

# The Art of Site Reliability Engineering (SRE) with Azure

Building and Deploying Applications  
That Endure

—

Unai Huete Beloki

*Foreword by Peter De Tender*

Apress®

# **The Art of Site Reliability Engineering (SRE) with Azure**

**Building and Deploying  
Applications That Endure**

**Unai Huete Beloki**

*Foreword by Peter De Tender*

Apress®

# ***The Art of Site Reliability Engineering (SRE) with Azure: Building and Deploying Applications That Endure***

Unai Huete Beloki  
San Sebastian, Spain

ISBN-13 (pbk): 978-1-4842-8703-3  
<https://doi.org/10.1007/978-1-4842-8704-0>

ISBN-13 (electronic): 978-1-4842-8704-0

Copyright © 2022 by Unai Huete Beloki

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr  
Acquisitions Editor: Smriti Srivastava  
Development Editor: Laura Berendson  
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image by Freepik ([www.freepik.com](http://www.freepik.com))

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail [orders-ny@springer-sbm.com](mailto:orders-ny@springer-sbm.com), or visit [www.springeronline.com](http://www.springeronline.com). Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail [booktranslations@springernature.com](mailto:booktranslations@springernature.com); for reprint, paperback, or audio rights, please e-mail [bookpermissions@springernature.com](mailto:bookpermissions@springernature.com).

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub (<https://github.com/Apress>). For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

# Table of Contents

<b>About the Author .....</b>	<b>vii</b>
<b>About the Technical Reviewer .....</b>	<b>ix</b>
<b>Acknowledgments .....</b>	<b>xi</b>
<b>Foreword .....</b>	<b>xiii</b>
<b>Introduction .....</b>	<b>xv</b>
<b>Chapter 1: The Foundation of Site Reliability Engineering .....</b>	<b>1</b>
The History of Site Reliability Engineering .....	1
Why SRE Is Not DevOps 2.0 .....	3
Identify Best Practices Around SRE.....	6
Automate Everything .....	7
Identify Acceptable Service Levels.....	7
Be Focused on Engineering.....	8
Understand the Challenges of SRE .....	9
Clarify Prerequisites to the Role of SRE.....	12
Summary.....	15
<b>Chapter 2: Service-Level Management Definitions and Acronyms.....</b>	<b>17</b>
Why It's Not a Glossary .....	18
Risk Assessment .....	18
Understand Reliability .....	23
Service-Level Metrics.....	24
Unavailability Metrics .....	30
Postmortems .....	39

TABLE OF CONTENTS

- Toil ..... 40
- Hierarchy of Reliability ..... 41
- Summary..... 43
- Chapter 3: Azure Well-Architected Framework (WAF)..... 45**
- Understanding Well-Architected Framework (WAF) Concepts ..... 46
- WAF – Reliability Building Block..... 49
- Reliability Checklists ..... 58
- Testing Applications for Resiliency ..... 59
- Well-Architected Framework Assessment ..... 62
- Summary..... 68
- Chapter 4: Architecting Resilient Solutions in Azure..... 69**
- What Is Resiliency? ..... 69
- Azure Platform Resiliency..... 71
- Resiliency Based in Numbers ..... 74
- Resiliency on Application Design..... 76
- Mainly Used Components/Platform Features for Resilient Solutions ..... 78
- Resilient Architecture Examples ..... 82
- PaaS Resilient Architecture ..... 84
- Microservices Architecture ..... 86
- Testing Resiliency on Azure ..... 89
- Summary..... 89
- Chapter 5: Automation to Enable SRE with GitHub Actions/Azure DevOps/Azure Automation..... 91**
- Automation for SRE ..... 92
- CI/CD Automation with DevOps ..... 93
- What Is DevOps ..... 93
- Modern Deployment Strategies ..... 122
- Summary..... 142

<b>Chapter 6: Monitoring As the Key to Knowledge</b> .....	<b>143</b>
Operational Awareness .....	144
SLI/SLO/SLA.....	145
Error Budget/Burn Rate .....	147
Observability vs. Monitoring .....	148
Azure Service Health.....	148
Azure Monitor.....	150
Data Sources .....	151
Visualize .....	153
Analyze .....	160
Application Insights.....	167
Azure Monitor Alerts .....	178
[DEMO] Tracking SLI/SLO/SLA Using Application Insights and Log Analytics .....	182
Azure DevOps.....	187
GitHub .....	189
Summary.....	192
<b>Chapter 7: Efficiently Handle Incident Response and Blameless Postmortems</b> .....	<b>193</b>
Incident Response (IR) .....	193
Incident Response Pillars .....	196
Incident Tracking/Detection .....	199
Communication and ChatOps .....	202
Eradication/Remediation .....	204
Measuring Performance.....	211
[DEMO] Incident Response .....	214
Blameless Postmortems .....	218
Best Practices/Tips.....	221
Summary.....	222

TABLE OF CONTENTS

**Chapter 8: Azure Chaos Studio (Preview) and Azure Load Testing (Preview) ..... 223**

- Intro to Chaos Engineering..... 224
  - Chaos Monkey ..... 225
  - Principles of Chaos (Engineering)..... 226
  - Azure Chaos Studio ..... 228
- Load/Performance Testing ..... 261
  - Azure Load Testing ..... 262
- Summary..... 271

**Index..... 273**

# About the Author



**Unai Huete Beloki** is a Microsoft Technical Trainer (MTT) working at Microsoft, based in San Sebastian (Spain).

From February 2017 to July 2020, he worked as a PFE (Premier Field Engineer), offering support and education as a DevOps expert to Microsoft customers all around EMEA, mainly focused on the following technologies: GitHub, Azure DevOps, Azure Cloud Architecture and Monitoring, and Azure AI/Cognitive Services.

Since July 2020, he has worked as a Microsoft Technical Trainer (MTT) on the technologies mentioned above and served as the MTT lead for the AZ-400 DevOps Solutions exam, helping shape the content of the exam/course.

In his free time, he loves traveling, water sports like surfing and spearfishing, and mountain-related activities such as MTB and snowboarding.



# About the Technical Reviewer



**Peter De Tender** has more than 20 years' experience in architecting and deploying Microsoft data center technologies, having started his career on Windows NT4 Server in 1996. Since early 2012, he started shifting to cloud technologies (Microsoft 365, Intune) and quickly moved to the Azure platform, working as a cloud solution architect and trainer, out of his own company. Since September 2019, Peter has worked at Microsoft as part of the prestigious Microsoft Technical Trainer team, providing Azure readiness workshops to larger customers and partners across the globe. Peter recently relocated to Redmond, WA, to continue this role in the West US team. Prior to joining Microsoft, he was an Azure MVP for 5 years and a Microsoft Certified Trainer for more than 12 years. He is still actively involved in the community as a speaker, technical writer, and author. Follow Peter on Twitter at @pdtit and check his technical blog, [www.007fffllearning.com](http://www.007fffllearning.com).

# Acknowledgments

Writing this book has been an amazing journey. I have been talking and giving support to customers around the mentioned topics for more than five years already, but I had never imagined writing my first book about it would be so rewarding (and time consuming of course 😊), but worth it!).

I want to start with thanking my colleague and friend Peter De Tender. As mentioned in the Foreword, Peter was the one who came up with the idea to write a book about SRE on Azure. I was aware of it, as we have been providing DevOps/SRE education to customers (many times together) during the last two years. Due to Visa regulations, he was not allowed to continue and asked me to join this interesting project. He has provided amazing help based on his extensive book-writing and Azure experience.

Finally, I also want to thank my girlfriend and family for the support and encouragement during the last busy months.

# Foreword

Thank you for being interested in the amazing world of Azure and Site Reliability Engineering (SRE). While SRE has been around for more than a decade, most organizations are starting to look into it and adopt it only recently. The main reason for this, as I see it, is because public cloud is out of its infancy and finally seen as an interesting way for running your business-critical workloads, anywhere in the world, where your users or customers need access to it. At the same time, public cloud has often been positioned with a slightly wrong connotation as it would solve all on-premises challenges around security, high availability, and scalability. While those are definitely more than valid reasons to migrate to Azure, there is still a need for properly architecting your business-critical scenarios, deciding on what platform architecture to use such as Infrastructure as a Service or Platform as a Service, relying on serverless and microservices architectures such as Azure Functions, Containers with Kubernetes, etc., where also, once you have it running, you want to guarantee uptime.

To help you with the architectural aspects of Azure cloud onboarding and migration, this book touches on the Microsoft Well-Architected Framework, a set of guidelines on how to accurately map your business requirements with technical requirements and run your applications efficiently. Next, there are a lot of technical hands-on aspects woven into this book as well, based on several years' experience in the field helping customers in deploying and managing Azure-running workloads. Expanding with my passion for cloud automation and DevOps application and infrastructure life cycle management, this book also integrates two amazing new Azure services in preview: Azure Chaos Studio and Azure Load Testing.

Now that you know what the book is about from a technical perspective, allow me to add a more personal note to it. The book you have in front of you was supposed to be my ninth in nine years and my sixth work that's Azure related. I love talking (and writing) about Azure, Azure DevOps, and Azure Architecting in my day-to-day job as a Microsoft Technical Trainer at Microsoft, as well as presenting on the same topics at User Groups and conferences over the last couple of years. Besides the technical aspects of this, I also got the pleasure to travel across the world and meet with amazing customers and partners. It was their input and questions around cloud reliability and resilience

## FOREWORD

that moved me into dedicating my next book to the topic of SRE and Azure. But then I accepted an offer from Microsoft Corp., Redmond, USA, when I was about 40% into the writing process. As per the US Visa regulations, I was not allowed to continue writing this book.

But I also didn't want to scrap it, as the need for writing on the complex topic was definitely there.

Luckily, I have this awesome colleague from within my EMEA Trainer team at Microsoft, Unai Huete Beloki, who has been a friend since he joined the team more than two years ago (he's a great guy, really, although I'm sure his similar passion for Azure DevOps might have something to do with it), bringing in a lot of technical background and expertise from his previous role at Microsoft, where he helped customers across EMEA into using DevOps concepts. When I approached Unai and asked him if he was interested in taking over the project, he didn't hesitate a minute. Even more so, he immediately came back with great suggestions to make the book even more technical, integrating more detailed focus points, extending with his own experiences in both Azure DevOps and GitHub Actions for CI/CD, monitoring, and so much more. Apart from the outline structure and the high-level subject, he turned this into a great read for anyone who's interested in learning more about Site Reliability Engineering, in combination with Azure public cloud, resulting in understanding how to max out the cloud potential for your business-critical workloads. I'm actually happy not too many scribbles and draft chapters remained, as it really is his work.

You should thank Unai when you are reading this book; he did an amazing job. I'm sure you will enjoy the journey.

Peter De Tender  
Microsoft Technical Trainer – West US team, Microsoft Corp.,  
Redmond, USA

# Introduction

This book will help you gain a foundational understanding of SRE and learn its basic concepts and architectural best practices for deploying Azure IaaS, PaaS, and Microservices-based resilient architectures.

Based on Dickerson's hierarchy of reliability, it will cover how to set up the necessary practices by using tools provided by Azure and Microsoft.

The book starts with the basic concepts of SRE operations and developer needs, followed by definitions and acronyms of Service-Level Agreements in a real-world scenario. Moving forward, you will learn architecting resilient IaaS solutions, PaaS solutions, and Microservices architecture in Azure. Here you will go through Azure reference architecture for high-availability storage, networking, and virtual machine computing, describing availability sets/zones and scale sets as the main features. You will learn similar reference architectures for platform services such as App Services with Web Apps and also touch on data solutions like Azure SQL and Azure Cosmos DB.

Next, you will learn automation to enable SRE with Azure DevOps Pipelines and GitHub Actions, along with an understanding of how an open culture around postmortems dramatically helps in optimizing SRE and the overall company culture. Toward the end, you will learn incident management and monitoring practices by making use of Azure Monitor/Log Analytics/Grafana, which forms the foundation of monitoring Azure and Hybrid-running workloads.

As an extra, the book finishes by covering two new testing solutions: Azure Chaos Studio and Azure Load Testing. These solutions will make it easier to test the resilience of your services.

In terms of prerequisites, having some basic understanding of Azure platform will help you understand the concepts covered in the book. A basic understanding of DevOps could help you follow some demos using Azure DevOps and GitHub.

## CHAPTER 1

# The Foundation of Site Reliability Engineering

In this first chapter, I introduce you to the foundation of Site Reliability Engineering (SRE) as a practice, as well as what it means to be or become a Site Reliability Engineer (SRE). Starting from the baselines set out by Google years ago already, it touches on the core characteristics of IT operations and software development. Next, it highlights the importance of collaboration, helping organizations in running business-critical workloads without major downtime.

By the end of this chapter, you should be able to do the following:

- Understand the history of SRE
- Recognize DevOps vs. SRE
- Identify best practices around SRE
- Understand the challenges of SRE
- Clarify prerequisites to the role of SRE

## The History of Site Reliability Engineering

Historically, organizations relied on system administrators (sysadmins) to deploy and manage the data center components, from storage, networking, systems, and security. On the other side, developers were tasked to create the software and focus on the development aspects of it. Deploying the actual application workload was often executed by the system administrators as well.

This often created friction before, during, and after the workload got deployed into production. Even more, when something went wrong (imagine a web application

not running anymore, or a web app front end not being able to connect to a database back end), it was frustrating and typically time-consuming to troubleshoot. Sysadmins approached the problem from a system’s perspective, validating networking, firewalling, system processes, etc. Developers were targeting the outage from a software perspective, validating the code. Sounds familiar, right?

The fact that you are reading this book gives me the impression you are looking for a better way to guarantee uptime of your application workloads. That is the foundation of SRE.

Site Reliability Engineering (SRE), both as the reference to the practice and the job role, brings us back to Ben Treynor Sloss, VP of engineering at Google, as the originator of the term. The basic idea behind SRE is that technical teams who are responsible for operational tasks – and related outages – should treat these as software problems. The reason behind this was that Google’s technical teams were always hiring **engineers** for the job, no matter if they would land in the Operations teams or the Development teams. Google never really worked in terms of system and data center administrators, admins are also called engineers.

Therefore, the basic principle being that dealing with an operational problem is not different from dealing with a software problem also explains the confusion with DevOps. I will explain more in detail later on what I mean by that.

The **Site** in SRE has a double meaning. Originally, when Google was mainly focused on the Google web search engine, [www.google.com](http://www.google.com) (and local domains as well), Site literally referred to making sure that this main page, “*the Google site*,” was always up and running. Years later, however, it should be expanded to “Services.” Within Google, the SRE practice is obviously being used for many more services than the search engine website, but also used for Gmail, Google Office, and so many other services it is running. Looking at SRE outside of Google, it is fair to say it could easily refer to “Services” as well, reflecting on-premises data centers, as well as hybrid or public-cloud services.

---

**Note** An interesting side story I heard why Google engineers wanted to make sure the Google search website was always available is because it was used for so much more than search. What is the first site you try to connect to when you connect to a public Wi-Fi hotspot or want to validate 4G mobile connectivity? If the site would not respond, you would assume it is an issue with the connectivity, not the site itself being down (... *Google is never down, right?*).

---

The last part in the term, **Reliability**, refers to how well an application workload is running from a high-availability perspective. One of the IT-industry misconceptions over the last four decades is that all applications should be always 100% highly available. I personally like the short definition given by Microsoft for SRE (<https://docs.microsoft.com/en-us/azure/site-reliability-engineering/>):

*“Site reliability engineering is an engineering discipline devoted to helping an organization sustainably achieve the appropriate level of reliability in their systems, services, and products.”*

From the definition, we can learn that the core goal of SRE is working toward an *acceptable availability* for the business as a whole (or for a given application workload in particular), which hardly reflects 100% for all workloads. Simply put, your lunch-ordering app website where your employees can order sandwiches will be less critical than the e-commerce website you run toward your customers to order products.

## Why SRE Is Not DevOps 2.0

It is interesting to see how the SRE acronym has gained momentum in the last few years and is viewed as an updated version of DevOps. However, we need to take a few steps back to really understand they are not exactly the same.

As you know, in the meantime, SRE’s primary goal is optimizing reliability and availability of systems and running applications. DevOps, on the other hand, is a methodology allowing organizations to deploy applications in a faster way, resulting in a more optimized application life cycle management.

Ultimately, I would describe **DevOps as a way to realize SRE**. When your application workloads can be deployed using an automated process (pipelines, actions, tasks, jobs, etc., depending on the tooling you use), where both developers teams and operations teams are collaborating toward the same goal, it should help in having better uptime of your application workloads as well. Key here is “should” as it could also have impact on the availability.

We could also reference the sentence included in the first chapter of *The Site Reliability Workbook: Practical Ways to Implement SRE*:

*“Class SRE implements interface DevOps.”*

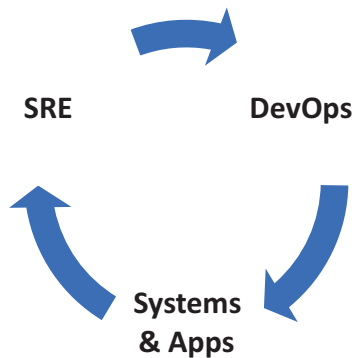
It explains a similar idea with different words. SRE implements the use of DevOps (together with other practices discussed later in this book) to obtain better reliability.



Not too long ago, I had a customer using a scenario where they had an end-to-end pipeline in place to publish a new snippet of code to a web application, across different regions in the world. From the pipeline perspective, the deployments were running fine, picking up source code artifacts, deploying to a validation environment, and eventually landing in the production environment. So 100% satisfaction from a DevOps (pipeline) perspective. However, the hosting provider was right in the middle of a maintenance window, causing downtime of the application in a specific region, only a few seconds after the pipeline published the new code. It was cumbersome to troubleshoot, since the pipeline showed up *green*, and the website was running fine in other regions, which got the same code published from the same pipeline.

This scenario brings me back to the motivation I used earlier, that ultimately, DevOps is a way to realize SRE, but it's not 100% fail-safe. But hey, didn't you learn that SRE is not aiming for 100% availability somewhere?

This somewhat simplified diagram (see Figure 1-1) reflects the base correlation between both methodologies and practices.



**Figure 1-1.** *SRE and DevOps*

- Start with DevOps, to optimize the deployment and release of systems and applications.
- Validate the availability requirements for your systems and applications and aim for the best SRE results possible.
- To realize the best SRE, you need to update/alter your DevOps processes, which in turn results in updated deployments of your systems and apps, which again will result in achieving a better SRE and so on.

What's important here is that SRE is a lot more than a wrong impression from the IT industry (or its consumers) to try and replace DevOps. In fact, it is not even close to trying to do that. The reason why the misconception is there is because the industry is talking a lot more about SRE lately than they do about DevOps (*Maybe we've been talking about DevOps enough? Too much?*), thus giving a false impression, if you ask me, about how the two streams are actually a lot more complementary than conflicting with or replacing one another. Let's take Donovan Brown's definition of DevOps ([www.donovanbrown.com/post/what-is-devops](http://www.donovanbrown.com/post/what-is-devops)):

*"DevOps is the union of people, process, and products to enable continuous delivery of value to our end users."*

If we compare it with the previously mentioned SRE definition, we can clearly see a difference between these two modern operations practices:

- **DevOps** focuses mainly on the **continuous delivery of value**.
- **SRE** focuses on achieving **sustainable reliability for your workloads**.

DevOps will mainly base its practices on Lean-based methodologies and CI/CD practices, whereas SRE will focus more on operational practices for your production workloads. Both definitions will have practices that overlap, but it can clearly be complementary in organizations nowadays.

Another important difference between SRE and DevOps is that DevOps is focused a lot more on cultivating a culture of collaboration across teams; it could be considered a philosophy. Even if we have DevOps engineers (personally, I consider them more DevOps architects) as a role, I consider **DevOps to be a culture** organizations will have to embrace top-down and transform bottom-up. On the other hand, SRE could be seen as a much more technical, engineering-specific process, which is less dependent on an organization's culture.

Does it mean culture is not an important aspect for SRE? Not at all. If we take a look at the typology defined by Westrum for organizational culture (see Table 1-1), we can identify three types: pathological, bureaucratic, and generative.

**Table 1-1.** *Westrum typology*

<b>Pathological (Power Oriented)</b>	<b>Bureaucratic (Rule Oriented)</b>	<b>Generative (Performance Oriented)</b>
Low cooperation	Modest cooperation	High cooperation
Messengers shot	Messengers neglected	Messengers trained
Responsibilities shirked	Narrow responsibilities	Risks are shared
Bridging discouraged	Bridging tolerated	Bridging encouraged
Failure ► scapegoating	Failure ► justice	Failure ► inquiry
Novelty crushed	Novelty problems	Novelty implemented

Taking the example of **blameless postmortems** (covered later in the book), as a practice to learn from mistakes, we can clearly conclude that generative culture environments will be more beneficial for our organization based, for example, on the following points:

- High cooperation/bridging ► better collaboration, breaking silos, cross-functional teams.
- Messengers trained/failure leads to inquiry ► employees will share potential issues as soon as they identify them, messenger is not punished, remove blame, failures lead to questions.
- Risks are shared ► quality, availability, reliability, and security are everyone’s responsibilities.

## Identify Best Practices Around SRE

It’s a bit scary to write about best practices, knowing there is a full chapter following this one, which touches on the core terminologies and definitions that make up Site Reliability Engineering. Think of this section as my way to influence you *feeling happy* about implementing SRE in your organization, as well as *motivating* you to read on in this book. What better way than sharing best practices to do this, right?

## Automate Everything

While it's not the ultimate goal of implementing an SRE practice, the best recommendation I can start with is moving to automation as much as possible. Knowing the foundation of SRE is maximizing the *acceptable resiliency* for a workload; it means you need to automate. Going back to the first paragraph in the history of SRE, where I talked about huge efforts being put in by sysadmins and developers for troubleshooting an incident, it mainly relates to the fact that troubleshooting and analyzing an incident typically requires manual labor. And manual labor is time-intensive and thus expensive.

What if you could rely on automation to deploy your application in a test environment, rely on automation to run tests in the test environment, integrate automation to redeploy to a production environment, and rely on automation to monitor and manage the reliability of your production workloads? And last, once more bring in automation to mitigate an outage in case of an incident? Sounds too good to be true? Hold on tight, it is what we cover in later chapters using Azure Automation, Azure DevOps, and GitHub Actions.

In the domain of SRE, doing any kind of work or performing a task relying on a manual approach is known as *toil*. So we could summarize this topic by saying that another goal of SRE is avoiding toil. (I'll explain more on this in the next chapter as part of the SRE terminology details).

## Identify Acceptable Service Levels

Knowing that I will discuss a lot more around Service Levels in the next chapter, together with several other important terminologies in an SRE world, it's crucial to understand your SRE implementation can only be successful if you identify *acceptable* Service Levels. Important here is the word "acceptable." It should already be clear from the opening paragraphs that aiming for 100% availability all the time and for all workloads doesn't make any sense. It never did, actually. What's more important, however, is mapping out what the different Service Levels are for the different services, systems, and application workloads your organization is running. Just coming up with a generic number still won't do it. It will take a decent amount of time and effort to discuss this topic with the business stakeholders, both technical (IT teams) and nontechnical.

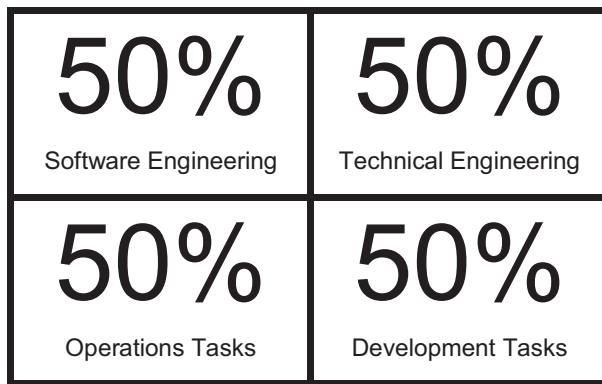
I also refer to the rather generic term "Service Levels," where you are probably thinking in terms of Service-Level Agreements - SLA. Although it is probably the most common and well-known indicator in the industry, it is actually the less technical one.

It is nothing more than a reference number from service providers to compensate you financially when the SLA is not met. In reality, the actual technical Service Level (the “real” availability) of the service might actually be even less than the foreseeable SLA.

## Be Focused on Engineering

As is clear from the SRE definition, having a focus on engineering is the key component to integrating a successful SRE practice. What this means in real life is that a successful SRE team has a good mix of software engineers (most would call them developers) as well as technical engineers (senior system administrators with an affinity for engineering). In the early Google SRE days, the technical engineers were mainly UNIX system administrators with an expertise in networking and operating systems but also were able to understand and write code. But not as much as being a full-time developer. Having the developer skills as a technical engineer is beneficial, since it allows you to integrate the *automate everything* we mentioned earlier in a faster and better way.

While there can be some deviation obviously, since there are so many dependencies when forming an SRE team, Google SRE guidance always recommends having a *double 50/50* guidance as the rule of thumb (see Figure 1-2).



**Figure 1-2.** Google guidance

- 50% of the team are software engineers; 50% are technical engineers.
- 50% of the team’s effort is spent on “Ops” work; 50% is spent on “Dev” tasks.

# Understand the Challenges of SRE

Many organizations are willing to adopt SRE, but they have no clear view on where to start, what the integration flow could look like, and when you can start seeing results.

I mentioned already that SRE comes with several dependencies, like pretty much everything else we have and discuss in the IT industry.

The following are a set of sample starting questions I always ask customers who are looking into embracing SRE for their workloads:

- Where are the workloads running (on-premises, hybrid, public cloud)?
- What is the current level of availability seen, and what is the expected one?
- What is the current practice for developing, testing, deploying, and managing applications and systems?
- What level of automation is already in place?
- Are you using DevOps-related planning methodologies already? Which one? (Agile, Scrum, Kanban, etc.)?
- What is your monitoring model like today?
- How do you manage incidents end to end?

So depending on the answers and the maturity level expressed, this might open up a lot more challenges to tackle first, or make moving up to an SRE integration a rather straightforward process.

Another challenge I've seen in the field is that every organization is rather unique. What works for some might not work for others. Expanding the list of questions mentioned previously and having a “meet-and-greet” moment to learn as much as possible about the current way(s) of working is probably worthwhile.

Implementing SRE is expensive. Whether you read this as a statement or as a question, you probably understand there is a cost related to the learning curve of integrating SRE into your environment. It starts with really understanding what SRE is about, taking the time to build up your inventory of systems and applications, and having repeatable and continuous conversations and meetings with the business to

gather as much information as possible. Once you have all that mapped out, the real work can start. How do you decide on the tool to use for automation, how do you train your existing technical teams to become ready, how large a good SRE team should be, etc.

On the flip side of cost, I hope I don't have to convince anyone in the IT industry that facing incidents and outages is awfully expensive as well. I remember seeing numbers by Gartner, going into the \$300,000 range for an hour downtime (in an enterprise environment with a business-critical workload). As I mentioned earlier, it obviously all depends. But to me, based on the enterprise-customer space I typically work in, this number feels fair and sometimes a bit underrated even to be honest. If you think of organizations such as Amazon e-commerce, Netflix, Spotify, and so many others, being active 24/7 on the Internet for reselling services and products for a few million an hour... yeah, the numbers can go up (or down) quite fast. The trade-off comes in where you have to estimate the cost for building an SRE team, what I talked about in the previous paragraph, and how it relates to the cost of an outage and if the business is willing to take that risk.

An organization will probably be more successful in adopting SRE for a specific workload, or for a specific department, as a starting point. Build up best practices and guidance and gain experience in a smaller capacity before expanding to the rest of the organization or the rest of the business-critical workloads you are running. Coming in with the big bulldozers and smashing down the present operations and development cycles and totally overhauling them with a new practice definitely won't work. I've seen organizations trying to do that and failing miserably. Also, it's immediately breaking down that (DevOps-influenced) culture, which is so important to make all this work. Try to position SRE team members as a smaller, more focused unit within the largest IT team, but not as a SWAT team like exclusivity. I personally don't like the idea that SREs are "better" than other team members, like sysadmins or developers. That's what I like so much about the original SRE approach from Google, saying that everybody is an engineer, everybody is equal, and there are no privileges. Because in the end, you all need to work together, you are all equal, and when a major incident occurs, you want everyone to look into the same direction - working toward fixing the incident. And don't forget about the blameless postmortems. No one is better than anybody else; no one is to blame, besides the whole SRE team, if something goes wrong.

Another challenge I've seen, which honestly was the biggest driver for me to write this book, is how unknown SRE still is in the field. Some of the confusing topics were already touched on earlier, especially the DevOps-SRE relationship (or lack thereof), but outside of that, there is a huge shortage on clear documentation, clear guidance, training material, best practices, and the like on what it takes to implement Site Reliability Engineering. A misconception is – since Google came up with this – that SRE is only relevant to large-scale, multigeography-deployed workloads. But I don't agree with this. Any organization that takes ownership of running its own data center or controlling its own hybrid or cloud-running services can benefit from SRE.

From what I have seen so far, the biggest SRE success comes when there is overall support and sponsorship from the whole organization, starting from the top C-level management, all the way down to the technical teams and application owners, extended with streamlined communication about processes, modes of operation, and setting clear expectations. This all sounds easy enough in a paragraph but is much harder to realize than one would think. That's why so many organizations are afraid of looking into SRE in the first place.

Touching on the challenge of monitoring already, hence why we dedicate a full chapter on that at the end of this book, it's always a surprising one for a lot of organizations. Simply put, you cannot manage what you do not monitor. While most organizations have monitoring in place, not all monitoring is equal. Monitoring is more than watching charts with CPU, memory, and disk load. One needs to work toward a 360-degree view, covering everything from the lowest level of the data center, all the way up to the end users connecting to the application. It is no longer acceptable to validate if the systems are up, but you should challenge your monitoring baseline to provide a services and services-level overview of what's going on. And always, you should be able to detect the problem before the end user reports it. So a big responsibility of implementing SRE will be around providing an end-to-end monitoring service to your business stakeholders.

The last challenge I'm throwing out here is the lack of incidents, which means there is no need for an SRE team. Yes, it does exist; organizations have been implementing their systems and applications so well that they are hardly facing downtime. This could actually bring in a false feeling of comfort, which could jeopardize mitigating an incident when it is eventually occurring, since the team is not prepared for handling the incident. Think of *acts-by-surprise* like the pandemic, an unexpected natural cause, a destructive



operation by an unhappy employee, etc. Even when your processes are that smooth and you never really had to fight fires, always make sure your team is prepared.

I got one last example, being the opposite of the previous one. What if organizations are that focused on hardly implementing changes, they don't have a risk in causing outages. And probably so many other organizational examples that fit perfectly in between both. No matter how you look at it, *any* organization can and will benefit from SRE if implemented correctly. The other extreme is facing the challenges during a major migration shift, which might be only every couple of years, when replacing servers or migrating data centers. Those are also typically the more traditional, super-legacy environments we know about, also avoiding hybrid or public cloud environment integration or extension as much as possible, all that to avoid "risk." With the focus of this book being on how to properly integrate and achieve SRE in Azure, I'm not sure if any of those organizations will go through this book in the first place.

## Clarify Prerequisites to the Role of SRE

In this last section of this introductory chapter, I would like to highlight a bit more what it takes to become, or be, an SRE, in a sense from the job role perspective, answering questions such as

- What does it take to become a Site Reliability Engineer?
- How many years of experience, and what experience exactly, are required?
- I'm a 100% developer, can I become an SRE?
- I'm a 100% sysadmin, can I become an SRE?
- My organization is not using cloud. Do we need an SRE?
- We think we have a solid DevOps methodology in place. Why should we turn into becoming SREs?
- I haven't done anything with DevOps. Can I become an SRE?

A Site Reliability Engineer switches caps all the time. At least 50% of an SRE's time should be spent on developer tasks, while the other 50% should be spent on operations and incidents work. Apart from the expertise-level technical skills in both domains, an SRE must have outstanding communication skills as well.

A good estimate for me is any person who mainly has a developer background, with notions of systems and networking – or an experienced system administrator who understands enough about software development. The expertise level is hard to define, but I would say that it takes you at least five years in either role before you should even consider moving into an SRE position. Please don't take this negatively; don't feel offended if you are currently working in the SRE field and not having a five-year experience. While ideally, an SRE team does not make a distinction in the work the whole team is doing, I've seen some specialization or focus domains within SRE teams. Taking containers and Kubernetes as a popular target environment for SRE, it is a rather locked-down scenario and touching on one specific architecture, while in reality, an organization is probably using so many more services, systems, and applications outside of the Kubernetes environment. I know a few individuals who are excellent in their role as an SRE for the Kubernetes clusters the customer is running, but they don't touch on the network side of things, they don't run operations against the identity solution in place, etc., which is totally fine in itself, since again, SRE is a team sport.

I'm confident a lot of you are interested in this book because you want to move into an SRE role within your organization or elsewhere and learn what it takes to get there.

What I brought up earlier already is that there is no clear path nor a clear readiness trajectory on how to “become a Site Reliability Engineer.” That's what makes the role and responsibilities of an SRE so diverse.

In the core, an SRE should feel super-comfortable in the systems world, understanding operating systems, TCP/IP networking, routing, firewalling, DNS, and the like as a starting point. Next, being familiar with at least one development language (Python, Java, C#, etc.) is definitely beneficial. Having a good understanding of the data center or cloud environment your organization is using is also key. Know how servers are communicating with each other, know how identity and governance are arranged, and identify incoming and outgoing communications toward and from the prime-running workloads. Understand the full application workload topology, end to end. Recognize the characteristics of scalability, performance, and high availability.

Preferably, you have built up experience in fixing incidents. I'm still convinced that the best way to learn is being with your boots on the ground, or maybe in the mud. If you never experienced the stress during an outage, the C-level executives asking every couple of minutes how much longer the workload will be down while you are doing your utmost best to get it up and running again, etc., you won't have the bandwidth or muscle to know how to respond during this kind of situation.

Taking it further, an SRE should have a good understanding of monitoring and observability, both from the general concepts of what logging, metrics, and tracing are about, but also a more in-depth understanding of the monitoring solutions you have in place within your organization, both running on-premises or in a cloud service. In the end, all systems and application workloads are connected at some point. So knowing where to go for which component could be crucial during the hours of the fire.

Lastly, become familiar (or an expert) with the DevOps practices, as well as the DevOps tooling being used within your organization. Know how to run automation pipelines to deploy workloads to different environments, and understand how approval processes are put in place, what is the fallback strategy when a deployment fails, etc.

Apart from the technical skills listed up here, allow me to go back to the more *soft skills side* of the SRE role. Being a good communicator is key, which I think is obvious. If you are locking yourself up behind a screen, hammering buttons during an outage to try and fix it, but you are not communicating what you are doing, it will only increase the stress level within the team and outside. An ideal SRE is self-confident (although not too confident) and well-organized and is used to handle mission-critical workloads (and corresponding outages). It's like being the secretary (organizational skills), the ER medical doctor (emergency), the firefighter (emergency with risk), and the nurse (calm) all in one person at the same time.

Lastly, having good presentation skills is definitely a benefit. As an experienced SRE, you typically need to present regular feedback to your teams (standup or retro calls), to the business stakeholders (crisis meetings and postmortems), as well as to "outsiders." This could be at a conference or symposium around SRE and could be at meetings with like-minded businesses or technical teams, where you are presenting your success as an SRE.

## Summary

In this chapter, I introduced you to Site Reliability Engineering as a practice, as well as a role. Starting from positioning some of the SRE history, you learned about the challenges and benefits of integrating SRE into your organization. We briefly touched on the confusing aspects of how SRE is different from DevOps, but not a replacement of it. Lastly, I tried to list some of the characteristics and prerequisites skills that make a good Site Reliability Engineer.

In the next chapter, we will spend a lot of time clarifying a lot of the typical and crucial terminology used within the Site Reliability Engineering domain.

## CHAPTER 3

# Azure Well-Architected Framework (WAF)

The two previous chapters primarily introduced you to the concepts of Site Reliability Engineering (SRE) and service-level definitions. Although this book is targeting how to apply and achieve SRE in an Azure environment, most of the information shared in the previous chapters is applicable to any data center scenario, whether running on Azure, Amazon AWS, Google GCP, on-premises, or hybrid. The following chapters will reiterate those concepts whenever needed.

From this chapter on, the focus on Azure will become much more apparent, starting with Azure Well-Architected Framework (WAF), a set of best practices from the Microsoft Azure Architecting teams, helping customers and partners in the journey and challenges of what it takes to establish a rock-solid baseline of Azure Services, and optimizing the application and service-layer workloads they are running on the platform.

While I touch on a lot of technical details in this chapter, it won't be that "hands-on" yet, but rather explaining the concepts from a "how-to-design" perspective. It's only in the following chapters that I will guide you through the actual scenarios, step by step, and a lot of examples on how to actually deploy resources, outline architectures, and integrate DevOps CI/CD pipelines as automation, and I will close with monitoring and observability.

By going through this chapter, you will learn the following:

- Understand Well-Architected Framework (WAF) concepts
- Recognize the Well-Architected Framework (WAF) building blocks
- Perform an assessment of an existing Azure environment using WAF
- Analyze a Well-Architected Framework (WAF) assessment and optimize Azure environments

# Understanding Well-Architected Framework (WAF) Concepts

The Azure Well-Architected Framework provides Azure customers with a set of guidelines and best practices on how to implement their Azure workloads “by the book.”

The framework focuses on five core objectives, which should be part of any proper Azure architectural design, no matter what workload or service will be deployed on the platform:

- Reliability
- Security
- Cost
- Operations
- Performance

The mindset behind the WAF is that it becomes the center of an Azure deployment plan, taking each of the objectives into account. Once all points of each objective have been touched on in the cloud solution architectural design, you are ready to deploy. Once the solution is deployed, the work is not done, however, as you can reapply the guidance and best practices of WAF for both new deployments, as well as already running workloads on Azure.

