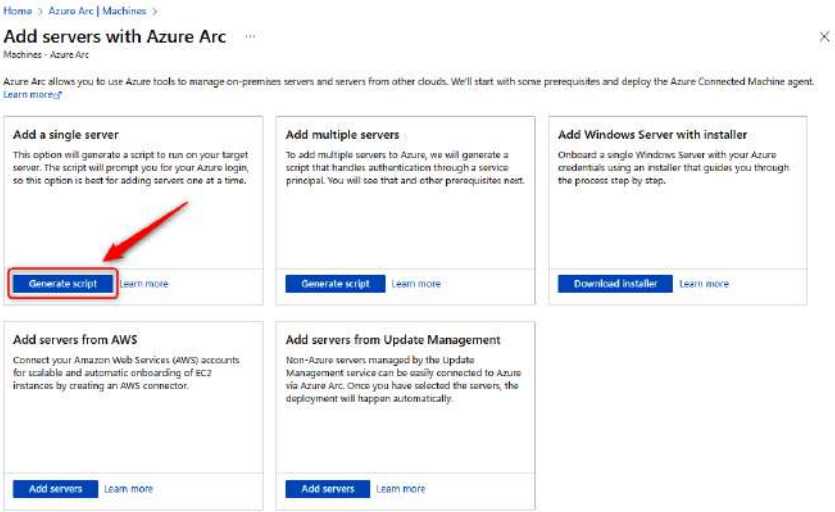

Figure 8.6 – Azure Arc onboarding options

3. Next, enter the required parameters as shown in *Figure 8.7*.

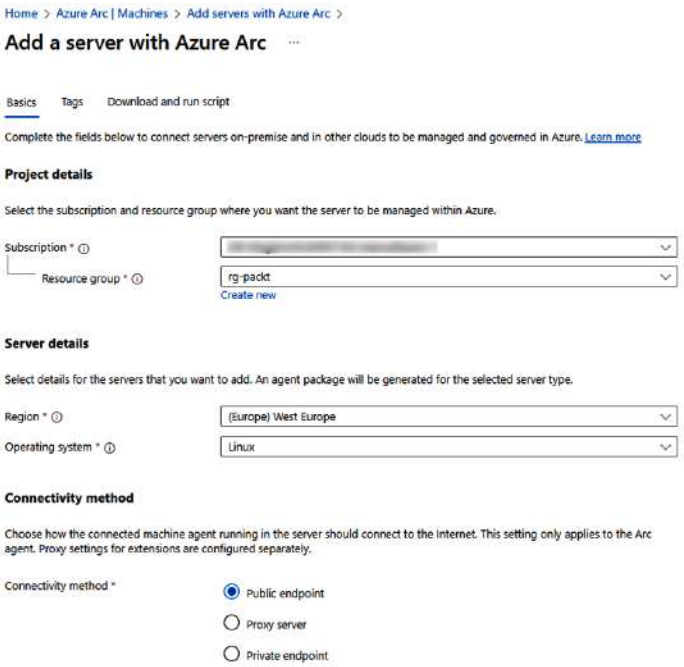

Figure 8.7 – Generate script for individual server

4. Azure Arc will then generate a bash or PowerShell script based on the chosen operating system.

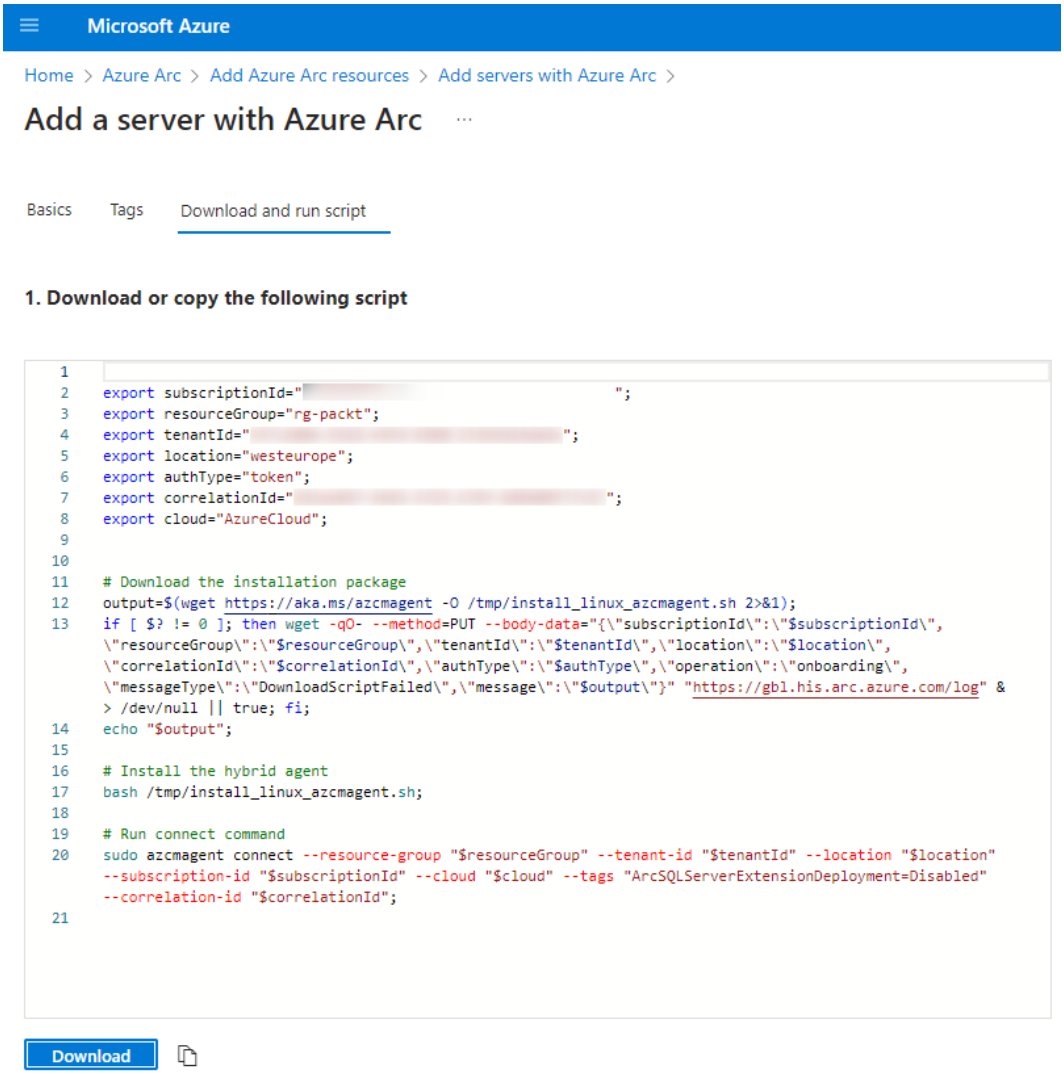

Figure 8.8 – Generated script for Linux server

5. Click **Download** and save it.

6. Connect to `packt-arc-vm` via **Secure Shell** (**SSH**). In our case, we will use **AWS Systems Manager** (**SSM**) to start an SSH session. Create a temporary `/tmp` directory and copy the script to a file called `OnboardingScript.sh`:

```
[root@ip-172-31-21-222 tmp]# ls
OnboardingScript.sh
[root@ip-172-31-21-222 tmp]#
```

Figure 8.9 – Copying the OnboardingScript inside the target machine

7. Next, make sure that you have the correct file permissions and run `OnboardingScript.sh` from the target server:

```
[root@ip-172-31-21-222 tmp]# ls
OnboardingScript.sh
[root@ip-172-31-21-222 tmp]# chmod +x OnboardingScript.sh
```

Figure 8.10 – Giving the current user execute permissions

The following figure shows some steps that the script will automatically execute during the onboarding process.

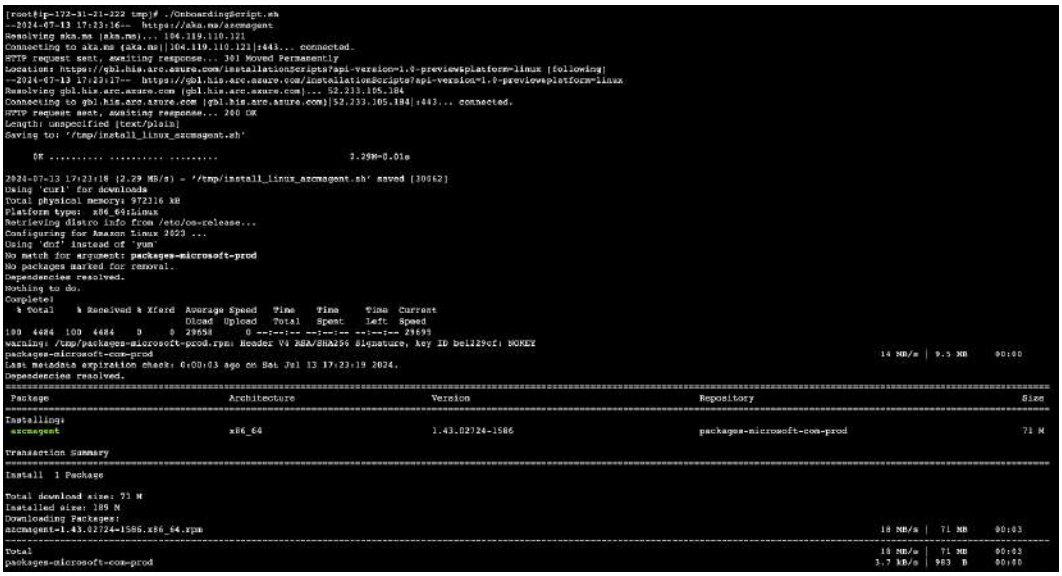

Figure 8.11 – Running OnboardingScript

During the onboarding process, a device code flow authentication is initiated and instructs you to open a browser page at `https://aka.ms/devicelogin`.

```
Transaction check succeeded.
Running transaction test
Transaction test succeeded.
Running transaction
  Preparing        :
  Running scriptlet: azcmagent-1.43.02724-1586.x86_64
Pre...Install
Creating himds group ...
Creating himds account ...
Creating arcproxy account ...

  Installing       : azcmagent-1.43.02724-1586.x86_64
  Running scriptlet: azcmagent-1.43.02724-1586.x86_64
Post...Install
Adding port 40342 to SELinux port policy
http_port_t                    tcp      40342, 80, 81, 443, 488, 8008, 8009, 8443, 9000
pegasus_http_port_t            tcp      5988
Created symlink /etc/systemd/system/multi-user.target.wants/himdsd.service → /usr/lib/systemd/system/himdsd.service.
Local configuration file is found
Getting status via systemd
Arc GC service is not running.
Configuring Arc GC service ...
Found systemd service controller...for Arc GC Service
Created symlink /etc/systemd/system/multi-user.target.wants/gcad.service → /usr/lib/systemd/system/gcad.service.
Service configured through systemd service controller. Gc Service
Local configuration file is found
Checked guest config disabled: 0
Getting status via systemd
Arc GC service is not running.
STARTING Arc GC
Getting status via systemd
EXT service is not running.
Configuring EXT service ...
Found systemd service controller...for Extension Service
Created symlink /etc/systemd/system/multi-user.target.wants/extd.service → /usr/lib/systemd/system/extd.service.
Service configured through systemd service controller. Extension Service
Local configuration file is found
Checked extd disabled: 0
STARTING EXT

  Verifying        : azcmagent-1.43.02724-1586.x86_64

Installed:
  azcmagent-1.43.02724-1586.x86_64

Complete!
Latest version of azcmagent is installed.
INFO    Connecting machine to Azure... This might take a few minutes.
INFO    Testing connectivity to endpoints that are needed to connect to Azure... This might take a few minutes.
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code FBMJ2BBWB to authenticate.
```

Figure 8.12 – Device code authentication flow

8. Enter the code displayed in your terminal as is shown in the following figure:

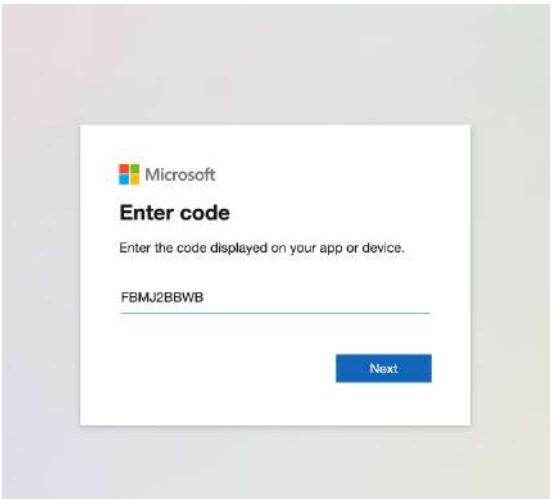

Figure 8.13 – Device code pasted at https://aka.ms/devicelogin

9. Next, sign in with your account credentials in the browser.
10. Once you log in successfully, the onboarding process will continue and if everything is fine, you will see the following details indicating the successful onboarding of the virtual machine to Azure Arc.
11. In case the agent does not start after the installation is complete, check the logs in the `var/opt/azcmagent/log` directory to analyze what happened.

Figure 8.14 – Successful onboarding of packt-arc-vm to Azure Arc

12. You can now check the `packt-arc-vm` agent status from the Azure Arc **Machines** portal, in the **Overview** tab of the machine.

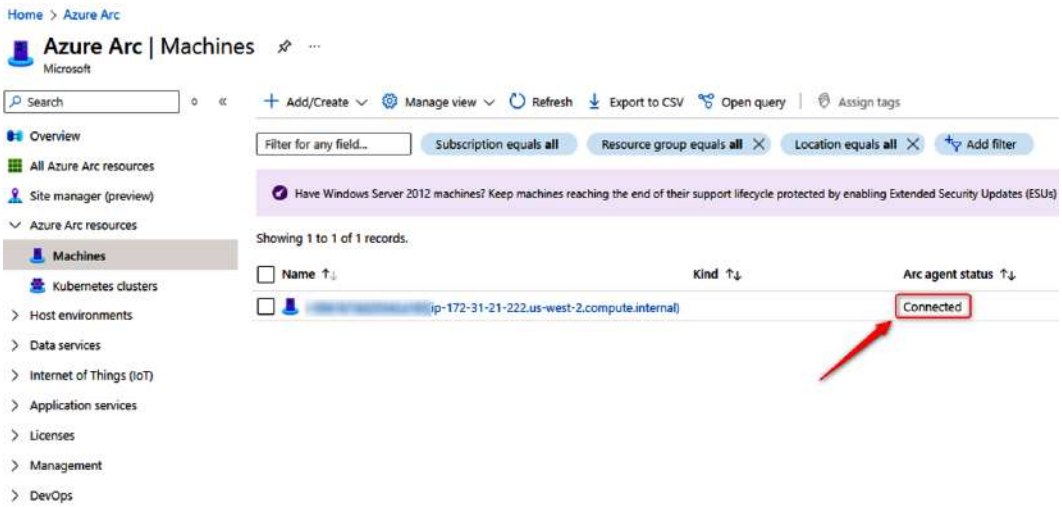

Figure 8.15 – Arc Machines portal

The following figure shows the `packt-arc-vm` agent status in the **Overview** tab of the machine.

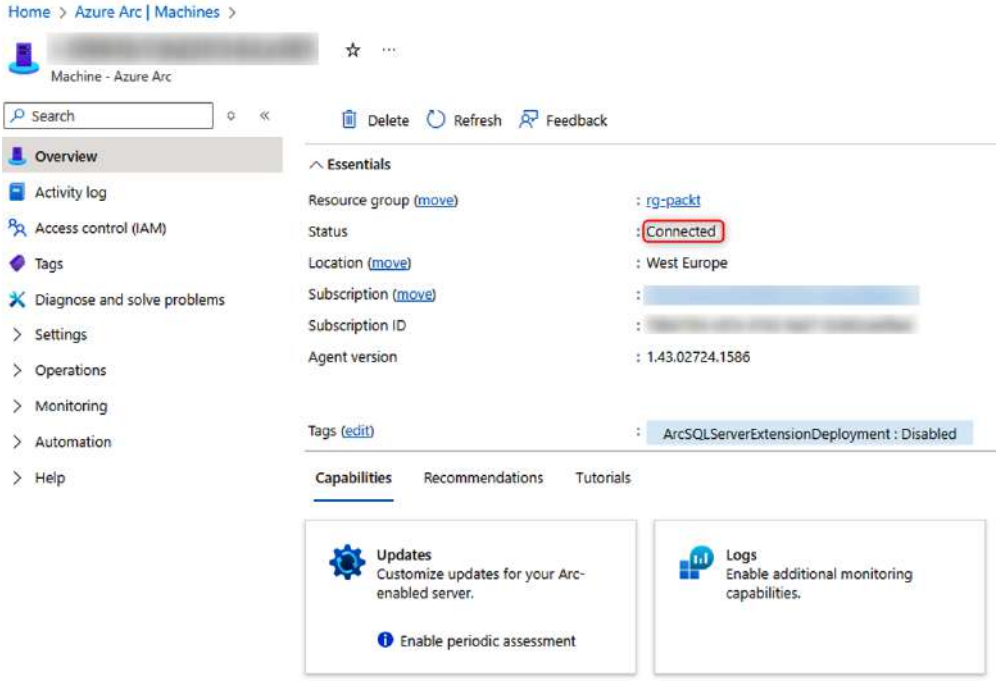

Figure 8.16 – packt-arc-vm Overview tab

The following figure shows more details of the `packt-arc-vm` **Overview** tab reported by Arc agent.

Figure 8.17 – packt-arc-vm Overview tab details

Now, you can manage `packt-arc-vm` from Azure Arc as you manage Azure VMs. This onboarding process has created a separate resource, and you can now monitor the VM, apply Azure policies, and so on.

13. To monitor this hybrid workload, we will apply the lessons of *Chapter 3* about DCR. We will use the DCR portal creation method for AMA installation. Note that this method not only creates the rule but associates it to `packt-arc-vm` and automatically installs AMA on it if it detects that the VM doesn't already have it.

    The following figure shows the first parameter tab of the wizard for DCR creation.

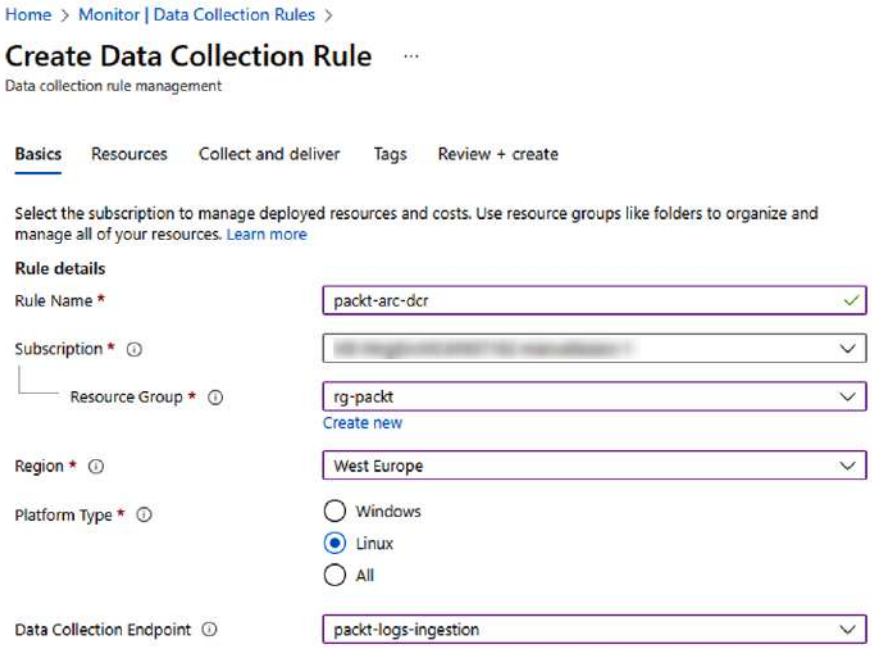

Figure 8.18 – packt-arc-dcr

14. Add `packt-arc-vm` as a resource in the DCR to collect data from.

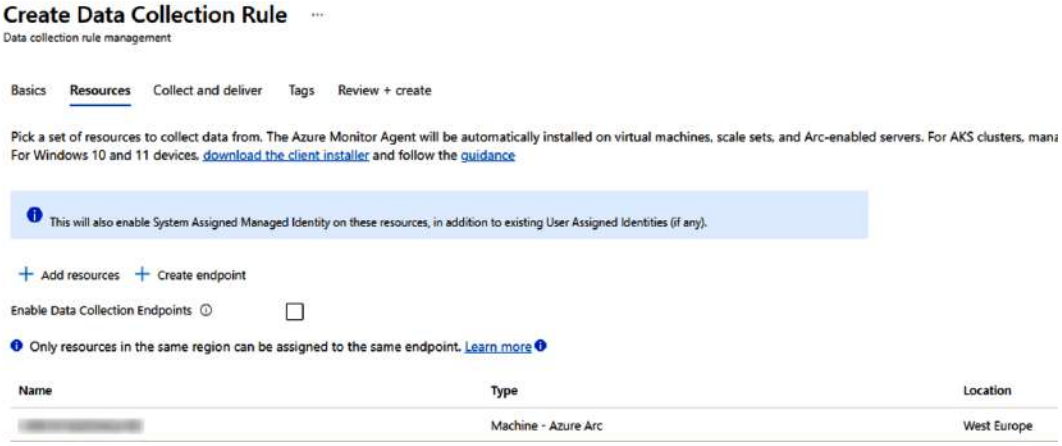

Figure 8.19 – Entering packt-arc-vm resource in the DCR

15. Configure the data sources you want to collect and the destination to send the data to. In our case, we choose basic performance counters and the `law-packt` log analytics workspace as the destination.

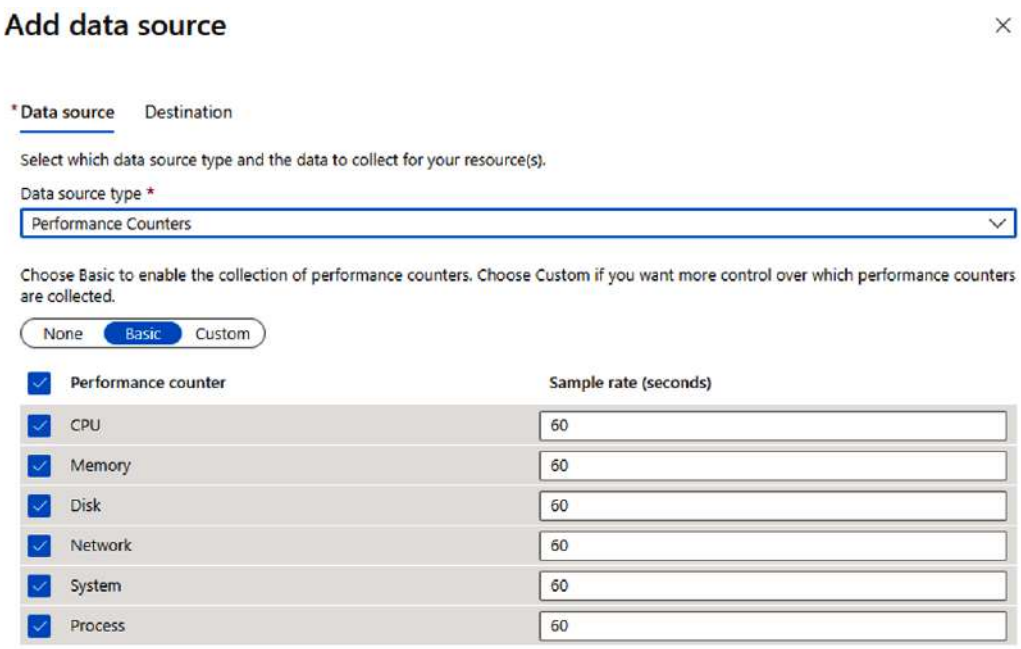

Figure 8.20 – DCR data sources

16. The following figure shows the destination configuration to send the data to.

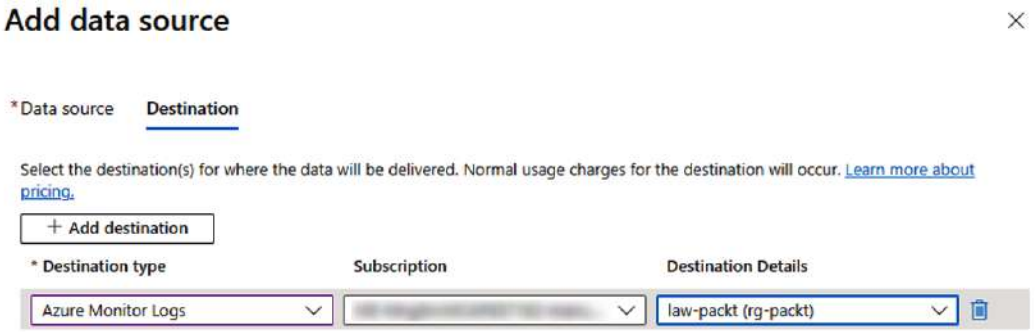

Figure 8.21 – DCR destination

17. Review the DCR configuration and create it.

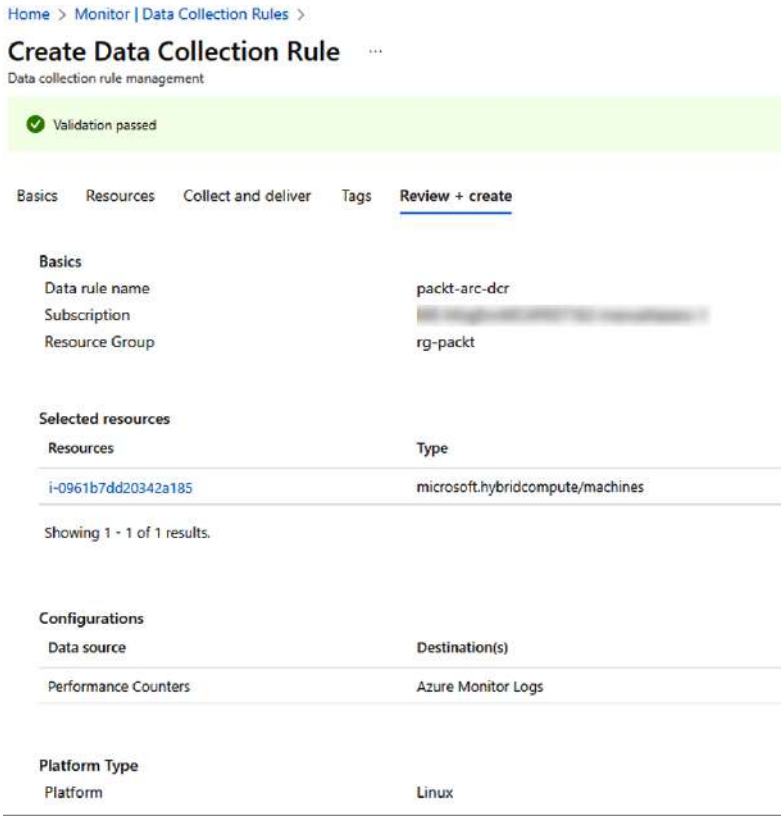

Figure 8.22 – DCR configuration review

We have concluded the AMA installation process, and you can now apply the techniques of analyzing and visualization we explained in previous chapters.