Ultimately, within your organization, you want to provide value to your business. The money the organization spends on Azure consumption to run their business workloads (critical and noncritical such as dev/test scenarios and disaster/recovery environments) is better well-spent. That’s the prime goal of the Well-Architected Framework, maxing out the characteristics and potential of each of the core five objectives (see Figure 3-1) to provide the maximum value back to the business.



**Figure 3-1.** WAF objectives

- **Reliability:** How healthy is your system running, and how fast can it recover from a failure?
- **Security:** Integrating threat protection to guarantee the security of your workload
- **Cost:** Managing the cost aspect without breaking down the quality of the architecture, but also without overspending
- **Operations:** Management tasks to keep systems running as healthy as possible
- **Performance:** Can your systems flexibly adapt to changes in behavior such as peak load and idle times without compromising the quality of the application?

Each of these WAF building blocks is equally important in the overall scope of integrating your Azure environments. However, given the SRE subject we emphasize in this chapter and the book overall, there will obviously be more focus on the reliability component. This chapter will cover the reliability topic in more conceptual way (what), while Chapter 4 covers how to implement it with Azure platform features (how).

Looking at the diagram, it seems like a linear flow from each of these objectives, where they are equally important. I guess that makes sense to a certain extent. Each objective is definitely as important as any other one. However, based on, or depending

on the workload type, the Azure resource as well as both technical and business requirements, the weight of each of the objectives might change.

For example, in the early days of onboarding a workload to Azure, organizations might be OK to see a higher consumption spent. There are additional costs in play for setting up a proof-of-concept (POC) environment; your DevOps teams might need more time to run multiple deployment cycles in a dev/test or staging environment before they land into production.

Once you have mapped your business case against each of the five objective categories (Microsoft calls them pillars), you most probably also need to identify the trade-offs. This might sound weird as it almost feels like I'm telling you to sacrifice certain objectives in favor of others. While not 100% true, it is actually what I see happening in a lot of organizations, and I have already seen this over the course of my career for both on-premises and cloud-based organizations.

Remember, I used the word "ultimately" in an earlier paragraph. In the most ideal and perfect world, you would emphasize each of these objectives in an equal way. Budget control is as important as reliability, as well as security and performance. This will all lead to operational excellence in the end. However, applications, systems, and, hey, business objectives overall are not always that clear. As a cloud solution architect, you can come up with the nicest, most-robust, best-performing architectural design, but if the business is not willing to pay for it, it's "bye-bye" design (or partial design), right?

Or what if there are unknown factors in place that neither you as the technical cloud team nor the business itself could foresee? One example coming to mind here is the COVID-19 pandemic that hit all continents in early March 2020. In several countries across Europe, physical stores, restaurants, bars, and other public places had to shut down. Literally overnight. Employees couldn't go to the office anymore, as they were not allowed to get to the office. Schools were closed for weeks (sometimes months), brick-and-mortar shops and stores were forced to close, etc. It's also where I remember seeing a huge cloud-adoption wave happening, resulting in a performance decrease of the Azure West Europe region, as well as other Azure regions quickly following, since organizations tried deploying their needed Azure resources somewhere else, or were executing disaster/recovery failovers across regions. Organizations were quickly migrating workloads to cloud, deploying new workload scenarios such as Virtual Desktop solutions, and building out Azure VPN to allow sysadmins and developers to still be able to connect to systems, where before it was going through a site-to-site or ExpressRoute tunnel from the corporate headquarters or branch offices, where now you were forced to work from home.



That's where a lot of organizations, including Microsoft as the Azure service provider, had to agree on some trade-offs to make the workloads running on Azure. Even now, two years after the start of the pandemic, I'm still hearing customers saying their newly deployed workloads are not 100% highly available yet in cloud, because they need to focus on other priorities such as security, or meet performance demand as a higher priority. Other teams are struggling with readiness, where they had to ramp up on Azure skills that fast; it was almost impossible to keep up with all the different services, architectural options, and management aspects that are part of running a workload on Azure. (It's also another motivator for me to write this book, helping you where possible.)

## WAF – Reliability Building Block

The **reliability building block** is the one that's closest to the overall topic of this book, since it extends the concept of Site Reliability Engineering. Remember, though, there are a lot of dynamics in place to decide which of the objectives are more important than others; and *ultimately*, they would all be equally important in the overall designing of your Azure Architecture designs.

Deploying reliable workloads involves the following concepts:

- On-premises architecture is (way) different from cloud architectures.
- Cloud platforms are not 100% reliable.
- Observability is key.
- DevOps and automation.
- Self-remediation of workloads.

When an application workload is **reliable**, it means it is running healthy, with no, or minimal, downtime. Sometimes, this is also mixed with **resilient**. However, it's not exactly the same. **Resiliency** typically refers to how fast a workload can be recovered in case of an outage. The more resilient, the shorter the recovery time. However, the more reliable, the lesser the downtime, which means you don't have to worry that much about resiliency. See, slightly different meaning, but crucially important for any workload.

Both will heavily determine the success factor of your SRE processes as well as the success of the SRE team in general. Chapter 4 will focus on showing some examples to achieve reliability/resiliency. Other chapters like Chapter 6 will also give some tips on how incident response practices can be applied for auto-recovery.

## On-Premises Is (Way) Different Than Cloud Architecture

Establishing a reliable cloud architecture is different from building a similar reliable architecture in your own on-premises data centers. What organizations have been doing over the last 40 years was typically investing in redundant physical components, starting from the physical storage that was typically equipped with intelligent storage replication features built in on the physical layer (anyone remembers disk mirror, stripe sets, RAID sets, etc.), physical servers with redundant power supplies, hot-swappable memory, and the like. From a software perspective, business-critical applications were deployed in a *cluster*, typically reflecting two or more servers running the same application, in an active/passive or active/active scenario.

Hypervisor solutions such as VMware and Microsoft Hyper-V were a welcome technology around 2003, where the IT industry noticed a shift from investing in redundant physical data center components into a more logical highly available software layer. Technologies such as VMWare vMotion and Hyper-V Live Migration – by which you could easily move your virtual machine workload from one physical host to another with minimal downtime – contributed a lot to how organizations looked into high availability and disaster recovery.

The preceding slide (see Figure 3-2) highlights what a typical data center disaster/recovery architecture looked like around 2008. One might think a lot has changed in 13 years, but actually it still looks rather familiar I’m sure to a lot of on-premises data center admins.

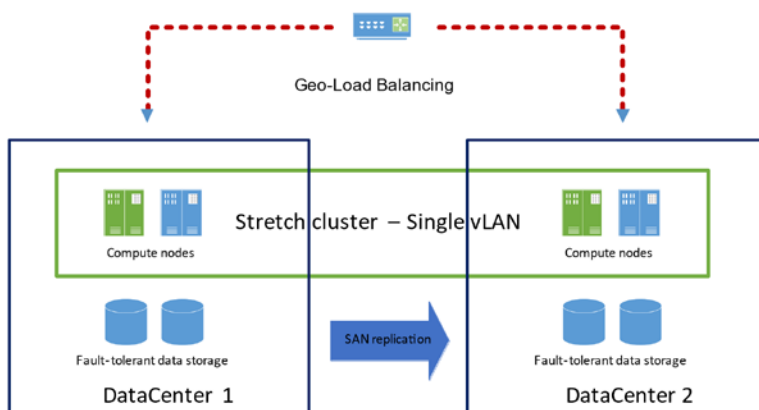
## “Familiar” DR Solutions

- Storage replication
- Stretch cluster using single vLAN with single address space
- Active / passive
- Dynamic incoming connection failover

Effective, although expensive to deploy and maintain

Normally tied to hardware vendor specific technologies

Difficult to perform partial failovers



**Figure 3-2.** High-availability data center architecture with disaster recovery solutions

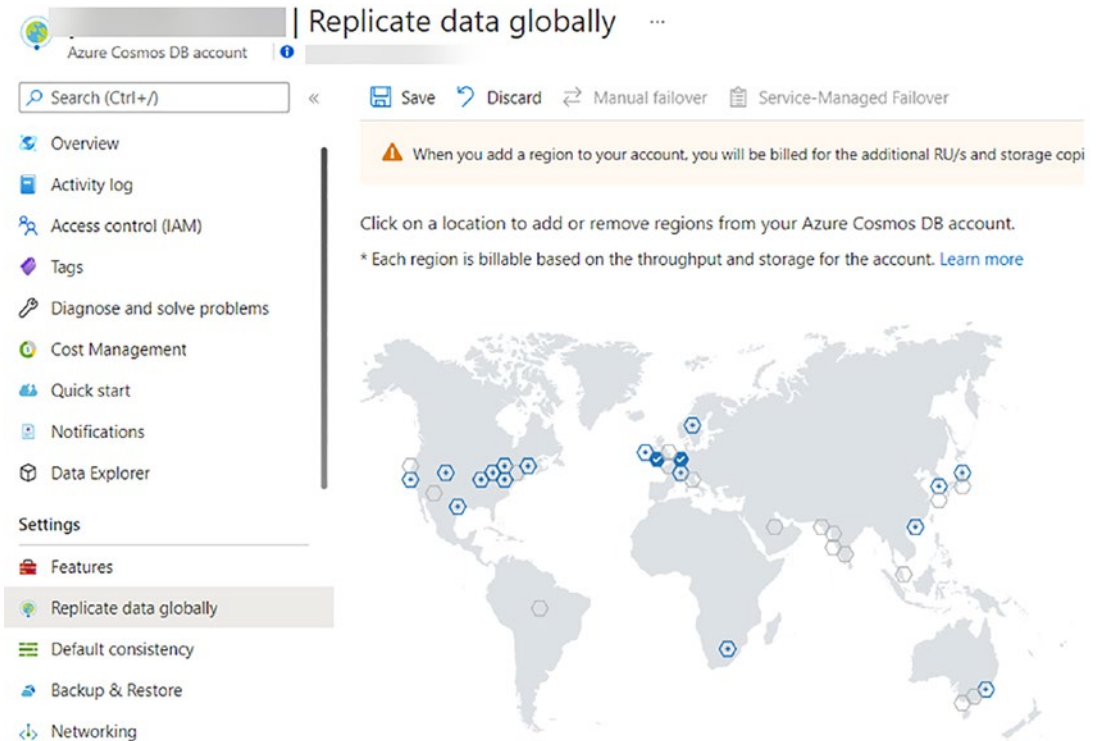
Switching to typical reliable cloud architectures today – which will be the core of what will be explained in the next chapter – is partly still identical, depending on whether you are using Infrastructure as a Service (IAAS) or Platform as a Service (PAAS). In the case of the first, from an Azure perspective, Microsoft takes ownership of providing you the redundant, physical data center components, up to the hypervisor (Hyper-V for Azure Virtual Machines), where you as the customer are hugely responsible for keeping your systems running, making the workloads highly available and redundant, and integrating backup and disaster/recovery as part of the solutioning, to name just a few core aspects. If you look at Platform Services such as Azure App Service, Azure SQL Database, Azure Cosmos DB, Azure Service Bus, and so many others, Microsoft is also taking responsibility for hosting the actual underlying services’ layer, such as the Web App services, the SQL database server instances, and the like. For you as the customer, the responsibility comes in to deploy the actual application layer (web app source code or container), as well as the database layer (the actual database tables and content).

Furthermore, cloud environments are more dynamic than on-premises ones. In on-premises solutions, as a sysadmin, you could connect through remote desktop to the machines and diagnose possible issues by checking logs/events/performance metrics. On the contrary, in a cloud environment, especially a PaaS offering-based environment, the VM/Server instances used in the back end may change anytime; troubleshooting

is not based on connecting to the underlying infrastructure (not possible); you need to leverage tools like Application Insights, Log Analytics, and Metric Explorer to export and analyze those logs/metrics instead (covered in Chapter 6).

## Cloud Is Not 100% Highly Available

A big mind-shift that I always emphasize to customers is understanding and accepting that cloud platforms are **not 100% highly available** (although the marketing pitch is typically telling a different story sometimes). So instead of focusing on building the ultimate high availability, you should shift to minimizing damage in case of an outage. From a high-level perspective, this means relying on the redundant architectural components on all levels possible. This would start with deploying your workloads in at least two separate Azure regions, leveraging virtual machine high-availability architectures such as availability sets and zones (see the next chapter), enabling Azure SQL Failover Groups, and configuring Cosmos DB multiregion database replication (see Figure 3-3), to name a few out-of-the-box capabilities for different Azure platform capabilities.



**Figure 3-3.** Configuration of Cosmos DB global data replication

Following on the same mindset of cloud (Azure, but also Amazon AWS, Google GCP, and other public cloud providers) not being 100% highly available, is where you need to validate the contractual SLAs (Service-Level Agreements) for the different services offered, as briefly touched on already in the previous chapters.

Some examples:

- **Azure App Services:** Microsoft guarantees that Apps running in a customer subscription will be available 99.95% of the time. No SLA is provided for Apps under either the Free or Shared tiers.
- **Azure SQL Database:** Offers multiple SLAs to choose from, depending on the business criticality of your back-end tier. Azure SQL Database Business Critical or Premium tiers configured as Zone Redundant Deployments have an availability guarantee of at least 99.995%, where the non-Zone Redundant Deployments have an availability guarantee of at least 99.99%.

- **Azure Kubernetes Service:** For customers who have purchased an Azure Kubernetes Service (AKS) Uptime SLA, Microsoft guarantees uptime of 99.95% for the Kubernetes API Server for AKS Clusters that use Azure Availability Zones and 99.9% for AKS Clusters that do not use Azure Availability Zones. The availability of the agent nodes in your AKS Cluster is covered by the Virtual Machines SLA.

See <https://azure.microsoft.com/en-us/support/legal/sla/> for all Azure services and their corresponding Service-Level Agreements.

So the key takeaway here is, as an Azure Cloud Solution architect, you need to know and understand the different SLA implications in order to choose the correct Azure platform target to deploy. As an SRE, you will need to work together with cloud architects to emphasize the importance these choices have on the reliability of the solutions. Equally important is understanding and knowing the business uptime requirements from your organization before you can map them with the Azure service of choice. A business-critical e-commerce application requiring an SLA of 99.999% requires a different collection and architecture of Azure services than a less-business-critical HR application, requiring an SLA of 95% for example.

Chapter 4 will provide a lot more technical insights on several common Azure services, both Infrastructure as a Service (IAAS) and Platform as a Service (PAAS), and how to design them from a reliable state perspective using Azure Product Team Reference Architectures as the main guideline, together with my own observations from designing, architecting, and deploying Azure environments at customers over the last few years.

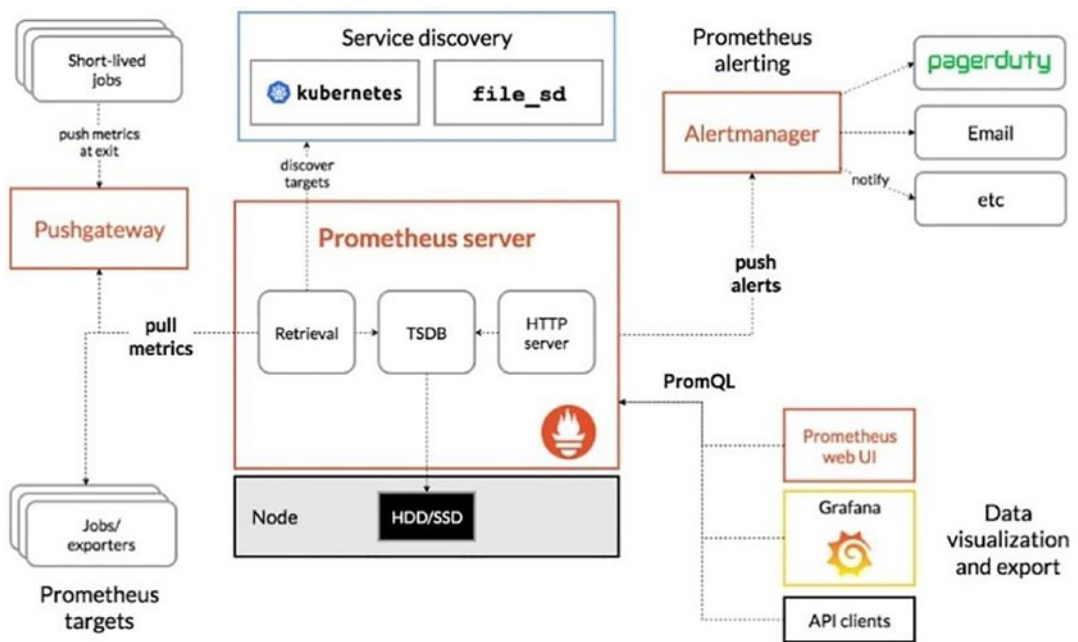
And remember, don't forget about the decision implication trade-offs around cost, security, and performance as well when considering reliability. A more reliable topology typically entails a more complex architecture, spread across different Azure regions. More services within the topology also mean more potential security threats and risks to take into account, more components that can fail, latency might occur in this kind of architecture more frequently than in a less complex scenario, etc.

## Observability Is Key

Apart from exploring a redundant cloud services architecture, following the outlined SLAs from the platform, together with the required SLAs from the business, there are other tactics needed that will optimize reliability. One of them is **observability**, sometimes referred to as monitoring in a more traditional context (difference explained in Chapter 6).

Monitoring is what gets covered in Chapter 6 later on, but in short, it helps IT Operations teams to not only identify how healthy a workload is running (or not running) but also how outages can be predicted or foreseen. Without a decent monitoring solution in place, IT teams have no means of identifying issues or incidents, besides waiting for an end user or customer to tell you the workload is not available or not responding as it should be. Interestingly enough, by relying on your monitoring metrics and logs, IT Operations and DevOps teams could often also detect and mitigate issues a lot faster, or even better, avoiding outages at all.

Built-in Azure solutions such as Azure Monitor, Azure Log Analytics, and Application Insights are able to provide you with all necessary details about your Azure services (and often hybrid as well for virtual machines running Windows or Linux). For specific workload scenarios like Azure Kubernetes Services (AKS), there is full supportability for third-party observability solutions like [Prometheus](#) and [Grafana](#) (see Figure 3-4).



**Figure 3-4.** Observability using Prometheus and Grafana (source: <https://prometheus.io>)

## DevOps and Automation

While it's not the only factor for outages, the human being (your DevOps teams) is often the root cause of issues in an IT environment. Scenarios such as patching, monthly maintenance, disaster/recovery testing, and the like often start with good intentions (keeping systems healthy) but often lead to a more severe outage than what was planned for (or not planned for at all).

The more automation can be brought into the workload's life span, the more reliable it should run. Starting from a DevOps practice, where application source code gets stored in a source control system such as Azure DevOps Repos or GitHub Repositories, relying on feature branches, pull requests, and peer reviewing will contribute to the success and quality of the application code. Next, integrating adequate testing workloads as part of your Continuous Integration/Continuous Delivery (CI/CD) pipelines is another important aspect. No one wants to publish failing code, and definitely not finding out after the application workload got published. Talking about releasing workloads, nothing should be allowed to get published to production environments if it does not pass through Dev and Test and/or Staging environments first.

In Chapter 5, I'll guide you through several common CI/CD pipeline scenarios, both using Azure DevOps solutions as well as GitHub Actions.

Deploying the different environments should be done using Infrastructure as Code; think of Azure ARM Templates or Azure Bicep, or multicloud tools such as Terraform and Pulumi. Instead of manually (the human is the weak link) deploying your workloads, which is not only time-intensive but particularly error-prone, switching to automated deployments from scripts, Azure CLI or Azure PowerShell, would be a good step forward. Yet there are additional advantages when using Infrastructure as Code. Instead of going through your resource's deployment from a "this is what I'm going to deploy next" approach, IaC allows you to focus on the "this is what my end state should look like" characteristic of deploying resources. By using automation, deployments are also easily reproducible and repeatable, allowing IT teams to quickly run deployments across different environments but also relying on automation to redeploy a specific workload in case of a disaster. If you know you can deploy your virtual machine architecture, starting from the virtual machine resource and a fully patched operating system, integrating virtual network settings and network security group definitions, all the way to injecting automated software deployments using Custom Scripts or Configuration as Code, why would you not do that? Or, if the same deployment takes you 30 minutes to run, why would you spend 60 minutes troubleshooting a failing runtime?



That's the efficiency that DevOps and Automation can bring into your reliability objective.

## Self-Remediation

Self-remediation is where you rely on the automation aspect of a platform to fix itself in case of outages. This can be done in two different ways:

- a) **Combine automation and observability:** In this scenario, you rely on monitoring capabilities of the platform and cloud services, together with alerts and notifications, to trigger an automated action, which fixes the issue. A simple example is capturing an alert when a service is down in a specific region and triggering an Azure deployment in a different region because of this. Also covered in Chapter 6.
- b) **Use self-healing services:** I'm not sure if this is a 100% correct definition, but containerized workload scenarios such as Azure Container Instances (ACI) and Azure Kubernetes Services (AKS) are pretty close. One of the default settings of ACI is that – when a container runtime is failing – it stops the container and starts up a new instance. A similar behavior is what makes Kubernetes awesome, relying on its intelligent orchestration to detect failures in PODs (container runtimes in Kubernetes), starting them again when needed, all the way to spreading PODs across healthy nodes in case of an outage of a running node within the Kubernetes cluster. Even probes (liveness/readiness) could be defined to automatically check the health of your container workloads and heal.

The remaining chapters in this book extensively come back on each of these concepts, with much more practical examples and several hands-on scenarios for you to follow along.

## Reliability Checklists

In order to identify an application workload as reliable, it must meet the following requirements:

- **Availability:** What is the possible and required uptime in SLA terms?
- **High resiliency:** How quickly can the workload be restored or recovered in case of an outage?
- **Cost:** What is the additional cost to take to make the workload more highly available and reliable?

To be able to do that, one should start with a reliability checklist, for example:

- Identify availability as well as recovery targets to meet business requirements. This could include backup, data replication, Azure regions, availability zones, and the like.
- Integrate application resiliency and high availability by gathering requirements. Map both technical and business requirements to have a full view.
- Don't approve a go-to-production if the application and data platforms are not meeting your organization's reliability requirements.
- Establish redundant connectivity paths to guarantee cloud availability using Azure ExpressRoute and Site-to-Site VPN as fallback scenarios. Extend with VNet peering relying on the Microsoft Backbone across different regions.
- Use availability zones where applicable to improve reliability and optimize costs. Availability zones are supported for virtual machines but also for Azure Web Apps and other Azure services.
- Guarantee that your workload architecture is resilient to failures, following the Microsoft Azure Reference Architectures, as well as Well-Architected Framework guidelines.

- Establish a fallback scenario when the technical or business requirements of the available Service-Level Agreements are not met. Identify both the business risks, costs, and processes to get them in order as part of the solutioning.
- Identify potential weakness or failure points in the workload to enable resiliency. Scenarios such as Chaos Engineering, which we touch on in Chapter 8, could be a great tool to do this.
- Ensure that systems, data, and applications can still function during the absence of one or more of their dependencies. Architectures relying on serverless or Microservices could make this possible, together with load balancers and deploying workloads across multiple regions.

## Testing Applications for Resiliency

Once the architectural design is in place and the actual workload has been deployed according to the synergy of both technical and business requirements, preferably in a dev/test or staging environment, it is time to go through your testing scenarios, validating the level of resiliency (and reliability) one can achieve with the deployed architecture.

There is a strong opinion across different organizations on how and when application testing should be performed. In my opinion, the more testing you can execute, the better. **Shift left testing**, moving your testing practices the earliest possible in your application life cycle, has gained some momentum lately (more in Chapter 5).

The process starts already during the DevOps cycle, by running unit/integration testing of a component of an application during the build stage. However, this typically only runs code validation but doesn't check how reliable the workload will run once it is published.

Next to that, testing applications has the misunderstanding it should explicitly and exclusively occur in a dev/test environment (I know, I mentioned this myself only two paragraphs back). More and more, thanks to automated deployments such as Azure ARM Templates or Bicep, Terraform - Infrastructure as Code - as well as Configuration as Code with Chef, Puppet, Ansible, or PowerShell DSC, it allows organizations to run more testing in a faster way and makes this process far more repetitive. Remember, I

already highlighted the upcoming next wave of Chaos Engineering, which I explain in more depth in Chapter 8. While not really built as a testing scenario anymore – Chaos Engineering is really coming in, hammering your production-running workloads and bringing them down – it is still accepted as a testing scenario. And honestly, nothing would block you from running Chaos Engineering practices against your nonproduction workloads too (especially useful in preproduction, architectures as close as possible to your production design).

Another question is how frequently tests like Chaos or Load testing should occur (compared to unit/integration/UI tests that should run in an automated way many times per day on the defined CI/CD pipelines).

As Chaos/Load testing workloads may be more expensive and results may not vary that often, an option could be that at every intersection of a major change, such as an architectural change, a version or application upgrade, a migration to a new environment, etc., testing should be performed. Even if all looks fine in a dev/test or staging environment, we all know that there is always a (slight or big) difference between production environments. Make sure you go through detailed and thorough testing when deploying your workloads.

For some organizations and applications, it would be interesting to integrate a ring-based deployment strategy, slowly increasing the change-impacted user number by leveraging Canary/Early Adopter/General Availability groups/rings. Think, for example, what Microsoft is doing with Edge Browser or Windows 11 Preview. The benefit of this scenario is that you allocate a certain number of users (mainly voluntary) to get feedback (bugs/features) before reaching the general audience.

Another concept is Blue/Green deployment, where you typically deploy two identical environments at the same time. All users will connect to one environment, and when major changes are happening or new releases are getting introduced, they will get deployed to the other environment first. From there, they get tested and validated and eventually will be the endpoint for all users. In this time, the original environment gets typically scrapped and becomes the new location for updates, after which the users get redirected back to and so on.

In Chapter 5, I will explain a bit more on those concepts and how to integrate them from a DevOps perspective.

This brings me to the next question to answer: What kind of **reliability tests** should be performed?

There could be several tests positioned here (not including functional ones), depending on what you want to get out of it. In short, the following tests are common:

- **Performance testing:** This is where your application gets validated under user load and monitors its behavior. In relation to it, you could find load testing or stress testing practices; performance testing could combine both.

**Load testing** is mainly used for triggering scalability under certain expected user loads (try to simulate possible scenarios).

**Stress testing** is often used to validate the maximum load an application or system can withstand before it breaks (tries to find the system's limits). Think of CPU overclocking on gamers' PCs, to find out what the maximum performance is that can be reached by the computer under heavy load and displaying optimal graphics.

- **Fault injection testing:** This is the testing scenario where small faults are getting introduced to identify the weaknesses of the application architecture, or to bring it down. Fault injection is the **basis of Chaos Engineering**, although it typically takes it further by combining multiple failures and causing major outages. Easy scenarios here can be bringing down a virtual machine in an Azure VM Scale Set or Availability Set to see its impact, changing an access key to Azure Storage to validate if your application falls back on the secondary access key, changing the credentials of a Service Principal, disconnecting disks, interrupting networking connectivity, and the like.
- **Disaster/recovery testing:** This is probably the more complex of testing. In this scenario, your test or production environment will be restored from backup or data replication, typically in a completely new environment. Services such as Azure Backup and Azure Site Recovery (for virtual machines only) can be used here. The goal is to validate if your backup and disaster recovery solution is adequate to bring your production environment back up and running by relying on a backup or recovery process. It helps confirm metrics like Recovery Time Objective (RTO) and Recovery Point Objective (RPO)

defined for your solutions (covered in the next chapter). While this all seems obvious, I do remember several scenarios where a customer was running backup jobs, only to find out during an outage, nothing was actually stored on backup tapes.

## Well-Architected Framework Assessment

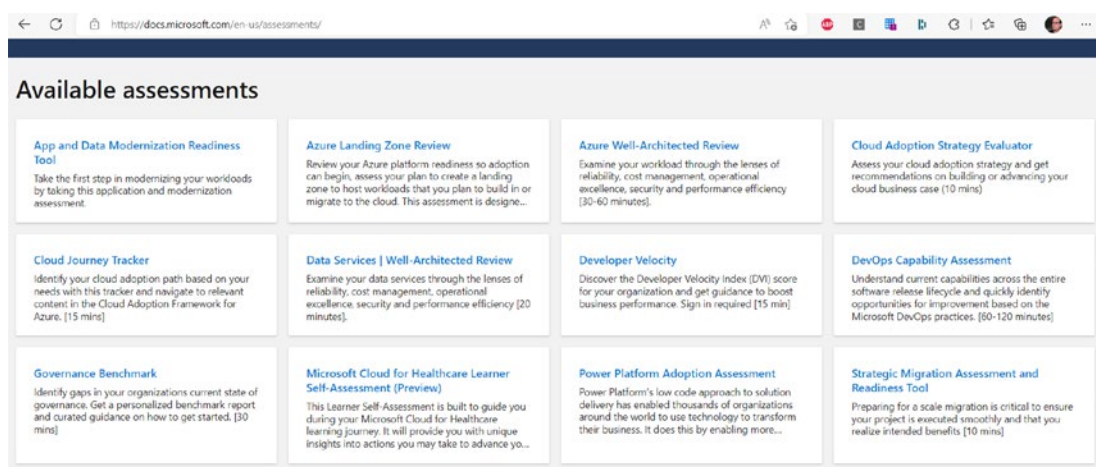
You should have a very good understanding of reliability and resilience of application architectures by now, as well as being able to identify how to thoroughly test your applications against failures and prevent outages.

In this last part of the chapter, I want to take you back to the start of the process, mapping out what level of Well-Architected Framework maturity your organization has.

Microsoft provides several different assessment tools and practices for you to go through, where the Well-Architected Framework is only one of them.

The following assessments exist at this link: <https://docs.microsoft.com/en-us/assessments/> (see Figure 3-5):

- App and Data Modernization Readiness Tool
- Azure Landing Zone Review
- Azure Well-Architected Review
- Cloud Adoption Strategy Evaluator
- Cloud Journey Tracker
- Data Services - Well-Architected Review
- Developer Velocity
- DevOps Capability Assessment
- Governance Benchmark
- Microsoft Cloud for Healthcare Learner (Preview)
- Power Platform Adoption Assessment
- Strategic Migration Assessment and Readiness Tool



**Figure 3-5.** Microsoft assessments

I wish we had the time and page resources to discuss each of these, as several are actually relevant to the scope of this book. But as you can imagine, I will walk you through the **Azure Well-Architected Review**.

1. Click the Azure Well-Architected Review from the available assessments.
2. Provide a descriptive name for the assessment or accept the default one (see Figure 3-6).

## Azure Well-Architected Review

Examine your workload through the lenses of reliability, cost management, operational excellence, security and performance efficiency [30-60 minutes].

Assessment name \*

Azure Well-Architected Review - Jun 20, 2022 - 9:16:07 PM

Start →

**Figure 3-6.** Assessment name

3. Click Start to continue and start the WAF assessment. The first question covers the WAF Configuration, asking what workload you want to evaluate. Select Core-Architected Review (see Figure 3-7).

### WAF Configuration

## What workload type do you want to evaluate?

- Core Well-Architected Review
- Azure Machine Learning
- Internet of Things (Preview)
- SAP On Azure (Preview)
- Analytics

[← Back](#)

[Next →](#)

**Figure 3-7.** Workload type

4. Click Next. In the next question (see Figure 3-8), select the Reliability pillar (although feel free to repeat the assessment for the other WAF pillars as appropriate for your organization).

### Core Pillars

## What pillars would you like to evaluate?

- Reliability
- Security
- Cost
- Operational Excellence
- Performance

[← Back](#)

[Next →](#)

**Figure 3-8.** Evaluation options

5. From here, you are presented with several more specific questions (see Figure 3-9), where each question offers multiple choice answers.



## What reliability targets and metrics have you defined for your application?

Availability targets, such as Service Level Agreements (SLA) and Service Level Objectives (SLO), and Recovery targets, such as Recovery Time Objectives (RTO) and Recovery Point Objectives (RPO), should be defined and tested to ensure application reliability aligns with business requirements.

- Recovery targets to identify how long the workload can be unavailable (Recovery Time Objective) and how much data is acceptable to lose during a disaster (Recovery Point Objective). ⓘ
- Availability targets such as Service Level Agreements (SLAs) and Service Level Objectives (SLOs). ⓘ
- Availability metrics to measure and monitor availability such as Mean Time To Recover (MTTR) and Mean Time Between Failure (MTBF). ⓘ
- Composite SLA for the workload derived using the Azure SLAs for all relevant resources. ⓘ
- SLAs for all internal and external dependencies. ⓘ
- Independent availability and recovery targets for critical application subsystems and scenarios. ⓘ
- None of the above.

← Back

Next →

**Figure 3-9.** Assessment questionnaire

*Each question provides a detailed description of the context, as well as multiple answers to choose from. Make sure you can identify the most accurate answer(s) for your specific scenario. The more accurate the answers given, the more useful the WAF assessment outcomes will be for your organization.*

---

**Note** You will be reminded of these recovery targets/metrics in Chapter 4.

---

6. We recommend you not to try and skip any of the questions, as it will obviously influence the accuracy of the results.

**Note that for the ease of this chapter, we only copy the remaining questions as a reference here, in case you want to learn more about the WAF assessment for now, without spending time on actually going through the assessment.**

*What reliability targets and metrics have you defined for your application?*

*How have you ensured that your application architecture is resilient to failures?*

*How have you ensured required capacity and services are available in targeted regions?*

*How are you handling disaster recovery for this workload?*

*What decisions have been taken to ensure the application platform meets your reliability requirements?*

*What decisions have been taken to ensure the data platform meets your reliability requirements?*

*How does your application logic handle exceptions and errors?*

*What decisions have been taken to ensure networking and connectivity meet your reliability requirements?*

*What reliability allowances for scalability and performance have you made?*

*What reliability allowances for security have you made?*

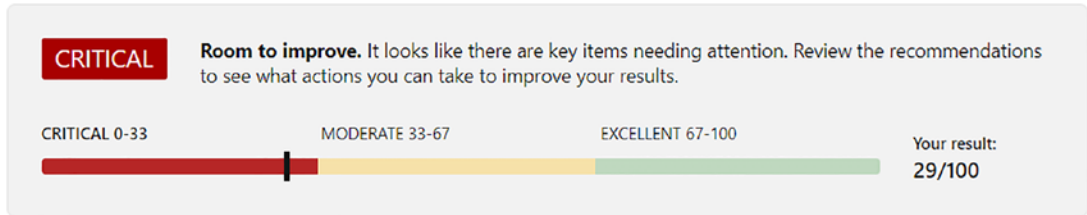
*What reliability allowances for operations have you made?*

*How do you test the application to ensure it is fault tolerant?*

*How do you monitor and measure application health?*

7. After completing the assessment questions, a summary of recommendations is presented (see Figure 3-10).

## Your overall results



## Categories that influenced your results

Reliability



You can find out how to improve on individual categories by reviewing the [recommendations](#) below in the report.

[Learn how to import](#) your CSV into Azure DevOps using a PowerShell script.

**Figure 3-10.** Assessment result

- There is also an overview of Next Steps (see [Figure 3-11](#)) to improve the results (outcome depends on the answers you provided).

### 33 recommended actions

Identify distinct workloads	Plan for expected usage patterns
Have clearly defined availability targets	Manage load balancer connections to avoid port exhaustion
Compute a composite SLA for your workload	Load balance traffic across availability zones
Perform a failure mode analysis	Automate key rotation
Decouple the lifecycle of the application from its dependencies	Put operational procedures into place for if data size exceeds limits
Collect and store logs and key metrics of critical components	Automate your tests

**Figure 3-11.** Recommended actions

9. Where for each of the Recommended actions, a link is provided to the official Microsoft Docs, Well-Architected Framework section. For example, the outcome of the “Perform a failure mode analysis” recommendation refers to this Microsoft Docs article:  
<https://docs.microsoft.com/en-us/azure/architecture/resiliency/failure-mode-analysis>
10. When you sign in with a Microsoft account, it is possible to store the assessment details for later retrieval. This will also allow you to go back and revise several of the questions, the answers, and how they impact on your overall reliability of your Azure workloads.

## Summary

In this chapter, you were introduced to the Azure Well-Architected Framework. Microsoft offers Azure customers and partners a set of guidelines, approached from five different pillars: Reliability, Security, Cost Optimization, Performance, and Operation Excellence.

This chapter mainly focused on the Reliability pillar, detailing the best practices of making your Azure-running workloads more reliable and resilient. After touching on several aspects of the Well-Architected Framework concepts, we guided you through the WAF assessment and how to interpret the outcome of the assessment.

## CHAPTER 4

# Architecting Resilient Solutions in Azure

While Chapter 3 reviews the main five pillars of Azure Well-Architected Framework, with a focus on the reliability pillar, on a conceptual way, this chapter focuses on the resiliency topic in a more practical way. It shows the main features Azure offers to make your solutions more resilient.

By the end of this chapter, you should be able to understand the following:

- What is resiliency
- Resiliency on Azure
  - Availability sets
  - Availability zones
  - Region pairs
- Resiliency based on numbers
- Resiliency on application design
- Mainly used Azure components and features for resiliency
- Resilient architecture examples
  - IaaS, PaaS/Serverless, and Microservices

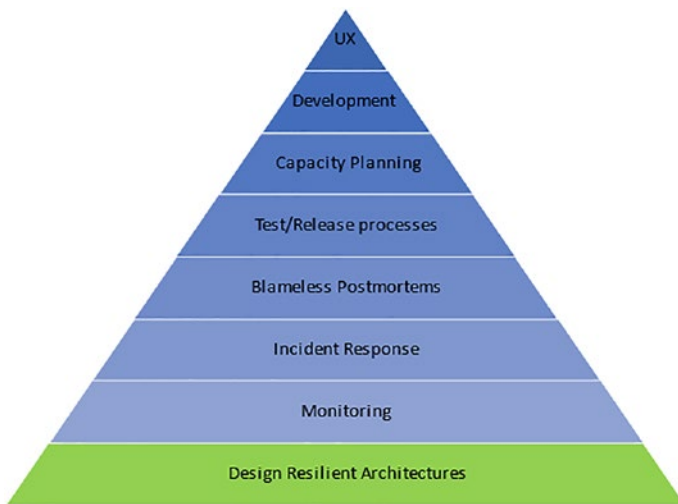
## What Is Resiliency?

Nowadays, everyone expects applications to be available 24/7 with instant access to their services. The previous chapter defined a **reliable service** as a solution that minimizes

downtime. Trying to achieve no incidents is an impossible task in cloud environments due to their complexity: dependency on external components, distributed systems cascade issues across components, hardware failures that cannot be avoided, etc.

As a result, a shift of mindset is needed. Temporary or large-scale failures should be expected, and solutions should be designed to recover from them. You need to design **resilient architectures**, resiliency defined as “the ability to recover from failures.”

Previous chapters mentioned Dickerson’s hierarchy of reliability as a base of practices to include on your day-to-day SRE activities. There is no directly linked topic on the pyramid related to resiliency; “capacity planning” would be the closest one, but it is mainly related to scalability, just one of the scenarios covered in this chapter. Personally, I would consider architecture design as the base of the pyramid (see Figure 4-1), putting together ideas covered in both Chapters 3 and 4, as this is an activity you can execute from the very beginning of a solutions life cycle (planning/design phase).



**Figure 4-1.** Customized hierarchy of reliability

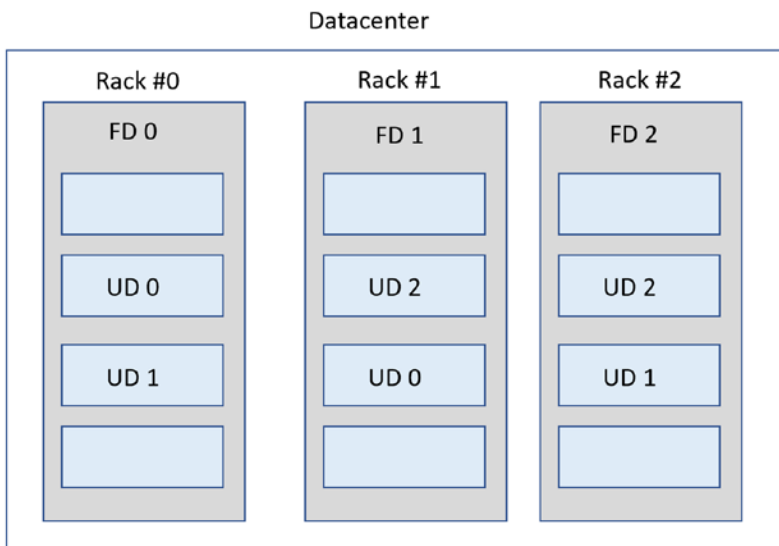
Resiliency practices should be applied at many levels of your architecture, like retrying mechanisms on your application code or global replicas for your data stores. In other words, **building resilient architectures is everyone’s responsibility**. It is critical to fully understand the cloud platform before digging deeper into the topic. Chapter 3 was focused on explaining the basic pillars of the Azure Well-Architected Framework. This chapter will focus on practical examples and deeper details of the **reliability** pillar. Let’s review some important aspects of Azure-provided services by analyzing the nature of Azure cloud platform.

## Azure Platform Resiliency

If we try to explain the Azure cloud platform in a simplistic way, Azure is composed of the following elements:

- Geographical regions offered to customers
- Data centers composing those regions
- Racks grouping physical server and storage resources used for the offered cloud services

Microsoft is running physical servers in the data centers; these ones may fail too! Take a virtual tour of the data center here: <https://news.microsoft.com/stories/microsoft-datacenter-tour/>. The good news is that Microsoft will offer you options you can easily implement to protect against them. What kind of issues? Mainly two: updates and hardware-related ones. These bring two important concepts to mind (see Figure 4-2).



**Figure 4-2.** Update and fault domains

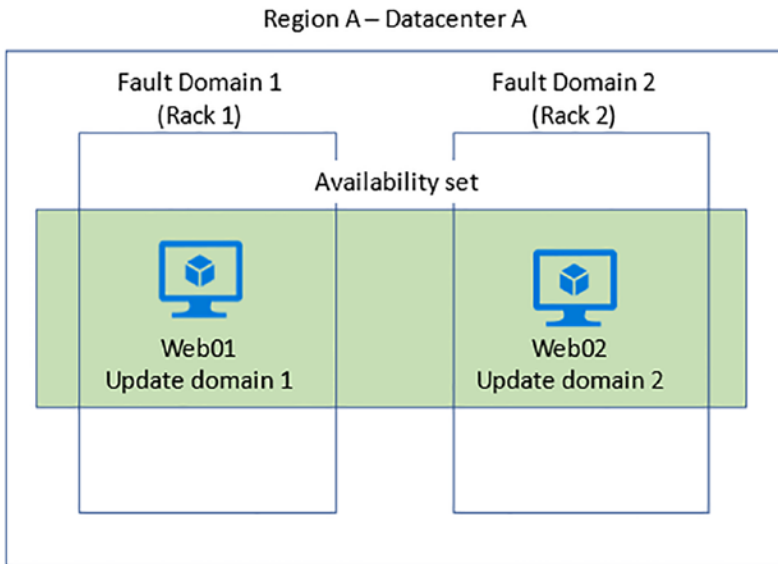
- **Fault domain:** A group of hardware components with a single point of failure (power or network). For example, if a rack fails, all server/storage (and services running on them) would fail with it.
- **Update domain:** A group of machines that are updated on the same cycle and cause problems in case it doesn't go the expected way.

Azure platform will offer the following setups to overcome platform-related issues.

## Availability Sets

They are an offering mainly related to Azure VMs, but services based on VMs like AKS and VMSS also offer them. They are a logical grouping of two or more VMs hosted in a way to provide higher availability. Two VMs on the same availability set (with two fault domains and two update domains) will be placed in a way where no updates of hardware failures will affect both at the same time (see Figure 4-3). Each availability set can be set up with a maximum of 3 fault domains and 20 update domains. Azure never updates two update domains at the same time, so you may want to go for a lower number than 20 (the maximum). Imagine your web solution will be running in ten instances and you need to have at least eight running while performing upgrades the fastest way possible. How many update domains would you need? The answer is five; machines would be placed the following way, only two being unavailable during updates: UD0 (VM1, VM6), UD1 (VM2, VM7), UD2 (VM3, VM8), UD3 (VM4, VM9), and UD4 (VM5, VM10).





**Figure 4-3.** *Availability set*

## Availability Zones

While availability sets let you protect against update and hardware-related issues, Availability zones take another step; they place your VMs in different data centers within the same regions, protecting your workloads against data center downtimes. Also known as **zone redundancy**, each zone has a distinct power source, network, and cooling system, highly connected between them, offering a latency of less than 2ms between zones. Proximity Placement Groups can be leveraged to offer even lower latencies.

## Region Pairs and Azure Site Recovery

Previous options could give you SLAs up to 99.99% ([https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1\\_9/](https://azure.microsoft.com/en-us/support/legal/sla/virtual-machines/v1_9/)), but what about regional disaster?

You can leverage Azure VMs deployed in multiple regions together with storage solutions like geo-redundant storage (GRS) for storage accounts to ensure business continuity. You can use Azure Site Recovery for it.

For regional replication, try to leverage Azure **region pairs**. Each region is paired with another region within the same geography, prioritizing one in case of broad outages and never updating services in both regions at the same time to avoid downtime. Normally, region pairs are placed in the same geography to meet data residency requirements. For example, East US-West US or North Europe-West Europe. You can see

the full list here: <https://docs.microsoft.com/en-us/azure/availability-zones/cross-region-replication-azure#azure-cross-region-replication-pairings-for-all-geographies>

## Resiliency Based in Numbers

There will always be a trade-off between the level of resilience desired and the budget assigned to the solution. As an SRE, your job is to define the resilience requirements to have an estimation of the budget needed to achieve your business goals.

Recall that Chapter 2 explained the most frequently used service-level metrics: SLI, SLO, and SLA (SLI being the tracked indicator, SLO being the objective defined for it, and SLA being the contract agreement signed with your customers).

If the SLA provided to our customers is 99.95% (21.6 minutes downtime per month), your workloads should be designed to always keep the SLI indicators above those numbers (normally defining a more demanding SLO for your teams, SLO>SLA, making sure there is some error budget before contract is violated).

Look at the related downtimes for the SLAs shown in Table 4-1.

**Table 4-1.** SLA/downtime

SLA	Downtime per week	Downtime per month	Downtime per year
99%	1.68 hours	7.2 hours	3.65 days
99.9%	10.1 minutes	43.2 minutes	8.76 hours
99.99%	1.01 minutes	4.32 minutes	52.56 minutes

The table clearly shows that for demanding SLAs, you cannot rely on fixing issues manually. You need to provide efficient incident-resolving and self-healing solutions.

Normally, SLAs are not only defined for the architectural availability perspective; when defining SLOs on your solution, you do it from the customer perspective (customer experience). For example, for an e-commerce website, you could define the following SLO: “for the payment API, customers should get 99.9% of successful transactions during the last 30 days.” These metrics are not only affected by the **architectural/infrastructure** components (e.g., zone and regional replicas) resulting in a composite SLA (as seen in Chapter 2); how your **application code** handles those transactions (e.g., using retry mechanisms) will also be critical to the success.

Decomposing critical and noncritical workloads will be a recommended exercise for your organization; many use tiers to logically separate workload requirements. Most probably not all workloads will have the highest resiliency needs and need for expensive globally replicated solutions. It is obvious if we say that an e-commerce website (money income) will not have the same requirements (tier) as the internally used HR website.

There are also **two recovery metrics** frequently used to define the requirements for workloads:

- **Recovery Time Objective (RTO):** Maximum time application/service can be unavailable after incidents.
- **Recovery Point Objective (RPO):** Maximum duration of data loss during a disaster.

These metrics will help you shape your needs in terms of resiliency of your solutions. RPO is focused more on data loss (prevented by data replication/backups/point-in-time restores), while RTO is more focused on availability (prevented by zone/regionally replicated services). For example, for RPO objectives, Azure VM disks recovery points can be created frequently by using Site Recovery offering.

Recall that Chapter 2 also covered the importance of availability metrics like **MTTR**, **MTTF**, and **MTBF**. In the previous case, if your Mean Time to Recover (MTTR) metric exceeds RTO, that could cause dangerous business disruption (money compensation defined in contracts) as it would not meet the promised restoring time (RTO).

When defining tiers for your workloads, your tiers will have SLA, RPO, and RTO metrics defined for each. For example, Microsoft for Azure SQL databases using zone redundancy offers an SLA of 99.995%, an RTO of 30 seconds, and an RPO of 5 seconds, giving you back 100% credit if SLA is not met! (<https://azure.microsoft.com/en-in/blog/understanding-and-leveraging-azure-sql-database-sla/#:~:text=That%20SLA%20comes%20with%20very%20strong%20guarantees%20of,in%20the%20industry%20offering%20a%20business%20continuity%20SLA>).

It is also important to remember the **shared responsibility** model (see Figure 4-4). On every cloud platform, depending on the selected offering, the provider will be responsible for certain layers of the solution, while others will be your responsibility, compared to on-premises solutions where full responsibility falls on the customer. Resiliency requirements will also depend on the chosen model: IaaS, PaaS, or SaaS. For example, for an Azure VM (IaaS), you will need to define the replication options mentioned before (availability sets/zones or global replicas), whereas for Azure DevOps (SaaS), the resiliency of the solution is taken care of by Microsoft.

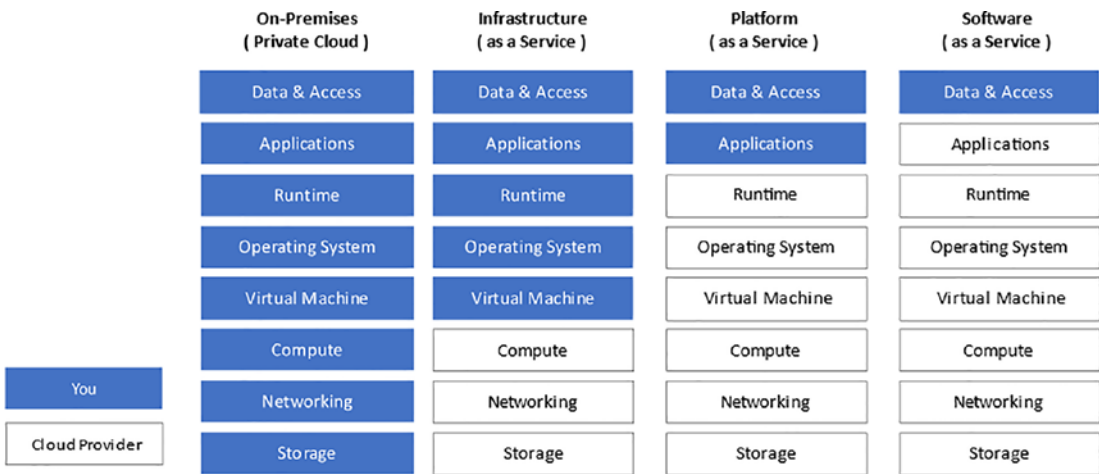


Figure 4-4. Shared responsibility model

## Resiliency on Application Design

Most times, when thinking about resiliency, people tend to think of ways to replicate data or add redundancy to infrastructure components. But what about application architecture? Incidents could happen at any layer of your solution.

Showing how to design application architecture is not the aim of the book, but as an SRE in an organization, you should make developers aware of this responsibility and encourage them to implement application patterns like the ones mentioned in the following:

- Retry/circuit breaker pattern:** Complex architectures are composed of multiple dependencies. The microservices running your solution may need to be calling external dependencies constantly, and **transient issues** are common. Your application should be able to handle those transient issues by implementing retry mechanisms. For example, if a service needs to call Azure Cosmos DB to read some objects and the reply you get is HTTP 429 (throttled request, not enough capacity), it may work retrying after a delay. If a reply like HTTP 401 authorized is received after the first call, retrying without changing the authentication properties may not get you the answer.

- Retry mechanisms should evaluate the response codes and implement retry operations for codes that reflect temporary issues. Implement timeouts to avoid blocking operations. Circuit breakers will avoid executing calls that are likely to fail (open circuit), testing out the endpoint after a delay and reclosing if successful
- Many built-in libraries support retry mechanisms: <https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/infrastructure-resiliency-azure#built-in-retry-in-services>.
- **Endpoint monitoring:** Use functional check to evaluate the health of services offered and needed by other parts of your solution. Kubernetes, for example, restarts pods based on health probes: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>. Azure load balancers (many of them covered later) can also distribute traffic based on the health of endpoints.
- **Queue based, publisher/subscriber pattern:** Components of the system should use asynchronous communication based on queues; it will help in load leveling (a producer faster than a consumer) and also in making sure messages are correctly processed by the consumer (e.g., using “peek and lock” mechanism used by Azure Service Bus queue, you can make sure the message is correctly processed before deleting).
- **Throttling:** It is a pattern to make sure clients only are allowed to use resources up to a limit, limiting the autoscaling up to some point. For example, hosting your APIs behind Azure API Management will let you apply throttling limits using policies.
- **Idempotent task:** Duplicated tasks should produce the same result. If a consumer is executed the same task, it should not lead to invalid results.
- **Fail back to different services:** For example, if your application is trying to store some data in Azure SQL and after some retries, it does not work, keep it temporarily in an Azure Cache for Redis (providing a replay for later writes).

- **Avoid affinity:** Avoid “sticky sessions,” as they may give problems in dynamic environments like the autoscaled solutions mentioned. Use services like Azure Cache for Redis to persist state.
- And many more that can be found, for example, here: <https://docs.microsoft.com/en-us/azure/architecture/patterns/>.

## Mainly Used Components/Platform Features for Resilient Solutions

Before checking some resilient architecture examples, it may be helpful to review the most frequently used Azure platform resiliency features and services. Most resilient architectures will share these capabilities; responsibilities will vary depending on what solution types you choose (IaaS, PaaS, or SaaS).

### Autoscaling

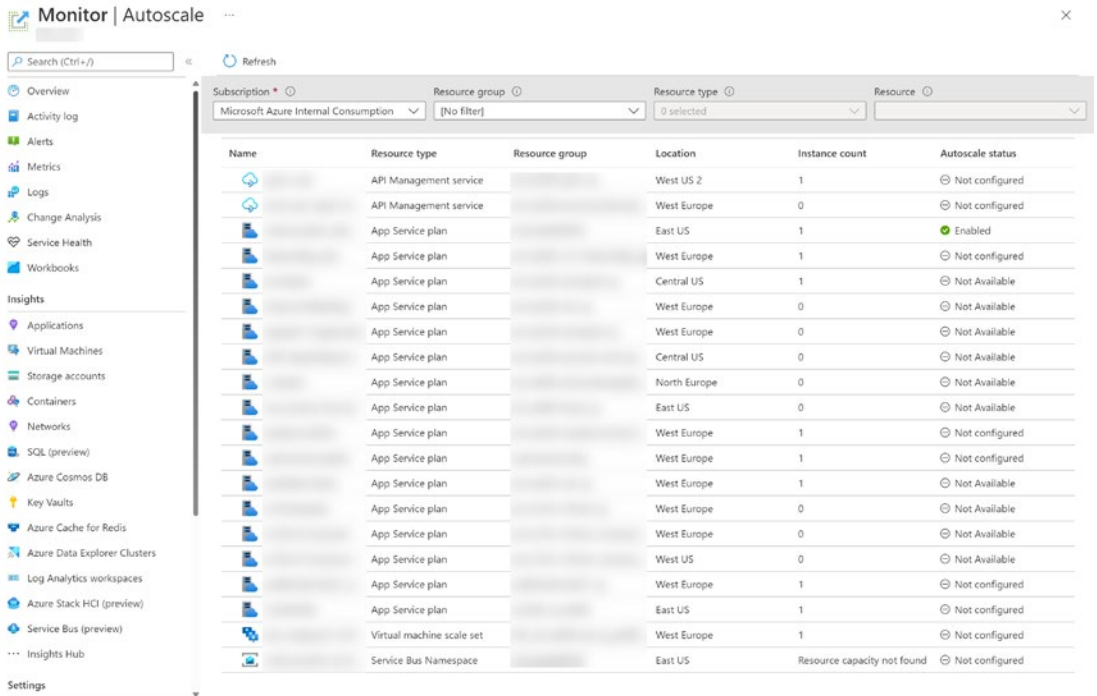
It is one of the most obvious measures you think about when trying to increase resiliency. There are two ways to scale services:

- **Vertical scaling (up/down):** Related to changing the capacity of the chosen resource, or simply put, changing the chosen pricing tier. For example, scaling up an Azure App Service Plan for B1 to S1. It often makes your solution unavailable!
- **Horizontal scaling (out/in):** Means adding or removing instance replicas. This is the scaling mode **used by autoscaling services**. It is mainly related to computing resources.

Autoscaling allows you to set up the correct amount of resource instances based on collected metrics/schedule. But it varies across resource types. Resources like Virtual Machine Scale Sets (VMSS) or App Service Plans let you define the minimum/maximum number of instances to run based on rules.

Other resources like Cosmos DB or Event Hub (auto-inflate) offer autoscaling options where rules are not chosen/defined (they are managed by the platform); only a maximum number of resources are defined (RU/s for Cosmos DB and TU for Event Hubs).

Azure Monitor (covered in Chapter 7) also offers a unified experience where all “scalable” (horizontal scaling) resources are shown, giving you the option to set up rules and monitor instance count (see Figure 4-5).



The screenshot shows the Azure Monitor Autoscale interface. The left sidebar contains navigation options like Overview, Alerts, Metrics, and Insights. The main area displays a table of resources with columns for Name, Resource type, Resource group, Location, Instance count, and Autoscale status.

Name	Resource type	Resource group	Location	Instance count	Autoscale status
[Redacted]	API Management service	[Redacted]	West US 2	1	Not configured
[Redacted]	API Management service	[Redacted]	West Europe	0	Not configured
[Redacted]	App Service plan	[Redacted]	East US	1	Enabled
[Redacted]	App Service plan	[Redacted]	West Europe	1	Not configured
[Redacted]	App Service plan	[Redacted]	Central US	1	Not Available
[Redacted]	App Service plan	[Redacted]	West Europe	0	Not Available
[Redacted]	App Service plan	[Redacted]	West Europe	0	Not Available
[Redacted]	App Service plan	[Redacted]	Central US	0	Not Available
[Redacted]	App Service plan	[Redacted]	North Europe	0	Not Available
[Redacted]	App Service plan	[Redacted]	East US	0	Not Available
[Redacted]	App Service plan	[Redacted]	West Europe	1	Not configured
[Redacted]	App Service plan	[Redacted]	West Europe	1	Not configured
[Redacted]	App Service plan	[Redacted]	West Europe	1	Not Available
[Redacted]	App Service plan	[Redacted]	West Europe	0	Not Available
[Redacted]	App Service plan	[Redacted]	West Europe	0	Not Available
[Redacted]	App Service plan	[Redacted]	West US	0	Not Available
[Redacted]	App Service plan	[Redacted]	West Europe	1	Not configured
[Redacted]	App Service plan	[Redacted]	West Europe	1	Not configured
[Redacted]	App Service plan	[Redacted]	East US	1	Not configured
[Redacted]	Virtual machine scale set	[Redacted]	West Europe	1	Not configured
[Redacted]	Service Bus Namespace	[Redacted]	East US	Resource capacity not found	Not configured

**Figure 4-5.** Azure Monitor autoscale

## Load Balancer

Traffic load balancers are critical resources in resilient architectures. They give you the capability to distribute traffic across those regional (and cross-regional) replicas.

Many PaaS services such as Azure App Services implement a load balancer that does not need to be configured. Azure platform takes care of load balancing based on the autoscaling rules. For example, for an Azure App Service called SREUnaiWebapp, a client would use a URL of <https://SREUnaiWebapp.azurewebsites.net> to call it, without bothering if one App Service Plan instance or multiple are used in the background. App Services even give you the option to distribute traffic balancing to slots, without requiring managing a load balancer as such.

When you want to have control over the load-balancing techniques that should be applied, Azure offers the following **four load-balancing services**:

- **Application Gateway**
- **Front Door**
- **Load Balancer**
- **Traffic Manager**

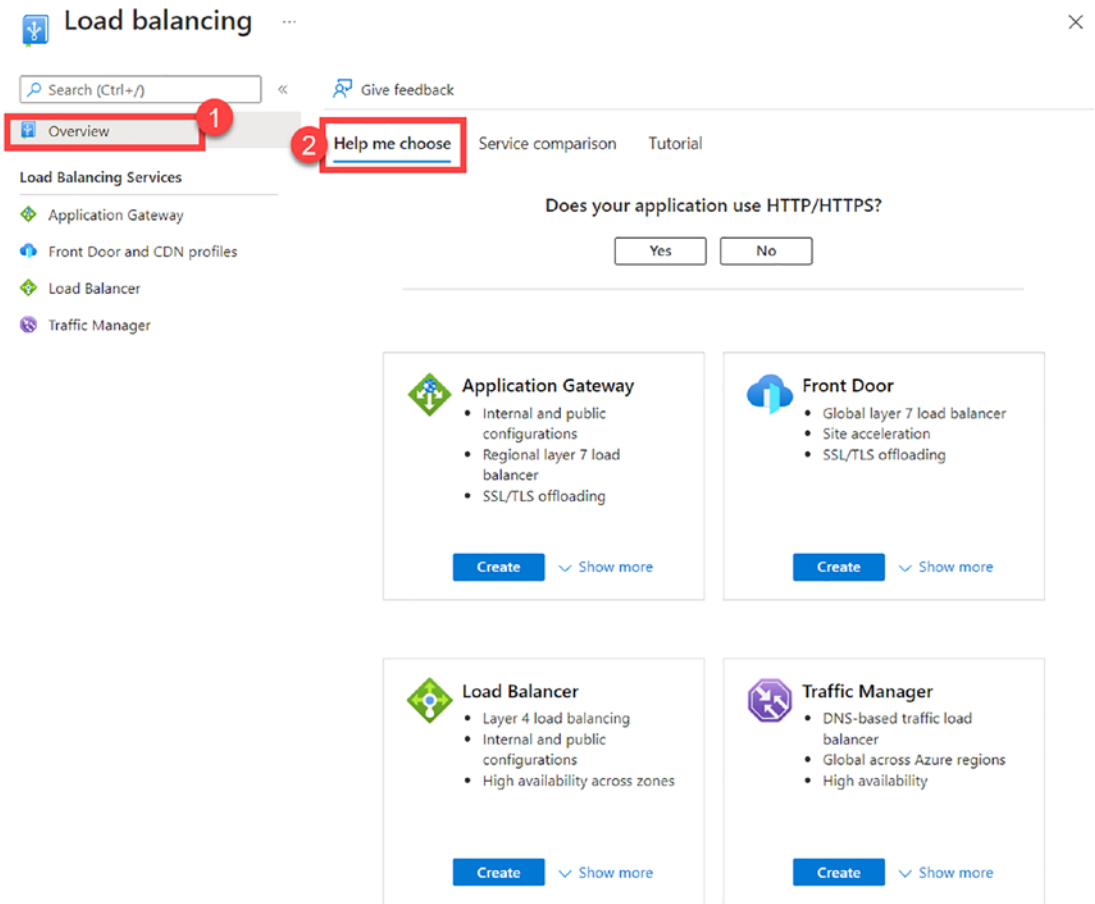
The easiest way to categorize these services is by two dimensions: global/regional and HTTP(S)/non-HTTP(S) (see Table 4-2).

*Table 4-2. Azure load-balancing services*

<b>Service</b>	<b>Global/regional</b>	<b>Protocols</b>
<b>Azure Front Door</b>	Global	HTTP(S)
<b>Traffic Manager</b>	Global	non-HTTP(S)
<b>Application Gateway</b>	Regional	HTTP(S)
<b>Azure Load Balancer</b>	Regional* (cross-regional in preview)	non-HTTP(S)

The portal also offers a section based on a questionnaire (see Figure 4-6) that provides a recommended service. Additionally, a more complete comparison table is provided.





**Figure 4-6.** Azure portal helper

As mentioned before, resilient architecture will frequently use the mentioned services to load-balance traffic across the deployed regional and cross-regional replicas, keeping track of the health of those endpoints for failover scenarios.

It is important to point out that your **load balancer could also fail**, leaving your replicated solution “useless.” Review the SLA offered by the load balancer service and determine if you may need to add another load balancer instance/solution as a fallback, updating CNAME records in DNS when failing.

## Replication/Redundancy

The Azure platform offers many ways to create redundant workloads, which vary depending on the service chosen. Remember from the first sections in the chapter, replicas can be running in different data centers by choosing availability zone features. Some services, mainly data-related ones, offer geo-redundancy features.

For example, when working with Azure App Services, the default experience places the App Service Plan VMs in a single zone. Optionally (available for premium plan v2 and v3), availability zone features can be enabled, distributing the VM instances across three zones in a selected region. In this case, geo-redundancy would have to be managed by the customer, placing one solution in each region and leveraging the previously mentioned load balancer.

Other data solutions like Azure Cosmos DB or Azure SQL provide geo-redundancy that can be set up easily with a “few clicks.”

When deploying solutions across regions, which may considerably increase costs, you will need to choose an **active-active** or **active-passive** configuration, depending on your defined RTO:

- **Active-active:** Both regions’ active and load-balancing requests.
- **Active-passive (hot standby):** Traffic is sent to the active region but secondary has allocated instances in case it needs to get requests.
- **Active-passive (cold standby):** Traffic sent to the active region and secondary regions’ instances not allocated until failover is needed. It decreases costs but increases recovery time.

## Resilient Architecture Examples

To provide practical ideas for the concepts explained previously, this section will provide some **resilient architecture examples** (focused mainly on infrastructure design) for different types of workloads grouped by different categories: IaaS, PaaS/Serverless, and Microservices.

## IaaS Resilient Architecture

Let's imagine the following IaaS-based scenario (see Figure 4-7). Let's analyze the components that make this architecture resilient:

- **Azure Virtual Machines:** The VMs in these solutions use the following capabilities:
  - **Multiregion:** The solution leverages multiple regions. Choosing region pairs is recommended. **Azure Site Recovery** lets you replicate VMs to another region.
  - **Availability zone:** Provides protection against data center failures.
  - **Virtual Machine Scale Set (VMSS):** Can be used to autoscale your VM instances based on metrics or schedule.
  - **SQL server in VMs:** Leverage **Always On Availability Groups** for database mirroring. It provides high availability and disaster recovery. **VNET peering** is needed for replicas hosted in different regions (addresses should not overlap).
- **Azure Traffic Manager:** You can implement either active-active or active-passive scenarios and use this service to load traffic. On this example, you could use **priority**-based traffic routing for primary and secondary regions. **Health probes** can be used to monitor each regional solution and failover automatically. The example does not use a backup solution for Traffic Manager; consider setting another load-balancing instance based on SLA and RTO metrics, changing the CNAME records in **Azure DNS** when failing.
- **Azure Load Balancer:** External (public) load balancer is used to distribute traffic to the front-end VMSS and internal (private) load balancer for the SQL back end.

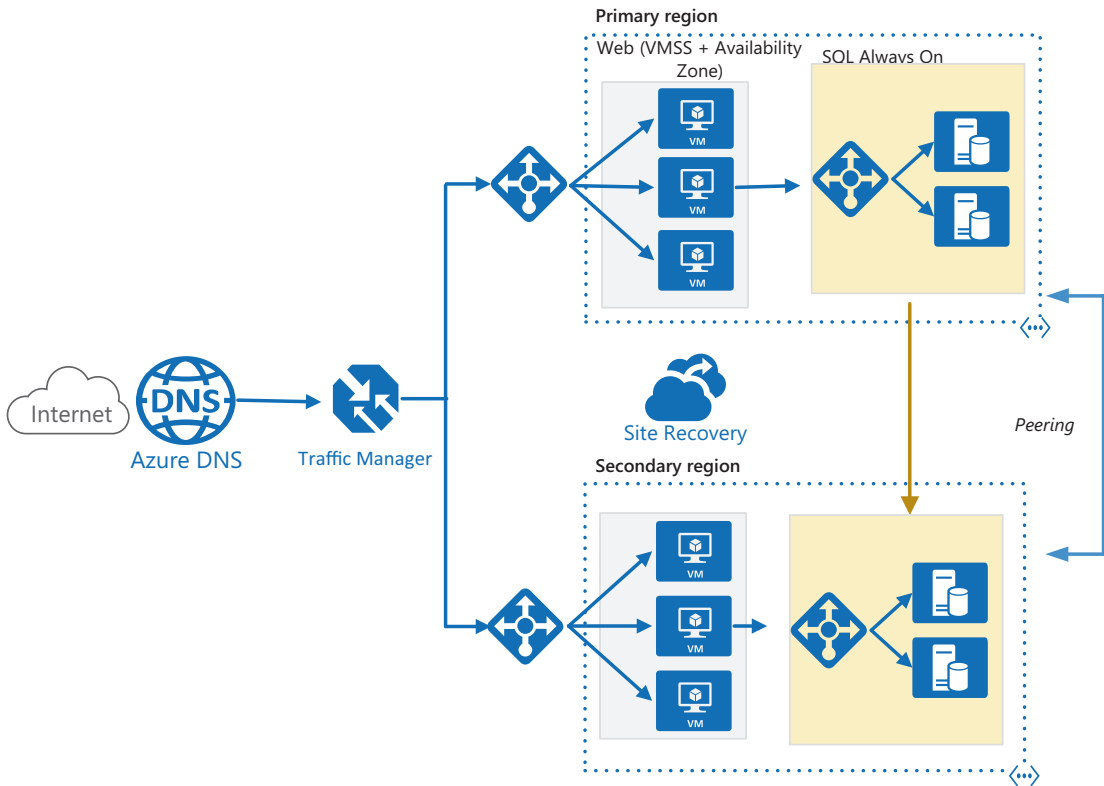


Figure 4-7. IaaS resilient architecture

## PaaS Resilient Architecture

When moving from on-premises solutions to cloud, most organizations start moving their workloads to IaaS solutions, as they offer a more familiar way to manage their solutions and lift-shifting on-premises VM images to the cloud seems more natural. But the “real” power of cloud providers resides on PaaS/Serverless offerings, as the provider takes care of more responsibility layers, offering easier ways to create resilient solutions.

Let’s decompose the following architecture example (see Figure 4-8):

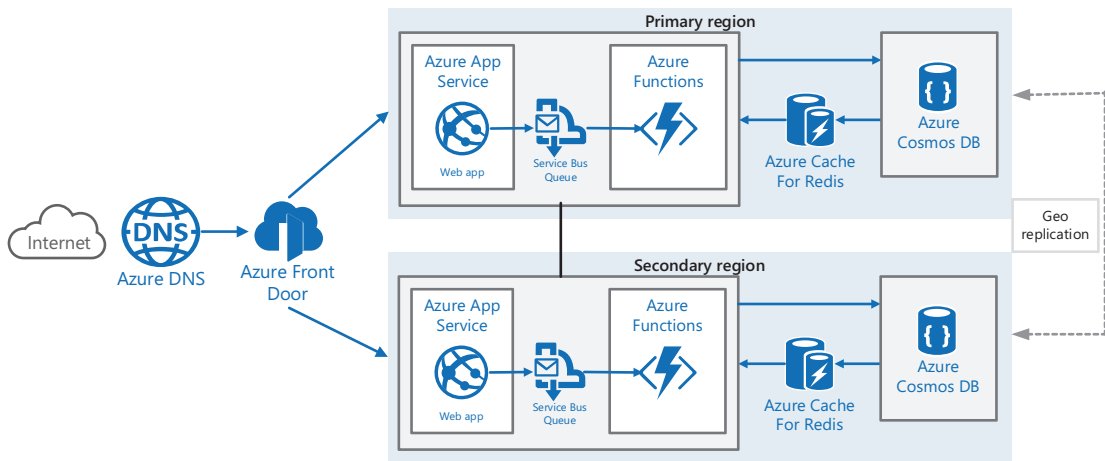
- Azure WebApp/App Service:** Confusing for many, WebApp=App Service. The service is composed of an App Service Plan (underlying VM) and the App Services hosted inside. An App Service Plan offers 99.95% of SLA (except for Free and Shared Tiers). Autoscaling rules should be defined to adjust instances to the load the WebApp

receives. Default autoscaling rules changes the number of App Service Plan instances, per app scaling could be defined if only selected WebApps should be affected. The proposed solution deploys the service in two regions for increased availability.

- **Azure Service Bus:** This is a premium messaging service offered by Azure. A Service Bus namespace can use an availability zone for even data center failures, and it also offers geo-disaster recovery for almost instantaneous failover (it does not replicate existing messages; for active-active configurations, check this guidance: <https://docs.microsoft.com/en-us/azure/service-bus-messaging/service-bus-federation-overview>). In this solution, the Web APIs delegate background tasks to the Azure Functions, and the queue helps in load leveling.
- **Azure Functions Apps:** This is a serverless service used to run “pieces of code,” letting Microsoft handle the rest, infrastructure setup, maintenance, and scaling. It offers three main hosting options: Consumption, Premium, and Dedicated (using an App Service Plan). Premium and Dedicated are the only options offering zone redundancy (data center failure protection). With the Premium option, the solution will also automatically scale (no rules needed). The example shown deploys our Function Apps in two regions for higher availability.
- **Azure Cosmos DB:** This is a fully managed NoSQL database. The SLA offered by the service can go up to 99.999% for multiregion (with writes) and availability zones setups. It offers an easy experience to set up multiple global replicas that can be used for both reads and writes. The service can be set up to automatically scale based on load. When using Cosmos DB with availability zones, it provides RTO=0 and RPO=0 even in zone outages!
- **Azure Cache for Redis:** This is an in-memory data store offered by Azure that is based on Redis open source software. Moving your Redis solutions has many advantages in terms of maintenance, scalability, and high availability. It is offered on multiple tiers and memory sizes, being the Premium and Enterprise ones, the ones

offering geo-replication and zone redundancy for higher availability. In this solution, it provides a faster mechanism to read database objects (cached data), offloading repetitive activity like frequent queries against the Cosmos DB, and it could also be used as a backup in case Cosmos DB writes do not work temporarily (will need to implement some rewrite mechanisms).

- Azure Front Door:** This is the load-balancing solution chosen for this example. It also offers CDN capabilities to cache web static content that increases performance and protects websites from threats using the Web Application Firewall (WAF) feature. Similarly to the previous example, based on RTO/SLAs, duplicated load-balancing solutions could be set up, changing DNS when failed over to backup.