Let's review and clarify some aspects after the installation process. As we mentioned, the DCR creation process from the Azure portal triggers several steps – among others, DCR creation, DCR association, and the AMA installation initiation process.

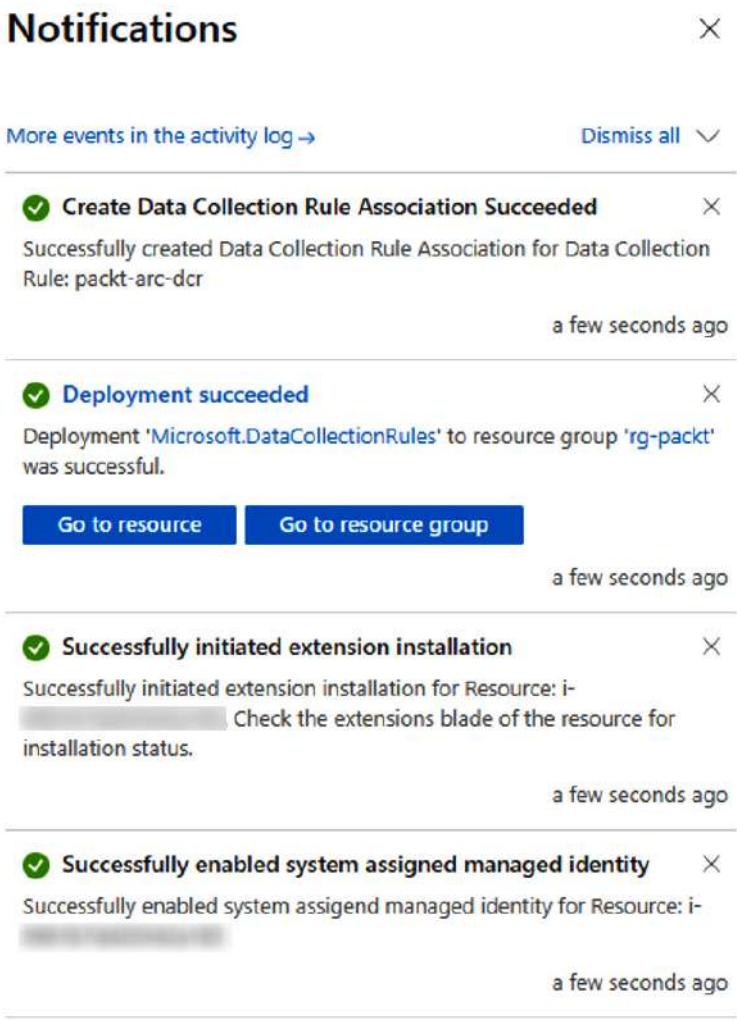

Figure 8.23 – DCR post-creation processes

As shown in the following figure, you can generate the ARM template of the resources deployed to check the properties we defined in the Azure portal process and review the schema of the DCR based on what you learn about DCR schemas in the *Exploring customization options for tailored monitoring* section in the *Appendix* at the end of this book.

```
    "dataFlows": {
        "type": "Array",
        "metadata": {
            "description": "The specification of data flows."
        }
    },
    "destinations": {
        "type": "Object",
        "metadata": {
            "description": "The specification of destinations."
        }
    },
    "tagsArray": {
        "type": "Object"
    },
    "apiVersion": {
        "type": "String",
        "metadata": {
            "description": "specifies the api version to use when deploying data collection rule template."
        }
    },
    "platformType": {
        "type": "String",
        "metadata": {
            "description": "The specification of the machine os type."
        }
    }
},
"resources": [
    {
        "type": "Microsoft.Insights/dataCollectionRules",
        "apiVersion": "[parameters('apiVersion')]",
        "name": "[parameters('ruleName')]",
        "location": "[parameters('location')]",
        "tags": "[parameters('tagsArray')]",
        "kind": "Linux",
        "properties": {
            "dataSources": "[parameters('dataSources')]",
            "destinations": "[parameters('destinations')]",
            "dataFlows": "[parameters('dataFlows')]",
            "dataCollectionEndpointId": "/subscriptions/                                /resourceGroups/rg-packt/providers/Microsoft.Insights/dataCollectionEndpoints/packt-logs-ingestion",
            "streamDeclarations": {}
        }
    }
],
"outputs": {
    "dataCollectionRuleId": {
        "type": "String",
        "value": "[resourceId('Microsoft.Insights/dataCollectionRules', parameters('ruleName'))]"
    }
}
}
```


Figure 8.24 – DCR ARM template

The following figure shows the `dataSources` parameters of the resources deployed.

Figure 8.25 – DCR ARM template dataSources parameters

The following figure shows the DCR ARM template `dataFlows` and `destinations` parameters of the resources deployed.

Figure 8.26 – DCR ARM template dataFlows and destinations parameters

You can go to the **Extensions** tab from `packt-arc-vm` to check the status of the AMA extension. Note that the agent status is **Failed**. If you click the **View details** tab, you will see the error message and why the agent installation failed. This was due to the version of the AWS Linux AMI deployed.

Figure 8.27 – Error message about why the AMA installation failed

We did this intentionally to show the importance of checking the requirements for installing not only the Azure Connected Machine agent but all agents, including the AMA. At the time of writing this book, AMA does not support the Amazon Linux 2023 operating system. You can select the previous Amazon Linux AMI version and repeat all the preceding steps to complete the lab successfully. After that, you will be able to apply the lessons learned in other chapters to successfully monitor a hybrid workload.

With this lab, we have shown how to connect Azure Arc with Amazon EC2 instances and how to deploy the AMA to achieve a completely unified multi-cloud monitoring experience. As we have mentioned, this process can be repeated at scale on multiple servers in different environments outside of Azure following the most appropriate deployment method and always taking into account the requirements of Azure Arc and AMA [1, 6].

Summary

In this chapter, we covered how Azure Monitor integrates with Azure Arc and SCOM Managed Instance to enable detailed monitoring of other cloud platforms and on-premises systems.

The chapter introduced Azure Arc as a solution for monitoring servers outside of Azure. It described how Azure Arc connects Windows and Linux servers running on other cloud providers or a corporate network and allows them to take advantage of Azure monitoring tools such as Log Analytics, VM insights, and Azure dashboards. In addition, we showed the different methods for deploying the Azure Monitor agent to Azure Arc-enabled servers, discussing the benefits and limitations of each method, including Azure Policy, the Azure portal, the Azure CLI, PowerShell, ARM templates, and Azure Automation. The chapter also provided sample commands and templates for each method.

We then moved on to Azure Monitor SCOM Managed Instance as an alternative for monitoring on-premises scenarios. The chapter explained the benefits of integrating Azure Monitor with SCOM Managed Instance, such as unified monitoring, advanced analytics, and enhanced alerting. The chapter also detailed the prerequisites and steps to set up the integration with Azure Log Analytics and concluded with best practices to optimize data ingestion, create custom dashboards, leverage automation, perform regular audits, and ensure IT team training.

Finally, we provided a lab practice on how to use Azure Monitor to monitor hybrid infrastructure servers, specifically Amazon EC2 instances. We showed how to connect Azure Arc with Amazon EC2 instances and how to deploy the Azure Monitor agent for a completely unified multi-cloud monitoring experience.

As we have concluded our introduction to the specific tools and techniques available in Azure for monitoring resources in hybrid and multi-cloud environments, we will now turn our attention to Azure Monitor integration with third-party tools. In the next chapter, we will explore the integration possibilities offered by Azure Monitor and how to leverage external solutions for enhanced observability.

Further reading

Here, you can find the links to expand your knowledge about specific concepts not covered in this book but referenced in this chapter:

- [1] Azure Arc supported operating systems: `https://learn.microsoft.com/en-us/azure/azure-arc/servers/prerequisites#supported-operating-systems`.
- [2] Plan and deploy Azure Arc-enabled servers: `https://learn.microsoft.com/en-us/azure/azure-arc/servers/plan-at-scale-deployment`.
- [3] How to create a Python 3.8 runbook: `https://learn.microsoft.com/en-us/azure/automation/learn/automation-tutorial-runbook-textual-python-3`.
- [4] Create an Azure Monitor for SCOM Managed Instance: `https://learn.microsoft.com/en-us/azure/azure-monitor/scom-manage-instance/overview#next-steps`.
- [5] Azure Connected Machine agent deployment options: `https://learn.microsoft.com/en-us/azure/azure-arc/servers/deployment-options`.
- [6] Azure Monitor Agent supported operating systems: `https://learn.microsoft.com/en-us/azure/azure-monitor/agents/azure-monitor-agent-supported-operating-systems#on-premises-and-other-clouds`.

9

Integrating with Third-Party Tools

Throughout the previous chapters, we have introduced the extensive features that Azure Monitor provides, exploring its capabilities in monitoring, logging, and alerting. We've seen how Azure Monitor can offer comprehensive insights into the performance and health of your applications and infrastructure.

However, we recognize that the built-in capabilities of Azure Monitor may not suffice for every scenario or requirement. Organizations often seek to extend their monitoring solutions to gain additional insights, leverage specialized tools, or integrate with existing systems.

In this chapter, we will explore the various options available to integrate Azure Monitor with third-party systems, such as Datadog, Elastic Cloud, New Relic, Dynatrace, or Logz.io, allowing you to enhance your observability and customize your monitoring strategy to meet your specific needs.

The chapter will cover the following topics:

- Exploring integration possibilities with Azure Monitor
- Leveraging external solutions for enhanced observability

Technical requirements

In this chapter, we will use a terminal together with Azure PowerShell and CLI tools to show how information can be extracted from Azure Monitor and integrated into third-party solutions. You would also need the Azure subscription used in previous chapters with an existing Azure Log Analytics workspace.

Exploring integration possibilities with Azure Monitor

As mentioned in the introduction, Azure Monitor provides a complete observability platform both inside and outside Azure. However, it may not be enough for every scenario. Every organization has unique monitoring requirements based on its infrastructure, applications, and business objectives. For instance, if your organization has invested in Splunk for security monitoring or any other third-party platform, integrating it with Azure Monitor allows you to leverage its advanced security features and ensure compliance with regulatory standards.

Furthermore, in a multi-cloud or hybrid cloud environment, achieving comprehensive visibility across all platforms can be challenging. Integrating Azure Monitor with third-party tools helps bridge this gap, providing a unified view of your entire infrastructure. This is particularly useful for organizations that use multiple cloud providers or manage on-premises and cloud resources. By centralizing data from different sources, you can gain a global view of your operations and make more informed decisions.

Ultimately, each third-party tool brings unique capabilities that complement Azure Monitor as your monitoring solution. To fully leverage those unique capabilities, it is essential to understand the various methods available for exporting data from Azure Monitor. Exporting this data ensures that the telemetry collected within Azure can be analyzed, visualized, and utilized within your preferred platforms, whether it's through direct integration using APIs, setting up data export to external systems, or utilizing intermediary services. In the following section, we will explore these options in detail, guiding you through the steps required to export and integrate your monitoring data. We'll explore three primary methods to export data from Azure Monitor – using the Azure Monitor REST API directly or through PowerShell/CLI, exporting logs to Azure Storage, and leveraging Event Hubs for real-time streaming.

Using Azure Monitor REST API

In *Chapter 3*, we explored the ingestion capabilities of the Azure Monitor REST API, which allows for the seamless collection of custom telemetry data from various sources, enabling comprehensive monitoring and observability. This API not only supports the ingestion of metrics or logs into Azure Monitor but also extends its functionality to extract this information.

By leveraging the Azure Monitor REST API, users can programmatically access and retrieve detailed monitoring data, facilitating integration with third-party tools and custom applications. This dual capability ensures that organizations can both centralize their monitoring data within Azure and efficiently export it to enhance their observability strategies, using external solutions.

Let's show an example of how you can retrieve data from each type of telemetry that Azure Monitor supports.

Extracting Azure metrics

The Metrics REST API supports not only the retrieval of metrics values but also the metric definition and the metric dimension values. Information can be retrieved from a single instance of a specific resource or multiple resources.

The first step to extract metrics information from a resource on Azure using the Azure Monitor REST API is authentication. It ensures that only authorized users can retrieve monitoring data, thus maintaining the security and integrity of your Azure environment. The primary method of authentication is via OAuth 2.0, using Microsoft Entra ID to obtain an access token. This token must be included in the header of your API requests to authenticate and authorize access.

In a production environment, you would probably use a service principal, as described in *Chapter 3* when ingesting custom data in Azure Monitor. However, in this case, we will show a simpler scenario for demonstration purposes, using the authentication token of your user. We will use the Azure CLI to obtain it, running the following command:

```
az account get-access-token
```

After that, the second step is to identify the resource and the metric we are interested in. The endpoint for retrieving metrics is structured as follows:

```
https://management.azure.com/subscriptions/{subscriptionId}/
resourceGroups/{resourceGroupName}/providers/
{resourceProviderNamespace}/{resourceType}/{resourceName}/providers/
microsoft.insights/metrics?api-version=2023-10-01&metricnames={metricN
ames}&timespan={timespan}
```

You need to replace placeholders with your subscription ID, resource group name, resource provider namespace, resource type, resource name, desired metric names, and the time span for the metrics.

As an example, let's obtain the CPU usage of the virtual machine we used in *Chapter 2* for the last 24 hours. We will use `curl` from the command line:

```
curl --location --request GET 'https://management.azure.com/
subscriptions/{subscriptionId}/resourceGroups/packet-book/providers/
Microsoft.Compute/virtualMachines/chapter2-vm/providers/microsoft.
insights/metrics?api-version=2023-10-01&metricnames=Percentage%20
CPU&timespan=2024-05-26T00:00:00Z/2024-05-27T00:00:00Z' --header
'Content-Type: application/json' --header 'Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJ…'
```

The output will contain a few details about the request you have made, together with a time series of the values available for the specific metric and period.

A detailed walk-through of the whole set of options for this type of request is available in the Microsoft Azure Monitor documentation, linked to in the *Further reading* section [1]. It contains more examples of retrieving metric definitions and dimension values, as well as examples of querying metrics for multiple resources.

However, when dealing with large-scale environments or the need to extract numerous metrics over extended periods, you might face challenges with rate limits. Azure Monitor imposes rate limits on the number of API calls you can make within a specific time frame. Exceeding these limits can result in throttling, where additional requests are temporarily blocked.

To address these challenges, Azure Monitor provides the **getBatch** API, which allows you to retrieve multiple metrics in a single request. This approach helps to reduce API calls by batching multiple metrics queries into a single API call and improving efficiency, minimizing the performance overhead through the consolidation of requests.

The authentication process is like the previous step; however, both the URL and the way to request the metrics are modified. Instead of using a GET HTTP request, the `getBatch` API uses a POST HTTP request, but both share a common set of parameters and response formats.

As an example, let's obtain again the CPU usage of the virtual machine we used in *Chapter 2* for the last 24 hours and another two different VMs.

The API endpoint will have the following structure:

```
https://{azureRegion}.metrics.monitor.azure.com/subscriptions/
{subscriptionId}/metrics:getBatch?starttime=2024-05-
26T00:00:00Z&endtime=2024-05-27T00:00:00Z&interval=PT1H&metricNames
pace=microsoft.compute%2Fvirtualmachines&metricnames=Percentage%20
CPU&api-version=2023-10-01
```

The body of the `POST` request will contain a JSON object with all the IDs of the resources we are interested in:

```
{
    "resourceids": [
       "/subscriptions/{subscriptionId}/resourceGroups/
       packet-book/providers/Microsoft.Compute/
       virtualMachines/chapter2-vm/",
       "/subscriptions/{subscriptionId}/resourceGroups/
       packet-book/providers/Microsoft.Compute/
       virtualMachines/chapter10a-vm/",
       "/subscriptions/{subscriptionId}/resourceGroups/
       packet-book/providers/Microsoft.Compute/
       virtualMachines/chapter10b-vm/"
    ]
}
```

We will use `curl` from the command line to submit our request:

```
curl --location --data '{"resourceids": [ "/subscriptions/
{subscriptionId}/resourceGroups/packet-book/providers/Microsoft.
Compute/virtualMachines/chapter2-vm/", "/subscriptions/
{subscriptionId}/resourceGroups/packet-book/providers/Microsoft.
Compute/virtualMachines/chapter10a-vm/", "/subscriptions/
```

```
{subscriptionId}/resourceGroups/packet-book/providers/
Microsoft.Compute/virtualMachines/chapter10b-vm/"]}' --request
POST 'https:// {azureRegion}.metrics.monitor.azure.com/
subscriptions/{subscriptionId}/metrics:getBatch?starttime=2024-05-
26T00:00:00Z&endtime=2024-05-27T00:00:00Z&interval=PT1H&metricNames
pace=microsoft.compute%2Fvirtualmachines&metricnames=Percentage%20
CPU&api-version=2023-10-01' --header 'Content-Type: application/json'
--header 'Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIs
[…]'
```

In a single request, we get all the information across the three virtual machines. Let's move on now to understand how to use the Azure Monitor REST API to extract log details.

Extracting Azure logs

The Azure Monitor REST API to extract Azure logs requires the same authentication as shown in the previous section. However, in this case, we need to use application credentials instead of our own token. The API endpoint is outside the default management domain, and we are required to authorize this application to read the data before the call can be executed.

After you have created your application, as covered in *Chapter 3*, you will need to go to the **API permissions** menu inside the properties of your application and look for **Log Analytics API**. After that, you should click on **Delegated permissions** and assign the **Data.Read** permission by clicking the checkbox next to it, as shown in the following screenshot.

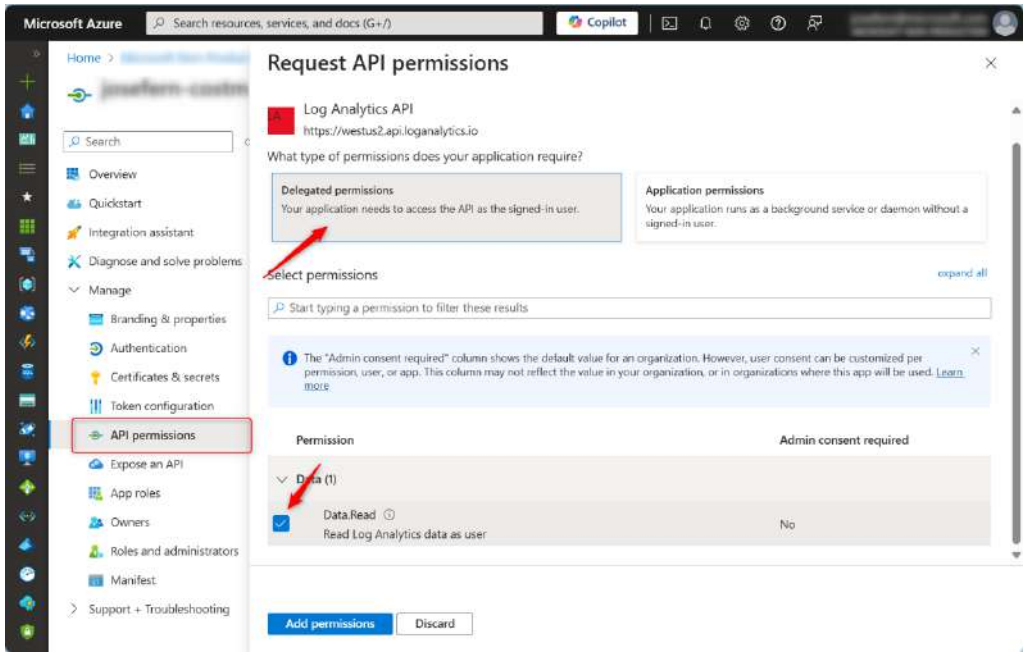

Figure 9.1 – Configuration of the API permissions

Once the permissions are assigned, you should be able to get an access token through the Azure CLI **after logging in with the application credentials** instead of yours.

The endpoint to query activity log data is structured as follows:

```
https://api.loganalytics.azure.com/{api-version}/workspaces/
{workspaceId}/query?[parameters]
```

You need to specify several key parameters in your API request. The `api-version` parameter, which identifies the version of the API you are using, should be set to `v1` to ensure compatibility with the latest features and updates. Additionally, you must include your `workspaceId`, which uniquely identifies your Azure workspace where the logs are stored. Along with these, the `parameters` parameter is essential, as it encapsulates the specific data required for the query, such as the time range, resource group, and the types of events you wish to retrieve.

When using this API to query data, you can use both `GET` and `POST` HTTP methods, depending on how you want to structure your request. For a `GET` request, the parameters are included directly in the query string. For example, let's obtain the average latency in milliseconds for our availability test, configured for the web app deployed in *Chapter 8*:

```
curl --location --request GET 'https://api.loganalytics.azure.com/v1/
workspaces/{workspaceId}/query?query=AppAvailabilityResults%20%7C%20
summarize%20avg%28DurationMs%29%20by%20Location' --header 'Content-
Type: application/json' --header 'Authorization: Bearer […]'
```

For a `POST` request, the body of the request must be valid JSON and must include the `Content-Type: application/json` header. The parameters are included as properties in the JSON body. If the `timespan` parameter is specified in both the query string and the JSON body, the timespan used will be the intersection of the two values. For example, to obtain the same information as with the HTTP `Get` call, we would need to submit the following query:

```
curl --location --request POST --data '{"query": "
AppAvailabilityResults | summarize avg(DurationMs) by Location"
}'  'https://api.loganalytics.azure.com/v1/workspaces/{workspaceId}/
query?query=AppAvailabilityResults%20%7C%20summarize%20
avg%28DurationMs%29%20by%20Location' --header 'Content-Type:
application/json' --header 'Authorization: Bearer […]'
```

In this case, the JSON body contains the query parameter. This approach is useful for more complex queries or when you need to include additional parameters in a structured format.

After covering this scenario, let's move on now to the last one – on how to use the Azure Monitor REST API to extract information about the activity logs.

Exporting activity logs

Before making requests to the Azure Monitor REST API to extract activity logs, you must authenticate using Microsoft Entra ID, as shown at the beginning of the *Extracting Azure metrics* section. In this case, you can use your own generated token or the one created for your application. The endpoint to query activity log data is structured as follows:

```
https://management.azure.com/subscriptions/{subscriptionId}/providers/
Microsoft.Insights/eventtypes/management/values?api-version=2015-04-
01&$filter={filter}&$select={select}
```

In this URL, you replace placeholders with your subscription ID, filter criteria, and selected properties. The `$filter` parameter is essential for narrowing down the set of returned logs. You can filter by time range, resource group, specific resource, or other criteria. The `$select` parameter allows you to specify which fields to include in the response, such as event name, operation name, status, event timestamp, correlation ID, and level, to reduce the payload size and focus on relevant data.