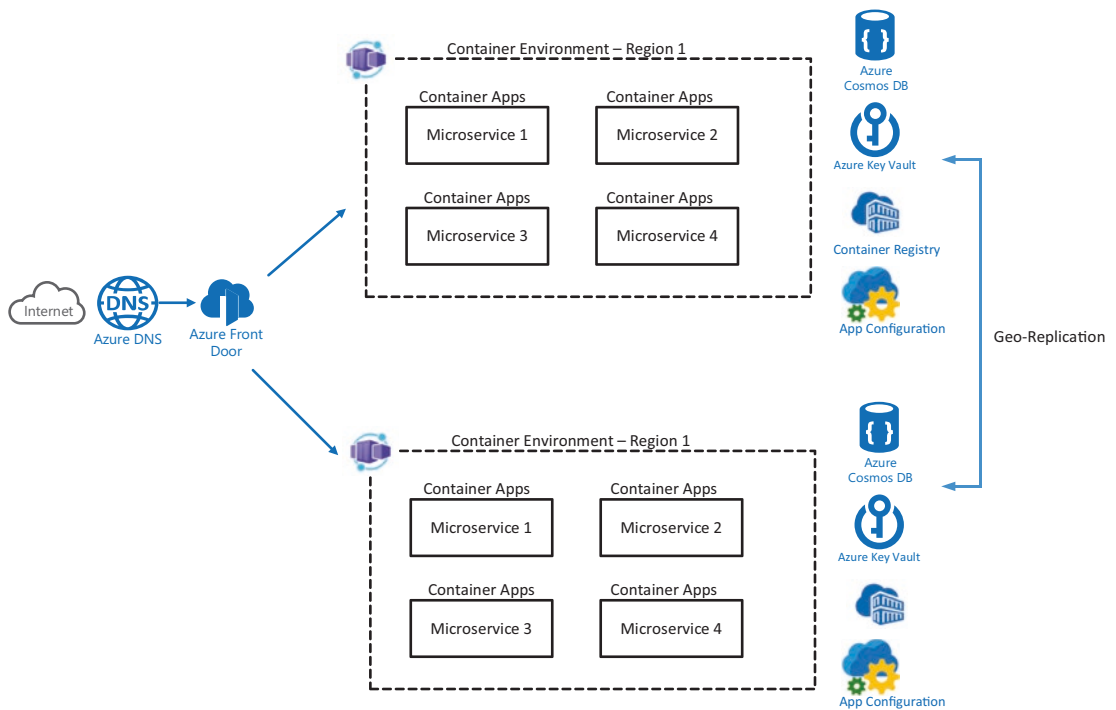
**Figure 4-8.** PaaS/Serverless resilient architecture

## Microservices Architecture

Complex architectures designed nowadays usually encapsulate business logic using microservices. Cloud Native Applications have gained popularity, and there are many solutions offered in the IT space, Docker being the most popular one for containerizing those services and Kubernetes being the most famous orchestrator in the market. Most of you would expect to see a Kubernetes-based architecture in this example, but I am

going to take another step. During the last months of 2021, Microsoft announced a new service called **Azure Container Apps** (<https://docs.microsoft.com/en-us/azure/container-apps/overview>). Azure Container Apps significantly reduces the complexity brought by solutions based on technologies like Kubernetes. It lets you forget about the underlying VMs and orchestrator and simplifies the cluster configuration, making it easier to update hosted microservices, define networking properties, or design autoscalable workloads.

Let's decompose the architecture shown in Figure 4-9.



**Figure 4-9.** *Microservices resilient architecture*

- Azure Container Apps:** As mentioned before, this service not only simplifies working on Kubernetes-based solutions by providing an infrastructure management layer (done by Azure Kubernetes Service - AKS) but also takes care of the Kubernetes configuration to apply: deployment and update of containers, networking properties, scaling, or secret management. No Kubernetes manifest files are needed; all can be done from ARM/Bicep files.

- **Environment:** Isolation boundary around a collection of container apps. Apps in different environments do not share resources (computing and networking) and cannot communicate with each other using DAPR. It takes care of the VNET setup (external/internal), but it also allows to use existing VNETs.
- **Container App:** This manages orchestration details for you (connection to container registry, container image to use, secrets, or traffic distribution). For scalability, it supports automatic scaling rules based on HTTP request, CPU/memory usage, or event-based rules using KEDA scalers. Health probes are also supported.
- **Azure Cosmos DB:** Explained in the previous section. It supports zone redundancy and geo-replication.
- **Azure Container Registry (ACR):** This is used for keeping the container images created by your Dev teams (using DevOps tools as shown in Chapter 5) and distributing those images (and updates) to the target environments, in this case, Azure Container Apps. The service also supports zone redundancy and geo-replication for higher resiliency and availability (only for Premium tier).
- **Azure App Configuration:** This is a service for centralizing configuration settings and being able to dynamically change application behavior without needing to restart/redeploy. It is also used for Feature Flags (covered in Chapter 5). The solution does not offer geo-replication, but you can deploy two instances (one in each region) and automatically sync both replicas using Azure Functions and Azure Event Grid: <https://docs.microsoft.com/en-us/azure/azure-app-configuration/howto-backup-config-store>.
- **Azure Key Vault:** This is the best place for securely storing and accessing secrets like passwords, certificates, or cryptographic keys. Key Vault content is automatically (no setup) replicated within the region and also in the Azure region pair. The service automatically fails over to another local or regional replica when an instance fails.
- **Azure Front Door and Azure DNS:** Covered previously, this is one of the load balancers that could be used.



## Testing Resiliency on Azure

The focus of the chapter was explaining the main ideas to design resilient architectures on Azure. But as you may imagine, resiliency and availability are not proven until tested. Remember, **availability** ensures uptime and **resiliency** measures how quickly your solution recovers.

Testing should happen on a regular basis; nothing should be just assumed. You don't want to have surprises on your production workloads.

**The last chapter** of the book will focus on showing some of the latest (both preview at the moment) Azure tools for resiliency testing: Azure Chaos Studio and Azure Load Testing.

## Summary

This chapter focused on explaining resiliency concepts based on Azure as the cloud provider. It covered the main resiliency patterns to include on both your infrastructure and application design, focusing mainly on infrastructure capabilities offered by Azure.

In addition, it covered the most frequently used recovery and resiliency metrics, which should be aligned with your business requirements.

Finally, it showed some architecture examples based on different service categories like IaaS, PaaS/Serverless, and Microservices.

## CHAPTER 5

# Automation to Enable SRE with GitHub Actions/Azure DevOps/Azure Automation

In this chapter, I will focus on the fourth pillar of Dickerson's hierarchy of reliability: automation (releases/testing). It will be a topic where SREs will need to work hand in hand with DevOps engineers in the organization. DevOps engineers will focus on continuous experimentation, whereas SREs will focus on the reliability of the solution, both by using shared tools and practices.

By the end of this chapter, you should be able to understand the following:

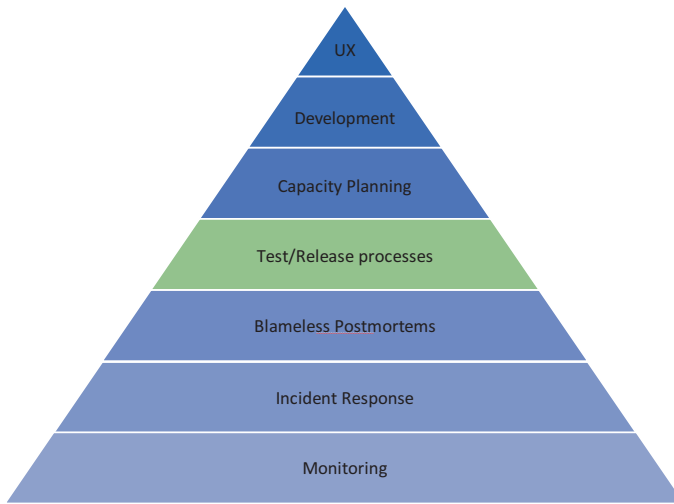
- ✓ What is DevOps and main concepts
  - CI and CD
  - Infrastructure and Configuration as Code
  - Shift-left testing
  - Secure DevOps
- ✓ Basics of GitHub Actions and Azure DevOps
- ✓ Modern deployment practices and tools

## Automation for SRE

This chapter will be covering the most important automation-related practices you need to at least “understand” as an SRE. Why such an emphasis on “understanding”? As discussed in the first chapter, DevOps and SRE practices sometimes do overlap, as this is the practice where the overlap happens more often. Your responsibilities as an SRE will vary from understanding to advising (or fully implementing) these automation approaches depending on your organization.

Organizations that are mature on DevOps practices will probably need the help of an SRE to focus more on the reliability of the releasing process (more complete automated testing approach or better deployment practices). On the other hand, nonmature DevOps organizations will require more help from the SRE of automation practices.

Remember, this chapter will mainly focus on the **testing/releasing** pillar of Dickerson’s hierarchy of reliability (see Figure 5-1).



**Figure 5-1.** *Dickerson’s hierarchy of reliability*

This chapter will focus on the three main areas of automation:

- **CI/CD DevOps processes:** How are the changes continuously built, tested (functional testing, quality, and security), and deployed to our services

- **Operational/desired state automation:** Covering options for executing runbooks and managing the desired state of virtual machines
- **Modern deployment practices:** Which will provide experimentation tools to DevOps teams as well as zero-downtime deployments (and more control) to SREs

## CI/CD Automation with DevOps

### What Is DevOps

DevOps has been one of the hottest topics in the last years. Development practices have been slowly moving from monthly (even yearly) releases to a few weeks. Even more, after all those months of effort, many times teams struggled by not meeting user expectations. Deployment of new code was “painful.”

Scrum, Agile, Kanban, and similar methodologies have completely shifted the way you develop software, enabling engineering teams to add incremental value very rapidly. In order to be able to deploy value to our product before the competition, you need a new way of deploying code changes.

DevOps shows us there is a better/faster/more reliable way of building/testing/deploying our code changes.

### Continuous Integration (CI)

Continuous Integration is the process that focuses on validation of code changes. Every time a change (new experimentation) is applied to your source code, you build, test (mainly unit testing at this point), and publish new code artifacts.

One of the pillars of Continuous Integration is your Version control (or Version Control) setup, defining the way we track and manage changes to software code. Choosing a proper branching strategy will be the key to experiment faster in a controlled way and move your proposed changes forward to more stable branches (and using CI/CD ► automatically deploy for validating in different environments).

Even if Source Control and Continuous Integration have been popular practices for developers and application code, nowadays, many IT professionals are encouraged to work with them. Any aspect that defines your running solution is encouraged to use Version Control and CI practices, for example:

- Application code
- Infrastructure
- Configuration
- Documentation
- Pipelines (mainly YAML)

Any change applied to any of these components could modify (or even break) your running solutions. Having a proper version control architecture is the key to experiment faster in a controlled way, with the help of version control technologies like Git and CI systems like Azure DevOps and GitHub.

## Continuous Delivery/Deployment (CD)

CD is the process focused on deployment practices and later testing practices like integration tests, UI testing, E2E testing, performance testing, etc. But what does CD really stand for?

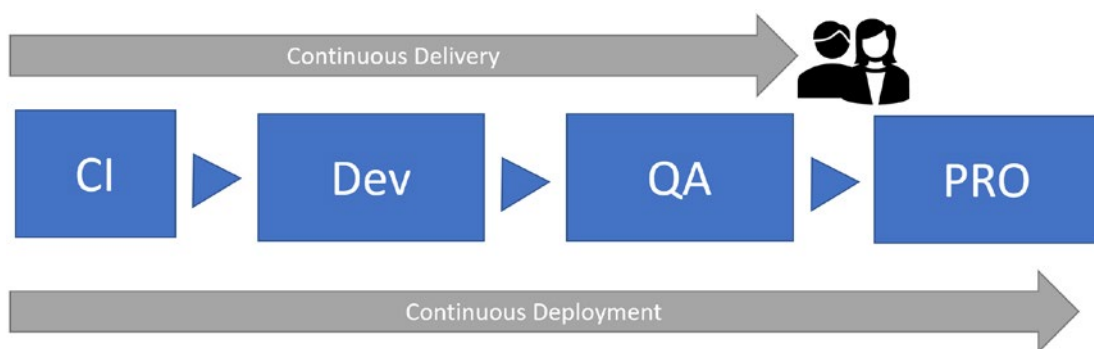
One of the first question I used to ask my customers during their DevOps transformation was the following:

“Are you looking forward to implement CD practices?”

Of course, the answer was mostly “Yes.” Then, my next question was the following:

“Do you want to implement Continuous Deployment or Continuous Delivery?”

A confused expression crossed their face. I am sure many were thinking “Isn’t it the same?” The answer is “No.” Let’s take the simple example shown in Figure 5-2.



**Figure 5-2.** *Continuous Delivery vs. Deployment*

- **Continuous Delivery:** The process that takes care of deploying newly created artifacts (mainly CI outcome) to desired environments. It will include some **manual** step in the process before the code change reaches the Production environment.
- **Continuous Deployment:** Full automation from a code change (commit) to production. It is more demanding as you will need to have proper automated testing mechanisms (unit, integration, UI tests, etc.) and modern releasing strategies in order to make sure code changes reach production work properly.

---

**Warning** Some organizations/teams switch the definition of these practices, making Continuous Delivery the full automation one. The most important outcome from this topic should be the following:

---

- Document the definition for your team/organization (common understanding).
- Clearly define the objective, as you have seen, going for the full automation will be more demanding.

## Shift-Left Testing in DevOps

Testing is a huge topic inside DevOps practices. Dickerson’s hierarchy of reliability mentions testing as one of the pillars for obvious reasons: it ensures code works as expected. Testing helps us identify bugs on our code before deploying it to production. Then, what is “shift-left testing”?

Shift-left testing means moving your testing efforts earlier on the life cycle, writing your test at the lowest level possible. Why? The following are the main reasons:

- The lower the level of the test ► the lower the impact (a bug is found earlier).
- Lower-level tests are more stable and run faster, the easiest way to get feedback around new functionality.
- Most testing is done before merging changes with the “main” branch.

A research from Ponemon Institute shows that early found vulnerabilities cost an average of \$80, compared to an average of \$7600 in Production. These numbers indicate the power of investing your testing efforts in unit/integration testing, rather than huge manual testing suites by the end of the life cycle (not fitting properly within fast-paced Agile/Scrum teams).

Take a look at the shift-left testing transformation taken by the Azure DevOps product team at Microsoft: <https://docs.microsoft.com/en-us/devops/develop/shift-left-make-testing-fast-reliable#a-case-study-in-shifting-left>.

Your team will need to define a proper testing taxonomy that specifies the right types of test to use in each scenario (CI/CD pipeline), based on dependencies or running time required. Take a look at this example taxonomy: <https://docs.microsoft.com/en-us/devops/develop/shift-left-make-testing-fast-reliable#test-taxonomy>.

Now let’s take a look at two different services you can use for fulfilling your CI/CD needs.

## Secure DevOps

Secure DevOps (also called Rugged DevOps or DevSecOps) is a practice that ensures security activities will be implemented through the full life cycle of your application. It focuses on bringing security practices the earliest possible on the life cycle of your solutions (shift-left security).

As a Site Reliability Engineer, this should be a practice you have to understand and encourage. In the end, catching vulnerabilities before they reach production will increase the reliability of our solutions, and that is something you need to fight for.

There are many different practices related to Secure DevOps; here, I include some of them:

- **Thread modelling:** An activity where the architecture of your solution is drawn and potential thread is analyzed, mitigated, and validated. It can be run from the planning phase; your solution could even be in the design phase (not even existing yet). Microsoft offers a free thread modelling tool: [www.microsoft.com/en-us/securityengineering/sdl/threatmodeling](https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling)
- **Automation-related tools:**
  - **CI:** During continuous integration, two main types of tools can be used:
    - **Software Composition Analyzers (SCA):** Will help in analyzing vulnerabilities and licensing restriction (for open source components) related to the libraries used by your application code (NuGet, Maven, npm, etc.). Tools like [WhiteSource](#) and [Dependabot](#) can be used with Azure DevOps and GitHub.
    - **Static Analyzers:** Will help in identifying vulnerabilities, bad practices, and duplicated code and even measuring the technical debt of your application code. Tools like [SonarCloud/SonarQube](#) and [CodeQL](#) can be used with Azure DevOps and GitHub.
  - **CD:** During the deployment phase, we can also run dynamic analyzers that will identify potential threads on your running environments. Tools like [OWASP ZAP](#) can be used to identify threads based on the Open Web Application Security Project (OWASP)-based recommendations.
  - **Monitoring/operation:** During the last phase, Azure tools like [Microsoft Defender for Cloud](#) can be used to detect and mitigate threads. It can be used in two different ways:



- **Reactive approach:** Resolving detected threads.
- **Proactive approach:** Based on the given recommendations for the existing architecture, create Azure Policies that will apply those best practices to new and existing environments.

## Infrastructure as Code (IaC)

High-performing DevOps organizations design their processes to achieve speed and agility. They surely provide automated build and release to make sure high-quality and consistent code gets deployed. But what about the environments where those systems run?

Infrastructure as Code (IaC) focuses on applying software engineering practices (like versioning, validating, or testing) to your infrastructure deployments in order to make them consistent, repeatable, and automated.

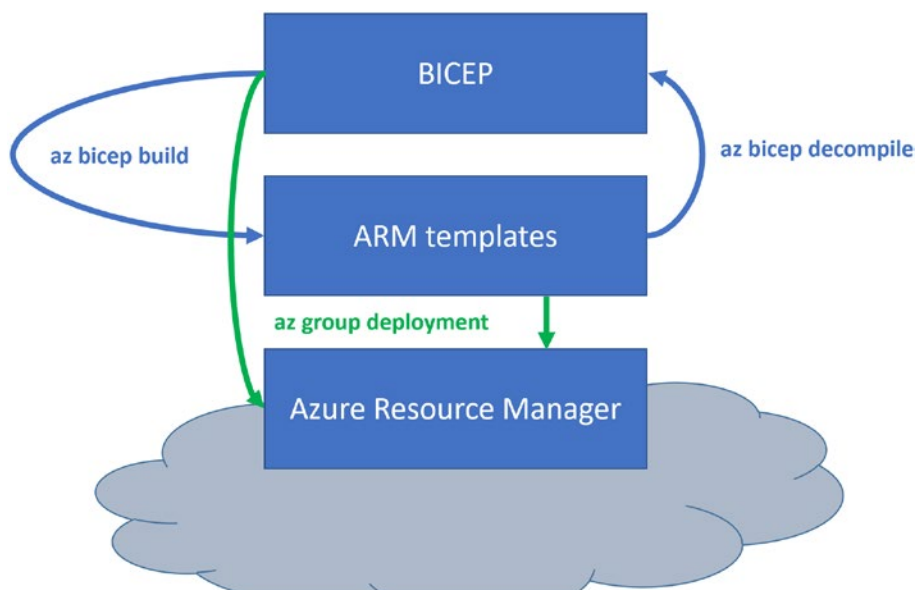
There are two different approaches to IaC:

- **Declarative:** You simply define how your infrastructure should look like (**WHAT**), and the cloud provider takes care of deploying the solution with such properties. Example tools:
  - **Azure native:** ARM templates or Bicep
  - **Third party:** Terraform, Ansible, Chef, Puppet, Pulumi, etc.
- **Imperative:** You are the one defining the steps to follow (**HOW**) to get the solution you are looking for. Example tools:
  - Azure CLI, Azure PowerShell, Azure Rest API, and Azure SDKs

No matter the tool you use, IaC process should aim for **idempotence**. What is idempotence? Simply put: the same IaC templates generate the same environments every time they are applied. Idempotence is the most important principle of IaC, key for automated CD pipeline successful execution. In other words, if the templates did not change, your pipelines will execute successfully, keeping the environments as it is. Based on previously mentioned tools, tools like **ARM/Bicep ensure idempotency**, whereas we could have **non-idempotent commands for imperative** tools (your script will need to be modified, include a condition to see if a resource/property exists, and only create/modify if it does not).

This book will focus on mainly showing both ARM templates and Bicep during the demos of the chapter.

- **ARM templates:** Azure Resource Manager is an exposed API service built in into Azure for provisioning purposes. You simply provide a JSON-based template with the resources (and configuration) you would like to provide. You can find many QuickStart templates at <https://azure.microsoft.com/en-us/resources/templates/>.
- **Bicep:** A new DSL (Domain-Specific Language) to deploy Azure resources by using declarative syntax. It provides an easier way to understand/create declarative templates with Azure, quite demanded by JSON “haters” 😊. It was designed to provide a “layer” on top of ARM templates to make IaC with Azure more intuitive; you can actually compile/decompile ARM↔Bicep templates (see Figure 5-3). Deployment can be done directly from Bicep or ARM files, for example, using *az group deployment* Azure CLI command.



**Figure 5-3.** *Bicep*

In the following, we can see the difference between an Azure App Service declared in ARM template and Bicep (see Listing 5-1). Bicep is more intuitive and easier to read and author.

**Listing 5-1.** ARM vs. Bicep**ARM Template**

```

{
  "$schema": "https://schema.management.azure.com/schemas/2019-04-01/
deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "metadata": {
    "_generator": {
      "name": "bicep",
      "version": "0.4.1008.15138",
      "templateHash": "13829464918613725405"
    }
  },
  "parameters": {
    "webAppName": {
      "type": "string",
      "defaultValue": "[uniqueString(resourceGroup().id)]"
    },
    "sku": {
      "type": "string",
      "defaultValue": "F1"
    },
    "linuxFxVersion": {
      "type": "string",
      "defaultValue": "node|14-lts"
    },
    "location": {
      "type": "string",
      "defaultValue": "[resourceGroup().location]"
    },
    "repositoryUrl": {
      "type": "string",
      "defaultValue": "https://github.com/Azure-Samples/nodejs-docs-
hello-world"
    },
  },

```

```

"branch": {
  "type": "string",
  "defaultValue": "master"
}
},
"functions": [],
"variables": {
  "appServicePlanName": "[toLower(format('AppServicePlan-{0}',
parameters('webAppName')))]",
  "webSiteName": "[toLower(format('wapp-{0}',
parameters('webAppName')))]"
},
"resources": [
  {
    "type": "Microsoft.Web/serverfarms",
    "apiVersion": "2020-06-01",
    "name": "[variables('appServicePlanName')]",
    "location": "[parameters('location')]",
    "properties": {
      "reserved": true
    },
    "sku": {
      "name": "[parameters('sku')]"
    },
    "kind": "linux"
  },
  {
    "type": "Microsoft.Web/sites",
    "apiVersion": "2020-06-01",
    "name": "[variables('webSiteName')]",
    "location": "[parameters('location')]",
    "properties": {
      "serverFarmId": "[resourceId('Microsoft.Web/serverfarms', variables
('appServicePlanName'))]",
      "siteConfig": {

```

```

        "linuxFxVersion": "[parameters('linuxFxVersion')]"
    }
},
"dependsOn": [
    "[resourceId('Microsoft.Web/serverfarms', variables('appServicePlanName'))]"
]
},
{
    "type": "Microsoft.Web/sites/sourcecontrols",
    "apiVersion": "2021-01-01",
    "name": "[format('{0}/web', variables('webSiteName'))]",
    "properties": {
        "repoUrl": "[parameters('repositoryUrl')]",
        "branch": "[parameters('branch')]",
        "isManualIntegration": true
    },
    "dependsOn": [
        "[resourceId('Microsoft.Web/sites', variables('webSiteName'))]"
    ]
}
]
}
}

```

### Bicep

```

param webAppName string = uniqueString(resourceGroup().id) // Generate
unique String for web app name
param sku string = 'F1' // The SKU of App Service Plan
param linuxFxVersion string = 'node|14-lts' // The runtime stack of web app
param location string = resourceGroup().location // Location for all
resources
param repositoryUrl string = 'https://github.com/Azure-Samples/nodejs-docs-
hello-world'
param branch string = 'master'
var appServicePlanName = toLower('AppServicePlan-${webAppName}')

```

```

var webSiteName = toLower('wapp-${webAppName}')
resource appServicePlan 'Microsoft.Web/serverfarms@2020-06-01' = {
  name: appServicePlanName
  location: location
  properties: {
    reserved: true
  }
  sku: {
    name: sku
  }
  kind: 'linux'
}
resource appService 'Microsoft.Web/sites@2020-06-01' = {
  name: webSiteName
  location: location
  properties: {
    serverFarmId: appServicePlan.id
    siteConfig: {
      linuxFxVersion: linuxFxVersion
    }
  }
}
resource srcControls 'Microsoft.Web/sites/sourcecontrols@2021-01-01' = {
  name: '${appService.name}/web'
  properties: {
    repoUrl: repositoryUrl
    branch: branch
    isManualIntegration: true
  }
}

```

You can find many QuickStart templates for both at <https://azure.microsoft.com/en-us/resources/templates/>. In addition, a great learning path can be found at MS Learn for Bicep at <https://docs.microsoft.com/en-us/learn/paths/fundamentals-bicep/>.

## Configuration as Code with DSC/Azure Automation/ Guest Configuration

The previous IaC topic was focused on deploying our desired infrastructure, but how about the configuration of those services we created? How do we make sure our Linux and Windows VMs have the configuration (OS settings, firewall rules, features, and software installed) in place?

Some could propose to just run a script right after creating those machines, but what if someone modifies the machine manually? We could be facing **configuration drift**: the machine not having the configuration we desire/expect. We would need to proactively run the same script ourselves.

Configuration as Code (CaC) consists of writing in a definition of the **desired state** you want your environment to have configured. It will do frequent consistency checks to make sure the desired state is met on a regular basis. In other words, configuration drift is detected and solved!

As part of the Microsoft tooling, we have a CaC engine called **Desired State Configuration (DSC)** as part of PowerShell (also called PowerShell DSC), available for Windows and Linux machines. Your machine needs to have PowerShell 4.0 or above installed. The service that will do the regular checks (based on the configuration) is called Local Configuration Manager (LCM).

When working with DSC, we are offered two working models:

- **Push model:** A DSC configuration file is applied on the selected VM (pushed), and it will take care to meet the configuration with regular inspections.
- **Pull model:** It involves using the **Azure Automation (<https://docs.microsoft.com/en-us/azure/automation/automation-dsc-overview>)** service and its State Configuration capabilities. It would work in the following way:
  1. Virtual machines are registered (and given a tag) on the Azure Automation account.
  2. PowerShell DSC configuration files are uploaded to the service and compiled. It will define the VMs (based on TAGs) where configurations should be applied.

3. Registered VMs will make sure of applying the latest configurations and being compliant to the latest setup. Configurations are rechecked every 15 minutes.

You can find many QuickStart DSC files on the Gallery feature of Azure Automation. The following example makes sure web server-tagged VMs will have an IIS web server installed (see Listing 5-2).

**Listing 5-2.** DSC example

```
configuration TestConfig
{
    Node IsWebServer
    {
        WindowsFeature IIS
        {
            Ensure          = 'Present'
            Name             = 'Web-Server'
            IncludeAllSubFeature = $true
        }
    }

    Node NotWebServer
    {
        WindowsFeature IIS
        {
            Ensure          = 'Absent'
            Name             = 'Web-Server'
        }
    }
}
```

Azure Automation can also be used for hosting and executing runbooks. **Runbooks** are scripts based on PowerShell (Graphical and Workflow) and Python, which can be executed based on schedule or webhook call. The runbooks should be used for **operational automation** (not CI/CD), for example, running scripts every day for start/stop of virtual machines or executing a script as a reaction to a specific alert being triggered (using Azure alert Action Groups).



## Azure Policy Guest Configuration

Azure released a new service called **Guest Configuration** (<https://docs.microsoft.com/en-gb/azure/governance/policy/concepts/guest-configuration>), providing audit and Configure-as-Code capabilities based on the Azure Policy service. The feature will be able to work with Azure and Arc-enabled virtual machines, offering the option to apply settings machine by machine or orchestrating using Azure Policy.

The service is using both PowerShell DSC and Chef InSpec on the back end to apply configurations on the target Linux and Windows VMs. New guest assignments are evaluated every 5 minutes and rechecked every 15 minutes. Multiple configurations can be applied to the same machine.

A tutorial to migrate from Azure Automation to Guest Policy is provided by Microsoft (<https://docs.microsoft.com/en-gb/azure/governance/policy/how-to/guest-configuration-azure-automation-migration>).

## Azure Pipelines

Azure DevOps is a product that provides many development-related services to customers, such as planning, collaboration, CI/CD, package management, and testing tools.

Azure Pipelines (<https://docs.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops>) is the service integrated on Azure DevOps that offers support for the Continuous Integration and delivery/deployment of your applications. But how does it work? The service is composed of many different components/parts; let's try to define them in simple questions:

### Where does the pipeline run?

Azure Pipelines is executed/triggered from the Azure DevOps service, but the defined automated tasks run on **agents**. What is an agent?

It is the computing infrastructure (mainly VM or container) that will be used to run pipeline **jobs**. It is called agent due to the software installed host machine to be able to listen to Azure DevOps service. Jobs can run on the agent's host machine or container.

Microsoft offers two main agent types:

- **Microsoft-hosted agents:** A group of agents maintained and upgraded by Microsoft. Offered both as VM and container, you get a fresh machine for each job. This machine comes with predefined

software installed: <https://github.com/actions/virtual-environments> (first “secret,” the same “agents” are used by GitHub Actions).

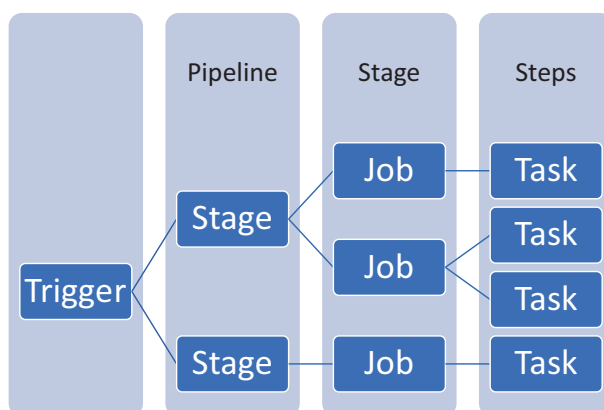
- **Self-hosted agents:** Due to networking, computing, or software needs, you may prefer to use your private infrastructure hosts. This option has been long available in the previous on-premises version of Azure DevOps named Team Foundation Server. It is also the only agent option in the actual on-premises Azure DevOps Server solution.
- It gives more control over networking, computing, cache, or software needs, but you need to maintain it. The agent (software) is offered for the three main operating systems: macOS, Linux, and Windows. You can also install it using Docker.

### Where/how do I define the automated task I want to run?

Azure DevOps offers two different ways to define your pipelines: YAML and Classic (also called Visual). In this book, we will focus on YAML, as the new features are mainly coming for this pipeline model.

YAML pipelines are defined in code on a file that is hosted on a GIT repository. It will have to follow the standard practices defined for source control (branching, pull request, etc.) and will benefit from this development practice in terms of collaboration, control, and change tracking. It also has a great advantage in terms of modularizing (creating templates) compared to Classic pipelines.

A pipeline is composed of the following components (a demo will be shown later) (see Figure 5-4):



**Figure 5-4.** Pipeline structure

- **Pipeline:** Context that defines the process; it can be CI, CD, or both. Pipelines can be linked to one another; not all the process has to be written in a single file (it could be). The pipeline will define **triggers** that will execute it (branch changes, pull requests, other pipeline successfully executed, etc.).
- **Stage:** It is a way of organizing the pipeline jobs. It is an optional feature; you could have pipelines only composed of jobs. Mainly used for separation of concerns, for example:
  - infrastructure deployment ► Website Publish ► E2E testing
  - build CI ► QA CD ► PRO CD

Stages and jobs do not have to be a linear process; you are free to design your flow with conditions and dependencies, linear or not.

- **Job:** A series of tasks you run sequentially inside the defined **agent** (explained before).
- **Steps:** Collection of tasks.
- **Task:** Prepackaged script to perform an action.
- **Other generic components:**
  - **Artifacts:** A mechanism to share files between jobs (remember, its job runs on a different agent selected from a pool) and a way to persist a result (e.g., build artifacts for each pipeline execution).
  - **Variables:** Variables can be defined for different scopes of the pipeline. They can also be defined on the UI. Variable groups can be used to make them available across pipelines. **System variables** are really useful too: <https://docs.microsoft.com/en-us/azure/devops/pipelines/build/variables?view=azure-devops&tabs=yaml>.
  - **Environments:** A reference made to your services (web apps, containers, virtual machines, etc.). It can be used to apply control on YAML pipelines that are referencing a predefined environment.

- **Service connection:** Authenticated connections to external or remote services that **task** needs to use. If your task want to use tools like SonarCloud for static analysis, check out code from GitHub or deploy/interact with Azure resources; they need to use authenticated connections on your pipelines. Simply put, the pipeline executed on an agent (Microsoft or private host machine) needs to know how to authenticate against those external components in order to be able to execute tasks successfully.
- **Checks and approvals:** The way to apply control, automated or manual, on a multistage YAML pipeline. What kind of controls? For example:
  - Manual approval (reviewer), business hours, evaluating artifacts, invoking Rest API calls, running Azure Functions, checking Azure alerts, or forcing the YAML template (process) the environment needs to use

It gives us great tools to make sure the change is ready to be applied. It can be applied in the following components:

- Agents pools
- Service connection
- Environments

Whenever one of those components is referenced on the YAML pipeline, Azure DevOps will look for checks and approvals link to it in order to fulfill those needs before following the execution of tasks.

### **Is it free? What is the pricing model?**

Azure DevOps offers the following pricing model: <https://azure.microsoft.com/en-us/pricing/details/devops/azure-devops-services/>.

For Azure Pipelines, the pricing is built around parallelism (how many jobs can we run in parallel). Private projects will get 1800 minutes/month of Microsoft-hosted agents and single parallel pipeline for Microsoft-hosted and self-hosted agents. When more parallel pipelines are purchased, the time limit disappears.

Public projects will get ten Microsoft-hosted parallel pipelines and unlimited minutes.

---

**Warning** You may need to fill out the form mentioned in this article to get free resources (due to a suspicious activity tracked lately on the product, mainly crypto mining): <https://devblogs.microsoft.com/devops/change-in-azure-pipelines-grant-for-public-projects/>.

---

## [DEMO] CI/CD Multistage YAML Pipeline

The following demo will be based on the following public repository from GitHub: [https://github.com/unaihuete-org/SRE\\_with\\_azure\\_devops\\_yaml](https://github.com/unaihuete-org/SRE_with_azure_devops_yaml). You need to fork/import it in your Azure DevOps project.

It contains

- A simple .NET core 3.1 website
- ARM templates to deploy Azure App Service
- Azure Pipelines YAML template

The demo will explain the previously described terms/concepts in a practical way. As we want to deploy our solution to a target Azure subscription, we will begin by creating a **Service Principal** that will be used to authenticate from our pipelines to the target Resource Group. First, make sure you have created the Resource Group to be used during the demo. What is a Service Principal?

It is a component offered by Azure Active Directory to offer applications (in this case, Azure Pipelines) as a way to authenticate against Azure environments. Normally, the Service Principal gets a role assigned (RBAC), which defines the permissions given to it once authenticated. Let's begin by creating a Service Principal that has a contributor role for an empty Resource Group (created for the demo). Open the Azure Portal ► Cloud Shell and run the following command:

```
az ad sp create-for-rbac -n ADO-pipelines --role contributor --scope /subscriptions/<SUBSCRIPTIONID>/resourceGroups/<RESOURCEGROUP-NAME> --sdk-auth
```

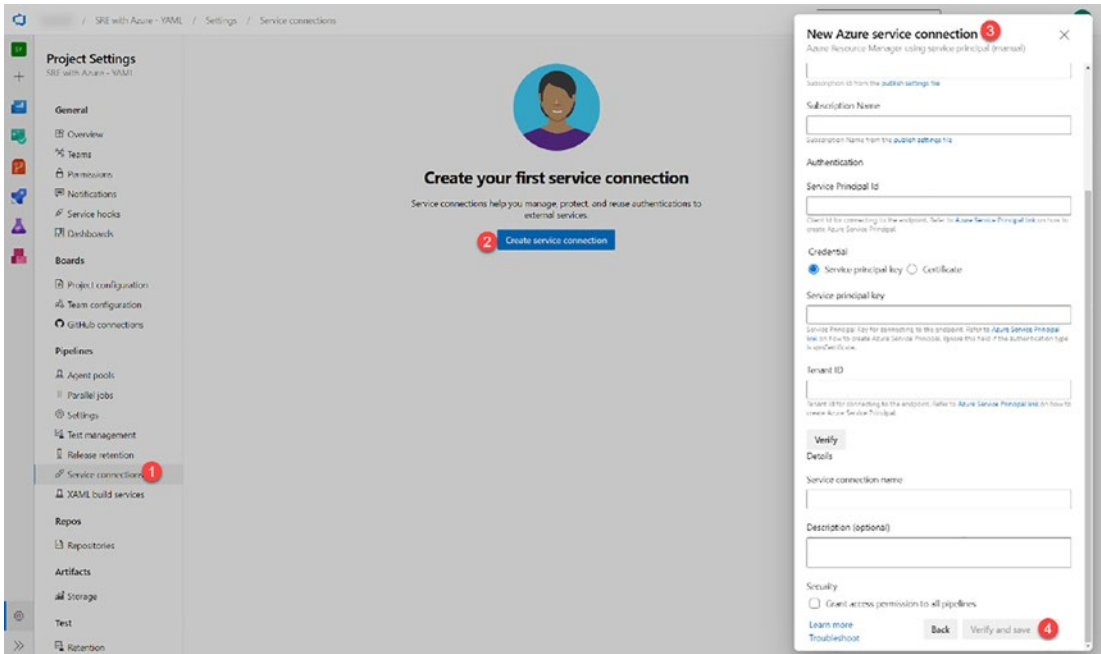
If the command executes successfully, it will output a JSON object with the main properties of the Service Principal. We mainly need the following four properties:

- ClientId (also known as Service Principal Id)
- SubscriptionId
- TenantId
- ClientSecret

**Warning** This information has to be treated really carefully; anyone with this information could impersonate the app and get access to our Azure environment!