For example, to get all the logs after a specific date, you should use the following query with the `$filter` parameter:

```
curl --location --request GET 'https://management.azure.
com/subscriptions/{subscriptionId}/providers/Microsoft.
Insights/eventtypes/management/values?api-version=2015-04-
01&$filter=eventTimestamp%20ge%202024-06-16T04%3A36%3A37.6407898Z'
--header 'Content-Type: application/json' --header 'Authorization:
Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng […]'
```

If you want to restrict the results to a specific resource group and return only the results for a specific resource group, you can add an extra condition to the filter:

```
curl --location --request GET 'https://management.azure.
com/subscriptions/{subscriptionId}/providers/Microsoft.
Insights/eventtypes/management/values?api-version=2015-04-
01&$filter=eventTimestamp%20ge%202024-06-16T04%3A36%3A37.6407898Z%20
and%20resourceGroupName%20eq%20%27packt-book%27' --header
'Content-Type: application/json' --header 'Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng […]'
```

If the information returned is more than you need, you can use the `$select` parameter to return only the specific fields relevant to you, as shown in the following example. Only four properties are returned from the whole available set:

```
curl --location --request GET 'https://management.azure.
com/subscriptions/{subscriptionId}/providers/Microsoft.
Insights/eventtypes/management/values?api-version=2015-04-
01&$filter=eventTimestamp%20ge%202024-06-16T04%3A36%3A37.6407898Z%20
and%20resourceGroupName%20eq%20%27packt-book%27&$s-
elect=eventName,operationName,status,eventTimestamp' --header
'Content-Type: application/json' --header 'Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng […]'
```

By using the Azure Monitor REST API and the Metrics Batch API, as shown in the previous examples, you can efficiently extract metrics information from your Azure resources, ensuring that you have the necessary data to monitor your cloud environment outside Azure Monitor.

While this approach provides a robust and flexible method to access detailed metrics and monitor data, integrating it into your scripts can sometimes be cumbersome. The need to manage authentication tokens, construct HTTP requests, and handle the responses programmatically requires significant effort, especially when dealing with large-scale environments or frequent data extraction tasks. To simplify this process, Azure offers alternative methods using PowerShell and the Azure CLI.

Using Azure PowerShell and CLI for log extraction

Azure PowerShell and Azure CLI tools provide streamlined commands and built-in functionality to extract logs and metrics, making it easier to incorporate monitoring data into your automation scripts and daily workflows. By leveraging PowerShell or CLI, you can efficiently retrieve the necessary information without the complexity associated with REST API calls, enabling quicker and more effective integration with your existing systems.

Using Azure PowerShell

Azure PowerShell provides the `Get-AzLog` cmdlet (check the *Further reading* section for more details [2]), which enables you to query and retrieve activity logs with ease. This cmdlet simplifies the process of extracting logs by encapsulating the necessary API calls into a single, user-friendly command.

Use `Connect-AzAccount` to authenticate and connect to your Azure subscription, and after that, use `Get-AzLog` to fetch logs for a specific resource group or time range. The following example retrieves the latest 50 records from the previous week:

```
Connect-AzAccount
Get-AzLog -StartTime (Get-Date).AddDays(-7) -EndTime (Get-Date)
-ResourceGroupName {YourResourceGroup} -MaxRecord 50
```

The response will provide a detailed activity log entry for each event with information about the HTTP request, its properties, the resource affected, and the status of the request. The `Get-AzLog` command provides a wide set of options to customize your request filtering by resource provider, resource group, or resource ID.

If you need to extract details about a metric instead of the activity logs, PowerShell provides the `Get-AzMetric` [3] command. The following example retrieves the information about the `Percentage CPU` metric used previously:

```
Get-AzMetric -ResourceId "/subscriptions/{subscriptionId}/
resourceGroups/packet-book/providers/microsoft.compute/
virtualmachines/chapter2-vm" -TimeGrain 00:01:00 -MetricName
"Percentage CPU"
```

Similarly, if you want to retrieve information from a log inside your Azure Monitor Log Analytics workspace, the `Invoke-AzOperationalInsightsQuery` command is provided by PowerShell to make queries to your workspace. Its name could be confusing, but as we discussed in *Chapter 1*, this was the old name of the service and PowerShell commands have kept it.

The following example retrieves the latest result entry from the `AppAvailabilityResults` table that stores all the availability information, from the web application deployed in *Chapter 8*:

```
Invoke-AzOperationalInsightsQuery -WorkspaceId
{LogAnalyticsWorkSpaceId} -Query "AppAvailabilityResults | take 1"
```

Let's now review what the Azure CLI offers to extract information from Azure Monitor.

Using Azure CLI

Azure CLI provides the `az monitor activity-log list` command, which offers similar functionality to retrieve activity logs. The CLI is a cross-platform tool that can be used on Windows, macOS, and Linux, making it a versatile option for automation.

Use `az login` to authenticate and connect to your Azure account and, after that, `az monitor activity-log list` to fetch logs for a specific resource group or time range. The following example retrieves the information for the last seven days inside the specific resource group included:

```
az login
az monitor activity-log list --start-time (Get-Date).AddDays(-
7)  --end-time (Get-Date) --resource-group {YourResourceGroup}
```

If you need to extract details about a metric instead of the activity logs, Azure CLI provides the `az monitor metrics list` command. The following example retrieves the information about the `Percentage CPU` metric used previously:

```
az monitor metrics list --resource {resourceID} --metric "Percentage
CPU"
```

Similarly, if you want to retrieve information from a log inside your Azure Monitor Log Analytics workspace, the `az monitor log-analytics query` command is provided by the Azure CLI to make queries to your workspace.

The following example retrieves the latest result entry from the `AppAvailabilityResults` table, as shown in the PowerShell example:

```
az monitor log-analytics query -w { LogAnalyticsWorkSpaceId }
--analytics-query "AppAvailabilityResults | take 1"
```

By using Azure PowerShell and CLI, you can streamline the process of extracting logs and metrics, integrating them more easily into your automation scripts and monitoring workflows. These tools eliminate the need for complex API calls, making log extraction more straightforward and accessible for users of all skill levels.

Let's continue by exploring other alternatives to export Azure monitoring information, without using the Azure Monitor APIs or its wrap for PowerShell and the Azure CLI.

Exporting logs and metrics to Azure Storage or Azure Event Hubs

Exporting logs to Azure Storage is a straightforward method to archive monitoring data and make it accessible for third-party tools. This method is particularly useful for long-term retention and batch-processing scenarios. On the other side, **Azure Event Hubs** provides a powerful platform for real-time data streaming, making it ideal for scenarios that require immediate processing and analysis of monitoring data.

Both export options are provided through the **Diagnostic settings** menu available for each resource, as shown in the following screenshot. It is possible to select not only a Log Analytics workspace as a destination for our logs but also to stream them to an Event Hub, or archive them to a storage account.

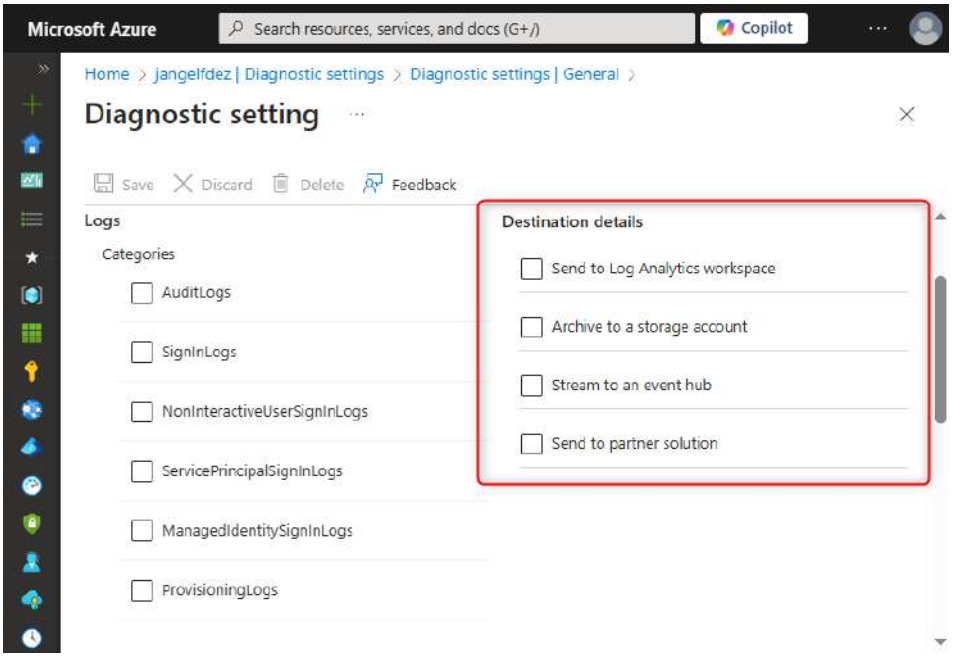

Figure 9.2 – Destination options for logs and metrics

Exporting the information to a storage account allows us to store large volumes of log data cost-effectively, provides durability through a reliable storage product with redundancy options, and improves its accessibility by third-party tools for ingestion and analysis. For example, an organization using Elastic for log analysis can periodically ingest log data from Azure Storage, enabling advanced search and analytics on the archived logs.

Streaming information to Event Hubs allows us to configure a low-latency pipeline, with near real-time data ingestion and processing that is scalable. This pipeline can handle large volumes of data with high throughput. For example, an organization using Splunk for security monitoring can stream log data to Event Hubs and set up Splunk to consume and process the data in real time, enabling immediate detection of and responses to security incidents.

In summary, to enable logs and metrics to be exported to Azure Storage, you need to do the following:

1. **Configure diagnostic settings**: In the Azure portal, navigate to your resource and configure the diagnostic settings to specify which logs should be sent to Azure Storage.
2. **Select a storage account**: Choose an existing storage account or create a new one to store your logs.
3. **Retain data**: Set retention policies based on your organization's requirements.
4. **Access logs**: Third-party tools can access the logs by reading from the storage account, using Azure Storage SDKs or the REST API.

Alternatively, to enable the forwarding of the logs and metrics to Event Hubs, you would need to do the following:

1. **Configure diagnostic settings**: In the Azure portal, set up diagnostic settings for your resources to send logs and metrics to Event Hubs.
2. **Create an event hub**: Ensure that you have an Event Hub namespace and an event hub to receive the data.
3. **Stream data**: Configure the diagnostic settings to route data to the event hub.
4. **Use consumer applications**: Use consumer applications to read and process the data from Event Hubs. This can be custom applications, Azure Stream Analytics jobs, or third-party platforms that support Event Hubs integration.

In both cases, logs and metrics are exported before they have been ingested inside your Log Analytics Workspace; however, in addition to exporting logs before they are ingested, Azure Monitor also offers a feature within Log Analytics to export data directly from selected tables. This capability allows you to route your monitoring data to Azure Storage for long-term retention, or to Azure Event Hubs for real-time streaming and integration with third-party applications.

The export feature in Azure Log Analytics allows you to continuously export data from specific tables in your Log Analytics workspace to Azure Storage or Azure Event Hubs as they arrive. This is useful for archiving data, performing additional analysis, or integrating with other systems that consume monitoring data. This feature uses Microsoft Azure's internal backbone, and information doesn't leave the internal Microsoft network.

To set up export to Azure Storage, navigate to your Log Analytics workspace in the Azure portal, and under the **Settings** section, select **Data export**. Click on **Create export rule**, as shown in the following screenshot.

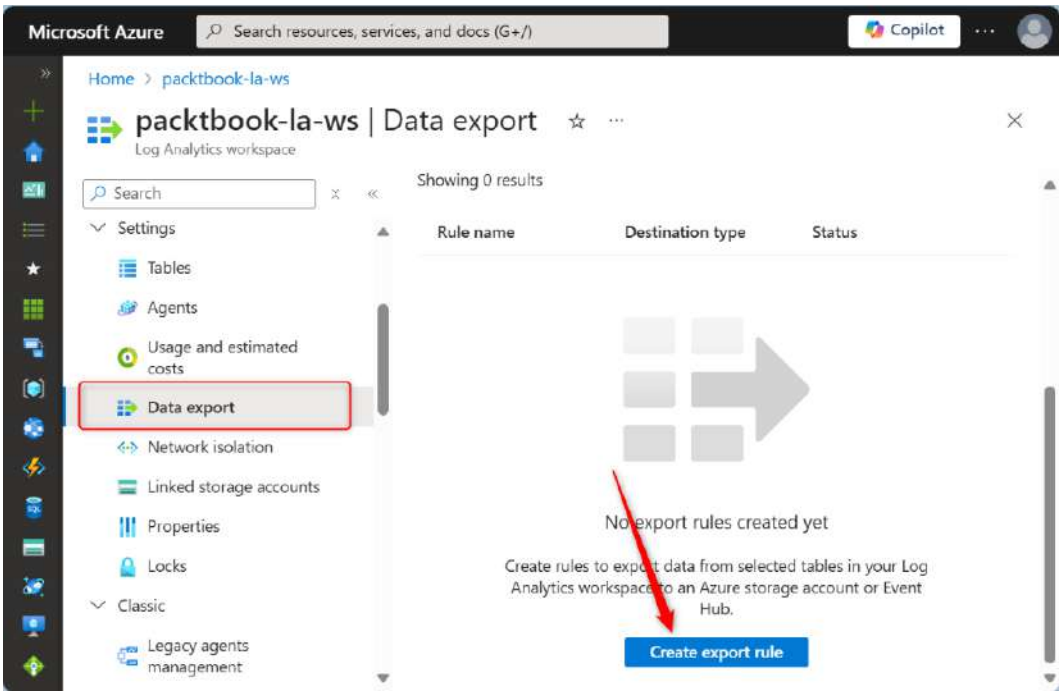

Figure 9.3 – Creating a new export rule for storage

A new configuration wizard will appear. You will need to provide a name for your new export rule and click **Next**. Then, a list of all the tables inside your Log Analytics Workspace will appear, allowing you to filter the ones to be exported, as shown in the following screenshot.

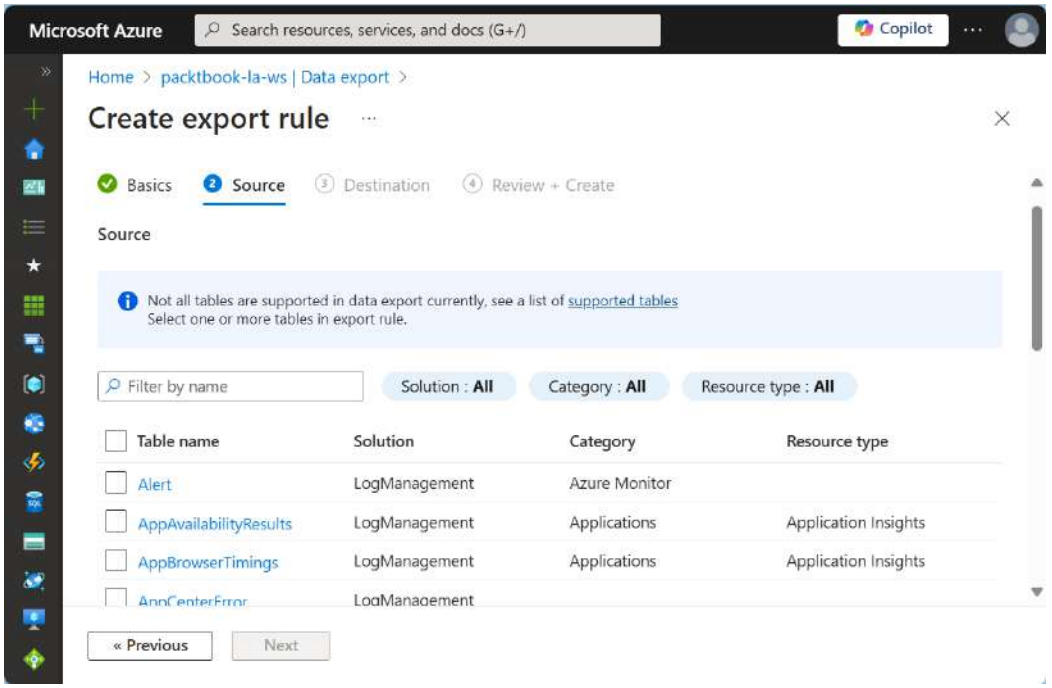

Figure 9.4 – The available tables as a source for data exporting

Select the ones relevant to you, and after clicking **Next**, the wizard will show you the destination options. Select the destination as **Storage account** or **Event Hub** and provide all the required details. If you choose Event Hub, ensure that it has the appropriate throughput units to handle the expected data volume.

> **Data export pricing model**
>
> Azure Log Analytics data export is not free. An extra export fee on top of the baseline price for Azure Storage or Event Hubs will be added based on the number of GB exported. Exported data is measured by the number of bytes in the exported JSON formatted data. As an equivalence, 1 GB equals 10^9 bytes.

As discussed in this section, Azure Monitor offers exporting capabilities that allow seamless integration with third-party tools, enhancing the monitoring, logging, and alerting functionalities beyond Azure's native capabilities. Additionally, many external solutions have developed native integrations in collaboration with Microsoft, providing even more streamlined and effective ways to unify monitoring data. In the next section, we will explore how these external solutions can be integrated with the monitoring data available inside Azure.

Leveraging external solutions for enhanced observability

As organizations strive to maintain robust and comprehensive monitoring solutions, leveraging Azure Native **ISV** (**Independent Software Vendor**) services becomes increasingly valuable. These services are specifically designed to integrate seamlessly with Azure, providing enhanced monitoring, analytics, and management capabilities that complement Azure's native tools. By incorporating ISV solutions, organizations can take advantage of specialized features, advanced analytics, and tailored monitoring capabilities that address unique business needs and operational requirements.

In this section, we will explore the Azure Native ISV services available for monitoring. We'll discuss the available service integration with Azure Monitor, their distinct advantages, and the added value they bring to your observability strategy. We will explore some of those services provided by Datadog, Elastic, Logz.io, Dynatrace, and New Relic. We'll discuss the options these services provide to integrate with the Azure platform, as well as the benefits they offer.

Azure Native Datadog

Azure Native Datadog is a powerful, cloud-native monitoring and security platform that integrates seamlessly with Azure. Designed to provide comprehensive visibility into the health and performance of your applications and infrastructure, Datadog offers robust features such as real-time metrics, advanced analytics, and customizable dashboards. With Azure Native Datadog, organizations can monitor Azure resources alongside other cloud and on-premises environments, enabling a unified approach to observability.

Datadog's integration with Azure enables the automatic discovery and monitoring of Azure resources, including virtual machines, databases, and services. It provides real-time monitoring through continuous collection and analysis of metrics, logs, and traces from your Azure environment. It supports both **IaaS** and **PaaS** environments, thanks to its extensive integration with more than 40 services.

Information collected can be used for advanced analytics and custom dashboards. You can utilize machine learning algorithms to detect anomalies and forecast trends, gain insights into application performance, and create detailed visualizations tailored to your specific needs, combining data from Azure and other sources.

Security is also relevant, thanks to its alerting and incident management capabilities. Set up proactive alerts and manage incidents efficiently to minimize downtime and impact. Improve your security inside Azure through its Cloud Security management features.

By leveraging Azure Native Datadog, organizations benefit from single-pane-of-glass visibility in hybrid and multi-cloud environments. Its costs are integrated into your Azure monthly bill directly, and access is transparent through the single sign-on integration.

Metrics and activity log ingestion are automatically configured, and installation of the custom Datadog agents can be automated for your virtual machines.

More information is available at `https://learn.microsoft.com/en-us/azure/partner-solutions/datadog/create`.

Azure Native Elastic Cloud

Azure Native Elastic is an integrated solution that combines the power of Elasticsearch, Kibana, and other Elastic Stack components with Azure's cloud capabilities. Elastic offers robust search, observability, and security solutions that help organizations gain deep insights into their Azure environments. By using Azure Native Elastic, you can seamlessly ingest, search, and visualize data from Azure resources, enabling advanced analytics and improved operational efficiency.

Elastic's integration with Azure provides a seamless experience for deploying and managing its Cloud-Native Observability Platform. It is provided as a **Software-as-a-Service** (**SaaS**) application through the Azure Marketplace, which centralizes log, metric, and trace analytics, simplifying the monitoring of Azure environments for Elastic clients.

Users can manage Elastic solutions directly through the Azure portal, implementing monitoring for cloud workloads via a streamlined workflow. Provisioning Elastic resources is facilitated by a custom resource provider, allowing the creation, provisioning, and management of Elastic resources within Azure, with Elastic managing the SaaS application and associated accounts.

It provides a similar experience to the previous solution through a single-pane-of-glass visibility platform, with a unified billing experience integrated into your Azure bill and transparent access to Elastic solutions through single sign-on integration. Metrics and activity log ingestion are automatically configured, and installation of the custom Elastic agents can be automated for your virtual machines.

More information is available at `https://learn.microsoft.com/en-us/azure/partner-solutions/elastic/create`.

Azure Native Logz.io

Azure Native Logz.io is a cloud-native observability platform that combines the best open-source tools – OpenSearch, OpenTelemetry, and Prometheus – in a unified solution. Logz.io provides advanced log management, metrics monitoring, and tracing capabilities, helping organizations achieve comprehensive observability across their Azure environments. With seamless integration and powerful analytics, Azure Native Logz.io enhances your ability to monitor and troubleshoot applications and infrastructure.

Logz.io's integration with Azure simplifies the deployment and management of observability tools. It is also provided as a SaaS application through the Azure Marketplace, which centralizes log, metric, and trace analytics. You can now provision the Logz.io resources through a custom resource provider that creates, provisions, and manages Logz.io resources through the Azure portal. Logz.io runs the SaaS, and Azure provides the interface to manage the resources.

Azure Native Logz.io empowers organizations to enhance their observability strategy, ensuring the reliability and performance of their applications and infrastructure through integrated log, metric, and trace management.

More information is available at `https://learn.microsoft.com/en-us/azure/partner-solutions/logzio/create`.

Azure Native Dynatrace

Azure Native Dynatrace is a comprehensive observability platform designed to provide deep insights into the performance and health of your Azure applications and infrastructure. Dynatrace leverages artificial intelligence and automation to deliver precise answers, helping organizations optimize their operations and improve user experiences. With seamless Azure integration, Dynatrace offers monitoring capabilities across cloud and hybrid environments.

Dynatrace's integration with Azure enables the automatic discovery and monitoring of Azure resources, offering a rich set of features such as AI-driven monitoring, using AI to automatically detect anomalies, identify root causes, and predict potential issues, or full stack observability that monitors the entire stack, from infrastructure to applications, in real time.

Azure Native Dynatrace provides the same key benefits discussed in the previous solutions related to integration, billing, and automation of agent deployment and information collection.

More information is available at `https://learn.microsoft.com/en-us/azure/partner-solutions/dynatrace/dynatrace-create`.

Azure Native New Relic

Azure Native New Relic is a powerful observability platform that offers comprehensive monitoring and analytics capabilities for your Azure applications and infrastructure. Designed to provide real-time visibility and actionable insights, New Relic integrates seamlessly with Azure, enabling organizations to monitor the performance and health of their environments with precision. By leveraging Azure Native New Relic, you can optimize application performance, enhance user experiences, and ensure operational excellence.

New Relic's integration with Azure allows effortless monitoring of Azure resources, featuring continuous monitoring of applications and infrastructure for real-time insights, powerful analytics to gain a deeper understanding of performance metrics and user behavior, custom dashboards to visualize key performance indicators and trends, and distributed tracing to track and analyze end-to-end transactions across distributed systems, helping you to identify performance bottlenecks.

Adopting Azure Native New Relic provides the same key benefits discussed in the previous solutions related to integration, billing, and automation of agent deployment and information collection.

You can learn more information at https://learn.microsoft.com/en-us/azure/partner-solutions/new-relic/new-relic-create.

Additional third-party services for integration

In addition to Azure Native ISV services, numerous third-party services also offer robust integration capabilities with Azure Monitor. These integrations extend the functionality of Azure Monitor, providing specialized features and advanced analytics that enhance your observability strategy. Leveraging these third-party services allows organizations to tailor their monitoring and security solutions to meet specific business needs, ensuring comprehensive visibility and control over their Azure environments.

Those third-party services are as follows:

- **IBM QRadar** is a leading **Security Information and Event Management** (**SIEM**) solution that helps organizations detect and respond to security threats. Integrating QRadar with Azure Monitor allows you to centralize security event data from your Azure environment and gain deeper insights into potential security incidents. You can read more about it at https://www.ibm.com/docs/en/qsip/7.5?topic=extensions-azure.
- **Splunk** is a powerful platform for searching, monitoring, and analyzing machine-generated data. Integrating Splunk with Azure Monitor enables you to collect, analyze, and visualize data from your Azure resources, enhancing your ability to monitor performance and detect issues. More information about this is available at https://splunk.github.io/splunk-add-on-for-microsoft-cloud-services/.
- **Sumo Logic** is a cloud-native, continuous intelligence platform for log management and analytics. Integrating Sumo Logic with Azure Monitor allows you to aggregate, monitor, and analyze log and metric data from your Azure resources, improving operational and security insights. More information is available at https://help.sumologic.com/docs/send-data/collect-from-other-data-sources/azure-monitoring/.
- **ArcSight** is a leading SIEM solution that provides advanced threat detection and response capabilities. Integrating ArcSight with Azure Monitor allows you to centralize security event data and gain actionable insights to protect your Azure environment. Read more about it at https://www.microfocus.com/documentation/arcsight/arcsight-smartconnectors/#gsc.tab=0.
- **Syslog** servers are a critical component of many IT infrastructures, providing centralized logging for network devices, servers, and applications. Integrating Syslog servers with Azure Monitor allows you to collect, store, and analyze syslog data from your Azure environment, improving visibility and operational efficiency. Further information is available at https://learn.microsoft.com/en-us/azure/azure-monitor/agents/data-collection-syslog.

Summary

In this chapter, we explored the features provided by Azure Monitor. Recognizing that built-in capabilities may not suffice for every scenario; we explored the various options available to integrate Azure Monitor with third-party systems. This chapter covered the methods and benefits of integrating Azure Monitor with third-party tools to extend observability beyond Azure's native capabilities.

We discussed how external solutions such as Datadog, Elastic, and Splunk can be seamlessly connected to Azure Monitor to enhance monitoring, analytics, and management capabilities. Additionally, we identified common challenges that might arise during the integration process and provided strategies to address them effectively. This chapter provided you with the knowledge and skills to enhance your monitoring strategy by leveraging the strengths of both Azure Monitor and third-party tools, ensuring a robust and comprehensive observability solution.

In the next chapter, we will move on from integrating third-party tools to developing a comprehensive monitoring strategy. Based on best practices from the Well-Architected Framework and the Cloud Adoption Framework from Azure, we will explore how to build a monitoring strategy that aligns with your organization's goals and requirements.

Further reading

Here, you can find the links to expand your knowledge about the specific concepts not covered in this book but referenced in this chapter:

- [1] Azure monitoring REST API walkthrough – Azure Monitor | Microsoft Learn: `https://learn.microsoft.com/en-us/azure/azure-monitor/essentials/rest-api-walkthrough?tabs=portal`.
- [2] `Get-AzLog` (`Az.Monitor`) | Microsoft Learn: `https://learn.microsoft.com/en-us/powershell/module/az.monitor/get-azlog?view=azps-0.10.0&viewFallbackFrom=azps-12.0.0`.
- [3] `Get-AzMetric` (`Az.Monitor`) | Microsoft Learn: `https://learn.microsoft.com/en-us/powershell/module/az.monitor/get-azmetric?view=azps-0.10.0`.

10
Building Your Monitoring Strategy

Throughout this book, we have seen that the complexity and dynamism of today's cloud environments not only require advanced monitoring tools and services to maintain the reliability, performance, and security of the system but also that it is necessary to create a robust observability strategy. In this chapter, we delve into the key elements needed to develop a comprehensive and effective observability approach with Azure Monitor for your organization's needs.

You will begin learning the fundamental principles that ensure your monitoring solutions deliver deep and actionable insights. These principles will guide you in creating systems that are not only monitored but also observable and adaptable to evolving technological landscapes. We will also cover strategies you can put in place to maintain visibility and control in expanding and rapidly changing environments due to the complexity of organizations' cloud infrastructures.

We will introduce a shared responsibility model that is the foundation for ensuring that all aspects of your organization's infrastructure are properly monitored and managed. With this model, you will understand how monitoring responsibilities are divided between cloud providers and customers.

The chapter will introduce the Azure Well-Architected Framework and Azure landing zones that will provide you with the best practices and guidelines for designing and implementing robust, scalable, and secure monitoring solutions within Azure.

By the end of this chapter, you will have a comprehensive understanding of the key components and strategies needed to build an effective monitoring strategy, ensuring that your systems align with monitoring industry standards and best practices.

In this chapter, we'll cover the following topics:

- Design principles for enhanced observability
- Strategies for scaling monitoring in large and dynamic cloud environments
- The shared responsibility model and your monitoring strategy
- The role of the Azure Well-Architected Framework and Azure landing zones

Technical requirements

To follow up on all the examples available in this chapter, you need a basic knowledge of Azure Policy and an Azure account with an active subscription, for deploying **Azure Monitor Baseline Alerts** from the **Azure landing zone accelerator** for new Azure landing zone deployments.

Design principles for enhanced observability

In *Chapter 1*, we explained the concept of observability without going into detail about the principles for building and maintaining a robust observability platform. In this section, we will address the core principles that will make your observability strategy not only solid but also adaptable to constantly evolving technological changes:

- **Comprehensive monitoring**: Start by prioritizing monitoring all layers of your system (infrastructure, applications, and network) to achieve a complete view of the health of your system. In conjunction with system layers, focus on ensuring complete visibility from frontend to backend systems, including third-party services used. In *Chapters 2* and *3*, we exposed how Azure Monitor collects and consolidates data from every layer and component in your system and this data is stored on a common data platform for consumption by a set of tools capable of correlating, analyzing, visualizing, and responding to data. You can use the techniques seen in that chapter to cover the principle of Comprehensive Monitoring.
- **Data collection, management, and maintenance**: For a long time, the collection, purification, organization, and maintenance of datasets have not been considered a principle of observability; however, over time it has become a fundamental aspect to allow organizations' IT teams to analyze and use data quickly to draw meaningful conclusions. The collection, purification, organization, and maintenance of datasets constitute what is called **data curation**. To achieve proper data curation, your cloud operations team should establish a clear data collection and storage strategy to ensure they manage data effectively. This strategy should include as its main pillars the identification of the type of data, the definition of data structure and format, and the methods and locations to store data and how it will be accessed.

Establishing effective data management processes that guarantee the quality of the data should not be forgotten. Data management processes should include at least automated data validation, implementation of automated controls to validate the accuracy and relevance of data, implementation of procedures to remove irrelevant or erroneous data from datasets, and finally, continuous maintenance and updates of data. You can use the techniques seen for Azure Monitor in *Chapter 3* to cover the principles of data collection, management, and maintenance.

- **Near real-time data processing and analysis**: Perform an analysis of the need for near real-time data processing to enable rapid responses to incidents and performance issues. Combine the preceding analysis with the creation of real-time dashboards and reporting for continuous insights to aid decision-making processes. You can use the techniques seen for Azure Monitor in *Chapters 4* and *5* to cover the principle of near real-time data processing and analysis.
- **Full context and granularity**: The observability platform should offer two fundamental capabilities: a high-level overview of application performance and the ability to drill down into granular details with the overall context of the application. The transition between the two views should be seamless and intuitive, making it easy for operations teams to go from a more global view of the application problem to a detailed view that drills down to see the specific exceptions or pieces of code that are causing the problem.
- **Intuitive design and accessibility**: A platform without intuitive user interfaces is not capable of providing users with friendly and fluid navigation and interaction with the observability platform; so, an intuitive design is a very important point to guarantee the best user experience with the platform. Additionally, the ability to customize dashboards and reports is crucial to offering all operations and monitoring teams within the organization the option to customize the platform to their specific needs, ensuring they can access the most relevant information efficiently and effectively when they need it.
- **Ready-to-use experience and the ability to stay up to date**: The rapid and efficient response by the monitoring teams is not only based on the ability of the observability platform to provide an intuitive and ready-to-use experience for the different teams that are resolving the incident or error, but also how quickly the observability platform is able to update itself periodically to ensure that new elements, such as agents, logs, metrics, and traces, are available and traceable out of the box. In the case of Azure Monitor, it is updated periodically to ensure that the platform components are always ready with the latest improvements for end users.
- **Scalable and adaptable architecture**: When it comes to scaling an observability platform, the main approaches are the same as for almost any platform, and they are horizontal and vertical scalability. Azure Monitor offers by design an appropriate balance between the two will guarantee that the observability platform will be distributed and provided with high availability and good performance. Equally important is the need for adaptive architectures in observability systems that ensure that the observability framework can scale efficiently and continue to provide accurate, real-time information regardless of changes in the monitored environment.

- **Actionable insights and automated response**: A fundamental design principle of the observability platform is that it is able to transform raw data into actionable information. This transformation process involves collecting and analyzing data from various sources to provide clear, insightful metrics and visualizations that help understand system performance and issues. Additionally, the platform must incorporate advanced analytics and machine learning that offer proactive suggestions and highlight potential problems before they become serious and, together with automated recommendations, improve the decision-making process. You can use the techniques seen for Azure Monitor in *Chapters 4*, *5*, and *6* to cover the principle of actionable insights and automated response.
- **Platform security**: An important design principle concerns platform access control and granularity in access to the control plane and data plane of the platform. Azure Monitor integrates with **Azure role-based access control** (**Azure RBAC**) by providing built-in roles for monitoring that you can assign to users, groups, service principals, and managed identities. In the case of permission to access data in a Log Analytics workspace, this is defined by configuring the access control mode in each workspace [1].
- **Data security and privacy**: Data security and privacy are critical in any observability strategy, so the platform must make use of strong encryption protocols for data in transit and at rest to protect sensitive information from unauthorized access and breaches. Likewise, compliance with regulatory standards such as the **General Data Protection Regulation** (**GDPR**) and the **Health Insurance Portability and Accountability Act** (**HIPAA**) is equally essential, ensuring that data handling practices comply with legal requirements and safeguard user privacy. Azure Monitor covers a wide list of certifications and attestations [2] and offers several capabilities for managing personal data, such as purging personal data and data retention [3].

To ensure the security of data in transit to Azure Monitor, it is highly recommended to configure AMA to use at least **Transport Layer Security** (**TLS**) **1.3**. Once ingested by Azure Monitor, data remains logically segregated across all components, with each piece of data tagged per workspace. Managed by Microsoft personnel, Azure Monitor maintains comprehensive logging of all activities for audit purposes and adheres to a strict incident management process. Regarding data security, data within Azure Monitor is encrypted using Microsoft-managed keys, with the option for customers to use their own encryption keys for enhanced control over access to logs and saved queries. Additionally, Azure Monitor Logs utilizes Azure Storage in specific scenarios, where customer-managed storage accounts with personalized encryption can be employed. Finally, regarding network security, Azure Private Link networking enables secure connections between Azure Monitor and virtual networks via private endpoints, enhancing the overall security of the service [2].

- **Seamless integration with hybrid environments and third-party systems**: An observability platform with an API-based design has the advantage of guaranteeing rapid and scalable integration with a variety of systems and technologies today. This approach allows for easy communication and data sharing, facilitating a smooth integration process. It is also important that the integration of observability platforms with more legacy systems or other clouds takes into account well-defined and careful strategies to avoid interrupting the existing workflows

of these systems. These integrations should prioritize compatibility and provide tools or agents that ensure strong integration, allowing organizations to enhance their hybrid or multi-cloud observability capabilities without compromising the stability of their current infrastructure. In *Chapter 3*, we explained that Azure Monitor is an API-based design and how you can use it for rapid and scalable integration with a variety of systems. In *Chapter 8*, we explained how you can use Azure Monitor together with Azure Arc to integrate with hybrid environments. Finally, in *Chapter 9*, we showed the possibilities of integrating Azure Monitor with third-party tools (e.g., Datadog, Elastic, Grafana, Splunk, Prometheus).

- **Continuous platform improvement**: Users provide suggestions and fundamental information that can improve the performance and use of the observability platform, which is why a good design for incorporating these feedback mechanisms helps identify areas for improvement and ensures that the platform evolves to meet the needs of operations teams. Another very important aspect is that the platform adapts in an agile and robust manner to technological changes to maintain effectiveness, and for this, it is recommended that organizations implement strategies to keep the platform updated with the latest technologies and best practices, and this will ensure that the observability tools remain cutting-edge and capable of addressing emerging challenges.

We recommend applying these principles to your organization's observability systems so that they are effective, efficient, and prepared for the changing technological future. As we saw in *Chapter 1*, monitoring and observability complement each other, and one of the aspects that constitutes an important challenge in organizations is understanding how to scale monitoring systems in large and dynamic cloud environments.

In the next section, we will address the main practices that organizations can implement to address these challenges.

Strategies for scaling monitoring in large and dynamic cloud environments

Expanding monitoring in large cloud environments involves more than just increasing capacity, especially as those environments become more complex and dynamic, so effective monitoring strategies must evolve to ensure observability, performance, and reliability. Here, we will expose several strategies for expanding monitoring in such environments:

- **Automated monitoring and alerting**: In large and dynamic cloud environments, an approach based on manual monitoring is not practical due to the large volume of data and rapid pace of change. The recommendation is the use of automated monitoring tools that can collect, analyze, and respond in real time. These tools must be configurable to automatically detect anomalies, generate alerts based on dynamic thresholds, prioritize critical alerts over false positives, and trigger automated remediation processes. In the case of Azure Monitor, it has the capability to collect, analyze, and respond in real time using AMA, KQL, and action groups. You can use the techniques seen for Azure Monitor in *Chapters 4*, *5*, and *6* to cover that.

- **Distributed and scalable data collection**: To effectively manage the large amount of data produced by large cloud environments, monitoring systems must be distributed and scalable. This is exemplified by Azure Monitor with its deployment of numerous data collection agents across multiple regions and services to ensure complete coverage. Some key aspects include placing monitoring agents close to data sources to minimize latency and improve data accuracy, employing horizontal scaling by adding more agents or nodes as the environment expands to handle higher loads and aggregating data from multiple sources into a centralized platform for comprehensive analysis and reporting.
- **Centralized logging**: In these large cloud environments, logs are produced by a multitude of services and components, requiring a centralized logging solution for efficient log aggregation, storage, and analysis. In the case of Azure Monitor, centralized logging ensures streamlined management of log data across the cloud environment, using robust log analysis tools to find, analyze, and visualize log data for pattern identification and problem resolution, and establishing retention policies to manage log data volume and retention of critical logs for analysis.
- **Advanced analytics and machine learning**: Leveraging advanced analytics and machine learning can greatly improve monitoring capabilities in large cloud environments by providing deeper insights and predictive capabilities beyond traditional monitoring systems. This is the case, for example, of Azure Monitor, which includes the use of machine learning models for predictive analysis to predict potential problems and allow proactive measures, the use of advanced analytics for rapid root cause analysis, and the use of historical data along with machine learning to forecast resource utilization trends and automate it through auto-scaling services.
- **Unified monitoring dashboard**: In large cloud environments, it is especially important to have a unified monitoring dashboard that consolidates data from various sources into a single panel, offering a comprehensive view of the entire environment. This approach simplifies tracking and makes it easier to identify and resolve issues.

  Additionally, as we have seen on visualization and dashboards in *Chapter 3*, it is essential that they be customizable dashboards that show key metrics and alerts adapted to different roles within the organization, with the implementation of real-time data visualization to provide up-to-the-minute information on system performance and health.

  Finally, it is also important that monitoring dashboards integrate with other tools and systems used within the organization so that a coherent workflow is created.

- **Continuous improvement approach**: A dynamic environment such as the cloud requires a continuous process of improvement and adaptation. This is why it is necessary to establish a feedback loop that allows organizations to refine their monitoring strategies based on lessons learned and evolving requirements. Some key points that should be part of this process of continuous improvement and adaptation are conducting periodic audits of the monitoring system to identify gaps and areas for improvement, collecting feedback from users and stakeholders of cloud teams to understand their needs, and challenges, and implementing iterative improvements to ensure that the monitoring system remains aligned with the organization's objectives and the dynamic nature of the cloud environment. It is also important to track **key performance indicators** (**KPIs**) to measure the effectiveness of tracking practices and identify opportunities for improvement.

  Finally, it is recommended to continually evolve, refine, and optimize monitoring processes, tools, and methodologies.

By adopting the preceding strategies, your organization can effectively scale its monitoring practices to meet the demands of large and dynamic cloud environments. This not only enhances monitoring and performance but also ensures that potential issues are identified and addressed quickly, maintaining the overall health and reliability of the cloud infrastructure.

Thus far, we have detailed and exposed strategies in cloud environments; in the next section, we will explain how the implementation of all these solutions, strategies, and good practices must be well aligned with the **shared responsibility model** in the cloud.

The shared responsibility model and your monitoring strategy

The **shared responsibility model** is a critical concept in cloud computing, delineating the division of security and operational responsibilities between the cloud provider and the customer. Understanding this model is crucial for developing a comprehensive monitoring strategy, as it clarifies which aspects of the cloud infrastructure require your attention and which are managed by the cloud provider.

Understanding the shared responsibility model

In the shared responsibility model, a cloud provider, such as Microsoft Azure, is responsible for the security and maintenance of the cloud infrastructure itself. This includes the physical security of data centers, hardware management, network infrastructure, and foundational services. On the other hand, customers are responsible for securing and managing their data, applications, and operating systems that run on the cloud infrastructure. This split responsibility ensures that both parties focus on their respective areas of expertise, leading to a more secure and well-maintained environment.

For example, in **infrastructure as a service** (**IaaS**), Azure manages the physical infrastructure, networking, and hypervisor, while the customer is responsible for the operating systems, applications, and data. In **platform as a service** (**PaaS**), Azure also manages the operating system and middleware, leaving the customer to focus primarily on their applications and data. In **software as a service** (**SaaS**), Azure takes on even more responsibility, managing the entire stack except for the data and user access.

Implications for your monitoring strategy

Developing a robust monitoring strategy within the shared responsibility model involves focusing on areas under your control while leveraging the monitoring tools and services provided by Azure. This dual approach ensures comprehensive coverage of your entire cloud environment:

- **Monitoring Azure's responsibilities**: Azure provides a range of built-in monitoring tools to help organizations keep track of the infrastructure and services they manage. Some services, such as Azure Monitor, **Azure Resource Health**, and **Azure Service Health**, provide insights into the status and performance of the underlying cloud infrastructure. These services alert you to any issues within Azure's domain, ensuring you are aware of problems that might impact your resources.
- **Monitoring customer responsibilities**: On the customer side, it is essential to implement monitoring solutions that focus on your applications, data, and any customer-managed aspects of the cloud environment. This includes application performance monitoring, alerting, notifications, and remediation. There are services such as Azure Monitor, **Application Insights**, and **Log Analytics** that can be configured to provide detailed insights into your custom responsibilities.
- **Integration and automation**: Effective monitoring requires integrating Azure's monitoring tools with your organization's own solutions. Setting up automated alerts and responses helps in proactive management. For example, integrating Azure Monitor alerts with an alert and ticketing management system can automate incident response workflows, reducing reaction times and improving operational scores. For example, Azure Monitor can be integrated with ServiceNow, which is a well-known alert and ticketing management system in the industry. It can be also integrated using action groups with any other ticketing system that supports webhooks or REST API calls.
- **Compliance and auditing**: Regulatory compliance is a shared responsibility. While Azure provides compliance certifications and tools to help meet various regulatory requirements, customers must ensure their configurations, data management practices, and application deployments comply with relevant regulations. The use of services such as **Azure Policy** and **Azure Environment Deployments** can help enforce compliance across your resources.
- **Operational insights and optimization**: Continuous monitoring provides operational insights that can drive optimization. By analyzing data from both Azure's managed components and customer-managed components, you can identify inefficiencies, forecast capacity needs, and optimize performance and cost.

- **Security and control access**: Access control and security is a shared responsibility. While Azure is responsible for Azure Monitor platform security, customers must ensure that their control plane and data plane access control configurations meet with relevant regulations and the data security requirements of their organizations.

Once we have seen how to establish a solid shared responsible monitoring strategy, in the next section, we address what the best practices are and the appropriate steps to follow to implement them.

Recommended approach for implementing a shared responsible monitoring strategy

The implementation of a shared responsible monitoring strategy has become essential but at first, it can be slow due to doubts about where to start. Here, we explore the recommended approach to implementing such a strategy, highlighting key steps and best practices:

- **Leverage Azure's native services**: Using monitoring services and other native Azure tools to monitor deployed workloads can make it faster and easier to have a comprehensive view of your infrastructure thanks to the native integration of these services. Additionally, the issues and challenges caused by integration between non-native components are greatly reduced. If you require the integration of third-party or customized solutions, make sure they follow observability design principles to ensure that the observability of your systems is comprehensive and effective.
- **Customize your monitoring solutions**: Deploy custom monitoring solutions for your applications and data whenever you can. To do this, use Application Insights for application performance management and Log Analytics for detailed log query and analysis capabilities. Support the customization of alerts and automated responses to incidents within your infrastructure. Use machine learning models to predict the behavior of your infrastructure and anticipate problems that may arise. Finally, use the visualization best practices we presented to get a comprehensive view of your infrastructure.
- **Automated responses**: Reduce risk by using an approach based on automated responses to common problems to minimize downtime and improve resilience. Use **Azure Automation** and **Azure Logic Apps** to create workflows that respond to alerts. Constantly improve incident remediation orchestration flows and introduce machine learning model outputs to improve the impact of these responses.
- **Regular reviews and updates**: An important part of the accountability model is that you periodically review and update your monitoring configurations to adapt to changing requirements and evolving threats. Make sure your monitoring strategy evolves with your cloud environment. Solidly define architectural improvements in the observability platform and quickly and safely introduce new functionalities that make the platform more robust and functional for your cloud operations teams.

- **Training and awareness in the organization**: Make sure your team is well-versed in both Azure monitoring tools and your custom solutions. In this way, it will reduce the learning curve and errors in the use of monitoring systems, greatly reducing adaptation time and problems in the systems. Adopt measures that focus on continuous training with regular training and awareness programs that help maintain a high level of operational excellence.

With this section, your organization's cloud teams will be able to understand the shared responsibility model and effectively implement a comprehensive monitoring strategy, ensuring that your cloud environment is secure, efficient, and resilient. Balancing monitoring efforts between Azure's built-in capabilities and your custom solutions will provide a comprehensive view of your infrastructure, enabling proactive management and continuous improvement.

Moving away from the shared responsibility model and your monitoring strategy, let's now consider the role of the Azure Well-Architected Framework and Azure landing zones in the observability ecosystem.

The role of the Azure Well-Architected Framework and Azure landing zones

Transitioning to the cloud without principles, good practices, or a foundation to build on is complicated and can pose many problems in an organization's digital transformation process. That's why Azure offers two fundamental tools that together facilitate a structured and efficient approach to cloud adoption and management:

- The **Azure Well-Architected Framework** is a set of guiding principles and best practices designed to help organizations build and maintain optimal cloud architectures. It covers five key pillars: *cost management*, *operational excellence*, *performance efficiency*, *reliability*, and *security*. By following these principles, organizations can ensure that their cloud solutions are well designed, resilient, efficient, and secure. For more information about the Azure Well-Architected Framework, refer to [4] in the *Further reading* section.
- The **Azure landing zones** are predefined, scalable, and secure environments within Azure that provide a foundation for building and deploying applications. These landing zones integrate various Azure services and components, aligning with the Well-Architected Framework to support best practices and ensure consistency across deployments.

In this section, we will see how observability is required within the design principles for building and maintaining cloud architectures recommended by the Well-Architected Framework, and how Azure landing zones include a monitoring baseline for helping organizations in their observability journey and maturity process.

Azure Well-Architected Framework

Observability is mentioned in several pillars of the Azure Well-Architected Framework, but in particular, we are going to go deep into two design principles from two different pillars.

The first of them is the **design for operations** principle, which is found within the *reliability* pillar. In *Further reading*, you can see the details and guidelines of this design principle [5]. However, we will focus on the *Shift left in operations to anticipate failure conditions* guide and how this relates to the observability aspects we have highlighted throughout the book.

This design principle emphasizes the importance of early and frequent testing for performance impact on reliability and highlights the need for shared visibility and insights from observable systems.

This design principle aligns with what has been insisted upon as fundamental throughout the book, which is to accurately identify the causes of why a cloud application or workload has failed. In this sense, the observability of each of the components can help in the granular identification of the underlying problems, but it is not enough to have a comprehensive view of the general state of the application. For that reason, we need aggregate observability, which combines data from multiple components and correlates them with user flows.

This holistic approach is fundamental because it allows you to understand how different application components interact and impact each other, leading to more agile and effective problem resolution. For example, an issue that appears to be isolated within a single component could actually be a symptom of a larger issue that affects multiple parts of the application. By taking an aggregate observability approach to data, you can identify patterns and correlations that would be missed by looking at components in isolation.

The results derived from aggregate observability provide engineers in organizations' cloud operations teams with essential data as they prioritize their error remediation efforts in cloud applications. When analyzing multiple problems, these teams must be able to quickly and accurately analyze and evaluate which are the most critical problems and prioritize them. That is why, in this process, an aggregate observability approach helps them make these decisions by providing them with a clear view of how different problems affect the overall performance of the application and the user experience. Furthermore, this approach allows them to go from a global view to a more detailed one, down to the various underlying problems instead of an approach focused on the symptoms of malfunction of some individual components.

Furthermore, adopting an aggregate observability approach aligns with the continuous improvement principle we looked at in the previous section, as it allows organizations to identify recurring problems and implement long-term solutions to prevent them from reoccurring. Likewise, this approach helps in the proactive maintenance of applications since the aggregation of the dataset and its analysis can identify trends and patterns that can highlight potential problems before they become major interruptions in the future.

The second principle is the **evolve operations with observability** principle, which is found within the *operational excellence* pillar. In *Further reading*, you can see the details and guidelines of this design principle [6]. In the same way as with the previous principle, we will focus on the *Gain visibility into the system, derive insight, and make data-driven decisions* guide and how this relates to the observability aspects we have highlighted throughout the book.