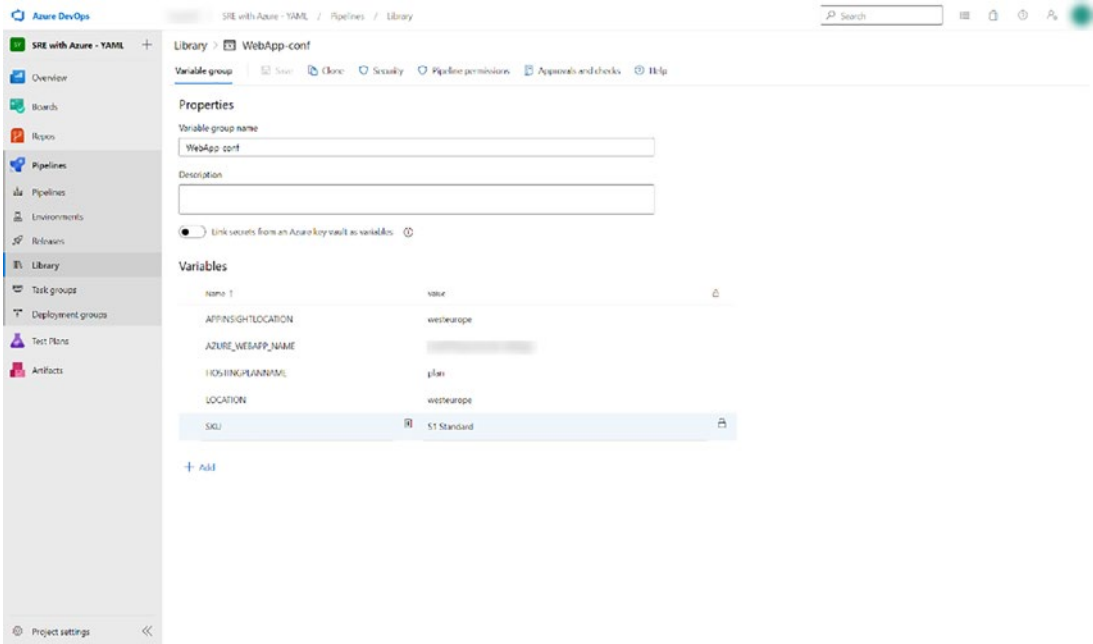
Now let's create a **Service Connection** in Azure DevOps (see Figure 5-5) in order to use the recently created Service Principal in our pipelines. Go to *Azure DevOps* ► *Project Settings* ► *Service Connections (under Pipelines)*, and click *Create Service Connection*. Choose the option *Azure Resource Manager* ► *Service Principal (manual)* and click *Next*.

Fill in the fields with the information taken from the previous JSON object (you also need the subscription name). Give the connection a name (e.g., *azure-sre-rg*), and check *Grant access permission to all pipelines* and *Verify and Save*.



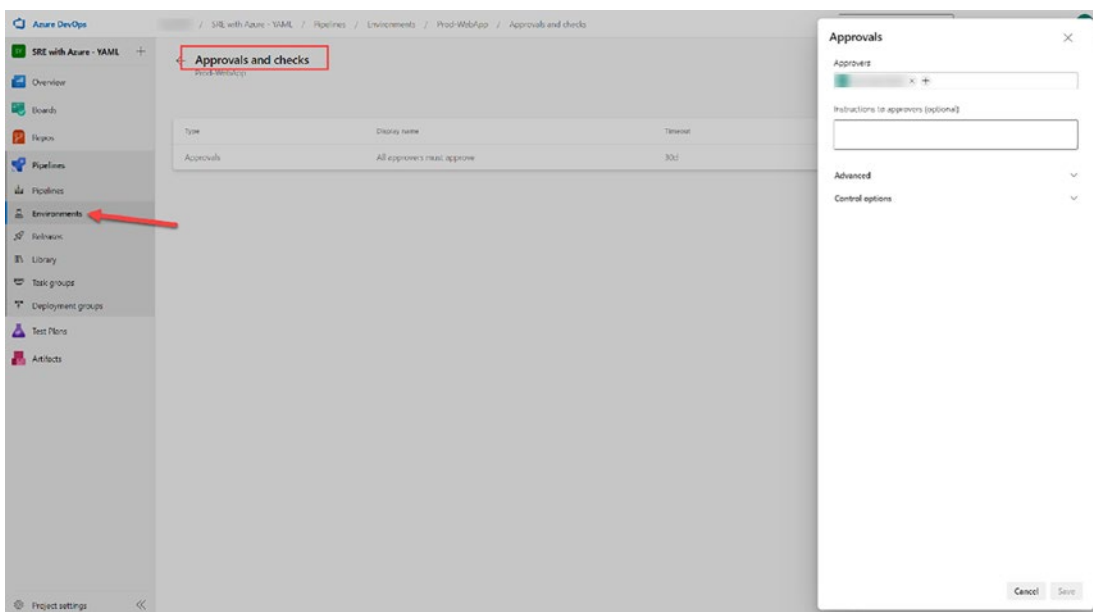
**Figure 5-5.** Service connection

Our pipeline will also use some variables, in our case, **Variable Groups** (see Figure 5-6). If we have sensitive data (like passwords, access keys, or connection strings), we need to provide to pipelines, variables offer a linking option to Azure Key Vault.



**Figure 5-6.** Variable group

We will also make use of **environments**. As explained before, environments provide features like **checks and approvals** for automated/manual control flows (see Figure 5-7). Our pipeline will ask for manual approval before web app publish happens.



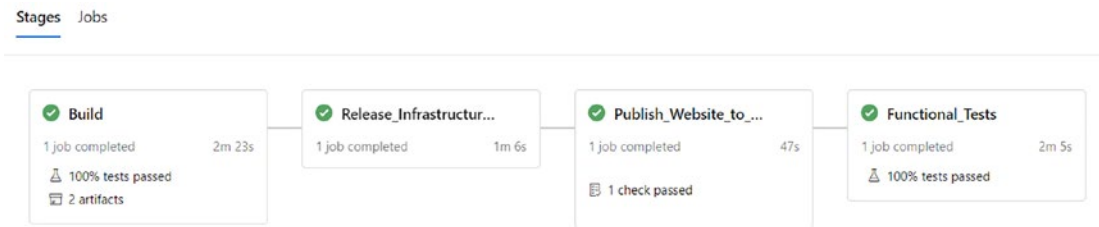
**Figure 5-7.** ADO environments

Now let's take a look at the YAML pipeline and explain it.

- **Trigger:** The pipeline is automatically executed for any change applied on the master branch.
- **Variables:** The pipeline will use “local” variables and the variable group (WebApp-conf) defined before.
- **Stages:** It is composed of four stages (see Figure 5-8):
  - **Build:** It takes care of building, unit testing, and publishing the .net core website. It creates two pipeline artifacts to be used later: ARM templates and published website. We can see in the following the *100% test passed* and *2 artifacts* created.
  - **Release\_Infrastructure\_to\_Azure:** It downloads the ARM templates from the first stage and deploys an Azure WebApp to Azure based on the ARM templates used.
  - The ARM deployment task uses the **Service Connection** created before and overrides template parameters using **Variable Groups**.



- Publish\_Website\_to\_WebApp:** It will use the **environments** feature defined previously and ask for manual approval (see *1 check passed* shown in Figure 5-8). When approved, it downloads the published website artifact from the first step and publishes the solution to the selected Azure App Service using the previously defined **Service Connection**.
- Functional\_Tests:** It will run functional tests against the running environment using Selenium.



**Figure 5-8.** Pipeline stages

**Listing 5-3.** Sample multi-stage YAML pipeline definition

```
trigger:
- master

variables:
- name: buildConfiguration
  value: Release
- group: WebApp-conf

stages:
- stage: Build
  jobs:
- job: Build
  pool:
    vmImage: 'windows-latest'
  steps:
- task: DotNetCoreCLI@2
  displayName: Restore
```

```

  inputs:
    command: restore
    projects: '**/*.csproj'

- task: DotNetCoreCLI@2
  displayName: Build
  inputs:
    projects: '**/*.csproj'
    arguments: '--configuration $(BuildConfiguration)'

- task: DotNetCoreCLI@2
  displayName: Test
  inputs:
    command: test
    projects: '**/*UnitTests/*.csproj'
    arguments: '--configuration $(BuildConfiguration)'

- task: DotNetCoreCLI@2
  displayName: Publish
  inputs:
    command: publish
    publishWebProjects: True
    arguments: '--configuration $(BuildConfiguration) --output
"$(build.artifactstagingdirectory)"'
    zipAfterPublish: True

- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(Build.ArtifactStagingDirectory)'
    ArtifactName: 'Website'
    publishLocation: 'Container'

- task: PublishBuildArtifacts@1
  inputs:
    PathtoPublish: '$(System.DefaultWorkingDirectory)/ArmTemplates'
    ArtifactName: 'Templates'
    publishLocation: 'Container'

```

```

- stage: Release_Infrastructure_to_Azure
  dependsOn: Build
  jobs:
  - job: Release_Infrastructure_to_Azure
    pool:
      vmimage: 'windows-latest'
    steps:
    - checkout: none

    - download: current
      artifact: Templates

  - task: AzureResourceManagerTemplateDeployment@3
    inputs:
      deploymentScope: 'Resource Group'
      azureResourceManagerConnection: 'azure-sre-rg'
      subscriptionId: 'YOUR-SUBSCRIPTION'
      action: 'Create Or Update Resource Group'
      resourceGroupName: 'SREwithAzure-ADOYAML'
      location: 'West Europe'
      templateLocation: 'Linked artifact'
      csmFile: '$(Pipeline.Workspace)\**\*.json'
      overrideParameters: '-webAppName $(AZURE_WEBAPP_NAME)
        -hostingPlanName $(HOSTINGPLANNAME) -appInsightsLocation
        $(APPINSIGHTLOCATION) -sku "$(SKU)'"
      deploymentMode: 'Incremental'

- stage: Publish_Website_to_WebApp
  dependsOn: Release_Infrastructure_to_Azure
  jobs:
  - deployment: VMDeployment
    displayName: "Publish Website to Azure App Service"
    environment:
      name: "Prod-WebApp"
    strategy:
      runOnce:
        deploy:

```

```

  steps:
  - checkout: none

  - download: current
    artifact: Website

  - task: AzureRmWebAppDeployment@4
    inputs:
      ConnectionType: 'AzureRM'
      azureSubscription: 'azure-sre-rg'
      appType: 'webApp'
      WebAppName: '$(AZURE_WEBAPP_NAME)'
      packageForLinux: '$(Pipeline.Workspace)/**/*.zip'

- stage: Functional_Tests
  dependsOn: Publish_Website_to_WebApp
  jobs:
  - job: Functional_Tests
    pool:
      vmImage: 'windows-latest'
    steps:
    - task: replacetokens@5
      inputs:
        targetFiles: '**/*.runsettings'
        encoding: 'auto'
        tokenPattern: 'rm'
        writeBOM: true
        actionOnMissing: 'warn'
        keepToken: false
        actionOnNoFiles: 'continue'
        enableTransforms: false
        enableRecursion: false
        useLegacyPattern: false
        enableTelemetry: true

```

```
- task: DotNetCoreCLI@2
  inputs:
    command: 'test'
    projects: '$(System.DefaultWorkingDirectory)/Application/aspnet-core-dotnet-core.FunctionalTests'
    arguments: '-s $(System.DefaultWorkingDirectory)/Application/aspnet-core-dotnet-core.FunctionalTests/functionalTests.runsettings'
```

If the pipeline is executed successfully, you should see the resources on the target Azure Resource Group and the published WebApp/App Service (see Figure 5-9 and Figure 5-10).

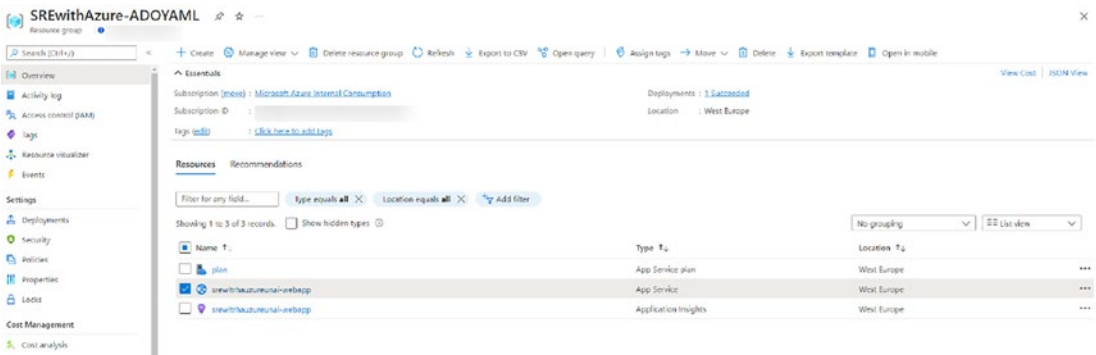
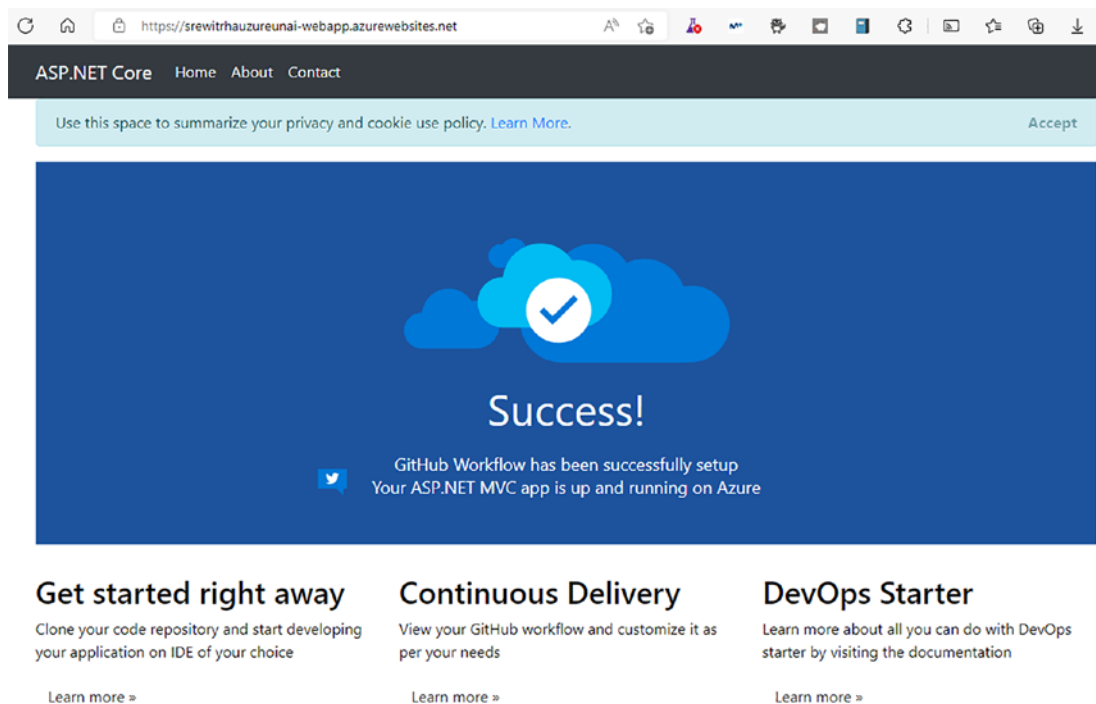


Figure 5-9. Azure Resource Group



**Figure 5-10.** Deployed WebApp

## GitHub Actions

During the last years, GitHub has been the home for millions of developers, mostly famous for offering version control services for Git repositories.

Microsoft acquired GitHub in June 2018 (<https://news.microsoft.com/2018/06/04/microsoft-to-acquire-github-for-7-5-billion/>), and few months later, on the GitHub Universe event in 2018, GitHub Actions was announced as a new automation platform (<https://github.blog/2018-10-17-action-demos/>).

Coincidence? I do not think so 😊.

Azure DevOps has been a DevOps/ALM tool providing many development life cycle services (like planning, repository hosting, testing, and CI/CD tooling) for many years already. It started long time ago with the on-premises *Team Foundation Server* solutions. GitHub has been the home for developers and especially open source communities for years too, but it had a big gap: no CI/CD service offered.

GitHub Actions has been quickly onboarding many of the Azure Pipelines YAML features (no classic/visual offered), and due to their similarities, I will try to explain it with the following comparison table (see Table 5-1).

Table 5-1. Azure DevOps and GitHub Actions

Azure DevOps	GitHub Actions
<p><b>YAML</b> and Classic pipelines</p>	<p><b>YAML workflows</b> (workflow=pipeline) need to be located under the .github/workflow folder                      Workflow syntax: <a href="https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions">https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions</a></p>
<p><b>Pricing</b> based on parallelism</p>	<p><b>Pricing</b> based on execution minutes (<a href="https://github.com/pricing/calculator#actions">https://github.com/pricing/calculator#actions</a>) Parallel/concurrent jobs based on license, starting with 20 for Free plans (<a href="https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration#usage-limits">https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration#usage-limits</a>)</p>
<p><b>Agents:</b></p> <ul style="list-style-type: none"> <li>• Microsoft-hosted</li> <li>• Self-hosted</li> </ul>	<p>Called <b>Runners</b> in GitHub</p> <ul style="list-style-type: none"> <li>• GitHub-hosted</li> <li>• Self-hosted</li> </ul>
<p><b>Pipeline</b></p> <ul style="list-style-type: none"> <li>• <b>Stage/s</b></li> <li>◦ <b>Job/s</b></li> <li>■ <b>Steps</b></li> <li>• <b>Tasks</b></li> </ul>	<p><b>Note:</b> The same machines used for both platforms on the back end: <b>GitHub runners</b> and <b>Azure pipelines agent repo:</b> <a href="https://github.com/actions/virtual-environments">https://github.com/actions/virtual-environments</a></p> <p><b>Pipeline</b></p> <ul style="list-style-type: none"> <li>• <b>Job/s:</b> a mix of Job/Stage from Azure DevOps, both visually on the UI and execution flow</li> <li>◦ <b>Steps:</b> collection of actions</li> <li>■ <b>Actions:</b> the name GitHub gives to tasks</li> </ul>

**Triggers**

Triggers defined with an **On** element on workflow. One of the biggest differences is that you need to include **workflow\_dispatch** on the YAML definition for manual trigger to be possible.

**Variables**

Named **Environment variables** in GitHub, it also offers **default environment variables** (similar to system variables in Azure DevOps): <https://docs.github.com/en/actions/learn-github-actions/environment-variables#default-environment-variables>

**Variable Groups**

There is nothing exactly similar; you can use **GitHub Secrets** (for secrets or not) to share variables at the organization or repository level

**Environments**

Also called **Environments** but it misses the automated control flows offered by Azure DevOps

**Tasks:** mainly from marketplace (also custom ones can be created)

**Actions:** two variants offered on the YAML template:

- **Use:** predefined actions (marketplace or private). Based on JavaScript or Docker
- **Run:** directly executes shell commands on runner (on Azure DevOps, e.g., you need to use the specific task for PowerShell/Bash )

**YAML Templates**

GitHub also offers a mechanism to reuse workflows for a modularized approach. Reusing workflows: <https://docs.github.com/en/actions/using-workflows/reusing-workflows>

<https://docs.microsoft.com/en-us/azure/devops/pipelines/process/templates?view=azure-devops>

**Pipeline Artifacts**

Similar functionality offered with **Workflow Artifacts**: <https://docs.github.com/en/actions/using-workflows/storing-workflow-data-as-artifacts#uploading-build-and-test-artifacts>



These basic concepts will be demoed after the next section, as I want to be using GitHub Actions together with some of the modern deployment practices that will be covered next.

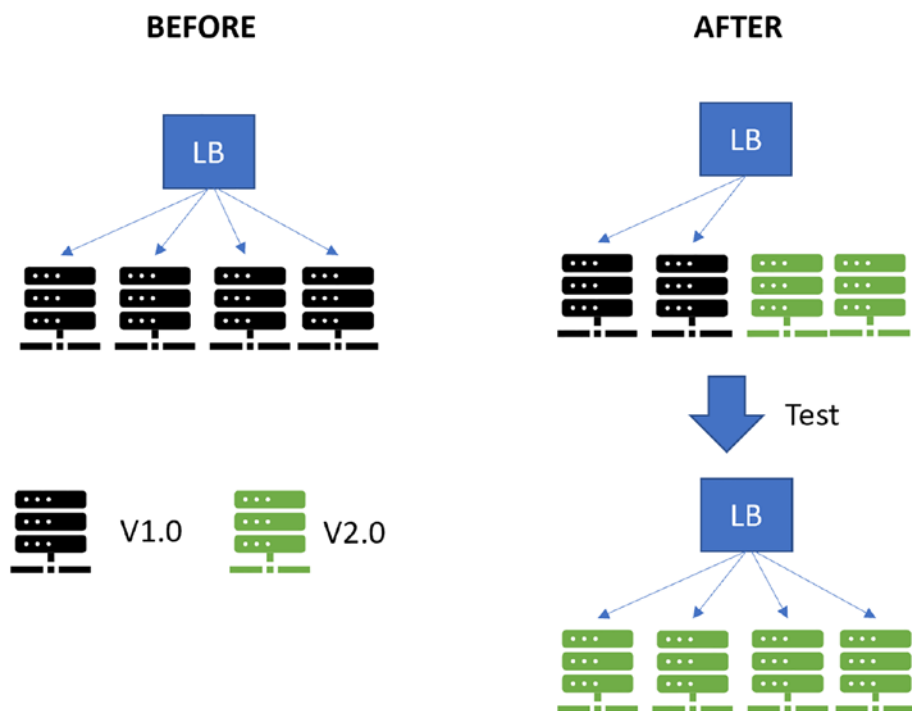
## Modern Deployment Strategies

Deployment strategies define the process to change/upgrade running instances of an application. Solutions nowadays, influenced by the agility introduced by Agile/Scrum methodologies, need to find ways to deploy updates more frequently without disrupting end users (zero downtime deployments). Furthermore, engineering teams are constantly striving to provide value to the end solution (remember the DevOps definition by Donovan Brown), and testing in production will be a critical practice in order to get valuable feedback.

On the other hand, as a Site Reliability Engineer, testing in production sounds scary, right? I will show how the following deployment practices will provide mechanisms to deploy updates with zero downtime and have control over the released features during runtime. Two main objectives will be defined: zero downtime deployments and testing in production (or production-like) in a safe way.

## Rolling Deployment

Rolling deployment is based on updating the instances of an application in an incremental way, node by node (see Figure 5-11). It is frequently used in VM-based solutions. Imagine our front-end solution is composed of four web server VMs. We could define a pipeline that gradually updates instances, for example, 50% first, and test it and the missing 50% later (modifying the load balancer to only point the selected VM instances).



*Figure 5-11. Rolling deployments*

As an example, check the tutorial for rolling deployments using Azure Pipelines YAML: <https://docs.microsoft.com/en-us/azure/virtual-machines/linux/tutorial-build-deploy-azure-pipelines?tabs=java>.

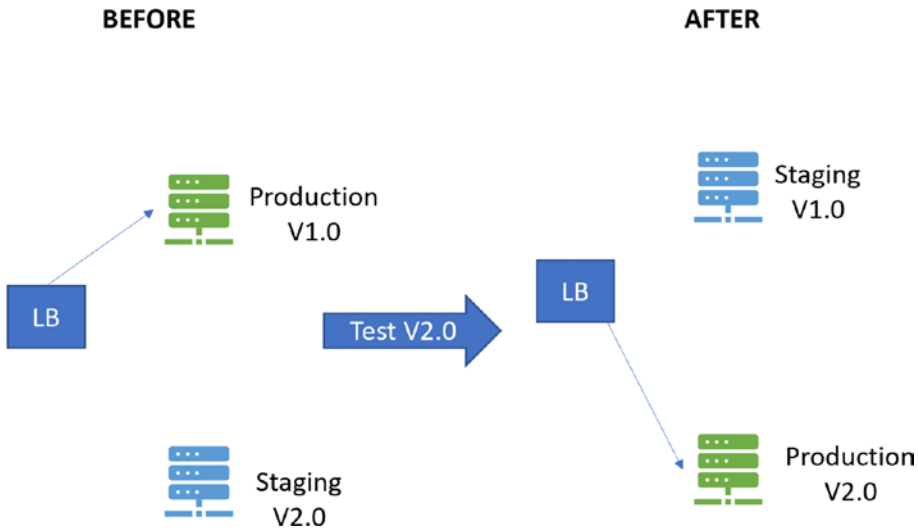
## Blue-Green Deployment

Blue-green deployment is a strategy based on having two identical production-like environments: staging (blue) and production (green).

Before any update, both instances are running the same code version, but only the “blue” one gets user traffic. When an update wants to be released to production, the process works in the following way (see Figure 5-12):

1. New code version is published to “staging” instance. QA teams (or automated E2E/UI tests) make sure the new version works fine.
2. Once “staging” is tested, traffic is shifted. Once deployment is successful, roles are switched too.

- Some people like to keep the “blue” one on the previous version in case rollback is needed.



**Figure 5-12.** Blue/green deployments

This strategy can be fulfilled in multiple ways (depending on the architecture used for your solution), but it is mainly based on having a load balancer that routes traffic to the two identical instances. In Azure, App Services offer a service called **slots** for creating identical app services that can be easily **swapped**; Azure takes care of the load-balancing needs, making it really easy to implement.

Azure App Service Slots: <https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots?msclkid=585ad452d07811ec988c98ced902b383>

## Feature Flags

Feature Flags (also called Feature Toggles, Feature Switches, or Conditional Features) are not a deployment strategy. They are actually a tool that will enable us to run deployment strategies mentioned in the following.

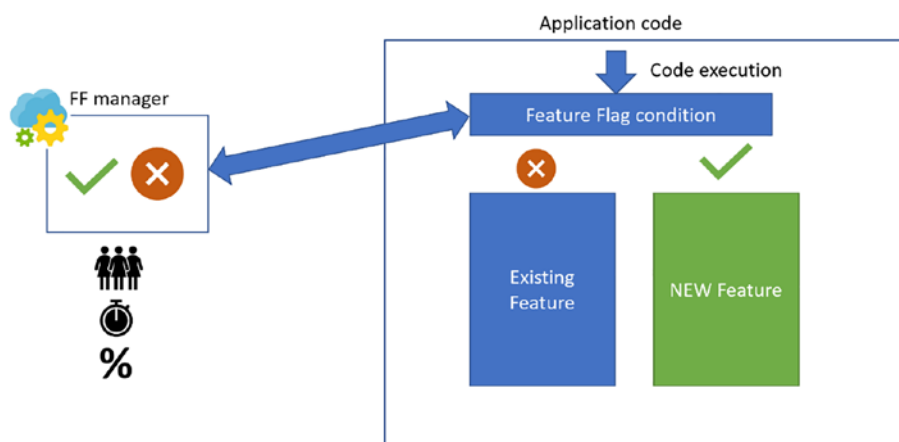
Feature Flags are a modern technique that enables you to deploy new features to production but restricts their exposure. They will enable us to continuously deploy changes to production but:

- Have the control to disable features if they are not executing as expected (no immediate rollback needed!)

- Just enable it to specific users (other filters supported too) for experimenting
- Disable certain features during high resource consuming periods

Feature Flags consist of two main components (see Figure 5-13):

- **Feature Flag manager:** The service used to decide what features will be enabled/disabled, optionally based on filter like random percentage of traffic, specific users, or schedule based.
- Your application code making a call to the Feature Flag manager in order to show/hide the new feature. As you can see, developers will need to use the SDKs provided by the Feature Flag manager tools.



**Figure 5-13.** Feature Flags

In Azure, a new service called **Azure App Configuration** (<https://docs.microsoft.com/en-us/azure/azure-app-configuration/overview>) was released, offering a centralized repository for flags (also used for centralized/dynamic configuration). It offers SDKs for languages like .NET, Java, Python, and JavaScript. An example will be shown in the demo of this section.

Feature Flags need to be maintained, as we can increase the technical debt of our solutions by leaving “dead” code in place. Take a look at the following article written by Martin Fowler: <https://martinfowler.com/articles/feature-toggles.html>.

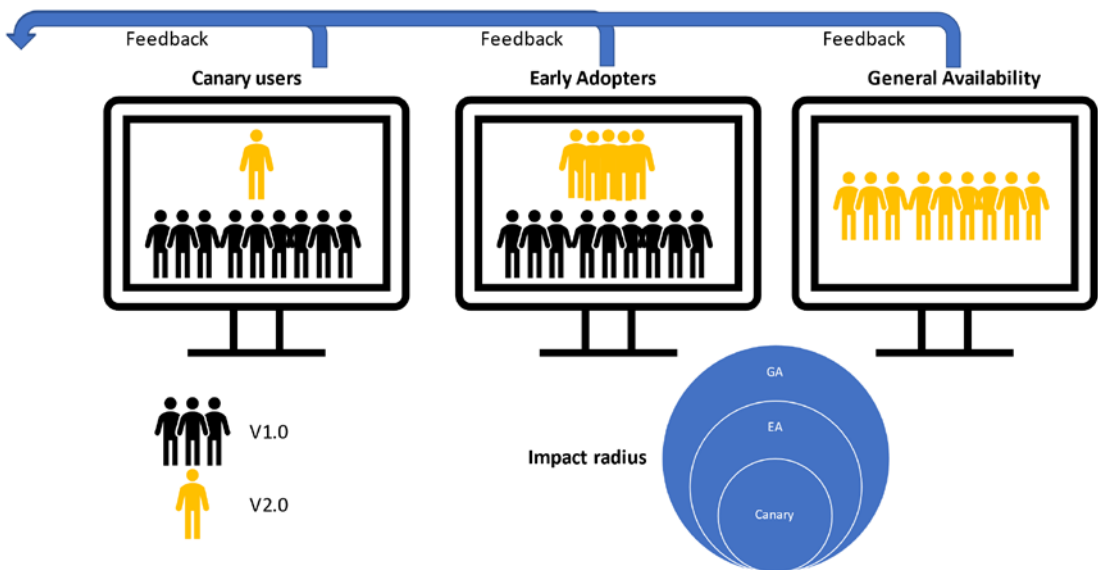
## Canary Deployments/Ring-Based Deployment

Canary deployment strategy is based on releasing a service incrementally to a subset of “voluntary” users in production. Users need to be aware they are going to be testing a “bleeding” version of code; their main purpose is to give feedback about recently exposed features in production.

Canary strategy can be fulfilled in multiple ways, depending on the architecture, based on load-balancing techniques, deployment slots, or feature flags.

It will be cheaper than blue-green (we do not need two identical production environments) but also more complex to implement as we are testing in production directly.

Based on feedback given by a canary group of users, features will be exposed to the next bigger group of users, usually called Early Adopters. Early Adopter is a group that should be expecting more stable features to test. **Ring-based** deployment is the strategy based on rolling the solution phase by phase, incrementing the amount of users “affected” by the change and getting valuable feedback from each ring. The idea is to slowly increment the “impact radius” (affected people) based on feedback from each ring (see Figure 5-14).



**Figure 5-14.** Ring-based deployments

## Dark Launching

This strategy is based on a similar idea as the canary strategy, but in this case, the users are not aware of “being used” to test the new features. Feedback will be gathered based on monitoring techniques mainly.

## A/B Testing

Compared to the previous strategies, A/B testing is focused on helping us decide between **two (or more) variants** of a new feature. Experiments can be implemented by using Feature Flags (showing option A or B based on the authenticated user or random traffic), traffic routing, or distinct deployments.

For example, let’s imagine we own an e-commerce website that only supports credit card payment. We are thinking on either supporting Apple/Google Pay or PayPal; for some reason, we cannot choose both. We could test the solution with both options, showing option A to some users and option B to others. By the end of the experiment, we should analyze the option that had a bigger impact (see Figure 5-15).



**Figure 5-15.** A/B testing

This strategy, compared to the other ones, is 100% focused on experimentation.

## [DEMO] Modern Deployments with GitHub Actions and Azure App Configuration

I will be using the following demo to show some of the recently explained strategies with GitHub Actions. The demo code can be found in the following GitHub repository: [https://github.com/unaihuete-org/SRE\\_with\\_Azure](https://github.com/unaihuete-org/SRE_with_Azure).

The demo will be using the architecture shown in Figure 5-16.

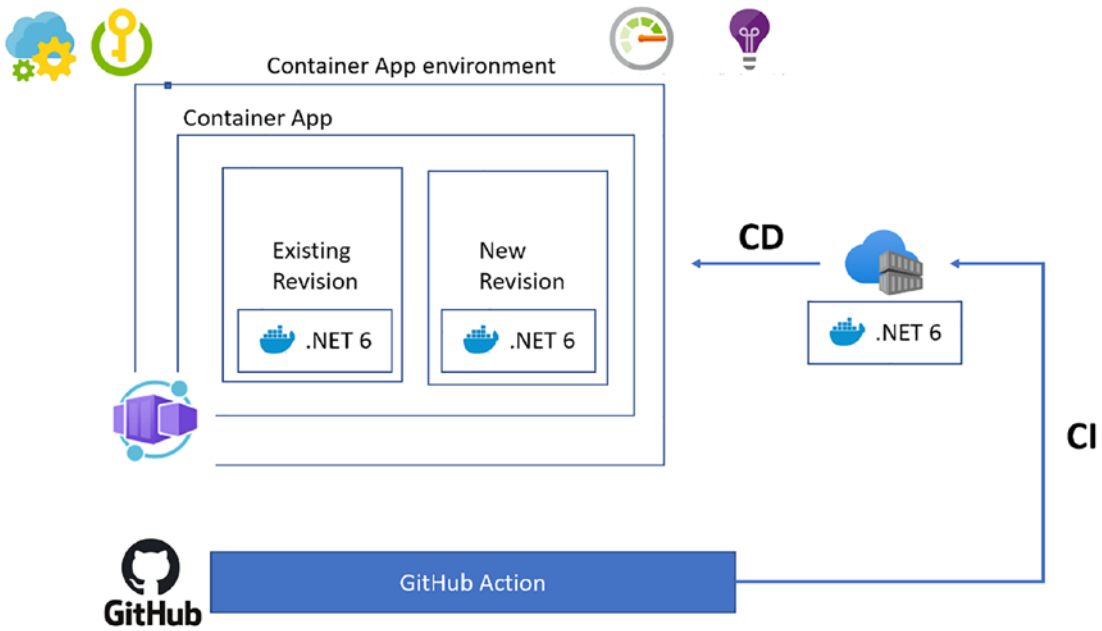


Figure 5-16. GitHub Actions demo architecture

- **.Net 6 website:** A simple .NET 6 website that will be containerized and pushed to the Azure Container Registry by GitHub Actions. We will pull multiple versions of the container running on Azure Container Apps.
- **GitHub Actions** will be used for CI/CD. Pipeline will be explained in the following.
- **Azure Container Apps (Preview)** (<https://docs.microsoft.com/en-us/azure/container-apps/overview?msclkid=0baebbe2d08811ec9752dd19779d46f2>) will be used to run our application. It is a service to run containerized applications on a serverless platform. It offers an easy way of running containers, leaving behind concerns like infrastructure management and complex orchestrators like Kubernetes. It is composed of the following components:
  - **Environment:** Isolation boundary around a collection of container apps (in our case, one single app). Apps in different environments do not share resources (computing

and networking) and cannot communicate with one another using DAPR.

- **Container App:** Manages orchestration details for you (connection to container registry, container image to use, secrets, or traffic distribution).
- **Revision:** An easy way to update running container instances. Every time the container image properties are changed, a new revision is created. You can easily define versions that should be active (accept traffic) and provide traffic distribution for each revision (for easily implementing canary or blue/green deployments).
- **Azure Key Vault:** Used to keep secret (sensitive) configuration values used by the website.
- **Azure App Configuration:** Used for
  - **Configuration management:** Providing the configuration to the website (together with Key Vault). App configuration keeps nonsensitive configuration data and references sensitive data hosted in KV. All the configuration is given to the website from this single service.
  - **Feature Flag manager:** A feature will be exposed based on rules applied to the resource.
  - **Azure Monitor/App Insights** will be used to monitor the solution (used on next monitoring-related chapters).
  - **Container registry:**

You will begin the same way as you did on the Azure DevOps one. We need a Service Principal (application identity) that will be used by GitHub Actions to interact with our Azure environment.

```
az ad sp create-for-rbac -n GH-Actions --role contributor --scope /subscriptions/<SUBSCRIPTIONID> --sdk-auth
```



Aside from GitHub, in this demo, you will reuse the same Service Principal (you could also create separated ones with different roles):

- Docker Container (website code) in Container App getting access to Azure App Configuration
- Docker Container (website code) in Container App getting access to Key Vault

On the actual application code, we can see how the Service Principal will be used for getting access to Azure App Configuration and Azure Key Vault: [https://github.com/unaihuete-org/SRE\\_with\\_Azure/blob/master/src/Program.cs](https://github.com/unaihuete-org/SRE_with_Azure/blob/master/src/Program.cs) (see Listing 5-4).

**Listing 5-4.** App Configuration setup in .NET

```
var endpoint="https://srewithazureunai-appconfig.azureconfig.io";
builder.Configuration.AddAzureAppConfiguration(options =>
    options.Connect(new Uri(endpoint), new DefaultAzureCredential())
        .ConfigureKeyVault(kv =>
            {
                kv.SetCredential(new DefaultAzureCredential());
            })
        .UseFeatureFlags());
```

The “DefaultAzureCredential” class is able to detect Service Principal properties like AZURE\_TENANT\_ID, AZURE\_CLIENT\_ID, and AZURE\_CLIENT\_SECRET on the environment variables to authenticate. In our case, the environment variables will be provided in the Dockerfile with the help of GitHub Actions. Those lines of code will give your app access to both sensitive (Key Vault) and nonsensitive (App Configuration) configuration values.

---

**Note** Give the Service Principal proper access roles/permissions on the Key Vault/Azure App Configuration level in order to have reading access (see Figure 5-17 and Figure 5-18).