This design principle indicates that operations must be designed to take into account observability due to the importance they have for various aspects of application maintenance and optimization. In particular, it insists on the importance that operations have in five aspects:

- **Proactive maintenance**: Observability enables continuous evaluation of system health, which is essential for proactive maintenance. By constantly monitoring application performance and resource utilization, potential problems can be detected and addressed before they become major problems. This proactive approach helps maintain system stability and minimize downtime, resulting in a more reliable user experience.
- **Guarantee quality and safety**: Continuous observability enables the detection of performance bottlenecks, errors, and other issues that could degrade application quality. Similarly, observability can identify unusual activities or patterns that may indicate security vulnerabilities or breaches. By ensuring that these issues are addressed promptly, observability helps maintain high standards of application quality and security.
- **Capacity planning**: Observability in operations provides historical analytics and real-time data on system usage and performance, allowing organizations to predict future resource needs and scale their infrastructure accordingly. This provision prevents performance degradation due to resource shortages and ensures that the application can handle increasing loads as user demand grows.
- **Product management**: One of the aspects that product or application managers value most is knowing which functions are most used by users and the user experience problems to make decisions that prioritize development efforts based on the current usage patterns and user feedback. Observability provides information about how users interact with the application and which features are used most; so, including observability in application operations can contribute to improving the organization's product and application management.

Having gained a comprehensive understanding of how the Azure Well-Architected Framework relates to observability, we are left to see in the next section how Azure landing zones, the other fundamental tool, include a monitoring baseline to help organizations on their observability journey.

Azure landing zones

Azure landing zones include several design areas, among which the management area stands out, where, among other topics, mention is made of the importance of monitoring the landing zone platform. Many of the necessary recommendations and good practices have already been outlined in previous chapters, such as the best Log Analytics workspace design practices, diagnostic settings configuration, configuration alerts, and so on. However, as we have seen, monitoring and observability are complementary to each other; so, ensuring total consistency in the alerting and monitoring of a landing zone is key to having good observability. This is why Azure is incorporating improvements in the Azure landing zone monitoring baseline, particularly in metric, activity log, and log query alerts following the best practices recommended by Azure in proactive alerting.

The solution provides a baseline of Azure Monitor metric and activity log alerts for a subset of resources, including **Azure ExpressRoute**, **Azure Firewall**, **Azure Virtual WAN**, **Azure Key Vault**, **Azure virtual machines**, **Azure Storage** account, and so on [7].

The implementation of the alerts is through Azure Policy. Each alert is compiled in an Azure Policy definition and grouped in the following Azure Policy initiatives:

- Alerting connectivity initiative
- Alerting management initiative
- Alerting identity initiative
- Alerting landing zone initiative
- Notification assets initiative
- Alerting Service Health initiative

Each of the preceding Azure Policy initiatives is respectively assigned to the reference management groups of the Azure landing zone architecture reference model [8]:

- Identity
- Management
- Connectivity
- Landing zones
- Sandbox

You can deploy Azure Monitor Baseline Alerts from the **Azure landing zone accelerator** for new Azure landing zone deployments (`https://aka.ms/alz/portal`). After clicking on the previous link, the Azure landing zone wizard will appear, as shown in *Figure 10.1*.

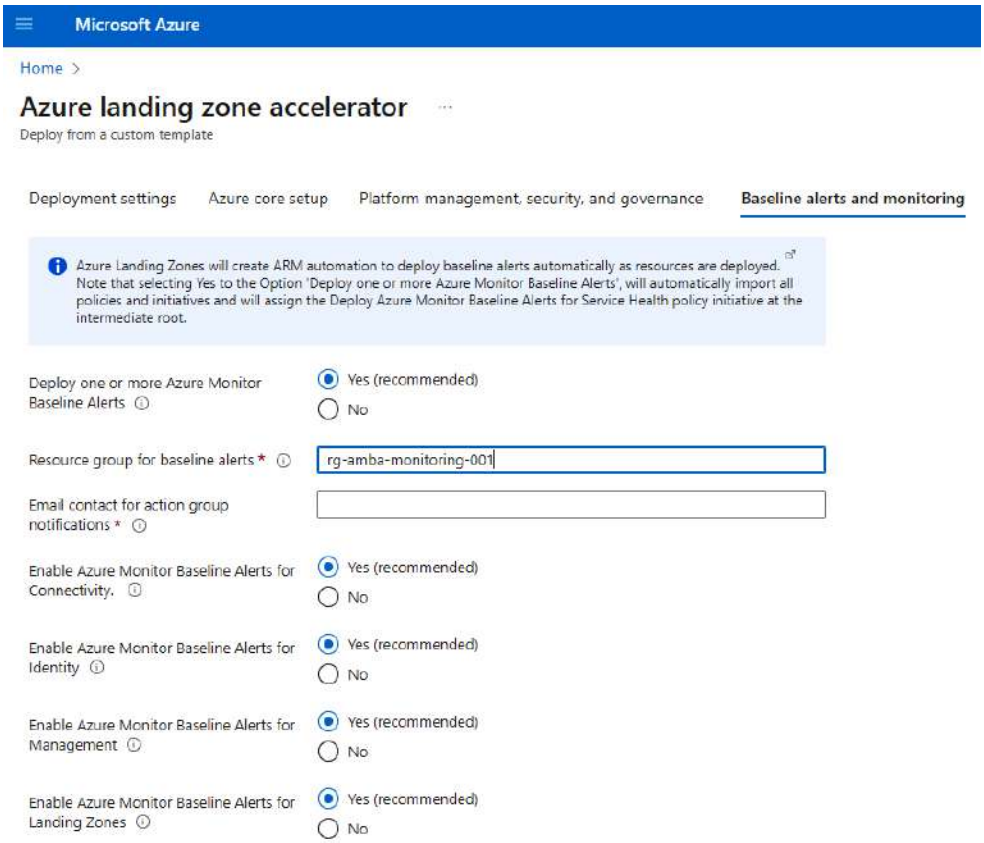

Figure 10.1 – Azure landing zone portal page for Azure Monitor Baseline Alerts

However, if you have a custom Azure landing zone model that is not aligned with the previous management group structure, you can import the Azure Policy initiatives and perform a custom implementation to your model using the templates from `https://azure.github.io/azure-monitor-baseline-alerts/welcome/`.

The solution offers flexibility in what alert rules can be implemented, the ability to change thresholds to suit, and it is also possible to disable alerts once they have been implemented. Surely, your organization's IT team wants to create additional alert rule policies and add them to the initiatives, which is also possible.

Deploying Azure Monitor Baseline Alerts templates is done using the `DeployIfNotExists` policy definition effect that runs after a configurable delay when a resource provider processes a create or update request for a subscription or resource and returns a success status code. If the related resources are missing or if `existenceCondition` doesn't evaluate to true, a template deployment is triggered. If you are not familiar with Azure policy definitions and effects, we recommend that you consult the official Azure Policy documentation to better understand how policy definitions and effects work for this scenario [9].

To exemplify this behavior, we will look at the policy definition called `Enforce VM CPU Alert` found within the landing zone initiative. You can download the policy from `https://azure.github.io/azure-monitor-baseline-alerts/services/Compute/virtualMachines/Deploy-VM-PercentCPU-Alert.json`.

```
},
"then": {
  "effect": "[[parameters('effect')]",
  "details": {
    "roleDefinitionIds": [
      "/providers/Microsoft.Authorization/roleDefinitions/b24988ac-6180-42a0-ab88-20f7382dd24c"
    ],
    "type": "Microsoft.Insights/scheduledQueryRules",
    "existenceScope": "resourceGroup",
    "resourceGroupName": "[[parameters('al                ourceGroupName')]",
    "deploymentScope": "subscription",
    "existenceCondition": {
      "allOf": [
        {
          "field": "Microsoft.Insights/scheduledQueryRules/displayName",
          "equals": "[[concat(subscription().displayName, '-VMHighCPUAlert')]"
        },
        {
          "field": "Microsoft.Insights/scheduledQueryRules/scopes[*]",
          "equals": "[[subscription().id]"
        },
        {
          "field": "Microsoft.Insights/scheduledQueryRules/enabled",
          "equals": "[[parameters('enabled')]"
        },
        {
          "field": "Microsoft.Insights/scheduledQueryRules/evaluationFrequency",
          "equals": "[[parameters('evaluationFrequency')]"
        },
        {
          "field": "Microsoft.Insights/scheduledQueryRules/windowSize",
          "equals": "[[parameters('windowSize')]"
        },
        {
          "field": "Microsoft.Insights/scheduledQueryRules/severity",
```

Figure 10.2 – The existenceCondition block of the Deploy-VM-PercentCPU-Alert.json alert

In *Figure 10.2*, you can see the policy definition content and that the `existenceCondition` block uses a single `allOf` to ensure that all conditions are true. Based on our previous comment, the template specified inside the `deployment` block, as shown in *Figure 10.3*, will be deployed only when `existenceCondition` doesn't evaluate to `true`. That means that only when at least one of the conditions inside the `allOf` operator evaluates to false does the deploy effect trigger.

```
    },
    "evaluationPeriods": {
      "type": "String"
    },
    "MonitorDisableTagName": {
      "type": "String"
    },
    "MonitorDisableTagValues": {
      "type": "Array"
    }
  },
  "variables": {},
  "resources": [
    {
      "type": "Microsoft.Resources/resourceGroups",
      "apiVersion": "2021-04-01",
      "name": "[[parameters('alertResourceGroupName')]",
      "location": "[[parameters('alertResourceGroupLocation')]",
      "tags": "[[parameters('alertResourceGroupTags')]"
    },
    {
      "type": "Microsoft.Resources/deployments",
      "apiVersion": "2019-10-01",
      "name": "VMCPUAlert",
      "resourceGroup": "[[parameters('alertResourceGroupName')]",
      "dependsOn": [
        "[[concat('Microsoft.Resources/resourceGroups/', parameters('alertResourceGroupName'))]"
      ],
      "properties": {
        "mode": "Incremental",
        "template": {
          "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
          "contentVersion": "1.0.0.0",
          "parameters": {
            "enabled": {
              "type": "string"
            },
            "alertResourceGroupName": {
              "type": "string"
            },
            "alertResourceGroupLocation": {
              "type": "string"
            },
            "UAMIResourceId": {
              "type": "string"
            }
          },
          "variables": {},
          "resources": [
            {
              "type": "Microsoft.Insights/scheduledQueryRules",
              "apiVersion": "2022-08-01-preview",
              "name": "[[concat(subscription().displayName, '-VMHighCPUAlert')]",
              "location": "[[parameters('alertResourceGroupLocation')]",
              "identity": {
                "type": "UserAssigned",
                "userAssignedIdentities": {
                  "[[parameters('UAMIResourceId')]": {}
                }
              },
```

Figure 10.3 – The deployment block of the Deploy-VM-PercentCPU-Alert.json alert

During each evaluation cycle, resources that match policy definitions with the `deployIfNotExists` effect are flagged as non-compliant, but no automatic remediation occurs. To address these non-compliant resources, a remediation task must be executed. This task will ensure that the necessary resources are deployed and compliance is achieved.

Regarding the methods of implementing the solution, we have already mentioned that it is possible through the Azure landing zone accelerator, as shown in *Figure 10.1*, but there are other options, such as automating a GitHub action, through CLI, PowerShell, or Bicep, or directly implementing ARM templates.

This section explained the importance of observability principles in designing new cloud architectures according to the Well-Architected Framework guidelines to ensure availability, reliability, and scalability, and the importance of monitoring landing zone components of your Azure platform using Azure Monitor Baseline Alerts components to maintain a stable environment, maximize performance, and meet changing organizational and business needs.

Summary

In this chapter, we covered the aspects to take into account to build a solid and effective monitoring strategy. We began by learning about design principles to improve observability, among others: comprehensive monitoring, near real-time data processing and analysis, full context and granularity, scalable and adaptive architecture, and so on. Design principles ensure that monitoring solutions provide deep, actionable information, making systems observable and adaptable to technological changes.

We then moved on to strategies for scaling monitoring. Monitoring and observability are complementary to each other, so strategies for scaling monitoring are necessary to maintain visibility and control in expansive and rapidly changing cloud environments, including automating distributed, scalable data collection and monitoring.

We also highlighted the importance of the shared responsibility model and how this model divides monitoring responsibilities between cloud providers and customers, ensuring that all aspects of an organization's infrastructure are properly monitored and managed.

Finally, we provided an analysis of how the Azure Well-Architected Framework and Azure landing zones provide best practices and guidelines for designing and implementing robust, scalable, and secure monitoring solutions within Azure.

In the next chapter, we will explore the critical aspects of cost management and optimization within Azure Monitor. We will discuss efficient resource utilization strategies, understanding Azure Monitor's pricing model, cost control measures in Azure Monitor, and a hands-on example of how to estimate your Azure Monitor Logs costs.

Further reading

Here, you can find the links to expand your knowledge about the specific concepts not covered in this book but referenced in this chapter:

- [1] *Manage access to Log Analytics workspaces*: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/manage-access?tabs=portal`
- [2] *Azure Monitor Logs data security*: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/data-security`
- [3] *Managing personal data in Azure Monitor Logs and Application Insights*: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/personal-data-mgmt`
- [4] *What is the Azure Well-Architected Framework?*: `https://learn.microsoft.com/en-us/azure/well-architected/what-is-well-architected-framework`
- [5] *Reliability design principle*: `https://learn.microsoft.com/en-us/azure/well-architected/reliability/principles#design-for-operations`
- [6] *Operational Excellence design principle*: `https://learn.microsoft.com/en-us/azure/well-architected/operational-excellence/principles#evolve-operations-with-observability`
- [7] *Monitor Azure platform landing zone components*: `https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/design-area/management-monitor`
- [8] *What is an Azure landing zone*: `https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/`
- [9] *Azure Policy definitions effect basics*: `https://learn.microsoft.com/en-us/azure/governance/policy/concepts/effect-basics`

11
Cost Management and Optimization

In today's cloud-driven world, effective cost management is essential for organizations striving to balance operational efficiency with financial prudence. Azure Monitor provides a robust platform for cloud observability, however, without careful cost management, the benefits of these monitoring capabilities can be overshadowed by escalating expenses.

Here, we cover the crucial aspects of cost management and optimization within Azure Monitor. We will explore strategies to efficiently utilize resources, control costs, and maximize the return on your monitoring investments. By understanding how to align your monitoring strategy with budgetary constraints and organizational goals, you can ensure a cost-effective and value-driven approach to cloud observability.

Whether you are an IT professional responsible for monitoring Azure resources or a financial manager seeking to optimize cloud expenditures, this chapter provides the insights and practical steps needed to manage and reduce costs effectively.

In this chapter, we will cover the following:

- Efficient resource utilization strategies
- Understanding Azure Monitor's pricing model
- Cost control measures in Azure Monitor
- Estimating your Azure Monitor Logs costs: a hands-on example

Technical requirements

For this chapter, you should have access to Azure Monitor, familiarity with the Azure portal, and basic knowledge of the main characteristics of Azure Monitor Logs and Azure Monitor Metrics, as covered in *Part 2* of the book. We will use the Log Analytics workspace created at the beginning of *Chapter 2* for further analysis.

Efficient resource utilization strategies

Efficient resource utilization is the cornerstone of cost management in Azure Monitor. By optimizing how resources are used, you can significantly reduce costs without compromising the effectiveness of your monitoring solutions. It's essential to understand how Azure Monitor's pricing works. Azure Monitor charges are primarily based on data volume ingested and retained, the number of metrics monitored, and the usage of alerts and notifications.

In this section, we will use that knowledge to explore several strategies grouped into four main principles:

- **Principle #1**: The biggest savings come from data you don't ingest
- **Principle #2**: More savings from data you avoid processing
- **Principle #3**: Reduce the data in use if it is not necessary
- **Principle #4**: Manage alerts and notifications wisely

Let's start with the first of them.

Principle #1 – The biggest savings come from data you don't ingest

Data ingestion costs can quickly escalate if not managed properly. One of the most effective ways to manage them is to minimize the amount of data ingested. By carefully controlling what data is collected and ensuring only essential information is ingested, you can significantly reduce your monitoring expenses. Here are several techniques to implement this principle effectively:

Filter and preprocess data

Before data is sent to Azure Monitor, filter out any logs or metrics that are not essential for your monitoring objectives. This can be done by setting up data collection rules that specify which data sources should be included or excluded. By only ingesting relevant data, you can avoid the costs associated with storing and processing superfluous information.

You can also implement preprocessing techniques at the data source to aggregate or summarize data before it is sent to Azure Monitor. For example, you can aggregate detailed log entries into summary logs that provide sufficient insight without the need for granular details.

Sample data

Use sampling techniques. **Sampling** involves collecting a subset of data points that represent the overall dataset. By implementing sampling, you can reduce the volume of data ingested while still maintaining a representative view of your system's performance and health. Sampling can be particularly useful for high-frequency metrics or logs where detailed granularity is not necessary.

You can also adjust sampling rates. Depending on the criticality of the data, you can adjust them to balance between cost and visibility. For less critical metrics, lower sampling rates can be applied, whereas more critical metrics can have higher sampling rates.

Optimize diagnostic settings

Be sure that you configure diagnostic settings appropriately. Azure resources often generate a large volume of diagnostic data. By configuring diagnostic settings to collect only the necessary logs and metrics, you can prevent unnecessary data from being ingested. This involves disabling verbose logging and ensuring that only critical events are captured.

You can also use **Azure Event Hubs** for diagnostic data. For scenarios where detailed diagnostic data is required but should not be continuously ingested into Azure Monitor, consider sending data to Azure Event Hubs. This allows you to collect and store detailed logs at a lower cost compared to the costs associated with direct ingestion into Azure Monitor. You can then process this data later as needed.

Azure Policy is a useful tool to enforce rules that ensure only necessary diagnostic settings are enabled across your Azure resources in a standard way across the company. This helps maintain consistency across your resources and prevents accidental ingestion of unnecessary data.

Regular review and adjustment

Conduct regular audits. Periodically review your data ingestion configurations to ensure they align with your current monitoring needs. Adjust filters, sampling rates, and diagnostic settings based on changing requirements and cost analysis findings. The definition established at the beginning of your projects could evolve with the passage of time.

Principle #2 – More savings from data you avoid processing

While minimizing data ingestion is the first step in controlling costs, significant savings can also be achieved by optimizing how data is processed before or while it is ingested. By reducing the amount of data processed and focusing on essential insights, you can further manage costs effectively. Here are several techniques to implement this principle.

Preprocess data at the source

Implement preprocessing techniques at the data source, to transform data before it is sent to Azure Monitor, which align with the analytics, reporting, and alerting needs. Focus on processing data that directly impacts decision-making and system performance and deprioritize or omit processing of less critical data. Although Azure Monitor supports log processing during the ingestion process through data collection transformations, it will add extra costs when more than 50% of the data is filtered or modified. Those costs can easily be avoided by moving the required processing to the source.

Optimize data querying

Use efficient queries that retrieve only the necessary data. Avoid broad or complex queries that process large volumes of data unnecessarily. Optimize query performance by using appropriate indexing, filtering, and aggregation techniques. While **analytics log** querying is included in the price, **basic log** querying has an extra cost per GB of data scanned.

You should also limit query frequency to minimize the data scanned on those types of logs. Schedule queries at intervals that balance the need for up-to-date information with cost considerations.

Principle #3 – Reduce the data in use if it is not necessary

Data retention is a critical factor in avoiding an exponential growth of your costs within Azure Monitor. By strategically setting retention periods and choosing appropriate storage options for your data, you can achieve substantial savings. This principle focuses on balancing the need for readily accessible data (interactive retention) with cost-effective long-term storage (archive and restore).

Set appropriate retention periods

Assess the importance of different types of data to your monitoring needs. Critical data that requires frequent access should have longer retention periods, whereas less critical data can have shorter retention times. Use Azure Monitor's retention policies to specify how long data should be kept available for interactive querying. Set shorter retention periods for non-essential data to minimize costs while maintaining sufficient historical data for analysis.

Interactive querying versus archive and restore

Data stored in hot storage within Log Analytics is readily accessible for querying and analysis. This is called **interactive querying**. However, it comes at a higher cost. Reserve hot storage for data that you need to access frequently or that is essential for immediate operational insights. By default, Azure Monitor includes the first 31 days for free in the price per GB ingested or 90 days if you are using **Application Insights** or the **Azure Sentinel** security solution. After that, you will be billed per GB per month stored inside Log Analytics.

For data that is less frequently accessed but still needs to be retained for compliance or historical analysis, utilize Azure's archival storage options. Archiving data is five times cheaper than keeping it in hot storage and is ideal for long-term retention. You need to be careful because searching and restoring information have extra costs. Apply the previous two principles to minimize the data ingested and processed to benefit from the savings.

Principle #4 – Manage alerts and notifications wisely

Alerts and notifications are vital for proactive monitoring and prompt response to issues within Azure Monitor. However, inefficient alert configurations can lead to unnecessary costs. By optimizing alerting and notification strategies, you can maintain effective monitoring while keeping expenses under control.

Consolidate alerts

Reduce alert noise – too many alerts can overwhelm your team and increase costs. Consolidate similar alerts to reduce noise. Use advanced alerting features, such as dynamic thresholds and smart detection, to minimize the number of alerts generated. Group related alerts together to provide a comprehensive view of an issue without generating multiple individual alerts. This approach reduces the volume of notifications and simplifies incident management.

Also, adjust the evaluation frequency. Set appropriate frequencies for alerts based on the criticality of the metrics being monitored. Less critical metrics can have longer evaluation intervals, reducing the number of times the alert logic is executed and lowering costs.

Tailoring notification channels

Choose notification channels that are cost-effective for your organization. Email notifications are generally less expensive than SMS or voice calls. Use premium channels sparingly for the most critical alerts.

Implement a hierarchy for notifications. Send alerts to on-call engineers or primary responders first and escalate to higher levels only if necessary. This reduces the number of notifications sent overall.

By implementing these cost-saving principles, you can optimize your Azure Monitor setup to balance performance with financial efficiency. Efficient resource utilization, minimizing unnecessary data ingestion and processing, optimizing data retention periods, and streamlining alerting and notifications are key strategies to manage costs effectively.

As we move forward, understanding the intricacies of the Azure Monitor pricing model is crucial for further optimizing costs. In the next section, we will jump into the details of how Azure Monitor charges are structured, providing you with the knowledge needed to make informed decisions and maximize the value of your monitoring investments. Let's explore how to leverage Azure Monitor's pricing model to enhance your cost management strategies.

Understanding Azure Monitor's pricing model

Effective cost management in Azure Monitor begins with a thorough understanding of its pricing structure. Azure Monitor charges are based on several components, each with its unique pricing model. This section will cover the pricing structure for **Logs**, **Metrics**, **Alerts**, **Notifications**, **Web Tests**, and **System Center Operations Manager** (**SCOM**) managed instances, providing insights into how to align their usage with cost-saving principles. Let's first start analyzing the pricing model for Logs.

We're using West Europe pricing at the time this book is being written as a reference for the explanations.

Logs

Azure Monitor Logs pricing is primarily based on data ingestion and data retention:

- **Data ingestion**: Charges are applied per GB of data ingested. The more data you ingest, the higher the cost
- **Data retention**: Default retention is included, but extending retention beyond the default period incurs additional charges per GB per month

> **Important note**
>
> Azure Monitor uses the equivalence of one gigabyte = 10^9 bytes. Charges are calculated based on the size of each record in the Log Analytics workspace. Calculation is done after transforming the different values inside each column to a string format. It is important to know that this volume is usually lower compared with the income JSON file for each registered event. On average, across all event types, Microsoft shares that the billed size is about 25% less than the incoming data size and can be up to 50% smaller for small events. Azure Monitor excludes specific columns and tables that are used internally for management or billing purposes.

As described in *Chapter 4*, Azure Monitor supports two types of logs: basic and analytics logs. Data ingestion pricing is based on a pay-as-you-go model by default in both cases. Selecting the right type of logs for your specific workload can have an impact on your savings of almost *5x*. We can use this as a reference:

- Basic log ingestion rate = $0.65 per GB
- Analytics log ingestion rate = $2.99 per GB

Analytics logs also offer reservation plans going from ingestion rates between 100 GB to 50,000 GB per day. Discounts vary from 15% to 36% over the standard pay-as-you-go prices.

After analytics log data is ingested and the data retention period included for free in previous prices has ended, an extra proportional cost per GB per month stored inside Log Analytics is added to your bill. As a reference, the cost is $0,13 GB per month. As mentioned in *Principle #3*, defining the right retention period is critical to avoid exponential growth of your costs.

Finally, if you want to export your logs to a third-party external solution, you have two options. Configure a diagnostic setting at the resource level with a cost of $0.325 per GB – that information is sent directly to the remote services without going through the Azure Monitor ingestion pipeline, or exporting your logs directly from the Log Analytics workspace at $0.13 per GB on top of the extra costs paid previously for inserting and transforming your logs inside your Log Analytics workspace.

Having covered the details that apply to Azure Monitor Logs, let's continue with the next service metrics.

Metrics

As discussed in *Chapter 2*, Azure supports different types of metrics. The first ones are platform-generated metrics, also known as built-in metrics. Those metrics are enabled by default and included for free in your Azure subscription during the supported 90-day retention period.

On the other hand, there are custom metrics where you can define and collect custom metrics tailored to their unique requirements using available APIs and SDKs. In this case, there are no free units included and pricing is based on samples ingested. As a reference, each 10 million samples are priced at $0.16.

Using **Azure Metrics Explorer** (aka Metrics Explorer) as described in *Chapter 4* to visualize and analyze your metrics is free. However, if you want to export those metrics through Azure Monitor APIs to integrate them in third-party systems such as the ones described in *Chapter 10*, Azure provides the first 1 million API calls for free, and after that, they are priced at $0.01 cent per 1,000 API calls.

Aligned with *Principle #2*, implement an efficient metric collection process and collect only the necessary custom metrics. Aggregate and sample metrics data as needed to reduce the number of data points before submitting them to Azure Monitor.

Once the information from both logs and metrics is stored in the workspace, configuring monitoring alerts is the next step.

Alerts

Alert pricing depends on the type and number of alert rules configured and the frequency of their evaluations. As described in *Chapter 5*, an alert rule observes your data and detects a signal that signifies an event occurring on the specified resource. The alert rule captures this signal and evaluates whether it meets the specified condition criteria. If the condition criteria are met, the alert is triggered. It's possible to identify two types of alerts:

- **Metric alerts**: Charges are based on the number of time series monitored and the evaluation frequency
- **Log alerts**: Charges are based on the number of log queries and their evaluation frequency

Azure includes the first 10 monitored metrics per month for free. After that, every metric is charged at $0.10. If you are using smart detection rules through a dynamic threshold, an extra fee of $0.10 is included.

Regarding log alerts, Azure includes 100 rules per subscription for free when monitoring activity logs – the ones created by Azure when a resource is created, modified, or deleted. However, other log alerts include only one for free and, after that, they incur extra costs based on the frequency at which the alert is analyzed. It goes from $0.50 for a 15-minute frequency to $3 for a 1-minute frequency. As described in *Principle #4*, adjust evaluation frequencies to balance cost and responsiveness.

Triggering an alert without anyone receiving it doesn't make a lot of sense. Let's move on to the pricing details for notifications.

Notifications

Notifications are sent after an alert rule is triggered. Notification costs are associated with the channels used for alert notifications, such as email, SMS, or voice calls, and the number of notifications sent:

- **Email notifications** include 1,000 free each month. After that, every 100,000 emails are billed at $2.
- **Push notifications** include 1,000 free each month. After that, they have the same price as email notifications for every 100,000 notifications.
- **IT service management (ITSM) events** submitted to a third-party platform also include 1,000 free each month, and after that, every 1,000 events are billed at $5.
- **Webhooks** include 100,000 per month. After that, every 1 million webhooks are billed at $0.60. If you need to use secure webhooks, the first one is included for free, and after that, every 1 million is billed at $6.

For voice calls and SMS, each destination country has a different policy. While the first 10 calls are included for free, prices for both voice calls and SMS vary based on the country code.

As mentioned in *Principle #4*, tailor your notification channels to choose channels that are cost-effective for your organization and meet your notification requirements.

The last two channels are less common and only applicable if you use specific features of Azure Monitor integrated services.

The first one is **Web Tests**. They monitor the availability and performance of web applications and services. Ping web tests are included for free. Standard ones are billed at $0.00065 cents per execution and multi-step web tests at $10 per test per month.

The second one is **SCOM Managed Instance** (**SCOM MI**). They incur charges based on the number of monitored instances of $6 per endpoint each month. Any other resource required to execute SCOM MI would be billed independently at a pay-as-you-go rate (i.e. **Azure Key Vault**, **SQL Managed Instance**, etc.).

Having gained a comprehensive understanding of the pricing structures for various Azure Monitor components, it's now crucial to apply this knowledge to effectively manage and control costs. Now, we will move on to the practical strategies for controlling Azure Monitor costs, utilizing tools and features such as the usage and estimated cost feature of Log Analytics, workspace-based resources from Application Insights, and the overarching capabilities of **Azure Cost Management and Billing**. These tools will enable you to control the cost-saving principles discussed earlier and ensure that your monitoring setup is both efficient and financially sustainable.

Cost control measures in Azure Monitor

Controlling costs in Azure Monitor requires a proactive and informed approach, utilizing various tools and features provided by Azure. The usage and estimated cost feature of Log Analytics is an invaluable resource for this purpose. By accessing this feature, you can gain detailed insights into your data ingestion and retention costs.

In *Figure 11.1*, you can see how the usage and estimated cost features allow you to track how much data is being ingested over time and the associated costs, providing a clear view of your spending patterns. In this small environment, consumption is being driven by both analytics and basic logs, although it is not big enough yet to be captured by the wizard.

With this information, you can identify which data sources are driving costs and adjust your data collection strategies accordingly. For instance, you might decide to filter out non-essential logs or adjust retention policies to archive older data more quickly, thus reducing storage costs.

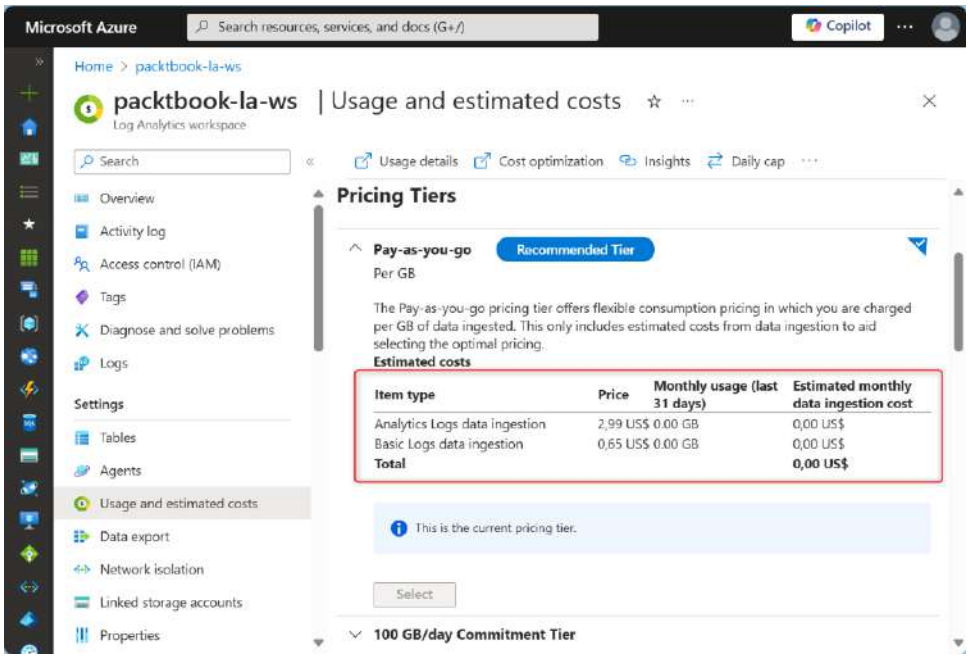

Figure 11.1 – Details of your Log Analytics workspace usage and estimated costs

The usage and estimate costs feature also shows you the recommended pricing tier based on your actual consumption. In my case, pay-as-you-go is the most effective; however, if my daily ingestion volume matches any of the available commitment tiers, the service will recommend updating my pricing tier to get the cheapest price available.

Scrolling down, it is possible to go further into the analysis of ingested data in the workspace. As you can see in the following screenshot, the size per day is around 40 KB – not a big number, and it is distributed across Azure Diagnostics and Azure Monitor Metrics (see *2* in the figure).

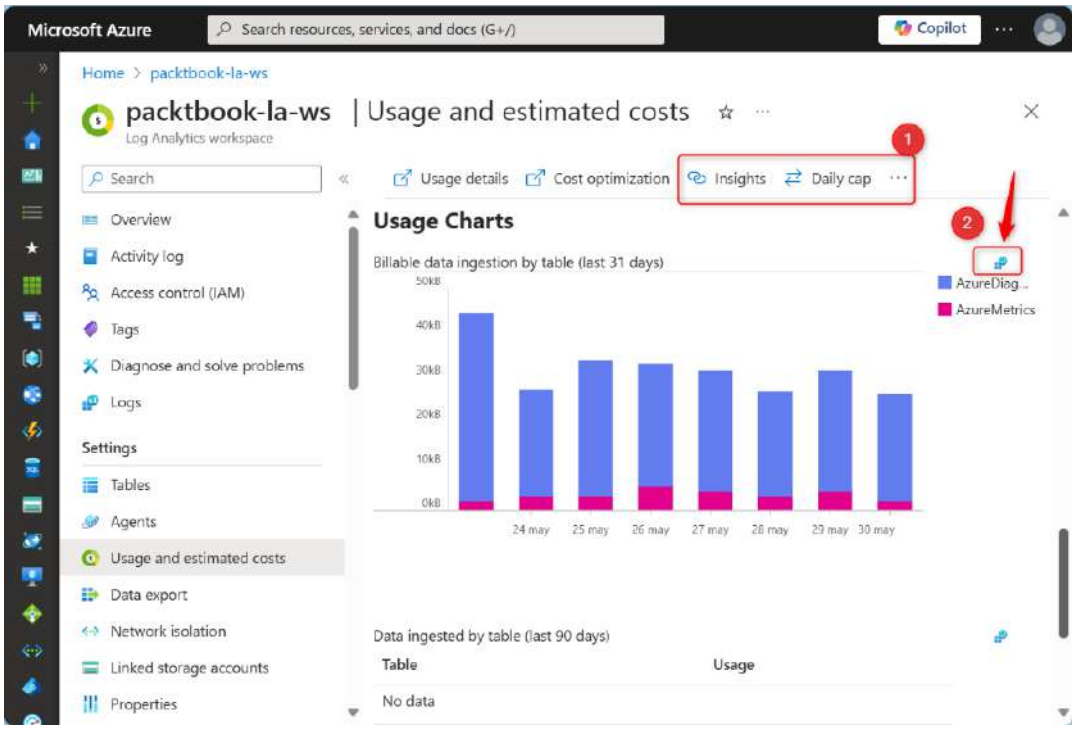

Figure 11.2 – Graphical representation of your Log Analytics workspace usage

If this information is not enough for us, we can go further in our analysis. Information provided in this report is extracted from the Log Analytics workspace itself through KQL queries. There are two possible options, as marked in the preceding screenshot: the first one is using the **Insights** menu option; the second one is editing the specific queries for the charts and tables.

In the first case, an Azure Monitor workbook is opened with the Log Analytics workspace **Insights** view. One of the scenarios available for visualization is described in *Chapter 9*. This workbook has multiple tabs that allow you to review the specific relevant details of your workspace, not only your usage, as shown in the following screenshot. It also supports the configuration of your time window for reporting details.

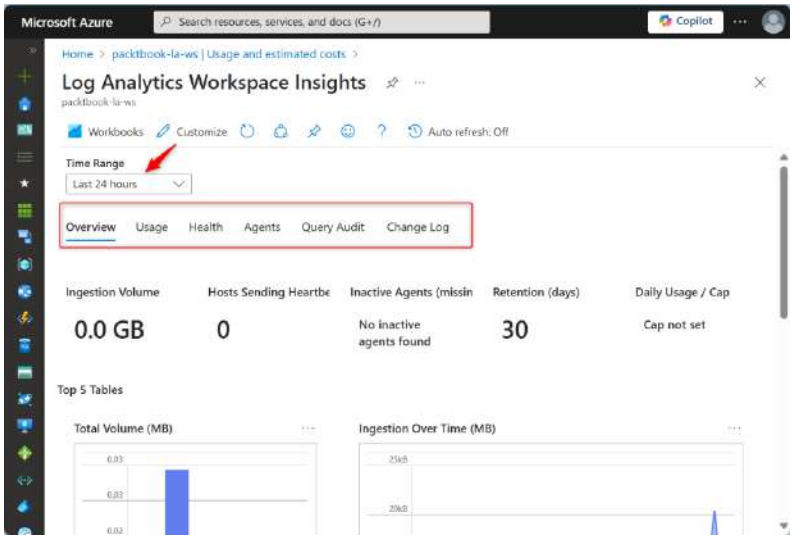

Figure 11.3 – Available details in Log Analytics workspace Insights

The second option provides direct access to the KQL editor inside the Log Analytics workspace to analyze the KQL query associated with the specific graphical representation shown in the portal. As shown in the following screenshot, you can modify the specific KQL query and customize the way the data is presented if needed. This way, you can customize your own workbook to your needs and save it, share it, or integrate it into your own custom dashboards and workbooks.

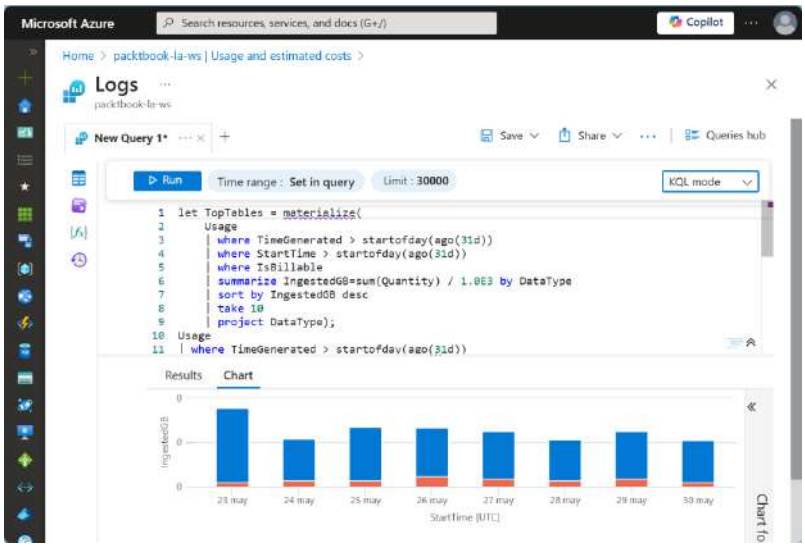

Figure 11.4 – Details of the associated KQL query in case you need to modify it

If you don't know how many logs you will ingest and you want to avoid the surprise of a big bill, the **Daily Cap** feature in a Log Analytics workspace will help you. It provides an option for controlling and managing data ingestion costs. It allows you to set a maximum limit on the amount of data that can be ingested into the workspace each day, helping to prevent unexpected cost overruns and ensuring that your monitoring expenses remain predictable and within budget.

When you configure a daily cap, you specify the maximum amount of data (measured in GB) that can be ingested into the Log Analytics workspace in a 24-hour period. Once this limit is reached, any additional data sent to the workspace will be discarded for the remainder of the day, preventing further ingestion charges. This helps you avoid scenarios where a sudden spike in data volume could lead to unexpectedly high costs.

It is recommended that you set up notifications to alert you when the data ingestion reaches the daily cap. This allows you to take proactive measures such as investigating and addressing the root cause of the increased data volume to avoid further impact in your monitoring strategy. While the daily cap helps control costs, it also means that data exceeding the cap will be lost for the day. That creates a dangerous situation of data loss for critical information and reduced capabilities for analysis and reporting.

In addition to Log Analytics, workspace-based resources from Application Insights offer another layer of cost control. The interface is like the Log Analytics one shown previously. It exposes an estimation of the monthly charges based on usage from the past month and the billable data ingestion by table from the past month.

Like the log experience, it's possible to go deeper in your investigation of Application Insights usage through the **Metrics** page using Azure Metrics Explorer. Information on data usage is stored under the **Data point volume** metric. It is possible to show data by each item type using the **Apply splitting** option for this multidimensional metric to split the data across dimensions.

Your last option is using the generic Azure Cost Management and Billing service. A comprehensive tool that provides a holistic view of your Azure expenditures, including those incurred by Azure Monitor. This tool enables you to set budgets and create cost alerts, helping you stay within your financial limits.

By reviewing cost analysis reports, you can identify trends and anomalies in your spending. Azure Cost Management also offers recommendations for cost optimization, such as resizing or shutting down underutilized resources. Consider the virtual machine we used in *Chapter 2* as an example. *Figure 11.5* shows that, for that case, Log Analytics consumption was small compared to the virtual machine.

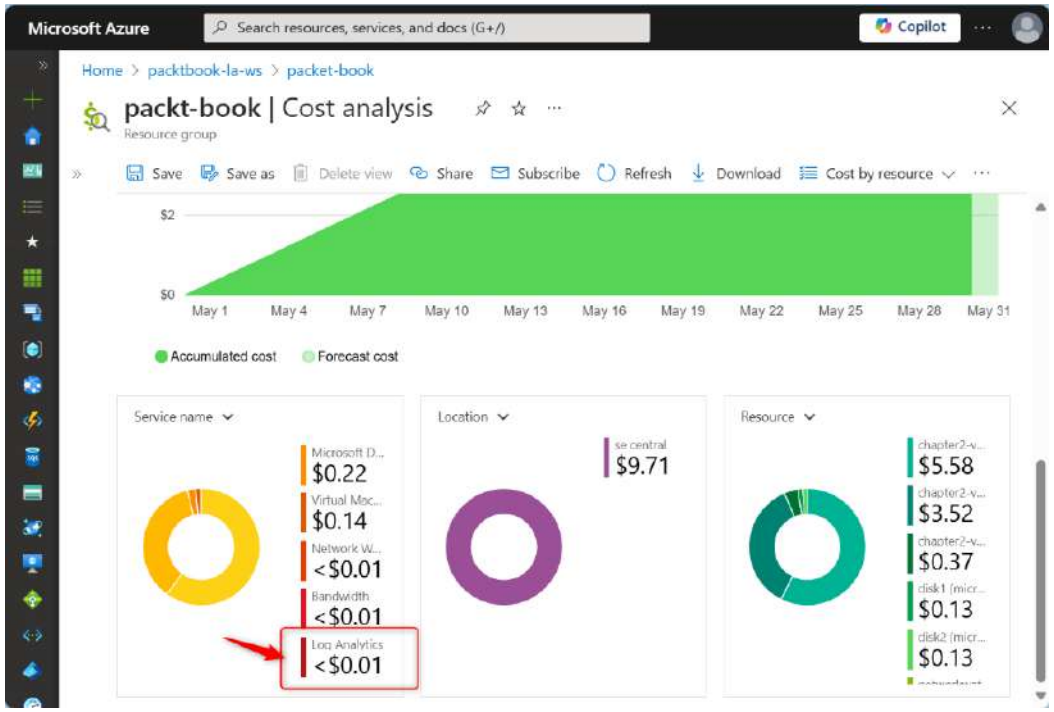

Figure 11.5 – Details of the costs in the Cost Analysis view

Additionally, the tool allows you to allocate costs to different departments or projects, providing visibility into which areas of your organization are driving Azure Monitor expenses. This level of granularity helps you make informed decisions about resource allocation and cost-saving measures.

By integrating these cost management features, you can effectively monitor and control your Azure Monitor expenses. Regularly reviewing the usage and estimated cost reports from Log Analytics, analyzing the performance insights from Application Insights, and leveraging the budgeting and alerting capabilities of Azure Cost Management ensures that your use of Azure Monitor remains aligned with your financial objectives. Adjusting your configurations based on the insights provided by these tools will help you maintain a cost-efficient monitoring setup, ultimately maximizing the value derived from your Azure investments.

After gaining a basic understanding of the pricing structures for various Azure Monitor components, you are now equipped to apply this knowledge to real-world scenarios. To illustrate how these pricing details translate into actual costs, we will walk through a practical example of cost calculation. This example will demonstrate how to estimate and manage expenses for your Azure Monitor setup.

Estimating your Azure Monitor Logs costs – a hands-on example

Imagine that you are responsible for defining the monitoring requirements for a new application that is going to be deployed in Azure. You need to provide an estimation of cost to your supervisor before the deployment is approved.

Microsoft provides the Azure Pricing Calculator (`https://azure.microsoft.com/en-us/pricing/calculator/`) as a reference tool to estimate the cost of your infrastructure on Azure. As shown in the following screenshot, you can search for any of the available Azure services and include them in your estimation.

If you look for Azure Monitor and add it to your estimation, a similar wizard to the one shown in the following screenshot will appear. As you can see, there are multiple options to introduce the details of the features in use to get your total estimate. The following example uses this calculator to estimate costs.

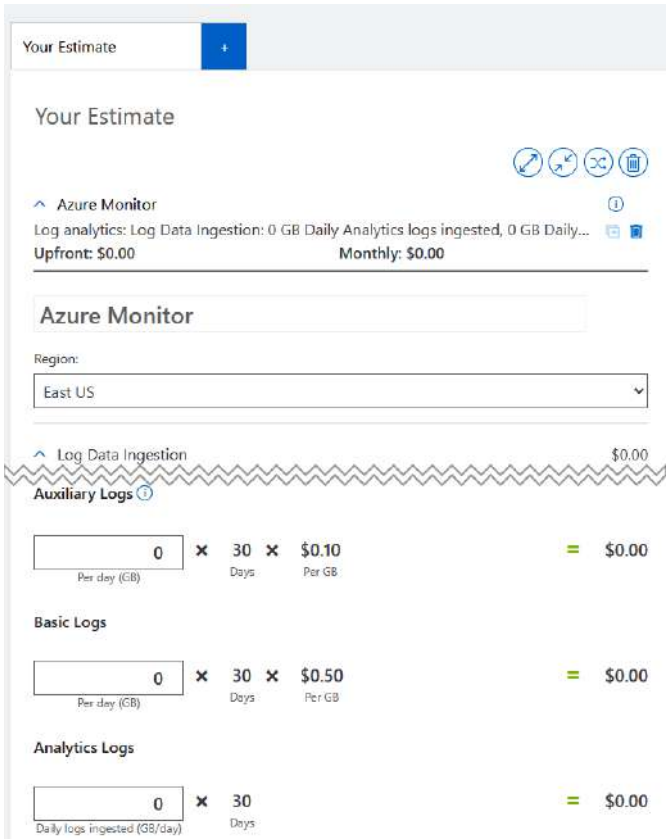

Figure 11.6 – Partial details of Azure Monitor options in Pricing Calculator

Imagine that our application is very verbose and generates an important volume of data per day – an initial estimation is around 200 GB per day. Initially, we did not have a specific requirement for the retention period, so let's start with the maximum of 2 years that Azure Monitor provides for interactive querying of that information. During the first month, our monitoring data would be stored for free thanks to the 31 days included in the ingestion price. After that, both storage and its associated costs would accumulate to the maximum retention defined – 2 years.

As shown in the following screenshot, each month, a total of 6.2 TB is being generated. We consider the 200 GB generated each day, with a month of 31 days. Although it adds an extra 7 days per year to that estimation, it simplifies the calculations due to Azure Monitor retaining the information for free for the first 31 days.

After 4 months, we already have more than 18 TB of logs retained, which grows to 68 TB after one year, and 148 TB after the second year.

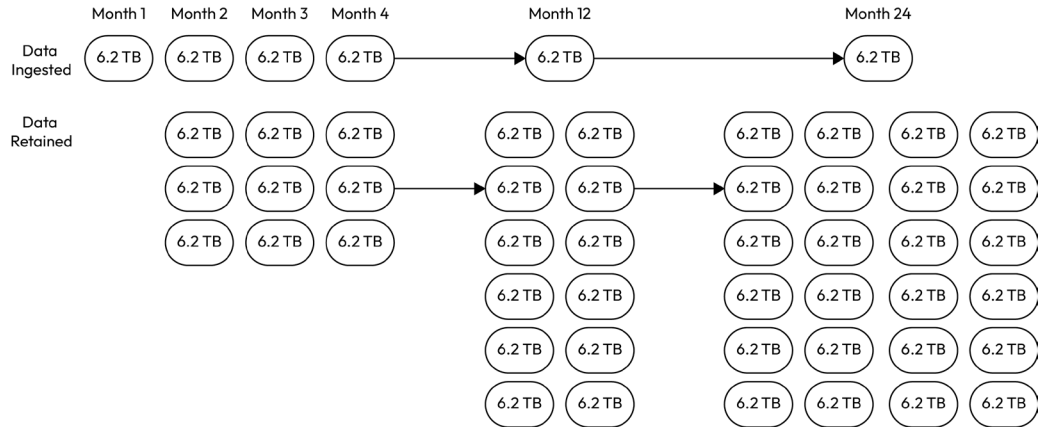

Figure 11.7 – Evolution of the total storage used in our initial scenario

Using the pricing mentioned in the previous chapter as a reference, your costs would start at $806 for month 2, but they would reach more than $19K a month at the end of the retention period.

Show me the money

If you want to go into more detail about the cost analysis described in this chapter, it's possible to download the associated spreadsheet from the GitHub repository. It contains a sheet with the three scenarios and the cost and total storage used by month.

Maybe two years is too much. In this case, we can try to configure the appropriate retention periods based on their importance and usage frequency. Imagine that, in the previous scenario, critical logs related to security have longer retention periods, while less critical logs can have shorter retention periods. We establish that a one-year-and-a-half retention period is required for critical logs and six months for less critical ones. Critical logs represent 35% of the total ingested information while less critical ones are the other 65%.

As you can observe in the following figure, the total amount of information has now been reduced compared with the previous scenario. After four months, we have the same as before, around 18 TB of logs retained. However, due to the change in our policy, we see that after the sixth month, fewer critical logs stop being retained, as in the previous example. As shown in the figure, only the six previous months are stored for less critical logs while the critical ones continue to grow. After a year, a total of 57 TB would be stored compared to the 68 TB in the previous scenario, a reduction of 16% of the total stored amount.