---

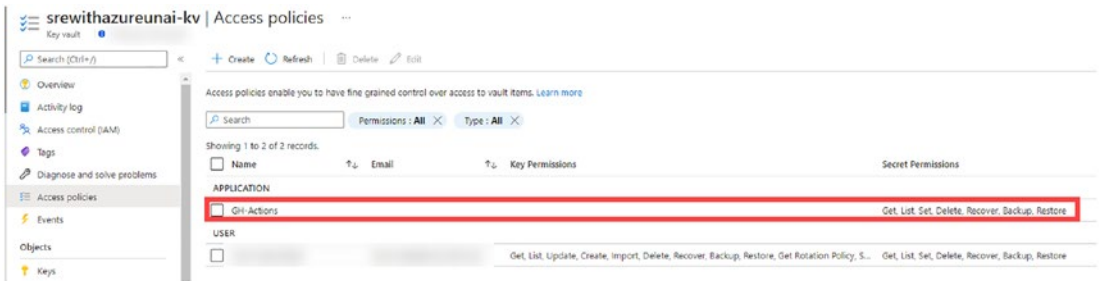


Figure 5-17. Key Vault access policies

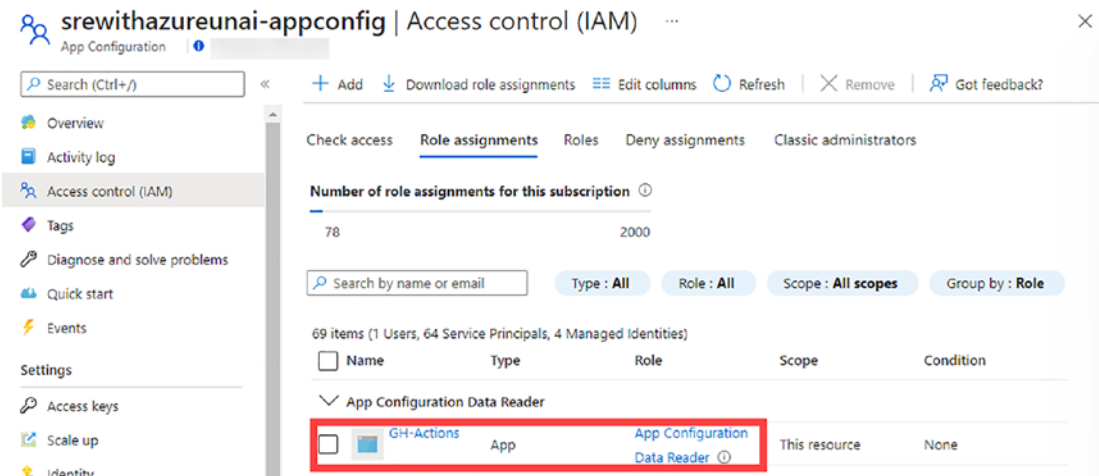


Figure 5-18. App Configuration access control

Let’s take a look at the GitHub workflows now, starting with the CI (see Listing 5-5):

**CI ([https://github.com/unaihuete-org/SRE\\_with\\_Azure/blob/master/.github/workflows/ci-build.yaml](https://github.com/unaihuete-org/SRE_with_Azure/blob/master/.github/workflows/ci-build.yaml) )**

**Listing 5-5.** GitHub Actions CI workflow

```
name: CI Build App
on:
  push:
    branches: [master]
    paths:
      - "src/**"
      - "tests/**"
```

```

- ".github/workflows/ci-build.yaml"
pull_request:
workflow_dispatch:

env:
  IMAGE_REG: ${ secrets.REGISTRY_LOGIN_SERVER }
  IMAGE_REPO: dotnet6
jobs:
  test:
    name: "Tests & Linting"
    runs-on: ubuntu-latest
    steps:
      - name: "Checkout"
        uses: actions/checkout@v2

      - name: "Run tests"
        run: make test-report

      - name: "Upload test results"
        uses: actions/upload-artifact@v2
        # Disabled when running locally with the nektos/act tool
        if: ${ always() }
        with:
          name: test-results
          path: tests/TestResults/

      - name: "Publish test results"
        uses: EnricoMi/publish-unit-test-result-action@v1
        if: ${ always() }
        with:
          files: tests/TestResults/TestResults.xml

  build:
    name: "Build & Push Image"
    needs: test
    runs-on: ubuntu-latest
    steps:
      - name: "Checkout"
        uses: actions/checkout@v2

```

```

# Nicer than using github runid, I think, will be picked up
automatically by make
- name: "Create datestamp image tag"
  run: echo "IMAGE_TAG=$(date +%d-%m-%Y.%H%M)" >> $GITHUB_ENV
# Replace tokens in Dockerfile
- uses: cschleiden/replace-tokens@v1
  with:
    tokenPrefix: '__'
    tokenSuffix: '__'
    files: ['**/Dockerfile']
  env:
    SP_AZURE_TENANT_ID: ${ secrets.SP_AZURE_TENANT_ID }
    SP_AZURE_CLIENT_ID: ${ secrets.SP_AZURE_CLIENT_ID }
    SP_AZURE_CLIENT_SECRET: ${ secrets.SP_AZURE_CLIENT_SECRET }
- name: 'Build and push image'
  uses: azure/docker-login@v1
  with:
    login-server: ${ secrets.REGISTRY_LOGIN_SERVER }
    username: ${ secrets.REGISTRY_USERNAME }
    password: ${ secrets.REGISTRY_PASSWORD }
- name: "Docker build image"
  run: make image
# Only when pushing to default branch (e.g. master or main), then
push image to registry
- name: Push to container registry
  # if: github.ref == 'refs/heads/master' && github.event_name
  == 'push'
  run: make push
- name: "Trigger ACA release pipeline"
  if: github.ref == 'refs/heads/master'
  uses: benc-uk/workflow-dispatch@v1
  with:
    workflow: "CD ACA"
    token: ${ secrets.GH_PAT }
    inputs: '{ "IMAGE_TAG": "${ env.IMAGE_TAG }" }'

```

On the CI workflow, we can see the following:

- **Trigger:** The workflow can be triggered, either providing changes to specific folders on the master branch, opening a pull request, or manually (workflow\_dispatch).
- **Variables (env):** It defines variables, making use of the **secrets** functionality to provide the endpoint of the Azure Container Registry.
- **Jobs:** Two jobs defined:
  - **Test:** Makes sure unit tests run properly and upload results.
  - **Build:**
  - **Replace tokens:** This one will replace the Service Principal details in the Dockerfile in order to be able to connect to Key Vault and App Configuration. Really useful when sensitive information needs to be replaced in a file. In this case, it will look for strings starting/finishing with “\_”, for example, “\_\_SP\_AZURE\_TENANT\_ID\_\_”, and replace it with the value kept at GitHub Secrets.
  - **Login:** Authenticates against the Azure Container Registry (ACR).
  - **Build & Push :** Builds the container based on Dockerfile and pushes it to ACR.
  - **Trigger CD:** It will trigger the next workflow for Azure Container Apps.

Let’s now take a look at the CD workflow for Azure Container Apps (see Listing 5-6):

[https://github.com/unaihuete-org/SRE\\_with\\_Azure/blob/master/.github/workflows/cd-containerapps.yaml](https://github.com/unaihuete-org/SRE_with_Azure/blob/master/.github/workflows/cd-containerapps.yaml)

**Listing 5-6.** GitHub Actions CD Workflow

```
#
# Deploy to Azure Container Apps
#
#
name: CD ACA
```

```
on:
```

```
workflow_dispatch:
  inputs:
    IMAGE_TAG:
      description: "Image tag to be deployed"
      required: true
      default: "21-04-2022.1534"
env:
  LOCATION: westeurope
  ACR_REPO: srewithazureunai.azurecr.io/dotnet6
  RG: SREwithAzure
  ACA_NAME: srewithazureunai-containerapp
jobs:
  #
  # Deploy Azure Container App
  #
  deploy-azure-green:
    name: Deploy Azure Container App to Azure Resource Group
    runs-on: ubuntu-latest
    # environment:
    #   name: AKS
    #output for next job
    outputs:
      LatestRevision: ${{ steps.get-latest-aca-revision.outputs.
        LatestRevision }}
      FirstDeployment: ${{ steps.first-deployment.outputs.
        FirstDeployment }}
    steps:
      # Checkout code
      - uses: actions/checkout@main
        # Log into Azure
      - uses: azure/login@v1
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}
        #does the cluster exist, first deployment detect
      - name: Get first deployment BOOL
```

```

uses: azure/CLI@v1
id: first-deployment
with:
  azcliversion: 2.35.0
  inlineScript: |
    az config set extension.use_dynamic_install=yes_without_prompt
    empty_answer=[]
    check=$(az containerapp list --query "[?name=='$ACA_NAME' ]")
    [ $check==$empty_answer ] && FirstDeployment=false ||
    FirstDeployment=true
    echo "FirstDeployment=$FirstDeployment" >> $GITHUB_ENV
    echo "::set-output name=FirstDeployment::$FirstDeployment"
- name: Get latest ACA REVISION
  uses: azure/CLI@v1
  id: get-latest-aca-revision
  if: steps.first-deployment.outputs.FirstDeployment != 'false'
  with:
    azcliversion: 2.35.0
    inlineScript: |
      az config set extension.use_dynamic_install=yes_without_prompt
      LatestRevision=$(az containerapp revision list -n $ACA_NAME
      -g $RG --query "[?properties.trafficWeight ==\`100\`][].name"
      --output tsv)
      echo "LatestRevision=$LatestRevision" >> $GITHUB_ENV
      echo "::set-output name=LatestRevision::$LatestRevision"
- name: ECHO previous REVISION
  if: steps.first-deployment.outputs.FirstDeployment != 'false'
  run: |
    echo "LatestRevision=$LatestRevision"
    # Deploy Bicep file (existing cluster)
- name: deploy
  #if: github.event.inputs.SKIP_INFRA == 'No'
  uses: azure/arm-deploy@v1
  if: steps.first-deployment.outputs.FirstDeployment != 'false'
  id: deploy

```

```

with:
  subscriptionId: ${ secrets.AZURE_SUBSCRIPTION }
  resourceGroupName: ${ env.RG }
  template: deploy/bicep/container-app.bicep
  parameters: image=${ env.ACR_REPO }:${ github.event.
  inputs.IMAGE_TAG } acrPassword=${ secrets.ACR_
  PASSWORD } newRevisionNumber=${ github.run_number }
  existingRevisionName=${ env.LatestRevision }
  failOnStdErr: false

# Deploy Bicep file (new cluster)
- name: deploy NEW
  #if: github.event.inputs.SKIP_INFRA == 'No'
  uses: azure/arm-deploy@v1
  if: steps.first-deployment.outputs.FirstDeployment == 'false'
  id: deploy-new
  with:
    subscriptionId: ${ secrets.AZURE_SUBSCRIPTION }
    resourceGroupName: ${ env.RG }
    template: deploy/bicep/container-app-new.bicep
    parameters: image=${ env.ACR_REPO }:${ github.event.
    inputs.IMAGE_TAG } acrPassword=${ secrets.ACR_
    PASSWORD } newRevisionNumber=${ github.run_number }
    failOnStdErr: false

# check GREEN deployment and witch with BLUE if working
switch-blue-green:
  name: switch BLUE/GREEN deployment
  runs-on: ubuntu-latest
  needs: deploy-azure-green
  if: needs.deploy-azure-green.outputs.FirstDeployment != 'false'
  environment:
    name: ACA

  steps:
  # Checkout code
  - uses: actions/checkout@main

```



```

# Log into Azure
- uses: azure/login@v1
  with:
    creds: ${{ secrets.AZURE_CREDENTIALS }}
- name: ECHO REVISION
  run: echo ${{ needs.deploy-azure-green.outputs.LatestRevision }}
# only deactivate when another revision exists (after first deployment)
- name: Replace the BLUE/GREEN traffic and deactivate old version
  uses: azure/CLI@v1
  with:
    azcliversion: 2.35.0
    inlineScript: |
      az config set extension.use_dynamic_install=yes_without_prompt
      az containerapp ingress traffic set --name $ACA_NAME -g
      $RG --traffic-weight latest=100
      az containerapp revision deactivate --revision ${{ needs.deploy-
      azure-green.outputs.LatestRevision }} --resource-group $RG

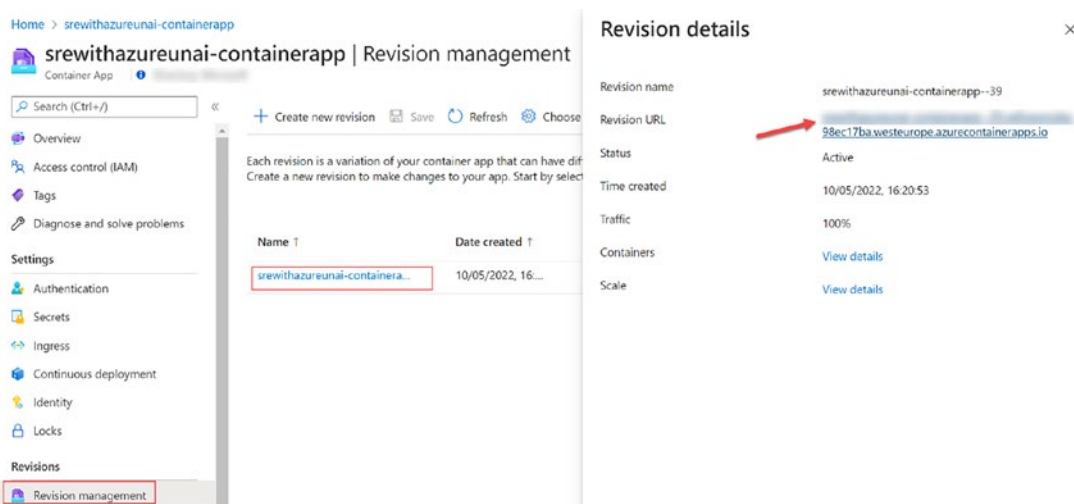
```

The CD workflow is composed of the following components:

- **Trigger:** Triggered either manually or by the CI workflow
- **Variables (env):** Defines variables for the target Azure environment
- **Jobs:**
  - **Deploy-azure-green:** This job will be executing a deploy based on Bicep templates. I have defined two different Bicep templates for Azure Container Apps (ACA): one for the initial deployment (no other revisions yet) and another template that will be used for blue/green deployment:
    - The job will authenticate against Azure using GitHub Secrets.
    - Running AZ CLI commands, it will detect if other deployments exist and create a workflow variable that will define the execution of the following actions:

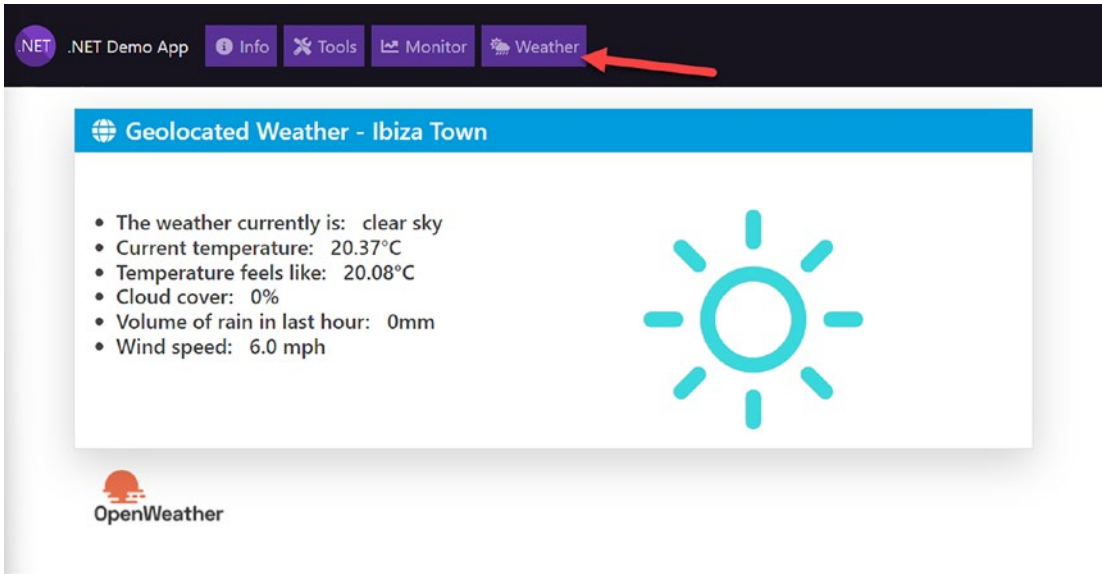
- **For first deployment:** Deploy the infrastructure (single container app revision) using the following template: [https://github.com/unaihuete-org/SRE\\_with\\_Azure/blob/master/deploy/bicep/container-app-new.bicep](https://github.com/unaihuete-org/SRE_with_Azure/blob/master/deploy/bicep/container-app-new.bicep)
- **For blue/green deployments (not first):**
  - It will identify existing ACA revision and deploy the revision. It will still keep 100% traffic on the existing revision and 0% on the new one until tested.
- Switch blue-green (not first):
  - The job references an **environment**. Based on the setup, the environment will force the user for a **manual approval** before execution.
  - The actions will switch traffic from the old revision to the new one and deactivate the old revision using Azure CLI commands.

As mentioned, revisions can be validated before switching (before environment approval), as they offer an endpoint to test it (see Figure 5-19).

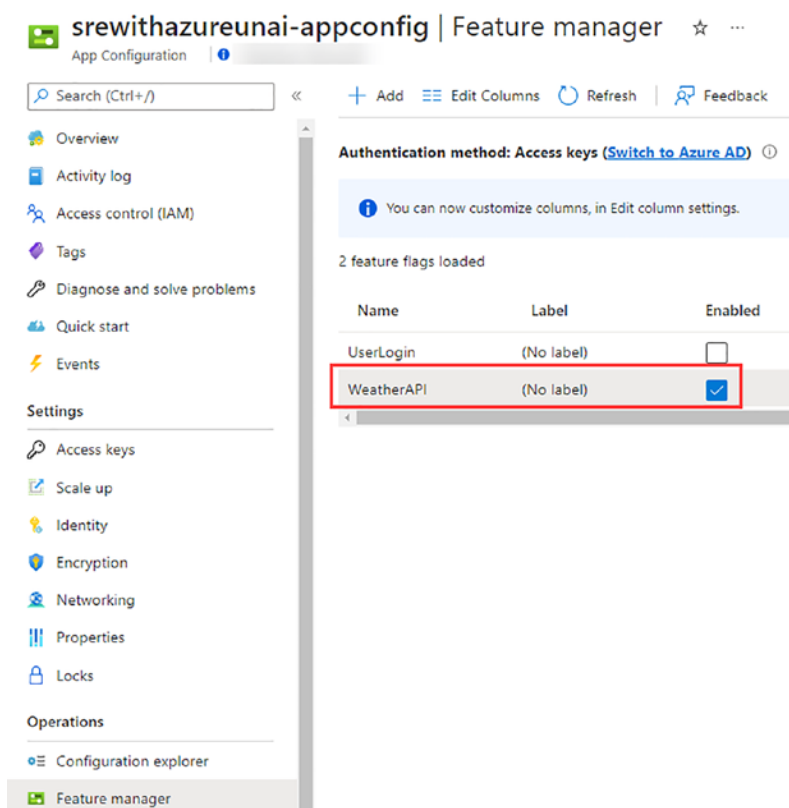


**Figure 5-19.** Container App revision URL

The Feature Flag defined in the solution will be able to expose/hide the **Weather** functionality of the website without needing to redeploy (updated in seconds, depending on refresh cycle configured on application code) (see Figure 5-20 and Figure 5-21).



*Figure 5-20. Website weather feature enabled*



**Figure 5-21.** App Configuration feature enabled

Using the Feature Flag together with authentication libraries, we could also enable the Feature Flag only for desired users/groups (canary/ring-based deployments): <https://docs.microsoft.com/en-us/azure/azure-app-configuration/howto-targetingfilter-aspnet-core>.

As you have seen, similar GitHub Workflow could be used, together with Feature Flags to implement the deployment strategies explained in the chapter.

## Summary

In this chapter, you were introduced to the automation topic, reviewing practices that will make your applications' incremental updates more safe, reliable, and efficient. As mentioned before, an SRE will need to be able to at least understand the concepts explained in this chapter. The implementation will depend on the DevOps maturity (and existing roles) of the organization.

The chapter focused on explaining the basic DevOps concepts and showing examples of how to implement them using tools like Azure DevOps and GitHub Actions.

I hope this chapter gives you a nice introduction to the topic; each concept defined could be further explained and researched (not the focus of this book).

## CHAPTER 6

# Monitoring As the Key to Knowledge

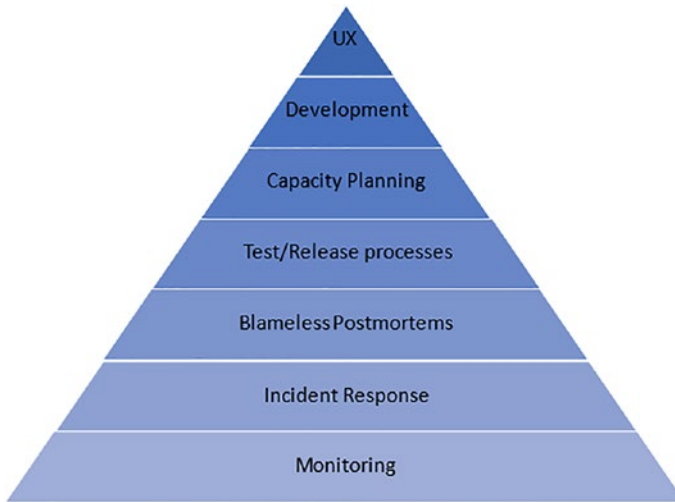
This chapter will cover the basics of a monitoring strategy, focusing on the concepts related to the topic and giving an explanation of the broad amount of Azure services that should be used for your observability goals.

By the end of this chapter, you should be able to do the following:

- ✓ Understand Operational Awareness and Monitoring vs. Observability
- ✓ Remember SRE concepts from Chapter 2: SLI/SLO/SLA and error budget
- ✓ Learn about burn rate (error budget)
- ✓ Learn about Azure Service Health
- ✓ Learn about Azure Monitor and Log Analytics
- ✓ Learn how to use Application Insights and customize it
- ✓ Learn how to use Kusto (KQL) queries
- ✓ Learn how to use Azure Alerts

## Operational Awareness

In order to identify possible issues, check performance, or provide proper maintenance to your solutions, you need to be able to see what is going on in your systems. You need to have a holistic view of a system's health. As Dickerson's hierarchy of reliability shows (see Figure 6-1), defining a good monitoring strategy is the base to give you the data you need to see how things are going and take action (incident response covered in the next chapter).



**Figure 6-1.** Dickerson's hierarchy of reliability: monitoring

It is obvious to think that **understanding your solutions** is critical to monitoring them in a proper way. This idea looks simple, but it is not that easy to implement. Solutions created nowadays have complex architectures, and the speed of change brought by DevOps/Agile methodologies makes it challenging to understand all the pieces that make your solutions. **Operational Awareness is key to know what needs to be monitored.** There are some questions that need to be answered before defining your monitoring strategy:

- What is it running on our environments? Understand the infrastructure architecture.
- Where do they run? Type of service (Azure VMs, WebApps, Container Instance, AKS, SQL DB, Cosmos DB, etc.), pricing tier, and regions.

- What services are they composed of? Understand application architecture.
- What are the dependencies of those services?
- How do you deploy changes to those services? Manual? CI/CD? Is there any automated/manual testing in place? Any deployment strategies in place? Understand DevOps processes applied.
- Who are the owners of those services? Stakeholders affected by possible incidents?

Once the solutions are fully understood, you can also define a baseline of the “normal” behavior of your solution based on past performance. The chapter will show how Azure tools will offer this experience powered by machine learning algorithms (see Application Insights Smart Detection later).

So what’s the next step? If you want to improve your solution’s reliability, you need to have a clear idea of the reliability concept. It may be helpful to review the “Understand Reliability” section in Chapter 2. Remember, there are many aspects related to reliability:

- **Availability:** Is the service “up” or “down”?
- **Latency:** Delay between requests and responses
- **Throughput:** Amount of data successfully processed
- And many others covered in Chapter 2, such as freshness, coverage, and correctness

These are the aspects of the solution you want to monitor for the services offered.

## SLI/SLO/SLA

It may also be helpful to remember the concepts covered in the “Service-Level Metrics” section in Chapter 2:

- **Service-Level Indicator (SLI):** An indicator measured to define the health of a service. Metric that is monitored. For example, number of successful requests (200 response and below 200ms) to an API every hour.



- **Service-Level Objective (SLO):** Based on the tracked indicator, the goal set as a team. Based on the previous example, an SLO could be 99.9% of successful requests to the API every hour. SLOs will be composed of
  - **Metric** to measure
  - **Target:** Can be improved/changed over time (agreed with stakeholders)
  - **Time interval/duration**
- **Service-Level Agreement (SLA):** Contractual agreement resulting in compensation if SLA is not met. Based on the previous example, SLA offered to the customer could be 99% of requests. Remember the trade-off between number of 9's (downtime), required investment on your services, and release velocity. The objective should always be reaching an **appropriate (or acceptable) level of reliability** (as seen in Chapter 1).

As an SRE, taking into account previously mentioned reliability aspects, **reliability should be measured from the customer's perspective**. When defining SLI/SLO/SLA, put yourself in the shoes of the customer. What are you expecting from a product? For example:

- Does a customer really care if the CPU load of our cluster is at 90% if the service is performing good enough?
- Does a customer really care if one of the workload instances goes down if our solution is running on multiple replicas and still provides responses fluently?

Of course, you should be alerted about an unusual behavior or potential incident in your apps; nevertheless, the defined SLI should mainly monitor metrics affecting customer experience: successful request percentage, response time for specific services, or freshness of data (sport events, elections, etc.).

## Error Budget/Burn Rate

Remember from Chapter 2, when defining SLOs, an **error budget** is given (1-SLO). You should create alerts that let you **react before the error budget is consumed!** When the error budget is consumed, DevOps teams may be encouraged to slow down the change velocity and work on stabilizing the existing solutions first.

Create alerts based on error budget/burn rate (how fast the budget is getting consumed). For example, imagine the following situation:

- **SLI** ► calls made to an API operation should be successful and have a response time lower than 100ms, measured the last month.
- **SLO** ► you define an objective of 98% for all monthly calls.
- **Error budget** = (1-SLO) ► 2%.
- If your API receives 100.000 calls per month, the **error budget** would be **2000 failed requests a month**.
- How to define alerts based on **burn rate**, how fast your error budget is consumed. A **burn rate of 1** for the monthly SLO period would mean consuming all budget by the end of the month. Check Google's video: [www.youtube.com/watch?v=t1BGo-1l1AM](http://www.youtube.com/watch?v=t1BGo-1l1AM):

**Burn rate = error budget consumed \* period/alert window**

**\*Error budget consumed being 1 for 100% used budget**

- If you want to measure a 2% burn of the error budget in the last hour (2% of 2000 ► 40 failed requests ► it would consume the budget in 50 hours!), it will give you a burn rate of  

$$\text{Burn rate} = 0.02 * (31d * 24h) / 1h = 14.88$$
- Alerts should be defined when evaluated with a short and a long window. For the preceding example, we could evaluate for both 1 hour (long) and 5 minutes (short) windows.

Then, how can we track/measure these indicators?

The aim of this chapter is to explain the tools provided by Azure to define ways to implement them and design the best monitoring approach possible for your solution.

## Observability vs. Monitoring

These two terms are often used (and confused) by IT teams nowadays. Distributed applications designed lately introduce challenges like tracing requests that may span multiple services and infrastructure components. But are monitoring and observability the same concept?

No. You can find many definitions in the community for these two terms, but let's try to explain it in the simplest way possible:

- **Observability:** Provides **ability to understand** a system internal state based on its external outputs. It is **informative**.
- **Monitoring:** The process of collecting data, analyzing it, and taking decisions based on the information. It is **actionable**.

After observability is achieved for complex solutions (full stack view), you will have the visibility needed to create actionable alerts, reports/dashboards, or even anomaly detection based on historical data using AIOps solutions. Three primary pillars can be identified for observability:

- **Logs:** Timestamped description of an event. Logs may need to be structured and parsed for further analysis (querying).
- **Metrics:** Point-in-time measurements, mainly numeric values (more consistent than logs), used for alerts and dashboarding.
- **Traces:** Getting details of particular requests and how they flow through the used components in a distributed system.

Let's see how to achieve our observability goals in Azure.

## Azure Service Health

Azure Service Health (<https://docs.microsoft.com/en-us/azure/service-health/overview>) is a service that provides a customized view of the health of the Azure services you are using. It gives you a view on the **platform/service level** and the health of services offered by the cloud provider. It is composed of the following capabilities (see Figure 6-2):

- **Service Issues:** Offers a way to look for real-time issues. Alerts can be defined for the issues.
- **Planned Maintenance:** Upcoming maintenance events information.
- **Health Advisories:** Changes in Azure services that may affect your workloads.
- **Security Advisories:** Security-related notifications.
- **Resource Health:** Reports the past and current health of your Azure services, relying on predefined signals to assess if a resource is unhealthy.

Home > Service Health | Service issues

Search (Ctrl+/) Save View Delete View Add service health alert Try preview

ACTIVE EVENTS

- Service issues
- Planned maintenance
- Health advisories (2)
- Security advisories (1)

HISTORY

- Health history

RESOURCE HEALTH

- Resource health

ALERTS

- Health alerts

Subscription: [Dropdown] Region: 22 selected Service: 209 selected

No service issues found  
See all past issues in the [health history](#).

Issues resolved in the past 7 days

Issue name	Subscription(s)	Services	Regions	Start time
RCA - Azure Active Director	16169bdf-109c-4eb7...	Azure Active Directory	Global	2022-05-23T23:49:45
Multi-Factor Authentication	16169bdf-109c-4eb7...	Multi-Factor Authent...	Global	2022-06-03T04:33:46
RCA - Delay in Sign in and A	16169bdf-109c-4eb7...	3 Services	Australia East,Australi...	2022-05-31T21:35:14

**Figure 6-2.** Azure Service Health

The Azure Status (<https://status.azure.com/>) website can also be used to get real-time information about service outages across regions (not only services/regions you are using) as you can see in Figure 6-3.

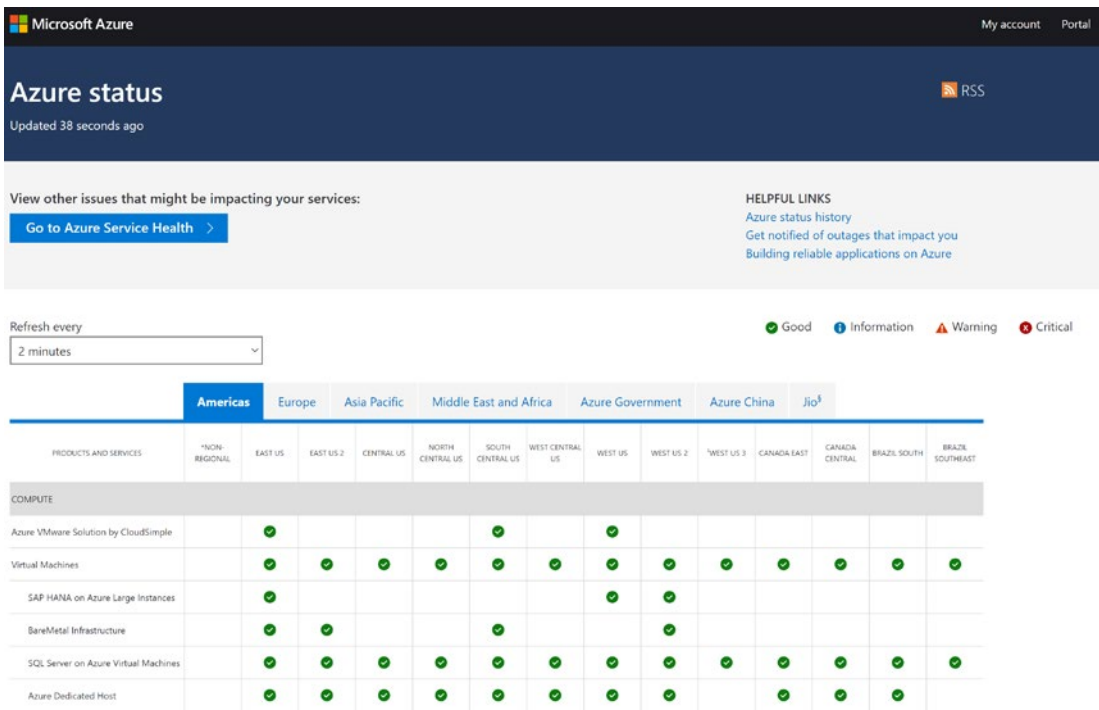


Figure 6-3. Azure Status website

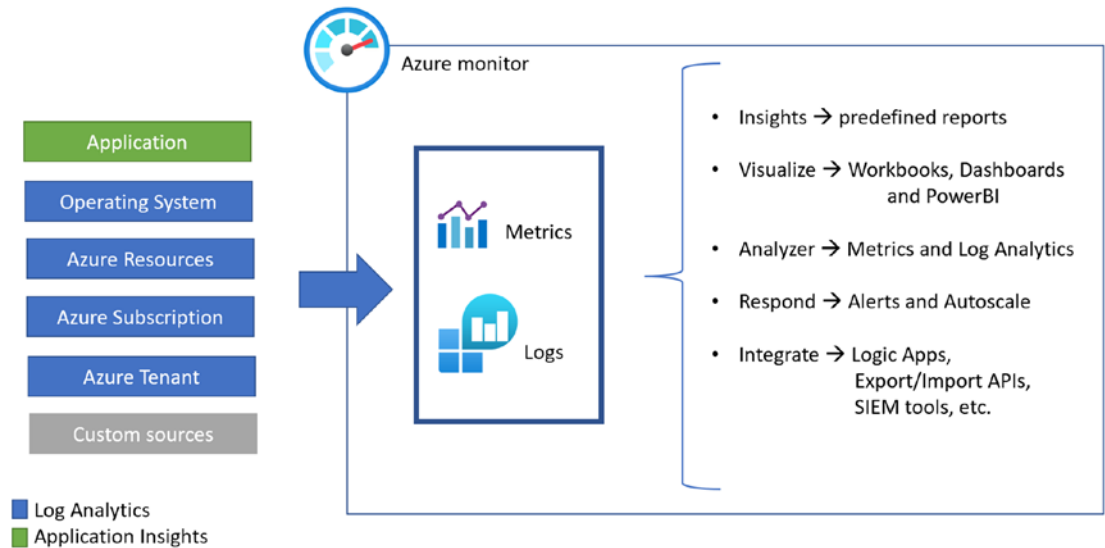
## Azure Monitor

When you host solutions in Azure, there are multiple sources you need to take into account for your observability approach. **Azure Monitor provides a unified experience** where you can have a total visibility of the health, performance, and other aspects of the running Azure services (also non-Azure workloads). **Metrics** are collected by default, but Azure Monitor makes use of the following two tools for **Logs and Traces** collection (remember observability pillars):

- **Log Analytics workspaces:** Solution used to hold logs offered by many Azure (and non-Azure) services (see Figure 6-4).
- **Application Insights:** Provides the tools to collect data about the performance and functionality of the application code: using traces, application logs, and user telemetry. New Application Insights instances use a Log Analytics workspace as the data store.

## Data Sources

Azure Monitor, by using the previously mentioned tools, gives you a way to collect the following sources/tiers for monitoring that can be collected and retained in a Log Analytics workspace for further analysis/reporting (see Figure 6-4):



**Figure 6-4.** Azure Monitor

- **Application:** As mentioned before, you can track the performance and use of your application by using Application Insights. Application Insights gives us different options for the collection of telemetry (no-code and SDK-based experience, covered later).
- **Operating System:** By using an Azure Monitor agent (AMA), we can collect logs and metrics from a guest operating system (Azure and non-Azure VM/servers).
- **Azure Resources:** Azure cloud platform offers the following data:
  - **Metrics:** Collected by default for 93 days. They can be retained longer if ingested in Log Analytics workspace.

- **Logs (or Resource Logs):** Several resources give information about “internal” operation; logs are created, but you need to export them (in case you want to collect) using **Diagnostic Settings** (see an example for Cosmos DB in Figure 6-5). Check supported services and schemas at <https://docs.microsoft.com/en-us/azure/azure-monitor/essentials/resource-logs-schema>.

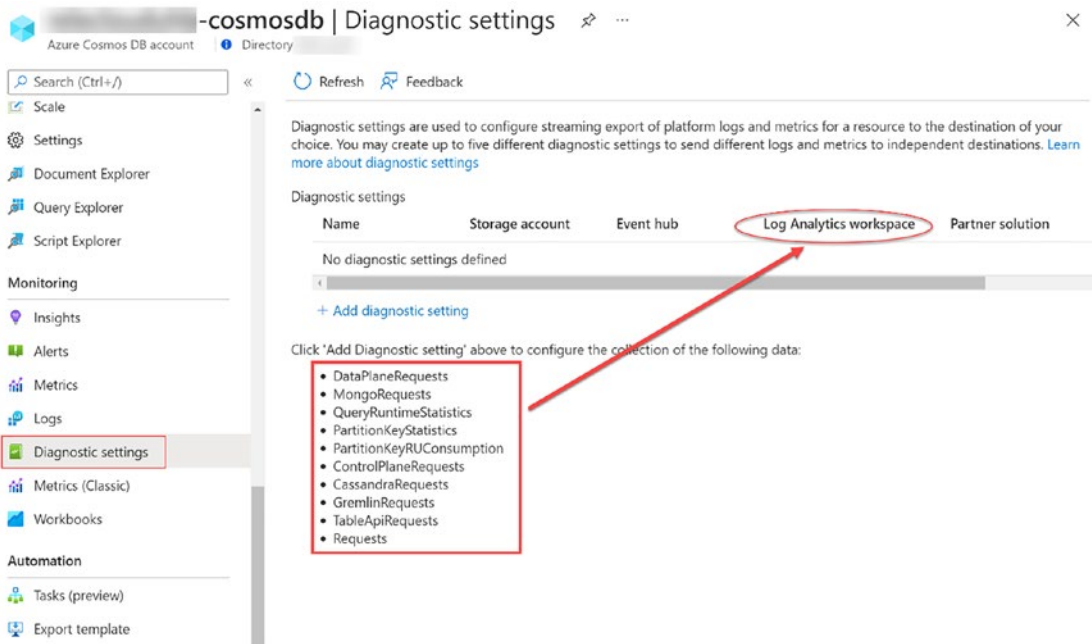
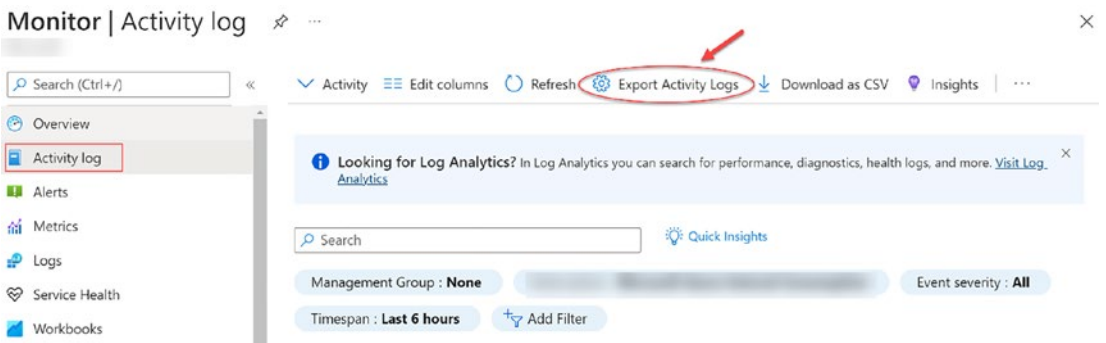


Figure 6-5. Diagnostic setting for Cosmos DB

- **Azure Subscription:** You can export subscription-related data about health and changes made to Azure resources to Log Analytics workspace using the export functionality from **Activity Logs** (see Figure 6-6).