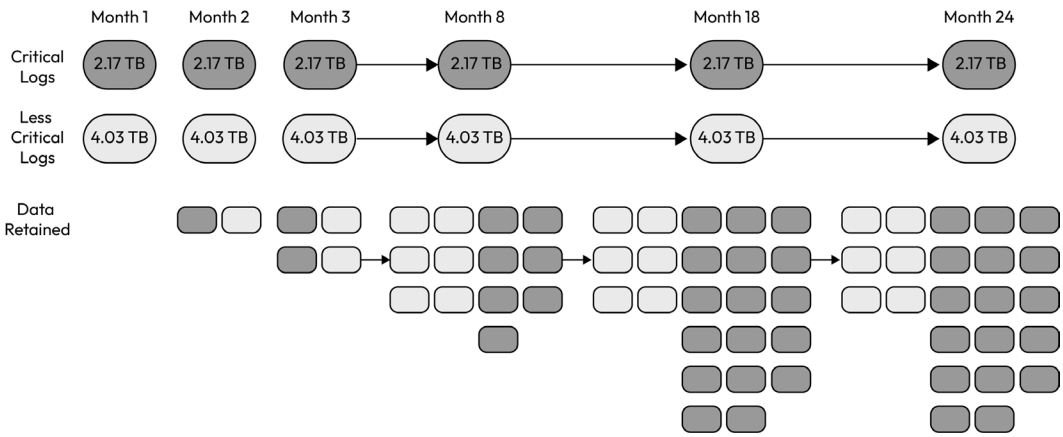

Figure 11.8 – Evolution of the total storage used in our second scenario

Something similar happens after the 18-month mark. Both types of logs reach the maximum retention time and old ones are discarded. After two years, 81 TB are retained compared to the 148 TB in the previous scenario. We have reduced the total amount of data in the service by 45%. Again, using the pricing reference, your costs would start at $780 for month 2, but they would reach a little more than $10K a month at the end of the retention period. With this initial change, we have reduced our bill by $8K.

However, after reviewing the information from other projects, you get the confirmation that while that retention period makes sense for business purposes, most data is not accessed three months after being generated. In that case, we can include the alternative of archiving that information.

As you can observe in the following figure, the total amount of information is now the same as in the previous one. However, we have distributed the data across two tiers. The last three months are in the interactive tier that allows you to have online access to that information, and the rest is archived.

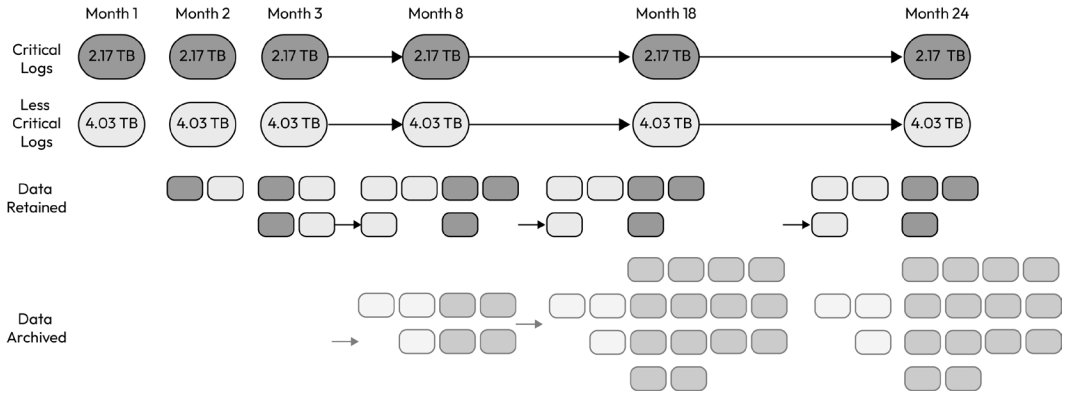

Figure 11.9 – Evolution of the total storage used in our third scenario

After archiving the information not accessed, the total bill has been reduced by another 77%. The total monthly cost at the end of the two-year analysis would now be $2,4K. A total reduction of 87% compared with our initial monthly costs.

If you need to search through archived logs due to an ongoing security investigation, asynchronous search jobs will help you with that and the price will still be cheaper compared to keeping them all the time with interactive querying.

Those assumptions can be modified across the lifecycle of our service. It's important to perform a regular review and adjustments as described in *Principle #1*.

Summary

In this chapter, we explored the critical aspects of cost management and optimization within Azure Monitor. Effective cost management is vital for organizations seeking to balance operational efficiency with financial prudence in a cloud-driven environment. This chapter provided comprehensive strategies for efficient resource utilization, understanding Azure Monitor's pricing model, and practical steps to estimate and manage monitoring costs effectively.

We introduced four principles to be considered when designing your monitoring strategy for effective cost management. Techniques to minimize data ingestion and processing costs, such as filtering and preprocessing data, using sampling methods, and optimizing diagnostic settings, were introduced.

After that, we went into more detail about how Azure Monitor charges for logs, metrics, alerts, notifications, and additional features and described a hands-on example to illustrate how to calculate and manage costs for Azure Monitor Logs setups, emphasizing the importance of appropriate data retention periods and the use of archiving to reduce expenses.

In the next chapter, we will explore future trends in the field of cloud monitoring and management. We will discuss emerging technologies, best practices for staying ahead of industry changes, and strategies for leveraging new tools and features to enhance your cloud monitoring capabilities.

12
Future Trends and Looking Ahead

Throughout this book, we have reviewed the rapidly evolving landscape of monitoring and observability and how it promises to transform how applications running in the cloud ecosystem operate. In this final chapter, we will delve into emerging trends and innovative approaches that will redefine the way we monitor, analyze, optimize, and operate our cloud environments.

This chapter explores how AI integration into observability platforms is revolutionizing the way we predict, detect, and solve problems. AI-driven analytics enable predictive observability, helping your organization anticipate issues before they arise, reducing downtime, and enhancing reliability. We will examine how tools such as Copilot for Azure, integrated with Azure Monitor, enhance predictive capabilities and support proactive monitoring. We will continue exposing how, as serverless computing grows, traditional monitoring struggles with its stateless and ephemeral nature. We'll discuss new approaches and tools required for monitoring serverless architectures and tackle the challenges they present. Finally, we'll cover evolving observability standards and best practices, emphasizing the importance of staying up to date to maintain effective monitoring. We'll also highlight emerging frameworks that are shaping the future of cloud monitoring.

By the end of this chapter, you will have a comprehensive understanding of why staying up to date with these standards is crucial for maintaining strong and effective monitoring practices.

In this chapter, we will cover the following topics:

- AI-driven analytics for predictive observability
- Serverless observability – monitoring the unseen
- Evolving standards in cloud observability

AI-driven analytics for predictive observability

Using AI and machine learning in today's dynamic cloud environments provides advantages over traditional monitoring tools due to the scalability and complexity of modern applications. They facilitate the analysis of historical and real-time data to forecast potential issues before they affect the application or system. This proactive approach is called **predictive observability**.

In this section, we discuss the capabilities of Azure Monitor to provide predictive observability and how tools such as **Microsoft Copilot for Azure** bring a new level of automation and intelligence to observability, helping organizations manage their cloud environments more effectively.

The components required for AI-driven analytics with Azure Monitor are as follows:

- **Data collection and aggregation**: Predictive observability begins with comprehensive data collection from multiple sources, including logs, metrics, traces, and events from across the application stack. This data should be aggregated into a centralized platform where it can be processed and analyzed at scale, such as **Azure Monitor Logs**. For this phase, we recommend reviewing *Chapters 3* and *7*, where we analyze in detail how this data collection and processing should be done in different ingestion scenarios.
- **Machine learning models**: Machine learning models are used to detect signs of potential or real-time problems. These models are trained with historical data to recognize patterns and anomalies. For Azure Monitor, **Kusto Query Language** (**KQL**) includes operators and machine learning functions for time series analysis, anomaly detection, forecasting, and root cause analysis. For more information on the use of these operators, refer to [1, 2, 3] in the *Further reading* section.

  The following figure shows an example of a network metrics forecasting scenario that uses the `series_decompose_forecast` operator in Azure Log Analytics. In the `NetworkIPs` table, there are four weeks (until 2024-06-30) of IP data in an hourly grain, with weekly seasonality and a stable trend. We then use `make-series` to create a time series from the `NetworkIPs` table showing the number of distinct IPs per hour and add another two empty weeks to the series (`24*7*2h`). Finally, the `series_decompose_forecast` operator is applied with two weeks (`24*7*2 points`), automatically detecting seasonality and trends and generating a forecast for the entire six-week period (until 2024-07-14).

Figure 12.1 – Network IPs forecasting scenario using the series_decompose_forecast operator

Similarly, using the techniques learned in *Chapter 4*, along with the forecast and anomaly operators (also check out references [1, 2, 3] in the *Further reading* section), you can perform series analysis, anomaly detection, and forecast scenarios on your dataset.

- **AI for forecasting and anomaly detection**: The use of AI-powered systems makes it possible to predict future trends based on historical patterns. For example, they can forecast resource utilization, identify potential bottlenecks, and predict application or system failures. Similarly, AI models can autonomously and continuously monitor data streams for anomalies, and once anomalies have been detected, can alert the cloud operations team before the issue becomes critical to the application. In Azure, multiple AI models can be integrated into observability platforms to provide these advanced forecasting and anomaly detection capabilities.

AI-powered assistants for observability

One of the most valuable aspects of generative AI is the ability it provides to innovate and deliver *human* solutions to users. Within this ecosystem is Copilot, an AI-powered virtual assistant that improves user productivity by helping humans with complex cognitive tasks, providing contextual suggestions, and generating data-rich insights. These copilots can be based on specific data and context from customers or third parties, offering the opportunity to create generative AI experiences that understand business-specific data.

In this ecosystem, there are different approaches to creating AI experiences within organizations depending on the unique requirements that need to be met. For this, Microsoft provides several approaches divided into patterns based on different scenarios that can be consulted in detail in the *Further reading* section at the end of the chapter [4, 5].

Our goal in this section is to present the benefits of these AI-powered assistants and how they can help in the observability ecosystem. In particular, we are going to present Microsoft Copilot for Azure, an AI-powered assistant that falls within one of the approaches mentioned before, and that has been

designed to help manage and optimize Azure cloud environments. It uses machine learning and natural language processing to provide intelligent recommendations, automate routine tasks, and improve observability.

In the case of observability, Microsoft Copilot for Azure can help us with multiple tasks:

- **Intelligent Recommendations**: Microsoft Copilot for Azure provides recommendations for performance improvements, cost optimizations, and configuration improvements based on an analysis of telemetry data. These recommendations can be adjusted to the context of a specific resource if necessary.
- **Automated Monitoring and Alerts**: Microsoft Copilot for Azure can help define and configure monitoring tools and establish automatic alerts following best practices and organizational requirements.
- **Incident Investigation and Management**: Microsoft Copilot for Azure can provide graphs on platform metrics for a specific resource, provide Azure Monitor query logs using natural language, list alerts for a specific time range, and even display the number of critical alerts in a time range. Additionally, Microsoft Copilot for Azure has a capability called **Azure Monitor Investigator** that allows you to run an investigation on a specific resource, even detect anomalies on a specific resource, and perform root cause analysis.

  The following figure shows how you can use Azure Monitor Investigator. The first step is to enable the Microsoft Copilot for Azure prompt window and then ask, `What is causing the issue in this resource?`. You can use a different prompt depending on your scenario and the information you want to get from Azure Monitor Investigator [6, 7].

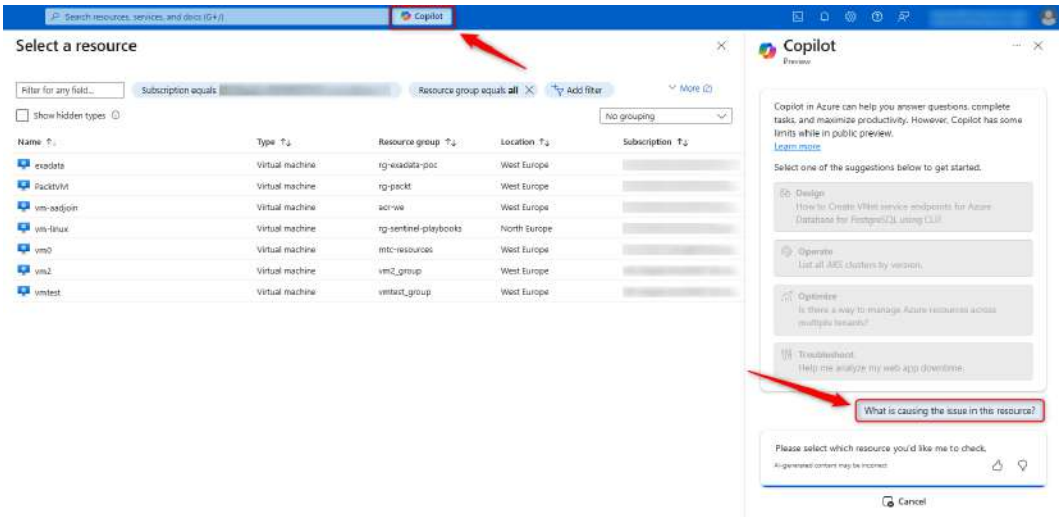

Figure 12.2 – Get information about a resource using Azure Monitor Investigator

As shown in *Figure 12.3*, Microsoft Copilot for Azure will ask you to establish resource context by selecting the resource you are interested in checking. It will ask you and show you different types of resource information for your research.

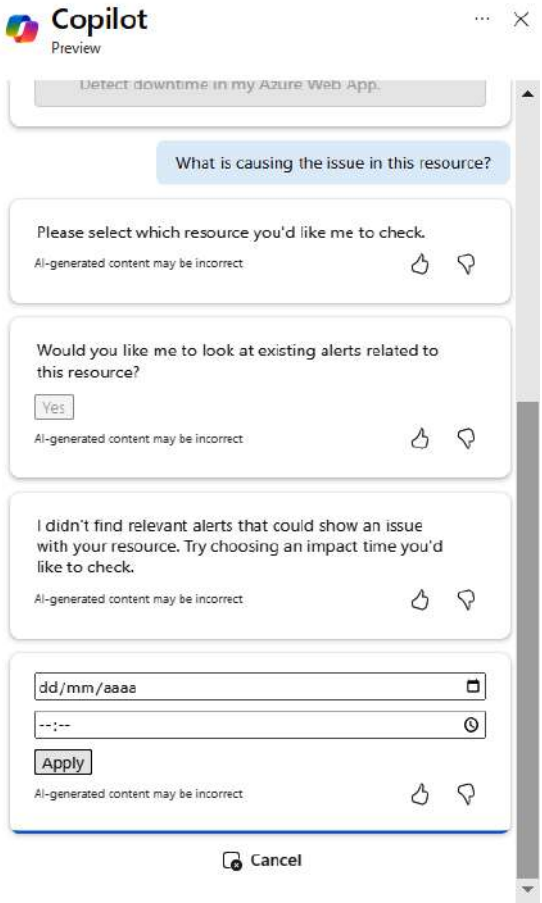

Figure 12.3 – Including the context of the issue for Azure Monitor Investigator

It is evident that the use of these solutions reduces the complexity of managing observability platforms and allows cloud teams to maintain a high level of visibility and control over the state of their applications. However, the skills that the Microsoft Copilot for Azure approach may not meet the needs or scenarios of your organization and may require taking another AI-powered assistant approach. As we mentioned, Microsoft offers different approaches based on different criteria. It is possible that you will want to build your own copilot with several very specific skill sets associated with different plugins to have end-to-end control of the AI experience.

In the next section, we will expand on the specific challenges of observability in serverless environments and see how what we have discussed in this section can help in those environments.

Serverless observability – monitoring the unseen

Serverless architectures provide a high level of abstraction that allows developers to focus on writing code without worrying about the underlying infrastructure. This abstraction, along with its ephemeral nature, has changed the way applications are created and deployed but presents a challenge in terms of the observability of this type of architecture. Traditional monitoring techniques are efficient in long-running, stateless server environments, but do not scale or are not efficient in many cases when applied to serverless architecture applications. In this section, we explain the strategies and tools needed to effectively monitor serverless architectures.

In serverless architectures, resources are dynamically created and destroyed in response to events, but this is just one example of the dynamic behavior of these architectures. Here, we present the main challenges to the observability of serverless architectures:

- **Lack of Infrastructure Control**: In serverless architectures, cloud development and operations teams don't have control of the underlying infrastructure, and it is the cloud provider that is responsible for managing the servers, network, and storage. This abstraction creates a challenge when identifying potential failures in application performance, latency problems between components, or limitations in the use of resources given the limited visibility of the underlying platform.
- **Ephemeral Nature**: Serverless functions, such as **Azure Functions**, are stateless in nature and exist only for a short period of time. Additionally, triggering functions are typically created based on events and scaled based on demand. All of this contributes to making it difficult to use traditional monitoring methods to track their life cycles.
- **Event-Driven Architecture**: Serverless applications are typically composed of multiple serverless functions that are triggered by events. This event-driven architecture creates a complex design that requires detailed logging and tracking to clearly understand the flow of interactions and dependencies that exist between the different components of the application architecture.
- **High Granularity**: In serverless architectures, many of the application tasks are assigned to a single function that is instantiated, so detailed monitoring is required to obtain the metrics and logs associated with that discrete unit of work.
- **High Volume and Typology of Metrics**: Serverless applications are typically composed of multiple interacting functions and microservices, each generating its own metrics and logs with various typologies and volumes. Traditional systems suffer from performance and cost issues when attempting to manage these needs with the capabilities they offer.

- **Distributed Tracing**: In *Chapter 7*, we mentioned what distributed tracing is and how, in serverless architectures, tracking the flow of requests between all the distributed components of the application is a challenge from the point of view of end-to-end visibility and diagnosing performance problems in the application stack.
- **Cost Management**: The ephemeral and autoscaling nature of serverless applications is a challenge in not only predicting costs but also detecting anomalies or unexpected costs resulting from inefficiencies in the code and very frequent function triggers, among other things. Visibility of the detailed uses and interaction between components is necessary to optimize the use of resources and perform precise and efficient cost control.

Techniques and tools for serverless observability

As we explained in the previous section, traditional monitoring and observability tools and practices face multiple challenges in providing the visibility necessary for the dynamism and ephemerality of serverless architectures.

Observability of serverless architectures requires innovative approaches and specialized tools based on techniques such as centralized logging, structured logging, and custom metrics that allow you to collect and analyze the dynamic and distributed nature of serverless functions.

In this section, we expose the methods and tools that are fundamental for effective observability in serverless architectures:

- **Enhanced Logging and Monitoring Solutions**: Logging and monitoring tools for serverless environments require a centralized logging approach, such as Azure Monitor Logs, where logs from serverless functions are consolidated into a single repository for streamlined analysis. It is important that the tools support the implementation of structured log formats such as JSON, which simplifies the query and analysis of log data. Additionally, adding custom metrics provides deeper insights into application-specific performance indicators, improving overall observability. In *Chapters 4*, *7*, and *8*, you can find more details and techniques for enhancing logging and monitoring.
- **Distributed Tracing**: In a serverless functions or microservices architecture, each service or function can fail independently for a variety of reasons, and it is important to understand what happened so you can troubleshoot. In this sense, distributed tracing tools are essential for tracing requests and isolating end-to-end transactions as they progress through the different components of a serverless application. **OpenTelemetry**, an open source observability framework, supports distributed tracing, metrics, and logging, and can integrate with serverless functions for end-to-end tracing. OpenTelemetry, along with **Azure Application Insights**, which provides distributed tracing capabilities, can be combined to provide developers with effective monitoring of serverless functions and their dependencies.

- **Cold Start Monitoring**: Cold starts of serverless functions influence application performance, so monitoring these starts by implementing metrics can help measure the frequency and duration of function initialization times. A good practice when working with cold-start services is to keep functions active through periodic invocations to reduce cold-start latency. Monitoring all these strategies is essential to ensure that application performance remains optimal.
- **Data Management**: Traditional monitoring systems are not equipped to handle the volume and diversity of data generated by serverless architectures. An effective approach for this scenario is data sampling, which collects representative samples of metrics, thereby reducing the volume of data without sacrificing essential data insights. Another approach is to use aggregate metrics to summarize detailed data, offering a high-level view of performance while allowing detailed analysis of specific issues. In previous sections, we have analyzed how Azure Log Analytics is a service designed to cover these data volumes and diversity needs for efficient and effective monitoring in complex environments.
- **Cost Monitoring and Optimization**: Cost monitoring tools are essential for managing and optimizing expenses for serverless applications. Setting up billing alerts ensures that cloud teams are notified when costs exceed predefined thresholds, allowing timely intervention to manage the issue. Analyzing resource usage patterns using Azure Log Analytics along with tools such as **Azure Cost Management** helps identify inefficient functions or excessive invocations, providing detailed cost insights. Additionally, refactoring inefficient code and optimizing feature configurations based on usage patterns revealed by these monitoring tools can lead to significant cost savings and improved performance. In *Chapter 11*, you can also find techniques that can help you with cost monitoring and optimization.
- **Proactive Incident Management**: Using AI and machine learning to predict and prevent issues before they impact your application contributes to proactive incident management. Using Azure Log Analytics incorporates anomaly detection algorithm operators that help identify unusual patterns or deviations in metrics and logs. Predictive analytics, which uses machine learning models, can forecast potential issues and resource needs by analyzing historical data to predict future trends and potential bottlenecks. This approach allows the organization's cloud teams to detect problems early, improving the stability and performance of applications.
- **Continuous Integration (CI) and Continuous Delivery (CD)**: Integrating observability into CI/CD processes is important to ensure that there is end-to-end visibility into the application development life cycle. Including automated tests in pipeline stages ensures that serverless functions generate logs and metrics, and these are collected and tracked correctly. Additionally, to ensure that new code does not introduce performance or reliability issues into the application, continuous deployment pipelines need to automatically deploy and monitor code updates.

In this section, we have explained how the observability of serverless architectures needs a different approach than traditional monitoring to address the multiple challenges of these architectures.

In the next section, we will introduce the evolving standards in cloud observability and their importance not only for serverless architectures but for any cloud application or system.

Evolving standards in cloud observability

Changes and evolutions in the cloud computing paradigm are constant, and ensuring robust monitoring and observability requires that development in observability standards continually adapts to new technologies and methodologies. The complexity of modern cloud environments requires more advanced and comprehensive observability practices to ensure the reliability, performance, and security of applications and systems, based on emerging guidelines and frameworks that shape the future of cloud monitoring.

Evolving cloud observability standards are essential for several reasons:

- **Coherence**: Standardized practices ensure consistency in monitoring across different platforms and services
- **Interoperability**: Standards facilitate the integration of various tools and technologies, enabling seamless data collection and analysis
- **Best Practices**: The emerging guidelines summarize industry best practices and help organizations implement effective monitoring strategies
- **Preparation for the Future**: Staying open to evolving standards ensures that observability practices remain relevant and able to address future challenges

For these reasons, it is essential to align monitoring and observability practices with industry benchmarks and innovations. Here, we will explore the main observability standards:

- **OpenTelemetry**: One of the most dynamic advances in observability is the development of OpenTelemetry. This open source project, a merger of **OpenTracing** and **OpenCensus**, provides a unified standard for collecting telemetry data (traces, metrics, and logs) from cloud-native applications. OpenTelemetry offers a single set of APIs and libraries for instrumenting applications, reducing the complexity of integrating different observability tools.

  One of the most interesting aspects of OpenTelemetry is that it is vendor-neutral, ensuring that telemetry data can be exported to any observability backend, providing flexibility, and preventing vendor lock-in.

  Finally, it should be noted that OpenTelemetry provides extensive coverage of programming languages and platforms, making it a versatile solution for various cloud environments.

- **OpenMetrics**: **Prometheus** is an open source monitoring and alerting toolset that has been instrumental in defining metrics collection standards. Now, Prometheus has inspired the OpenMetrics project, which aims to create a standard for transmitting metrics at scale.

  OpenMetrics establishes a form of standardized metrics that guarantee compatibility between different monitoring systems. Additionally, OpenMetrics integrates natively with Prometheus, simplifying the process of importing and exporting metrics data to and from Prometheus.

Finally, the standard has been designed with the scalability of modern cloud environments in mind to support efficient metrics collection and storage.

- **Cloud Native Computing Foundation (CNCF) Observability Framework**: The CNCF continually promotes improvements to the observability framework through best practices and guidelines to achieve end-to-end observability in cloud-native environments. This framework incorporates the perspectives and experiences of different members of the CNCF community. The promoted observability framework stands out for its holistic approach to monitoring cloud applications, emphasizing the importance of monitoring infrastructure, application performance, and user experience. Additionally, the framework advocates interoperability between different observability tools, ensuring seamless data flow and integration.
- **Distributed Tracing Standards**: **Distributed tracing** is a fundamental framework that addresses the most complex aspects of microservices architecture, such as understanding the flow of requests and their state in these architectures. Emerging standards in this area focus on improving the interoperability and usability of tracking tools. For instance, we have the **W3C Tracking Context** standard, which defines a common format for propagating tracking information between different services, improving tracking improvement and analysis.
- **Observability as Code (OaC)**: Just as **Infrastructure as Code** (**IaC**) has emerged as a concept for infrastructure provisioning and management, the concept of OaC is emerging and encouraging the definition and management of observability configurations using code. This approach means that observability configurations are defined in code, so organizations can version control their monitoring configurations, facilitating rapid rollback and auditing of changes if necessary.

  Additionally, code-based adjustments can be automated and deployed at scale, making deployment consistent and repeatable across all observability tools in all environments. Finally, treating observability as code encourages collaboration between development and operations teams, integrating observability practices into the CI/CD process.

There have also been several developments in these standards, with the following main trends:

- **Improved Security and Compliance**: Observability standards are increasingly incorporating guidelines to ensure compliance with a broad spectrum of regulations. In recent years, concerns about data security and privacy have increased, and new frameworks include in their definitions increasingly precise guidelines on how to cover these needs. Among the most important things that observability standards promote is the encryption of telemetry data both *in transit* and *at rest* to protect sensitive information. Added to this is the implementation of access controls and strong authentication mechanisms that prevent unauthorized access to observability data. Finally, it is essential to maintain detailed audit records of all activities related to observability that guarantee an investigation of incidents, if required.

- **Real-Time Monitoring and Alerting**: The evolution of observability standards is strongly influenced by the need for real-time information on the performance and state of applications and systems. This is why emerging standards are focusing on low-latency data collection and processing to have analyzed data in just a few minutes. This low latency in collection and processing is directly related to alerts, something else that the standards are defining so that their triggering is not only fast, but also incorporates machine learning and anomaly detection mechanisms that reduce false positive alerts and improve incident response times. Finally, standards are being developed with the principle of scalability to ensure that monitoring and alerting systems can scale efficiently with the growth of cloud environments.
- **Innovations in Observability Tools and Techniques**: AI and machine learning are among the most important innovations that observability standards are incorporating into their foundations to address the constant challenges and complexities that arise from cloud environments. The integration of AI and machine learning within observability tools provides predictive analytics, anomaly detection, and automated root cause analysis, which contribute to significantly improving monitoring efficiency and accuracy.

  Another innovation that is emerging strongly is serverless monitoring. Given the widespread use of serverless architectures, more and more monitoring tools are being adapted to provide visibility into serverless functions and services.

  Finally, microservices architectures are the design standard in recent times for many business applications, which is why distributed tracing is growing as a technique to track requests that flow through this type of architecture.

In this section, we have explained how by adopting evolving cloud observability standards such as OpenTelemetry, OpenMetrics, and the CNCF Observability Guidelines, organizations can ensure consistent, scalable, and secure observability practices. These standards not only improve the ability to detect and resolve problems but also enable proactive monitoring and continuous improvement in observability.

Summary

In this chapter, you have learned how the integration of AI-driven analytics and tools such as Microsoft Copilot for Azure represents a significant advancement in the field of cloud observability. Using these capabilities will allow your organization to apply predictive observability to your environments, thereby proactively addressing issues and optimizing your cloud environments.

With the techniques and tools mentioned for addressing challenges that serverless architectures present in the realm of observability, your organization's cloud development and operations teams can achieve end-to-end visibility into their serverless applications that helps them maintain performance, ensure reliability, and resolve issues quickly, which ultimately leads to a better user experience and more efficient operation of this type of architecture.

Finally, you have learned the importance of monitoring and adopting evolving standards in cloud observability to maintain effective monitoring practices. For this reason, we recommend maintaining awareness of and adopting new developments in emerging standards for those organizations seeking to optimize their cloud operations and provide reliable, high-performance services.

Further reading

Here, you can find the links to expand your knowledge about concepts not covered in this book but referenced in this chapter.

- [1] `make-series` operator: `https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query/make-seriesoperator`.
- [2] `series_decompose_anomalies` operator: `https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query/series-decompose-anomalies-function`.
- [3] `series_decompose_forecast` operator: `https://learn.microsoft.com/en-us/azure/data-explorer/kusto/query/series-decompose-forecast-function`.
- [4] Copilot experiences across the Microsoft Cloud: `https://learn.microsoft.com/en-us/microsoft-cloud/dev/copilot/overview`.
- [5] Creating Generative AI Experiences with the Microsoft Cloud: `https://learn.microsoft.com/en-us/microsoft-cloud/dev/copilot/isv/isv-extensibility-story#pattern-a-create-plugins-to-enhance-an-existing-copilots-functionality`.
- [6] Microsoft Copilot for Azure: `https://learn.microsoft.com/en-us/azure/copilot/example-prompts`.
- [7] Azure Monitor metrics, logs, and alerts using Microsoft Copilot for Azure: `https://learn.microsoft.com/en-us/azure/copilot/get-monitoring-information`.

Appendix

This appendix provides a detailed, hands-on guide to exploring customization options for tailoring pipelines to various monitoring scenarios, ensuring an optimized data flow for improved performance and efficiency.

By using this appendix, you will get a deep understanding of the specific schema for each **Data Collection Rule** (**DCR**) type. It is included in the book for reference purposes. We recommend coming back to this appendix when you need to create a custom DCR that requires a basic file structure to start with.

Technical requirements

To follow along with all the examples in this appendix, we recommend downloading the associated files available in the GitHub repository of the book. You can find the relevant files in the `chapter13` folder (`https://github.com/PacktPublishing/Cloud-Observability-with-Azure-Monitor/tree/main/chapter13`).

The Azure CLI will be used to deploy and configure the resources. You can install it by following the instructions available in the Azure official documentation (`https://learn.microsoft.com/en-us/cli/azure/install-azure-cli`) or using Microsoft Azure Cloud Shell (`https://learn.microsoft.com/en-us/azure/cloud-shell/overview`), which has all the tools required already installed.

Exploring customization options for tailored monitoring

In the *Understanding the data ingestion pipeline in Azure Monitor* section of *Chapter 3*, we mentioned the different types of DCRs that exist. However, we did not go into the specific details of the schema definition of each of them. This section provides a detailed overview of all the schemas by type of data source.

Logs Ingestion API DCR structure

The Logs Ingestion API is used to ingest custom logs into the Log Analytics workspace. The following list contains the relevant parameters for this scenario:

> **Tip**
>
> If you need to submit custom logs through the Logs Ingestion API as described in the *External Telemetry: Integrating Insights from External Sources* section of *Chapter 3*, this is the required schema for your DCRs. You can download the full file from the `chapter13/LogsIngestionAPIDcrStructure.bicep` folder in the book's GitHub repository.

- `location`: The region where the DCR is created must be the same as that of the **Data Collection Endpoint** (**DCE**) and the workspace.
- `dataCollectionEndpointId`: This is the resource ID of the DCE in the format `/subscriptions/<subscription_id>/resourceGroups/<rg-name>/providers/Microsoft.Insights/dataCollectionEndpoints/<dataCollectionEndpointName>`.
- `streamDeclarations`: This contains one or multiple streams that define the structure of the data sent to the ingestion endpoint. The streams must include the `Custom-` prefix in the name. The structure of incoming data does not necessarily have to match the destination table structure, as it is possible to apply a KQL transformation to align them. However, the output of the KQL transformation must match the structure of the destination table and include a `TimeGenerated` column.
- `destinations`: This indicates the destinations to which the data should be sent. In the case of this API, the destination is `logAnalytics`, and it is specified by including `workspaceResourceId` and the workspace name.
- `dataFlows`: This is responsible for matching streams with destinations and applying transformations if necessary. It consists of one or more data flows. Its properties are as follows:
  - `streams`: For each data flow, there can be one or multiple streams if the destination table is the same. If the data flow includes a transformation, two data flows must be created, each containing only one stream. The streams are indicated as declared in `streamDeclarations`.
  - `transformKql`: This is a KQL statement that transforms the structure of the input stream and aligns it with the schema of the destination table. The destination table must always have a `TimeGenerated` column, so the output of the transformation must include it.
  - `destinations`: One of the destinations defined in the previous `destinations` block, indicating where the output data of the transformation is sent.

  - `outputStream`: This indicates which table in the workspace will ingest the output of the KQL transformation. When dealing with a custom table, the nomenclature for `outputStream` is `Custom-<tableName>_CL`, whereas in the case of an Azure table, the nomenclature is `Microsoft-<tableName>`.

Azure Monitor Agent DCR structure (logAnalytics and azureMonitorMetrics as destinations)

Working with both logs and metrics requires the usage of the **Azure Monitor Agent** (**AMA**) data structure format. The following list contains the relevant parameters for this scenario:

> **Tip**
> If you need to submit custom logs or custom metrics using AMA into both Azure Log Analytics and Azure Monitor Metrics, this is the required schema for your DCRs. You can download the full file from the `chapter13/AMADcrStructureLAAMM.bicep` folder in the book's GitHub repository.

- `location`: The region where the DCR is created must be the same as that of the DCE and the workspace.
- `kind`: This indicates the type of platform for the source resources to which the DCR applies. In this case, options include `Linux`, `Windows`, or `All`, the latter being for if it applies to both.
- `streamDeclarations`: This contains one or multiple streams that define the structure of the data sent to the ingestion endpoint. The streams must include the `Custom-` prefix in the name. The structure of incoming data does not necessarily have to match the destination table structure, as it is possible to apply a KQL transformation to align them. However, the output of the KQL transformation must match the structure of the destination table and include a `TimeGenerated` column.
- `dataCollectionEndpointId`: This is optional since AMA does not use DCEs for standard Azure tables, except in cases where data sources such as **Internet Information Services** (**IIS**) logs, custom text logs, and custom JSON logs are collected. The use of a DCE increases network control for ingestion and configuration download endpoints. A higher level of network isolation can be achieved when combined with **Azure Monitor Private Link Scope** (**AMPLS**), but this is beyond the scope of this book. The format of the DCE's resource ID is `/subscriptions/<subscription_id>/resourceGroups/<rg-name>/providers/Microsoft.Insights/dataCollectionEndpoints/<dataCollectionEndpointName>`.

- `dataSources`: This contains one or multiple data sources that define the structure of data for each type of data source sent to the ingestion endpoint. Data sources can be as follows:
  - `logFiles`: Log files in `txt` or `json` format
  - `extensions`: Data sources associated with VM extensions, for example, `ChangeTracking`
  - `performanceCounters`: Performance counters from Windows and Linux VMs
  - `windowsEventLogs`: Events generated by the Windows operating system
  - `syslog`: Events generated by the Linux operating system
  - `iisLogs`: Information from the IIS logged on the local disk of the Windows VM
- `destinations`: This indicates the destinations to which the data should be sent. In the case of this DCR, destinations can be as follows:
  - `logAnalytics` indicating the `workspaceResourceId` value and the workspace name
  - `azureMonitorMetrics` as an additional destination only for `performanceCounters`
- `dataFlows`: This is responsible for matching streams with destinations and applying `transformations` if necessary. It consists of one or more data flows:
  - `streams`: For each data flow, there can be one or multiple streams if the destination table is the same. If the data flow includes a transformation, two data flows must be created, each containing only one stream. The streams are indicated as declared in `streamDeclarations`.
  - `transformKql`: This is a KQL statement that transforms the structure of the input stream and aligns it with the schema of the destination table. The destination table must always have a `TimeGenerated` column, so the output of the transformation must include it.
  - `destinations`: One of the destinations defined in the previous `destinations` block (logAnalytics or azureMonitorMetrics), indicating where the output data of the transformation is sent.
  - `outputStream`: This indicates which table in the workspace will ingest the output of the KQL transformation. When dealing with a custom table, the nomenclature for `outputStream` is `Custom-<tableName>_CL`, whereas in the case of an Azure table, the nomenclature is `Microsoft-<tableName>`. In the case where the stream of the data flow is a data source such as `extensions`, `performanceCounters`, `windowsEventLogs`, `syslog`, or `iisLogs`, and the destination table is defined by design for these data sources, it is not necessary to include `outputStream`.

Important note

To send logs in text or JSON format, custom tables must be created beforehand in the Log Analytics workspace. The same method used in the *Creating a custom table in a Log Analytics workspace* section of *Chapter 3* can be employed for this. Also, the DCR mentioned earlier needs to be associated with AMA following the example in the *Data collection rule associations* section of *Chapter 3*.

On the other hand, one of the strategies detailed in *Chapter 12*, is filtering the events ingested into the Log Analytics workspace. In the case of Windows Event Logs, using XPath queries allows filtering of the necessary events. It is recommended to use Event Viewer in Windows to obtain the specific XPath query, extract the path from the XML format, and include it in the DCR so that AMA only collects the desired events. For example, the following XPath queries filter all critical-level events from the system event log and all successful security events, respectively:

```
'System!*[System[(Level=1)]]'
'Security!*[System[(band(Keywords,90071'9254740992))]]'
```

AMA DCR structure (eventHubsDirect, storageBlobsDirect, and storageTablesDirect as destinations)

The previous two scenarios had a Log Analytics workspace as a destination. However, if you need to forward your custom logs and metrics outside it to a storage account or an event hub, you will need to use the following data structure. The following list contains the relevant parameters for this scenario:

Tip

If you need to submit custom logs or custom metrics using AMA into a storage account or an event hub, this is the required schema for your DCRs. You can download the full file from the `chapter13/AMADcrStructureEHBlobTables.bicep` folder in the book's GitHub repository.

- `location`: The region where the DCR is created must be the same as that of the DCE and the workspace.
- `kind`: In this type of DCR, the value is `AgentDirectToStore`.
- `streamDeclarations`: This contains one or multiple streams that define the structure of the data sent to the ingestion endpoint. Streams must include the `Custom-` prefix in the name.
- `dataSources`: This contains one or multiple data sources that define the structure of the data for each type of data source sent to the ingestion endpoint. Data sources can be as follows:
  - `logFiles`: Log files in `txt` or `json` format
  - `performanceCounters`: Performance counters from Windows and Linux VMs

- `windowsEventLogs`: Events generated by the Windows operating system
- `syslog`: Events generated by the Linux operating system
- `iisLogs`: Information from the IIS logged on to the local disk of the Windows VM

- `destinations`: This indicates the destinations to which the data should be sent. In the case of this DCR, destinations vary depending on the type of data:
  - Windows:
    - Windows Event Logs:
      i. `eventHubsDirect`: Indicating `eventHubResourceId` and the event hub name
      ii. `storageBlobsDirect`: An array with different containers, indicating `storageAccountResourceId`, the storage account name, and `containerName`
      iii. `storageTablesDirect`: An array with different tables, indicating `storageAccountResourceId`, the storage account name, and `tableName`
    - Performance counters:
      i. `eventHubsDirect`: Indicating `eventHubResourceId` and the event hub name
      ii. `storageBlobsDirect`: An array with different containers, indicating `storageAccountResourceId`, the storage account name, and `containerName`
      iii. `storageTablesDirect`: An array with different tables, indicating `storageAccountResourceId`, the storage account name, and `tableName`
    - IIS logs:
      i. `storageBlobsDirect`: An array with different containers, indicating `storageAccountResourceId`, the storage account name, and `containerName`
    - Text/JSON log files:
      i. `storageBlobsDirect`: An array with different containers, indicating `storageAccountResourceId`, the storage account name, and `containerName`
  - Linux:
    - Syslog:
      i. `eventHubsDirect`: Indicating `eventHubResourceId` and the event hub name
      ii. `storageBlobsDirect`: An array with different containers, indicating `storageAccountResourceId`, the storage account name, and `containerName`
      iii. `storageTablesDirect`: An array with different tables, indicating `storageAccountResourceId`, the storage account name, and `tableName`

    - Performance counters:
        i. `eventHubsDirect`: Indicating `eventHubResourceId` and the event hub name
        ii. `storageBlobsDirect`: An array with different containers, indicating `storageAccountResourceId`, the storage account name, and `containerName`
        iii. `storageTablesDirect`: An array with different tables, indicating `storageAccountResourceId`, the storage account name, and `tableName`
    - Text/JSON log files:
        i. `storageBlobsDirect`: An array with different containers, indicating `storageAccountResourceId`, the storage account name, and `containerName`
- `dataFlows`: This is responsible for matching streams with destinations and applying transformations if necessary. It consists of one or multiple data flows:
    - `streams`: For each data flow, there can be one or multiple streams if the destination is the same. Streams are indicated as declared in `streamDeclarations`.
    - `destinations`: One of the destinations defined in destinations.

> **Important note**
> This type of DCR is only supported for Windows or Linux Azure VMs, and transformations are not supported. For the data to be sent to Event Hubs, Blob Storage, and Table storage, it is necessary for the managed identities of the VMs where AMA runs to be assigned the roles Storage Blob Data Contributor, Storage Table Data Contributor, and Azure Event Hubs Data Sender. Also, note that the preceding DCR must be associated with AMA following the example from the *Data collection rule associations* section of *Chapter 3*.

Event Hub DCR structure

The previous scenario was forwarding the information from AMA into the specified event hub; however, most of the Azure services don't integrate with AMA. If you need to forward the logs to an event hub, you will need to use this data structure. The following list contains the relevant parameters for this scenario:

> **Tip**
> If you need to submit your information directly into an event hub, this is the required schema for your DCRs. You can download the full file from the book's GitHub repository in the `chapter13/EventHubDcrStructure.bicep` folder.

- `location`: The region where the DCR is created must be the same as that of the DCE and the workspace.
- `identity`: This indicates the identity used by the DCR to act as an event hub consumer group. This identity can be a managed identity or a user-assigned managed identity.
- `streamDeclarations`: This contains one or multiple streams that define the structure of the data sent to the ingestion endpoint. Streams must include the `Custom-` prefix in the name. The input structure of the stream is not customizable and must consist of the `TimeGenerated`, `RawData`, and `Properties` columns. The input data structure does not have to match the structure of the destination table, as it is possible to apply a KQL transformation to align them. The output of the KQL transformation must match the structure of the destination table and must have a `TimeGenerated` column.
- `dataCollectionEndpointId`: This is the Resource ID of the DCE with the format `/subscriptions/<subscription_id>/resourceGroups/rg-packt/providers/Microsoft.Insights/dataCollectionEndpoints/<dataCollectionEndpointName>`.
- `dataSources`: This contains the `dataImports` data source type, which specifies the configuration of the event hub.
- `destinations`: This indicates the destinations to which the data should be sent. In the case of this DCR, the destination is `logAnalytics`, indicating `workspaceResourceId` and the workspace name.
- `dataFlows`: This is responsible for matching streams with destinations and applying transformations if necessary. It consists of one or multiple data flows:
  - `streams`: For each data flow, there can be one or multiple streams if the destination table is the same. If the data flow includes a transformation, then two data flows must be created, each containing only one stream. Streams are indicated as declared in `streamDeclarations`.
  - `transformKQL`: This is a KQL statement that will transform the input structure of the stream and align it with the schema of the destination table. The destination table must always have a `TimeGenerated` column, so the output of the transformation must include it.
  - `destinations`: This is one of the destinations defined in the previous `destinations` block, indicating where the output data of the transformation is sent.
  - `outputStream`: This indicates which table in the workspace will ingest the output of the KQL transformation. In the case of a custom table, the nomenclature for `outputStream` is `Custom-<tableName>_CL`, while in the case of a supported Azure table, the nomenclature is `Microsoft-<tableName>`.

Important note

This scenario requires that the managed identity of the event hub or Event Hubs namespace has permission to send events to the DCR and the DCE. To achieve this, the assignment of the custom role or the Azure Event Hubs Data Receiver role is needed. Finally, for events to be ingested from the event hub into the ingestion pipeline, it is necessary to create an association of the DCR with the event hub, as we did in the *Data collection rule associations* section of *Chapter 3* for the AMA, specifying `eventHubResourceId` in the scope. It is important to note that an event hub only supports having one associated DCR.

Log Analytics workspace transformation DCR structure

The last scenario covered in this chapter is on-the-fly data transformation before the monitoring details are stored inside the Log Analytics workspace. These are the relevant properties for this scenario:

Tip

If you need the transformation to happen directly in your Log Analytics workspace before the data is stored, this is the required schema for your DCRs. You can download the full file from the book's GitHub repository in the `chapter13/LogAnalyticsWorkspaceDcrStructure.bicep` folder.

- `location`: The region where the DCR is created must be the same as that of the DCE and the workspace.
- `kind`: In this type of DCR, the value is `WorkspaceTransforms`.
- `destinations`: This indicates the destinations to which the data should be sent. In the case of this DCR, the destination is `logAnalytics`, indicating `workspaceResourceId` and the workspace name.
- `dataFlows`: This is responsible for matching the streams with the destinations and applying transformations if necessary. It is composed of one or more data flows, such as the following:
  - `streams`: For each data flow, there can be one or more streams if the destination table is the same. If the data flow includes a transformation, then two data flows must be created, each containing only one stream. The streams only include supported Azure tables and are indicated as `Microsoft-Table-<tableName>`.
  - `transformKql`: This is a KQL statement that will transform the structure of the input stream. The destination table must always have a `TimeGenerated` column, so the output of the transformation must include it.

- `destinations`: This is one of the destinations defined in the previous `destinations` block, indicating where the output data of the transformation is sent.
- `outputStream`: This indicates which Azure table in the workspace will ingest the output of the KQL transformation. The nomenclature for `outputStream` is `Microsoft-<tableName>`.

Just like we saw in the Logs Ingestion API DCR, it would be ideal if the transformation in the workspace DCR could modify the input data to store it with a structure different from the default structure of the destination Azure table. For this, there are a couple of prerequisites.

First, it is necessary to use the `Tables – Update` API to enable transformations in the destination table, regardless of whether custom columns will be added or not. The custom columns that are added must follow the format `<columnName>_CF`. For example, suppose we want to add the `AuthenticationMethod` custom column to the `SigninLogs` Azure table:

```
curl -X PATCH 'https://management.azure.com/
subscriptions/<subscription_id>/resourceGroups/rg-packt/providers/
Microsoft.OperationalInsights/workspaces/law-packt/tables/<supportedMi
crosoftTableName>?api-version=2022-10-01' \
-H "Authorization: Bearer eyJ0eXAiO...nbrgGJsnVT6Ngz9aDEA8fLkl1_
ljQmRlwA" \
-H 'Content-Type: application/json' \
-d @schema.json
```

Here, `schema.json` is as follows:

```
{
    "properties": {
        "schema": {
            "name": "SigninLogs",
            "columns": [
                {
                    "name": "AuthenticationMethod_CF",
                    "type": "string",
                    "description": "Type of authentication
                     method used.",
                    "isDefaultDisplay": true,
                    "isHidden": false
                }
            ]
        }
    }
}
```

Second, the DCR must be associated with the workspace through a call to the `Workspaces - Update` API:

```
curl -X PATCH 'https://management.azure.com/
subscriptions/<subscription_id>/resourceGroups/rg-packt/
providers/Microsoft.OperationalInsights/workspaces/law-packt?api-
version=2022-10-01' \
-H "Authorization: Bearer eyJ0eXAiO...nbrgGJsnVT6Ngz9aDEA8fLkl1_
ljQmRlwA" \
-H 'Content-Type: application/json' \
-d @schema.json
```

Here, `schema.json` is as follows:

```
{
  "properties": {
    "defaultDataCollectionRuleResourceId": "/subscriptions/
    <subscription_id>/resourceGroups/<rg-name>/providers/
    Microsoft.Insights/dataCollectionRules/
    <dataCollectionRuleName>"
  }
}
```

Important note

It is important to mention that a workspace only supports being associated with a single DCR.

All the aforementioned requirements are necessary in case the configuration of the DCR creation is not done through the Azure portal. If the Azure portal is used to create the DCR through the wizard, both the update of the table schema and the association of the DCR are done automatically.

Wrap-up

This appendix aimed to facilitate the creation of personalized DCRs tailored to your organization's unique requirements. Given the abundance of details associated with each data source, we suggest utilizing this appendix as a reference when crafting new custom DCRs. Doing so streamlines the process of initiating a blank slate equipped with a suite of predefined schemas that can be readily modified and expanded upon to accommodate your specific needs.

Index

A

B

C

D

E

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

packtpub.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 d
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.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:

Observability with Grafana

Rob Chapman, Peter Holmes

ISBN: 978-1-80324-800-4

- Understand fundamentals of observability, logs, metrics, and distributed traces
- Find out how to instrument an application using Grafana and OpenTelemetry
- Collect data and monitor cloud, Linux, and Kubernetes platforms
- Build queries and visualizations using LogQL, PromQL, and TraceQL
- Manage incidents and alerts using AI-powered incident management
- Deploy and monitor CI/CD pipelines to automatically validate the desired results
- Take control of observability costs with powerful in-built features
- Architect and manage an observability platform using Grafana

Implementing GitOps with Kubernetes

Pietro Libro, Artem Lajko

ISBN: 978-1-83588-422-5

- Delve into GitOps methods and best practices used for modern cloud-native environments
- Explore GitOps tools such as GitHub, Argo CD, Flux CD, Helm, and Kustomize
- Automate Kubernetes CI/CD workflows using GitOps and GitHub Actions
- Deploy infrastructure as code using Terraform, OpenTofu, and GitOps
- Automate AWS, Azure, and OpenShift platforms with GitOps
- Understand multitenancy, rolling back deployments, and how to handle stateful applications using GitOps methods
- Implement observability, security, cost optimization, and AI in GitOps practices

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 *Cloud Observability with Azure Monitor*, 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.

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/9781835881187`

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