**Figure 6-6.** Activity Logs export option

- **Azure tenant:** Collect data about Azure Active Directory–related operations (set up export from Azure Active Directory ► Diagnostic Logs).
- **Custom source:** Import logs/metrics to Log Analytics workspaces using REST APIs offered by Azure Monitor.

## Visualize

Once data has been collected, either using Application Insights, Log Analytics, or the metrics provided by default, you should start thinking on ways to visualize your data. These are some of the tools that can be used.

## Azure Dashboards

Azure Dashboards can be used to create an organized view of your cloud workloads. From a dashboard, you can monitor and even define quick access to resources. Many solutions, like Application Insights, provide default dashboards you can later customize (see Figure 6-7).



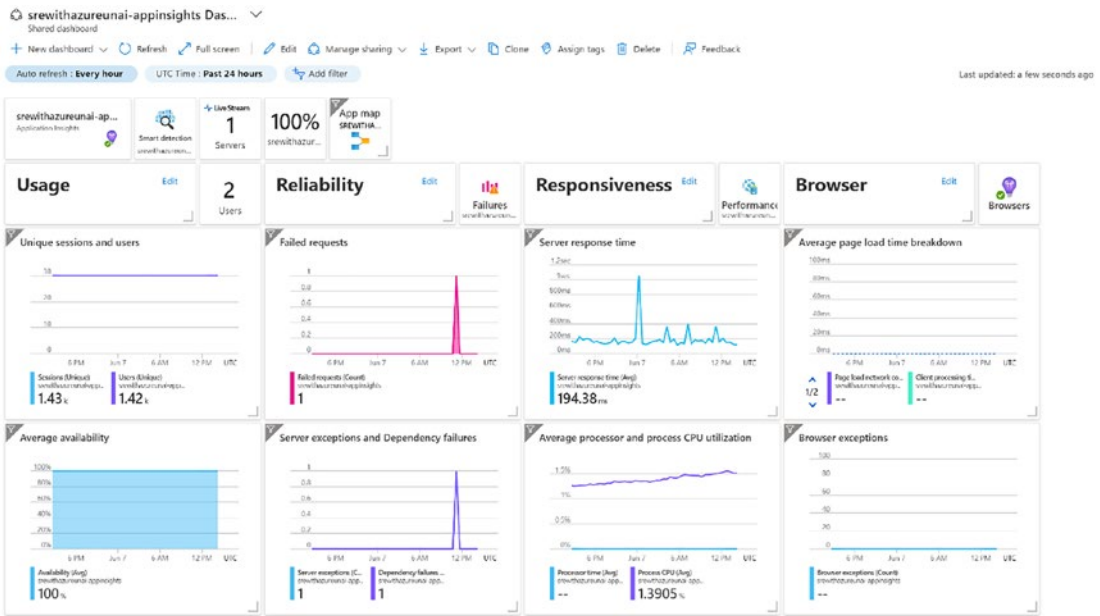


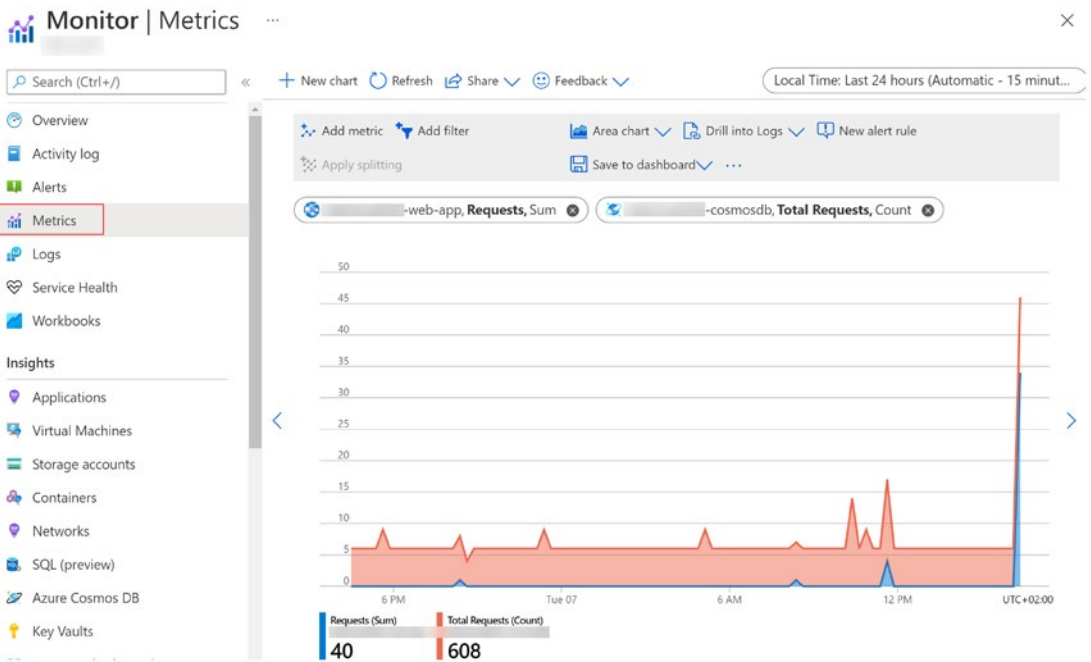
Figure 6-7. Application Insights default dashboard

Use Azure RBAC to allow others to see your dashboard. They can be treated as any other Azure resource. As the dashboard is represented by a JSON file (clicking on Export), you can also programmatically change and deploy them using, for example, Azure CLI commands. While editing, you can include customized markdown content or use predefined tiles from the Gallery.

## Metrics Explorer (Metrics)

It is a service used for plotting charts based on metrics that are collected from your running services. The solutions give the option to correlate metrics coming out of multiple Azure resources on the same chart and use filters to select the desired properties.

The charts can be attached to Azure Dashboards, Azure workbooks, and Grafana, and even Alert rules can be created from the tracked metrics directly from this service. Figure 6-8 is an example of a metrics chart for correlating both WebApp and Cosmos DB metrics.



**Figure 6-8.** Metrics chart for WebApp and Cosmos DB

## Azure Workbooks

Azure workbooks (<https://docs.microsoft.com/en-us/azure/azure-monitor/visualize/workbooks-overview>) provide a nice canvas that can be used for rich visuals and analysis. Gather multiple data sources in a single interactive “page” your engineering teams can use for monitoring.

You can combine markdown text, queries, and metrics chart, which can be interactively modified by the consumer using parameters you specify. Workbooks can also be treated as Azure resources; they offer a way to work with them as ARM templates (<https://docs.microsoft.com/en-us/azure/azure-monitor/visualize/workbooks-automate>).

Product teams also offer many default workbooks (see Figure 6-9) that can be customized if desired, or you can also start from a blank page.

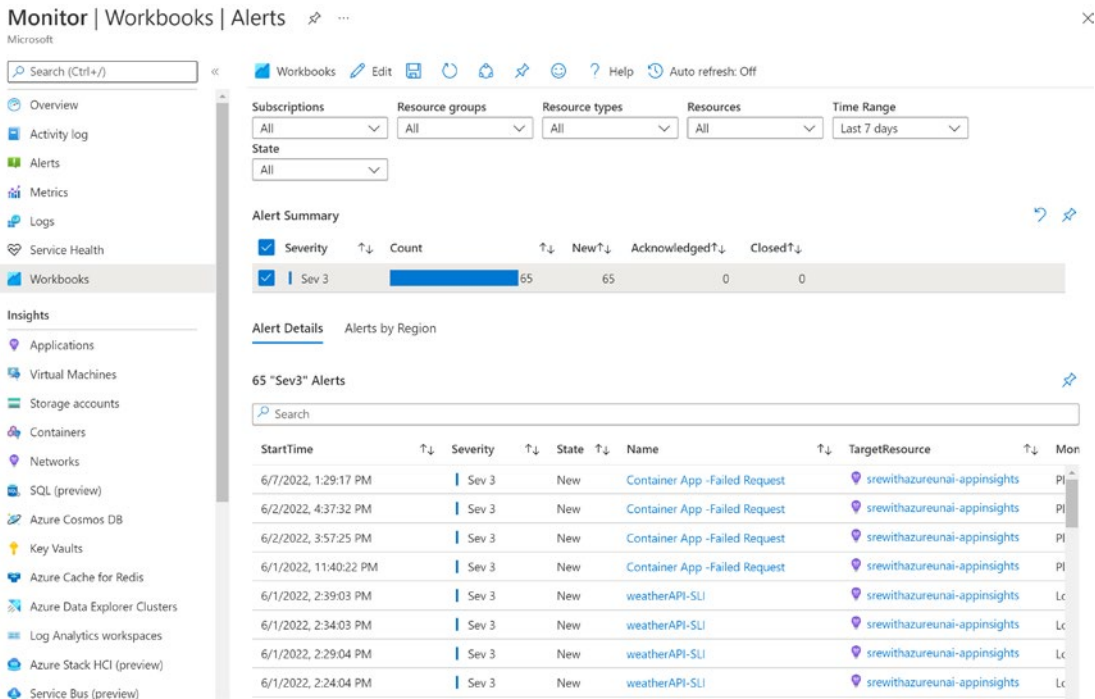


Figure 6-9. Alert management default workbook

## Azure Monitor Insights

Both from the Azure Monitor window (see Figure 6-10) and individual Azure resources (see Figure 6-11), you may find an **Insights** section. Insights provides a predefined workbook that can be customized, offering a general workbook from the Azure Monitor window and an individually scoped (more detailed) workbook from the Azure resource window.

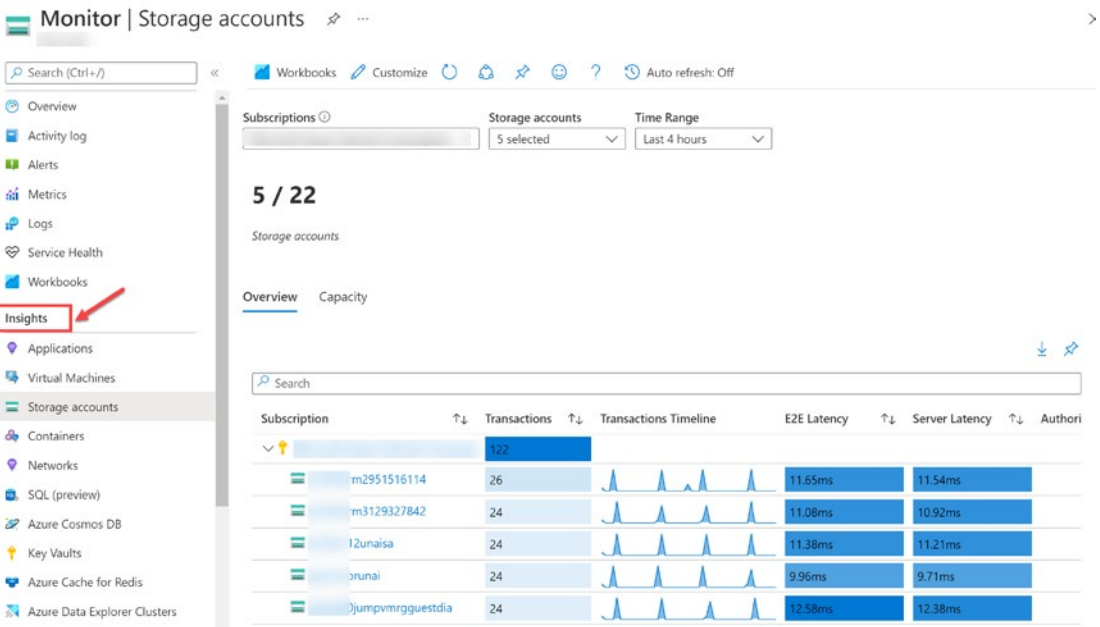


Figure 6-10. Insights from Azure Monitor

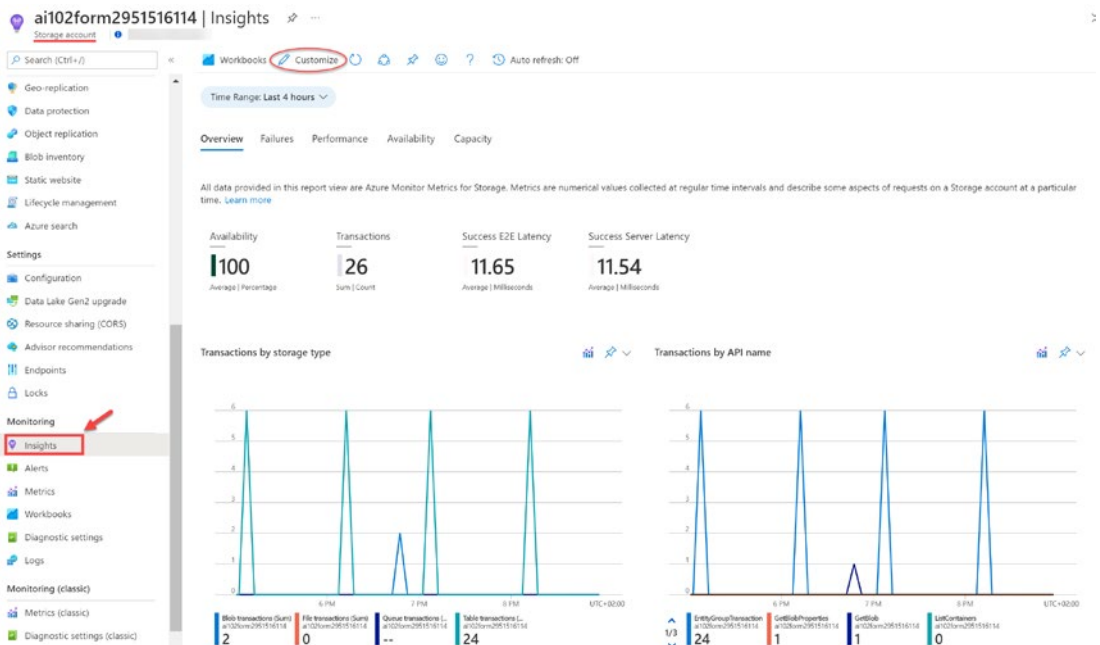


Figure 6-11. Insights for a storage account resource

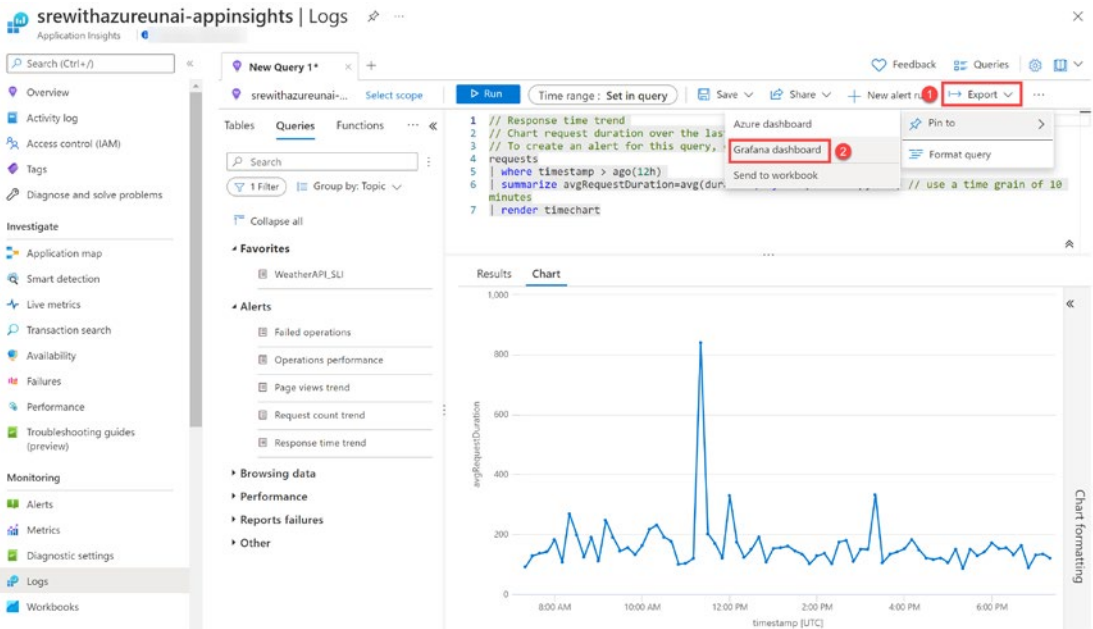
## Grafana

Grafana is a well-known open source solution for providing analytics and interactive visualization. Grafana can be connected to multiple data sources to act as the main observability platform.

In the case of using Azure Monitor (Log Analytics/Application Insights) for the data collection, you are offered the choice to connect to Grafana using the Azure Monitor data source plug-in (<https://grafana.com/docs/grafana/latest/datasources/azuremonitor/>). The plug-in will be able to retrieve the previously mentioned metrics/logs/traces, bringing them together into a single user interface in real time.

There are two different ways to set up the Grafana server:

- Ad hoc setup (in your selected VM/server) by installing the Grafana solution
- Grafana Cloud: managed service provided by Grafana Labs (<https://grafana.com/products/cloud/>)
- Create an **Azure Managed Grafana service** (<https://docs.microsoft.com/en-us/azure/managed-grafana/>) (in preview at the moment of writing):
  - It is fully supported by Microsoft, not dealing with setup and hosting.
  - You can share dashboards with people outside and inside the organization.
  - It uses Azure AD for authentication and access control.
  - It automatically connects to Azure Monitor as a data source using a Managed Identity for authentication.
  - We can directly import charts (Metrics and Kusto Log queries) from the Azure portal (e.g., see Figure 6-12 and Figure 6-13).



**Figure 6-12.** Export Log query chart to Grafana from Azure Portal



**Figure 6-13.** Exported query chart in Azure Managed Grafana

## Power BI

Similar to the previously mentioned Grafana, if you are looking for an external dashboarding solution, Power BI could be your choice. From Log Analytics queries, we are offered the option to export a **Power BI M query** (see Figure 6-14), which can be copy-pasted into a blank query in Power BI to create dashboards/reports in this tool.

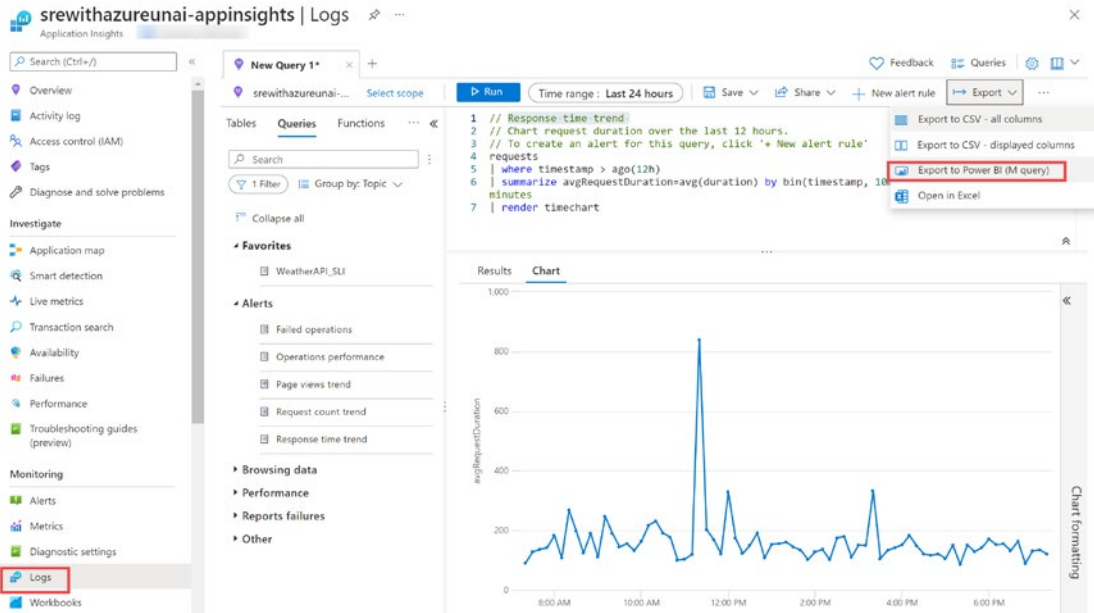


Figure 6-14. Export M Query for Power BI

## Analyze

### Azure Monitor Logs

There has been a lot of confusion lately with the terminology associated to these products. After consolidating many tools under the Azure Monitor scope (Log Analytics and Application Insights), the following naming changes were introduced (<https://docs.microsoft.com/en-us/azure/azure-monitor/terminology#february-2019---log-analytics-terminology>):

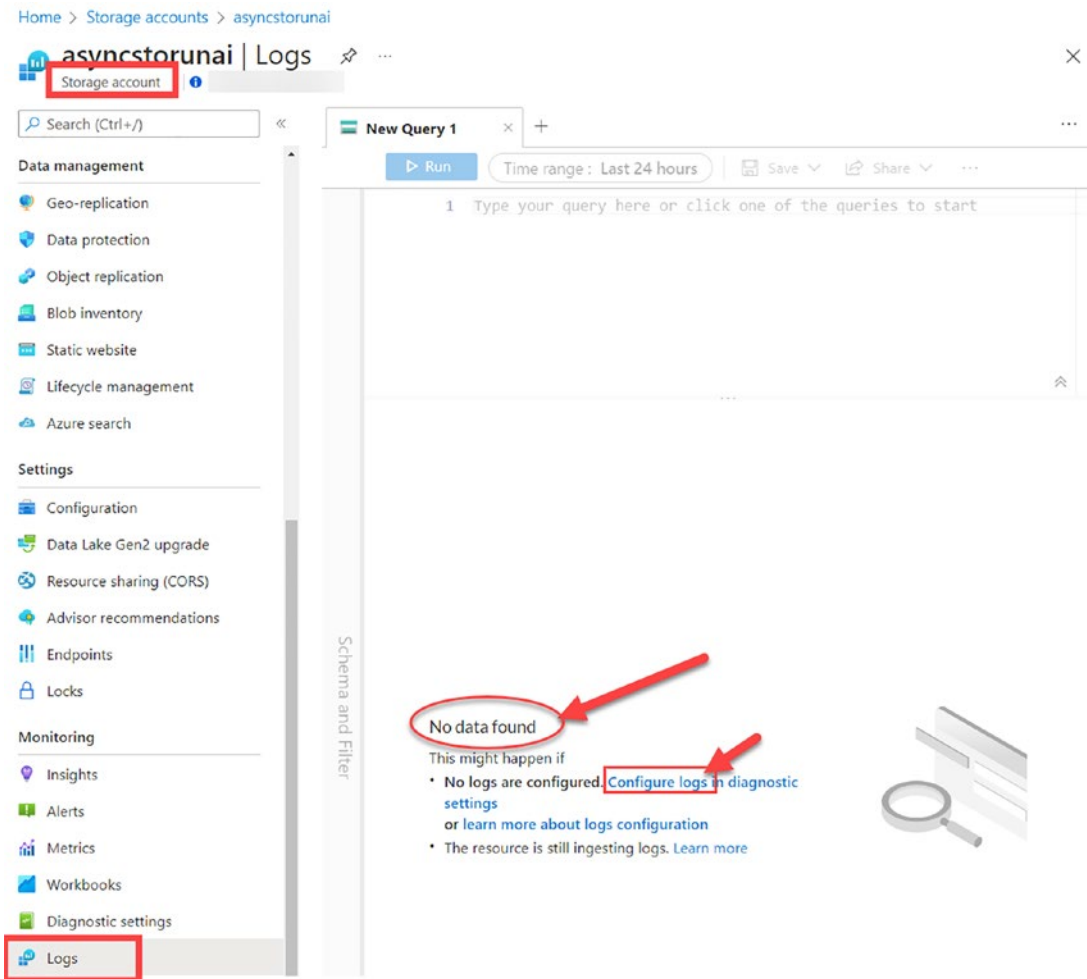
- **Log Analytics (before and still used) = Azure Monitor Logs (now):** Being the service in the Azure portal used to write and run Kusto Query Language (KQL) queries against collected logs (the equivalent to Metrics Explorer for Logs). It can be accessed from the **Logs** tab offered by many resources or from the **Logs** tab in **Azure Monitor**.

It should not be a surprise to see a “No data found” warning (see Figure 6-15) if log collection to a workspace has not been configured for the selected scope.

- **Log Analytics workspace (no name changed):** Still the solution used to store the collected Logs from selected sources. Access/permissions to workspace (and collected log tables) can be configured using Azure RBAC (Role-Based Access Control).

A single workspace may meet the requirements of your organization. Nevertheless, if you want to define different requirements (owners, costs, data retention, data limits, etc.), take a look at the best practices covered at <https://docs.microsoft.com/en-us/azure/azure-monitor/logs/workspace-design>. With the proper permissions, you will still be able to run cross-workspace queries: <https://docs.microsoft.com/en-us/azure/azure-monitor/logs/cross-workspace-query>.





**Figure 6-15.** No data found warning for Logs in Storage Account

## Log Analytics/Azure Monitor Logs

**Log Analytics (or Azure Monitor Logs)** gives the option to create and execute our own queries. You can find it in the **Logs** section of many resources. From **Azure Monitor** or the **Log Analytics workspace** resource, the **Logs** section will let you choose the **scope** to query. From a resource’s **Logs** section, the scope will be limited to the related resource.

The interface shows you the following options (see Figure 6-16):

1. **Scope** selected by the user.
2. **Query** to be executed using the **Run** button. Time range can be specified both in the query and the top option.

3. **Results/Charts** are shown for the executed query. The charts can be modified with the options offered on the right.
4. **Sidebar** offers multiple tools:
  - a. **Tables** shows the collected data for the scope; you can expand it to show the columns.
  - b. **Queries** shows example queries; you can execute or access the ones you have saved.
  - c. **Filter** tab will help you identify the fields to use to make your query more relevant (shown after running query).
  - d. **Functions** is a query that can be used in other queries as a command. You can use predefined functions or create your own to share with your team.
5. Extra options:
  - a. **New alert rule** lets you create an alert rule based on the results of your query.
  - b. **Export** gives you the option to move the query/charts to other solutions like Azure workbooks, Azure dashboards, Power BI, or Grafana.
6. **Queries** can be used to access predefined or custom-created queries that have been saved before. Queries can be saved in groups by using **Query Packs** (either the default one or custom ones) (<https://docs.microsoft.com/en-us/azure/azure-monitor/logs/query-packs>).
7. **Learning** content for KQL can be found on the top-right corner of the page.

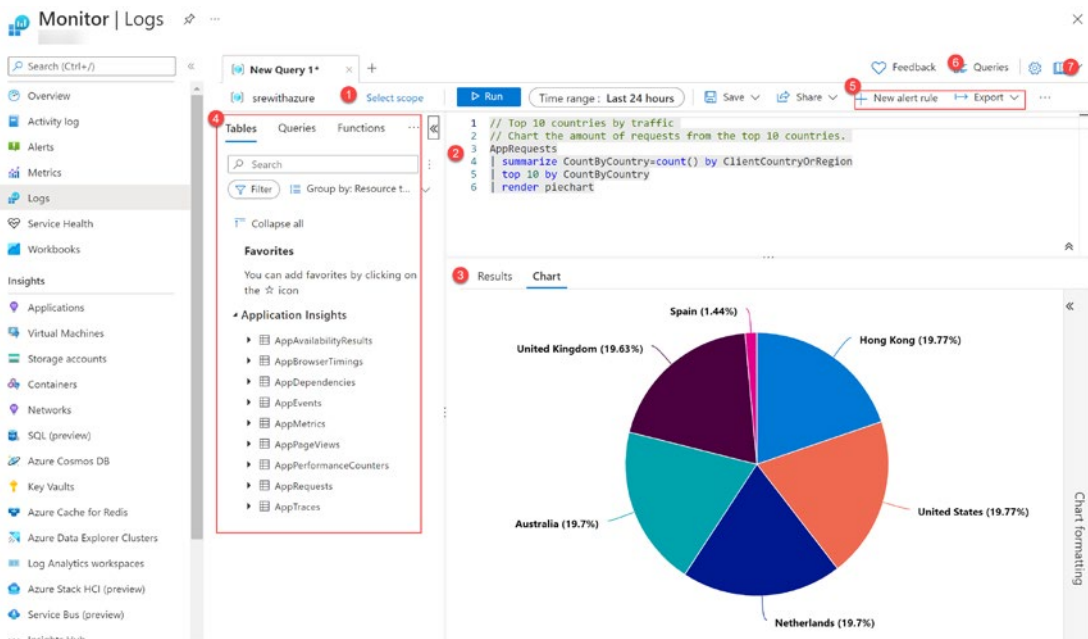


Figure 6-16. Logs user interface

## Kusto Query Language (KQL)

Kusto or KQL is a rich querying language designed to be easy to read and author. It is the querying language you use for analyzing the data collected in your workspaces. It is also used for Azure Data Explorer (<https://docs.microsoft.com/en-us/azure/data-explorer/data-explorer-overview>); actually, it was created for it, and now it has extended to Azure Monitor and other products (not all operators are supported).

For those of you having experience with SQL and Splunk, the product offers documentation to help you understand the differences between those languages:

- SQL to Kusto: <https://docs.microsoft.com/en-us/azure/data-explorer/kusto/query/sqlcheatsheet>
- Splunk to Kusto: <https://docs.microsoft.com/en-us/azure/data-explorer/kusto/query/splunk-cheat-sheet>

If you do not have experience with the previously mentioned tools (and even if you do), a lot of free documentations/tutorials are provided to learn Kusto language. A full book could be dedicated to teaching this querying language, but it is not the focus of the book.

Check the materials/links shared on the following link; it includes cheat sheets, Microsoft Learn modules, and community blogs/content: <https://docs.microsoft.com/en-us/azure/sentinel/kusto-resources>. Check out the mentioned GitHub repository maintained by Rod Trent (it points to his eBook, video channel, and even Merch store): <https://github.com/rod-trent/MustLearnKQL>.

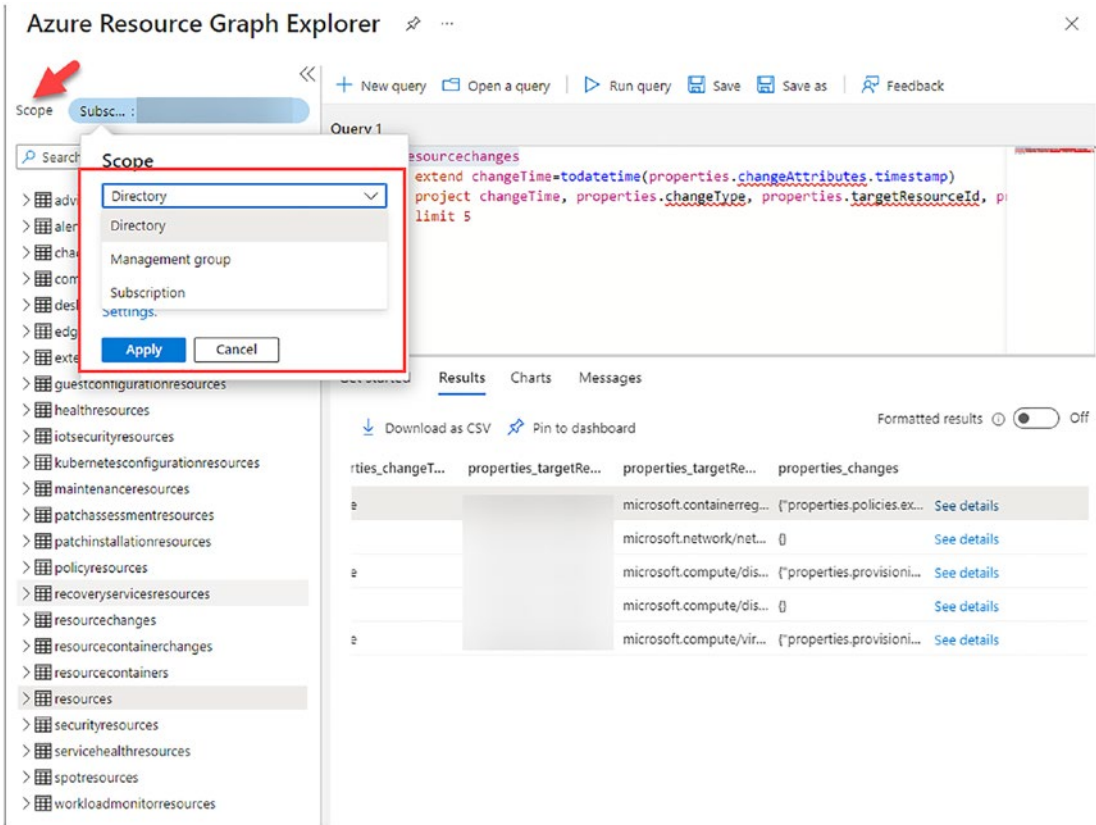
If you want to practice with KQL and you miss demo data, try this demo workspace: <https://aka.ms/LADemo>

The demo explained at the end of the chapter will show some practical ways to use this querying language.

## Azure Resource Graph

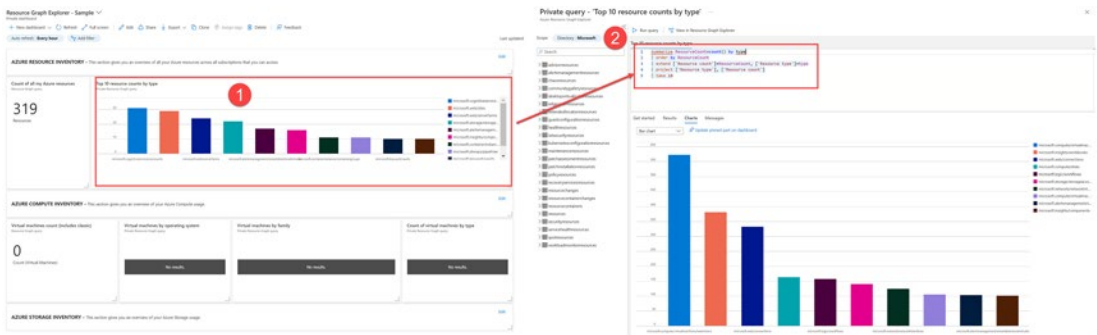
Azure Resource Graph (<https://docs.microsoft.com/en-us/azure/governance/resource-graph/overview>) is a service used to help you manage your Azure cloud environments, providing you a querying experience powered by Kusto queries (KQL). Explore resources (and their properties) across subscriptions to help you govern your environments. You can even query resource properties changed within the last 14 days.

Resource graph queries can be executed from tools like Azure CLI, PowerShell, and Portal. Also, SDKs are offered for the main programming languages. You can select the scope (see Figure 6-17) for your queries: Directory (selected tenant), Management Group, or Subscriptions.



**Figure 6-17.** Azure Resource Graph Explorer scopes

Check the following Azure Dashboard examples based on Resource Graph queries: <https://github.com/Azure-Samples/Governance/tree/master/src/resource-graph/portal-dashboards>. You can download the JSON files and directly import Azure Dashboards into your environment. Clicking on the tiles given will show you the query used (see Figure 6-18).



**Figure 6-18.** Azure Dashboard and related Resource Graph query

## Application Insights

Application Insights (<https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview>) is the tool provided by the Azure Monitor ecosystem for Application Performance Management (APM). It will help you understand all the moving parts of your complex application architectures.

Application Insights is a tool developers have to embrace in order to give visibility to the actions the application code is executing. SRE/DevOps should be embracing the cultural change that monitoring is not only the responsibility of operations engineers but is everyone's responsibility. When a new feature is designed for a solution, it should come along with the SLOs/SLIs for it and the indicators that need to be measured from code. Developers will need to use Application Insights SDK to obtain the full potential of this powerful tool.

## Instrumentation Options/Setup

Application Insights offers different options for data collection:

- **Auto-Instrumentation (No-Code):** Developers can instrument their applications with minimal effort, just by enabling automatic telemetry collection. It will be able to collect metrics, request, or dependency calls made by your application. You may have seen the Application Insights tab, for example, in Azure App Services or Azure Functions. Many environments/languages support this option; even

on-premises apps can be instrumented using the Application Insights agent. Check the full list here: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/codeless-overview#supported-environments-languages-and-resource-providers>.

---

**Note** Personally, I only recommend this option when using SDK is not an option (e.g., development is externalized, and you do not have access to customize the code). The SDK releases the full potential of the tool in terms of features and customization.

---

- **SDK:** Application Insights offers SDKs for multiple programming languages like .NET, Java, Node.js, Python, and JavaScript. **It releases the full potential of the tool**, as we can completely customize “what” telemetry will be collected, “how” it will be collected, and even how it will be sent to the Application Insights instance in the cloud. This section will provide some of those examples. **Almost any application written in a supported language and with connectivity to Azure, regardless if it is front end, middleware, or back end, can be monitored.**
- **OpenTelemetry (new, still in progress):** The future option based on an open source telemetry project(<https://opentelemetry.io/>) used to collect telemetry data (metrics, logs, and traces). Application Insights can be used as the data store when using OpenTelemetry vendor-neutral instrumentation SDKs. Nowadays, out of the three observability pillars, it only supports distributed tracing. **Metrics and logs are still in progress; as a result, the book will focus on the SDK option.** Check the website for future updates: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/opentelemetry-overview>

New Application Insights instances will be using a Log Analytics workspace as the data store (previously, classic option, used to store data on their own back end). When an instance is created, the following options are provided to authenticate/connect to your Application Insights instance from your application code (mainly when using SDK):

- **Instrumentation key:** Identifier for the Application Insights instance. Previous SDKs of Application Insights used to allow data ingestion just using the Application Insights instrumentation key. **Migrating to the next options is recommended (instrumentation key support ends in 2025):** <https://docs.microsoft.com/en-us/azure/azure-monitor/app/migrate-from-instrumentation-keys-to-connection-strings#new-capabilities>.
- **Connection string:** It defines where telemetry data has to be sent. Connection strings can be customized for proxy redirects (used in some on-premise solutions): <https://docs.microsoft.com/en-us/azure/azure-monitor/app/sdk-connection-string?tabs=net#scenario-overview>. Keep your connection string as environment variable (or Azure App Configuration) with the name APPLICATIONINSIGHTS\_CONNECTION\_STRING.
  - **Azure AD-based authentication (preview, recommended):** Used to only allow authenticated telemetry using Azure AD. You can disable local authentication to ensure only Azure AD identities (such as Managed Identities or Service Principals) are allowed to ingest telemetry: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/azure-ad-authentication?tabs=net>.

## Features

Let's cover some of the features offered by Application Insights.

### Application Map

Get a quick representation of your application and its components to quickly detect failures and bottlenecks. Components are mainly calls your application is making to external solutions like APIs, SQL databases, Cosmos DB, storage queue, storage table, etc. Even applications using different Application Insights resources could be shown together in the map if components are related.

When clicking an external component, the most frequent errors like failed or slowest request will be shown for further investigation. The Intelligent View (preview) option applies ML to help you identify possible issues based on past performance: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-map?tabs=net#application-map-intelligent-view-public-preview>.



Figure 6-19 shows the topology of an application, composed of various Azure services and availability tests created with Application Insights. The role name under the instance can be modified using SDK (covered later).

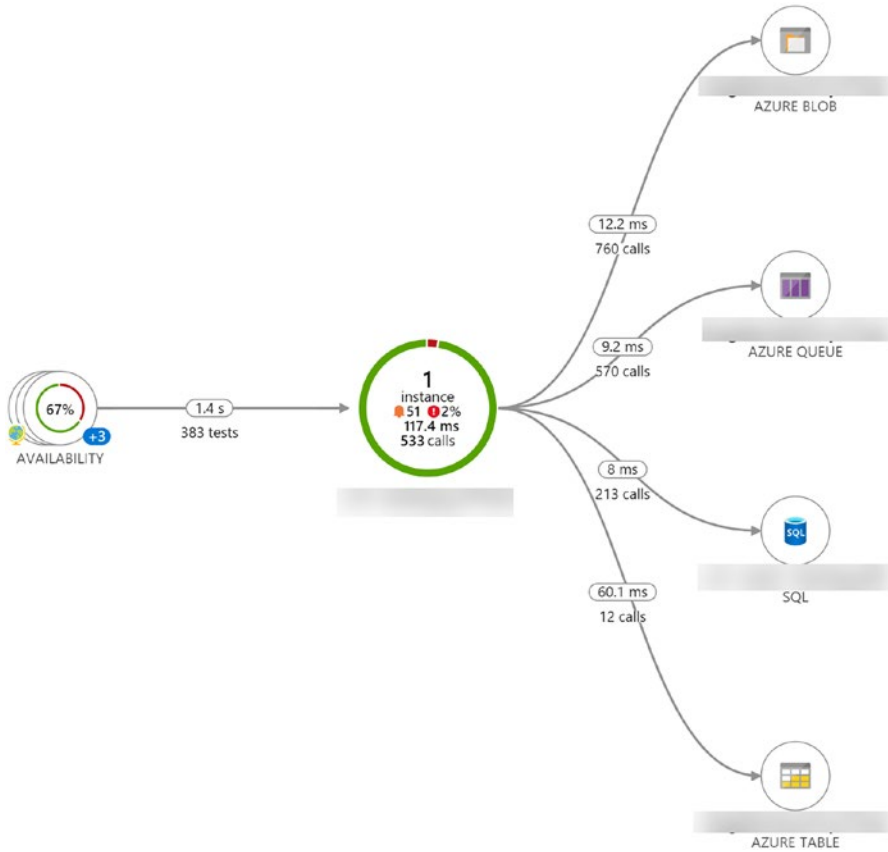


Figure 6-19. Application Map

## Smart Detection

The Application Insights Smart Detection feature learns from the past behavior of your application to detect anomalies using ML and alerts like slow responses or degradation of services. It has its own alerting system that sends notification emails, but now you can migrate to Azure Alerts that can use Action Groups/Action Rules for centralizing the alert management experience (preview): <https://docs.microsoft.com/en-us/azure/azure-monitor/alerts/alerts-smart-detections-migration>.

## Live Metrics Stream

Also known as QuickPulse, it will let you watch your application performance in real time using its noninvasive diagnostics tool (see Figure 6-20).

Live Metrics uses a different endpoint (<https://live.applicationinsights.azure.com>) compared to the data ingestion feature (<https://dc.applicationinsights.azure.com>). It also behaves differently to the Metrics Explorer or Log Analytics solutions. In the case of Live Metrics, data is only sent when the pane is opened; it is shown within a second, and once shown, it is discarded. For Metrics Explorer and Log Analytics, data is always collected; it is aggregated over minutes and retained for 90 days (default) up to 2 years.

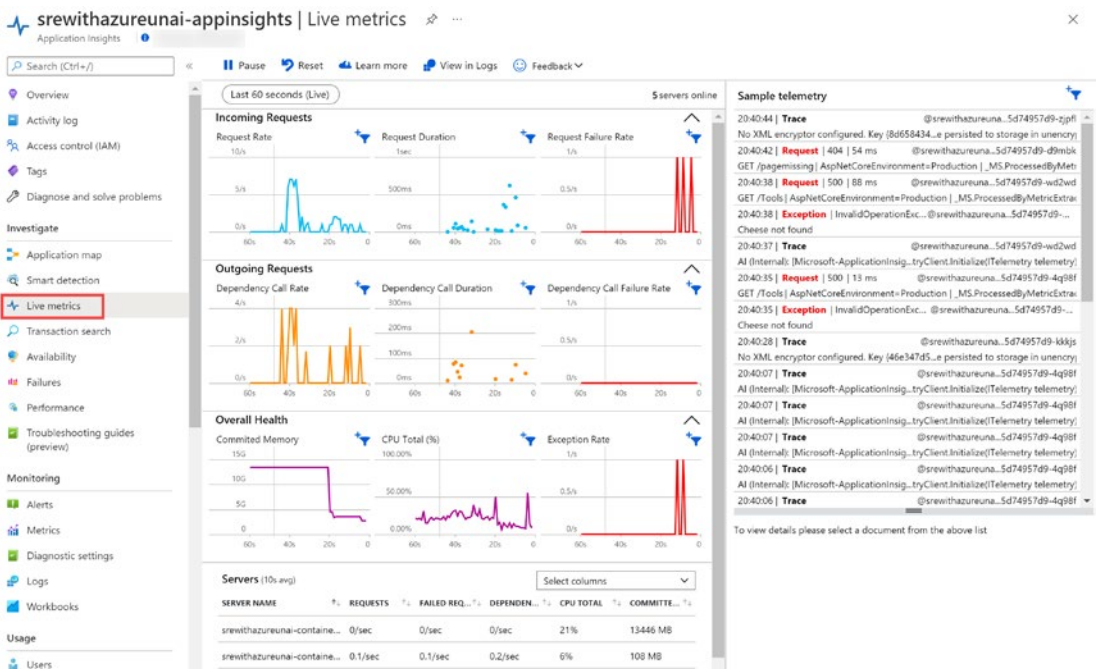


Figure 6-20. Live Metrics Stream

## Transaction Search

It is mainly used to confirm default and customized telemetry has been ingested (see Figure 6-21). For further analysis, use KQL queries.

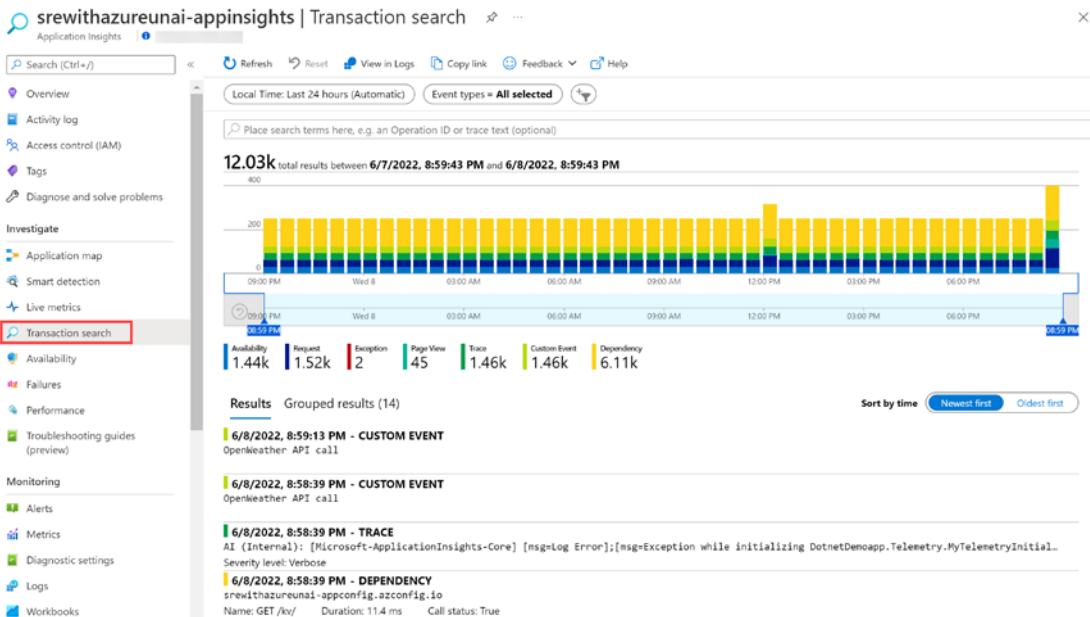


Figure 6-21. Transaction search

## Availability

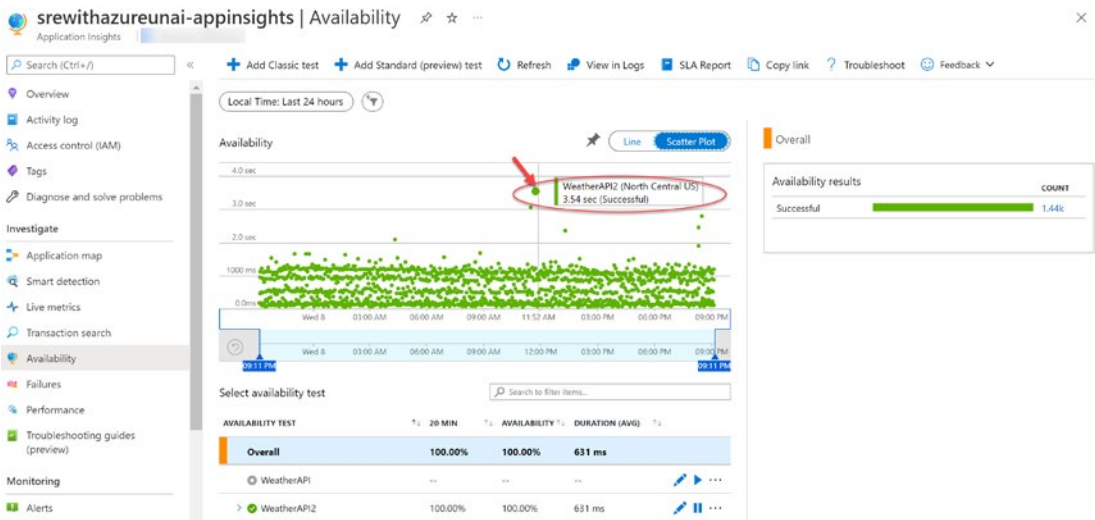
Application Insights gives you the option to run and monitor availability test running against your environment. The following options are offered:

- **URL ping test:** Simple test to validate an endpoint.
- **Standard test (preview):** New generation URL ping test that offers more customization.
- **Multistep web test (deprecated):** Not recommended as it has to be created with Visual Studio Enterprise 2019 and the next versions do not offer it.
- **Custom test using TrackAvailability:** Use Application Insights SDK to run your customized tests from any application and send the results.

The first three options use predefined machines from 16 different global points. It may not be the best option for solutions running behind firewall rules (you would have to open traffic for all those machines). Using the TrackAvailability option, you can run

tests from any infrastructure/application you want. For example, you could run tests using an Azure Function: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/availability-azure-functions>.

Alerts can be defined for failing, and the given reports can be used to review the executed tests. As shown in Figure 6-22, even if tests are successful, you may want to study why completion time was bigger for specific regions (study if performance could be improved for those users).



**Figure 6-22.** Availability tests

## Failures/Performance

Both Failures and Performance tabs will show latest information divided by Server/Browser views.

As part of the Performance option, a feature called **Profiler** can be enabled to identify “hot” code path performing “slower.” Profiler is supported for the following scenarios: <https://docs.microsoft.com/en-us/azure/azure-monitor/profiler/profiler-overview#supported-in-profiler>.

In the case of Failures, there is a feature called **Snapshot Debugger** that can automatically collect a debug snapshot when exceptions occur, giving you the chance to see the state of the code at the moment of failure. The following scenarios are supported: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/snapshot-debugger#enable-application-insights-snapshot-debugger-for-your-application>.

## Troubleshooting Guides

It is a solution based on Azure workbooks where developers can host Azure workbooks for both analyzing existing environments and documenting possible mitigations for known issues.

## Logs

The section under Monitoring (see Figure 6-23) is offered the same way as we see it in other Azure resources (nothing specific to Application Insights). **Logs** service will be scoped to the telemetry collected by Application Insights to run the desired KQL queries.

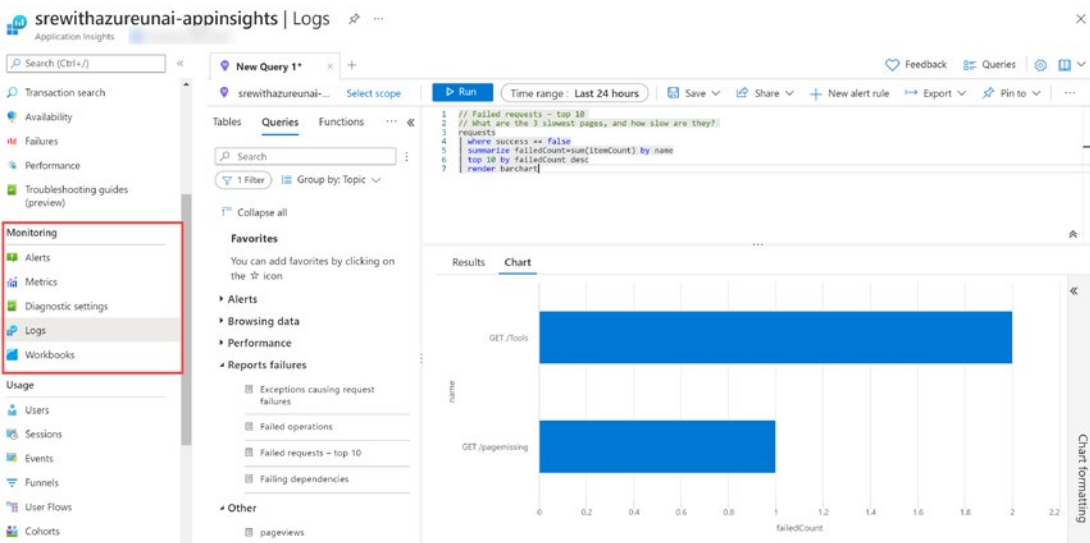


Figure 6-23. Monitoring section of Application Insights

## Usage (User Behavior)

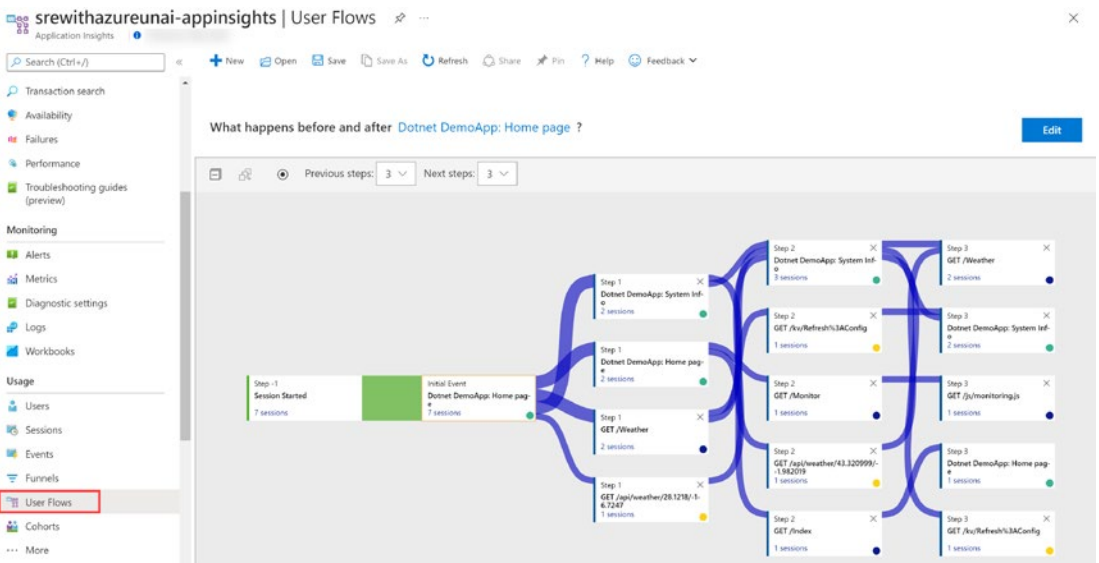
We mentioned before monitoring should always have a customer focus. The **Usage** section of Application Insights provides tools to understand how users are consuming our solution. For the best experience, both server-side and client-side code (snipped to be included in JavaScript) should be tracked.

The solution offers many different views (based on Azure workbooks) to analyze user behavior (**cohorts** can be defined to scope analysis to groups of users):

- **Users/Session** pages to understand where users are located and what browser and operating systems are used. This could be critical

information; for example, it could be used for localizing solutions closer to users or creating automated testing efforts that target the most frequent browser/OS combinations.

- **Retention** page to identify retention rates.
- **Events** page to analyze the most frequently used PageViews or custom events like purchase of items (using TrackEvent API from the SDK).
- **Funnels** page lets you define important workflows in your solution and helps you analyze conversion rates (and possible impact properties) from one step to next one. For example, if your application is an e-commerce website, you could define the steps and measure conversion rates: 1. Home Page ► 2. Product Add to Cart ► 3. Checkout Started ► 4. Order Successful.
- **User Flows** lets you choose an initial event, and it will show how your customers “move” around your application (see Figure 6-24).



**Figure 6-24.** User Flows

## Customized Application Insights Using SDK

As mentioned before, Application Insights SDK lets you use the full potential of the product, from behavior customization to custom telemetry options. Let's cover some of those options in this section.

### Application Insights API

Even if Application Insights is able to collect a lot of telemetry by default, using the SDK will be critical to collect application logic data. For example, by default, Application Insights will be able to detect many HTTP requests or dependency calls, but it does not really know if we are running an e-commerce website or a banking system. Using the APIs on the following reference, you have the power to decide the telemetry that is collected, using custom telemetry like `TrackDependency()` for dependencies not detected by default: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/api-custom-events-metrics#api-summary>

Properties (string values) and metrics (numeric values) can be provided to the customized telemetry, used later for creating charts and filtering. For example:

```
// Set up some properties and metrics:
var properties = new Dictionary <string, string>
    {{"game", currentGame.Name}, {"difficulty", currentGame.Difficulty}};
var metrics = new Dictionary <string, double>
    {{"Score", currentGame.Score}, {"Opponents", currentGame.
OpponentCount}};

// Send the event:
telemetry.TrackEvent("WinGame", properties, metrics);
```

### Advanced Configuration for Application Insights

An Application Insights behavior can be modified using the following options given by the SDK:

- Each language SDK will give you options to modify enabled/disabled services in code. You can also completely delete services by deleting modules in charge of collection automatic telemetry.

- **Sampling:** Adaptive sampling is enabled by default, but fixed-rate sampling is supported (<https://docs.microsoft.com/en-us/azure/azure-monitor/app/sampling#configuring-adaptive-sampling-for-aspnet-core-applications>).
- **TelemetryInitializer:** Can be used to enrich telemetry properties; for example, you could include the application version to each telemetry item.
  - **RoleName/RoleInstance:** These two properties are frequently defined on the TelemetryInitializer to identify the running instances when using Application Map. For example, if a front end for your solution is composed of three VMs, you could define RoleName=app-frontend and RoleInstance=app-frontend-vm1 for VM1.
- **TelemetryProcessors:** After the initializer is used, telemetry is created, and it can go through multiple processors to modify and filter telemetry before it is sent to an Application Insights resource.
- **Log Traces:** For customers using diagnostics tracing logs for ASP.NET solutions like ILogger, NLog, log4net, and System.Diagnostics.Trace, you can collect those traces in Application Insights.
- **Release Annotations:** You can create release annotations on Application Insights charts or workbooks (see Figure 6-25) by using the script mentioned here: <https://docs.microsoft.com/en-us/azure/azure-monitor/app/annotations#create-release-annotations-with-azure-cli>. Automate it when deploying updates using Azure DevOps or GitHub Actions Azure CLI task/action.



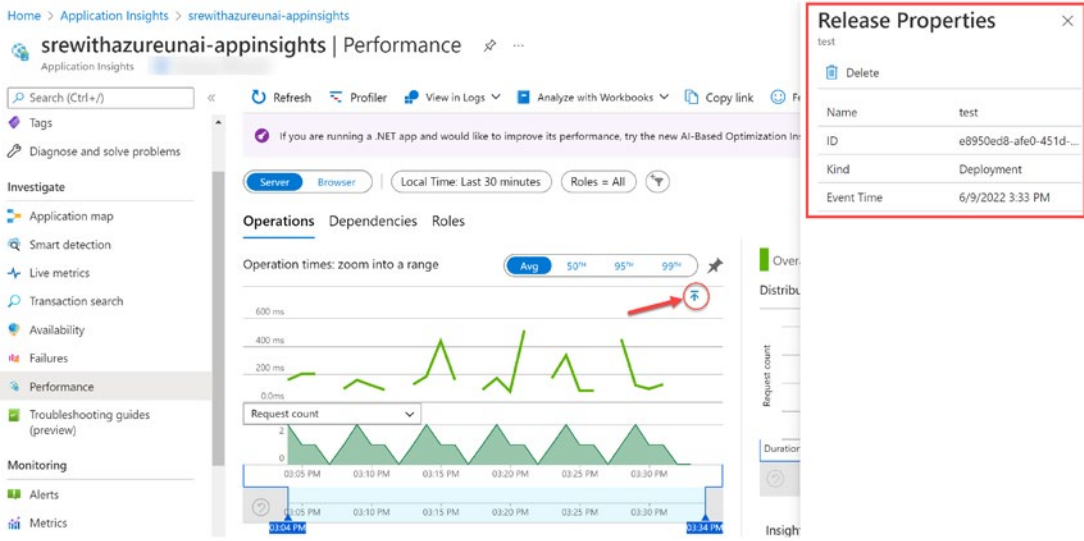


Figure 6-25. Release annotations in Application Insights

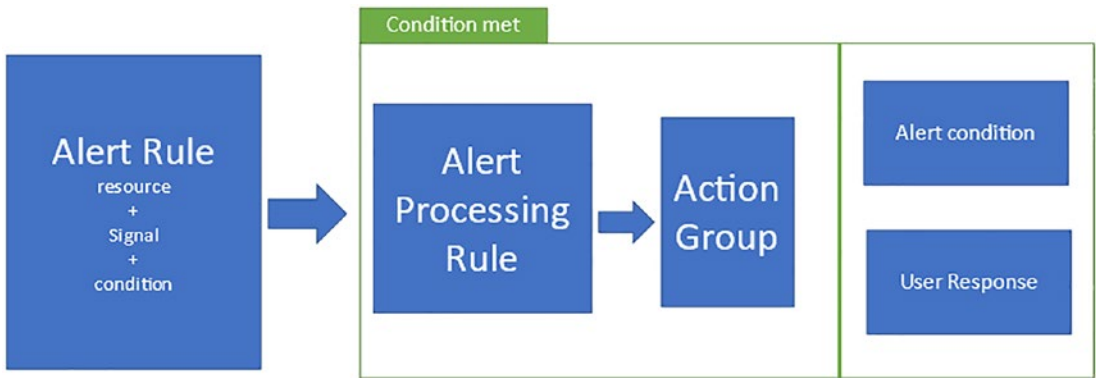
## Azure Monitor Alerts

Most of the monitoring tools mentioned previously are related to a “reactive” approach: fixing issues once they have happened. On the other hand, **Alerts** should be used for a “proactive” approach: getting notified before users notice those problems.

Azure offers an alert service that can be found on the **Alert** section of Azure **Monitor** (alerts centralized here) or individual Azure resources (alerts scoped to the resource). Alerts defined in Azure can use the following signals:

- Logs/Metrics/Traces collected by Log Analytics and Application Insights. Also, Smart Detection or Availability test offered by Application Insights
- Platform Metrics (default ones given for 93 days)
- Activity Log alerts

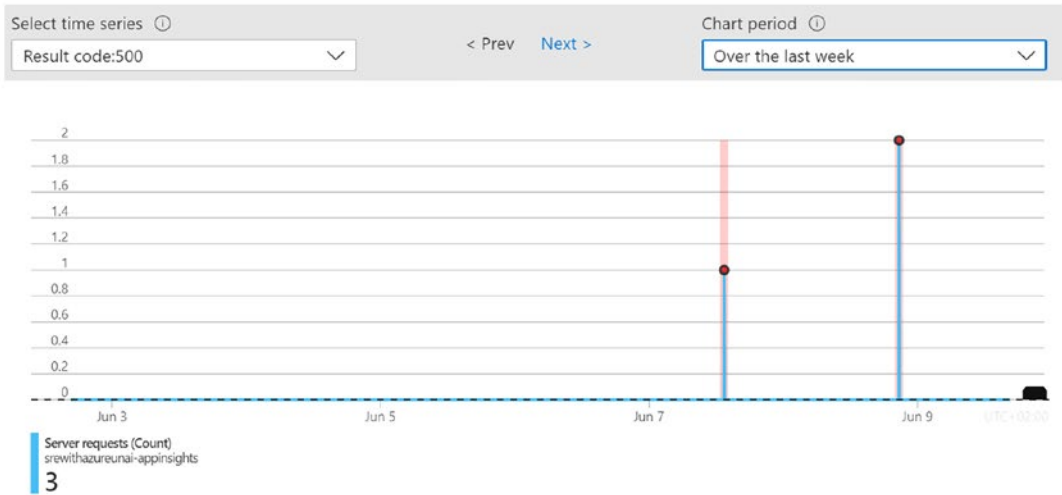
They are composed of the elements shown in Figure 6-26.



**Figure 6-26.** Azure Alerts

- Alert Rule:** It defines the resource, signal, and condition to monitor. Multiple conditions can be applied to the same rule. Target signals can be narrowed using dimensions. When defining the threshold, two types are supported: **static and dynamic**. Dynamic thresholds use ML technology to alert based on deviations, **really recommended to avoid noise and set up alerts when the threshold to apply is not clear yet**. The example shown in Figure 6-27 shows an alert rule based on an Application Insights server request with result code (dimension) different to 200. The threshold shows past situations where the alert would have been triggered.

## Configure signal logic



### Split by dimensions

Use dimensions to monitor specific time series and provide context to the fired alert. Dimensions can be either number or string columns. If you select more than one dimension value, each time series that results from the combination will trigger its own alert and will be charged separately. [About monitoring multiple time series](#)

Dimension name	Operator	Dimension values
Result code	≠	200

[Add custom value](#)

**i** Monitoring 2 time series (\$0.1/time series)

### Alert logic

Threshold **Static** **Dynamic**

Operator: **Greater than**    Aggregation type: **Count**    Threshold value: **0**    Unit: **Count**

### Condition preview

Whenever the count of server requests is greater than 0

### Evaluated based on

Aggregation granularity (Period): **5 minutes**    Frequency of evaluation: **Every 5 Minutes**

**Done**

**Figure 6-27.** Azure Alert rule

- **Alert Processing Rule:** When the rule is met, you can modify the expected behavior by suppressing alerts (e.g., useful for maintenance windows) or automatically assigning Action Groups to alerts based on filters (instead of doing it alert by alert). It helps with the management of alerts in your organization.
- **Action Group:** Triggers automation or notifications once the alerts are fired (and evaluated by processing rules):
  - **Notifications:** Email, push notification
  - **Automation:** Runbooks, Logic Apps, Functions, Webhooks, ITSM integration, or event hubs
- **Alert Condition:** Defined by the system to “fired” when a condition is met and moved to “resolved” when a condition clears (checkbox in Figure 6-28 should be enabled).

**Alert rule details**

Provide details on your alert rule so that you can identify and manage it later.


Alert rule name ⓘ Container App -Failed Request

Description Specify the alert rule description

Subscription ⓘ

Resource group ⓘ SREwithAzure

Severity \* ⓘ 3 - Informational

Automatically resolve alerts ⓘ  

**Figure 6-28.** *Automatically resolve alerts*

- **User Response (state):** The user can define the alert state to “New,” “Acknowledged,” or “Closed.”

Alerts defined for our complex environment should be “actionable.” Avoid creating alerts for successful actions/states; you just want to be notified about exceptional situations that may be putting your SLOs at risk.

Once your organization starts getting more mature on this practice, you will be able to filter noise (non-actionable alerts) and trigger automation that may remediate or fix issues raised by alerts automatically.

Resiliency helps you to recover from failures. Availability gives your users access to your workload at all times. Design monitoring solutions that expect failures and recover from them.

## [DEMO] Tracking SLI/SLO/SLA Using Application Insights and Log Analytics

The example shown in this chapter is using the demo architecture used in Chapter 5 (see Figure 6-29), which hosts a containerized .NET 6 website in an Azure Container. Let's see how Application Insights and Log Analytics can be used to track our solution as explained in the chapter. Demo files can be found here: [https://github.com/unaihuete-org/SRE\\_with\\_Azure](https://github.com/unaihuete-org/SRE_with_Azure).

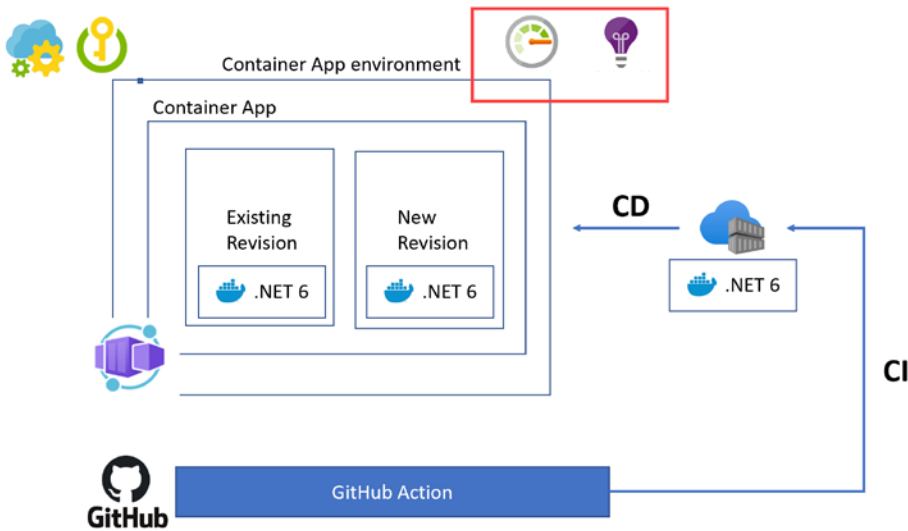


Figure 6-29. Demo solution

The .NET 6 website is using Application Insights SDK to collect telemetry from the running application. Application Insights is set up in the following way:

- A telemetry initializer ([https://github.com/unaihuete-org/SRE\\_with\\_Azure/blob/master/src/MyTelemetryInitializer.cs](https://github.com/unaihuete-org/SRE_with_Azure/blob/master/src/MyTelemetryInitializer.cs)) is used to include App version and RoleName properties in every telemetry.

- Initializer is used and Application Insights instance is connected in ([https://github.com/unaihuete-org/SRE\\_with\\_Azure/blob/master/src/Program.cs](https://github.com/unaihuete-org/SRE_with_Azure/blob/master/src/Program.cs)) using the connectionString taken from an Azure App Configuration resource.

Figures 6-30 and 6-31 show both telemetry initializer properties get exposed in the portal (role name and appVersion).

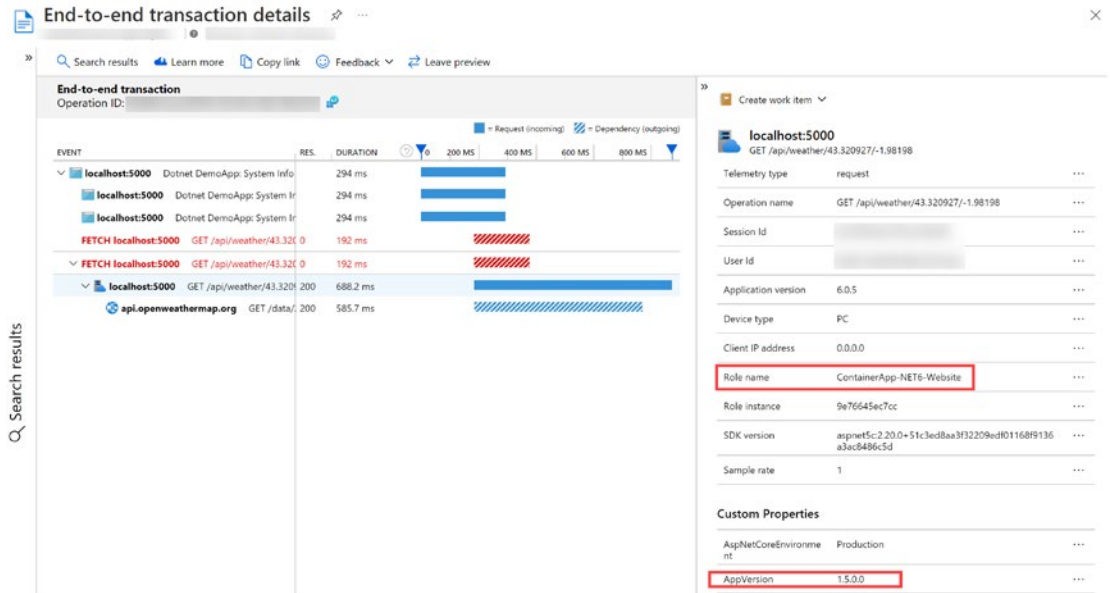


Figure 6-30. Transaction details

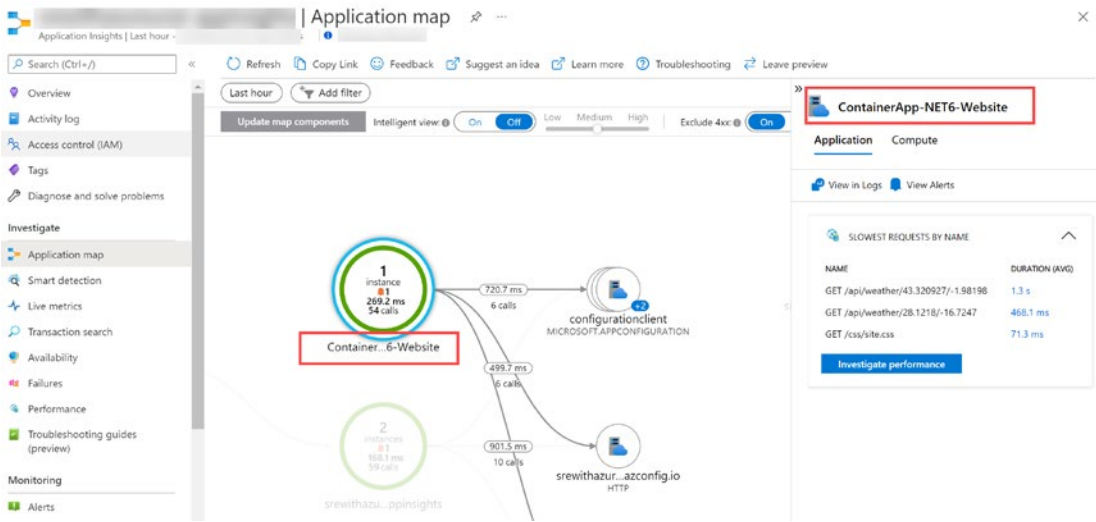


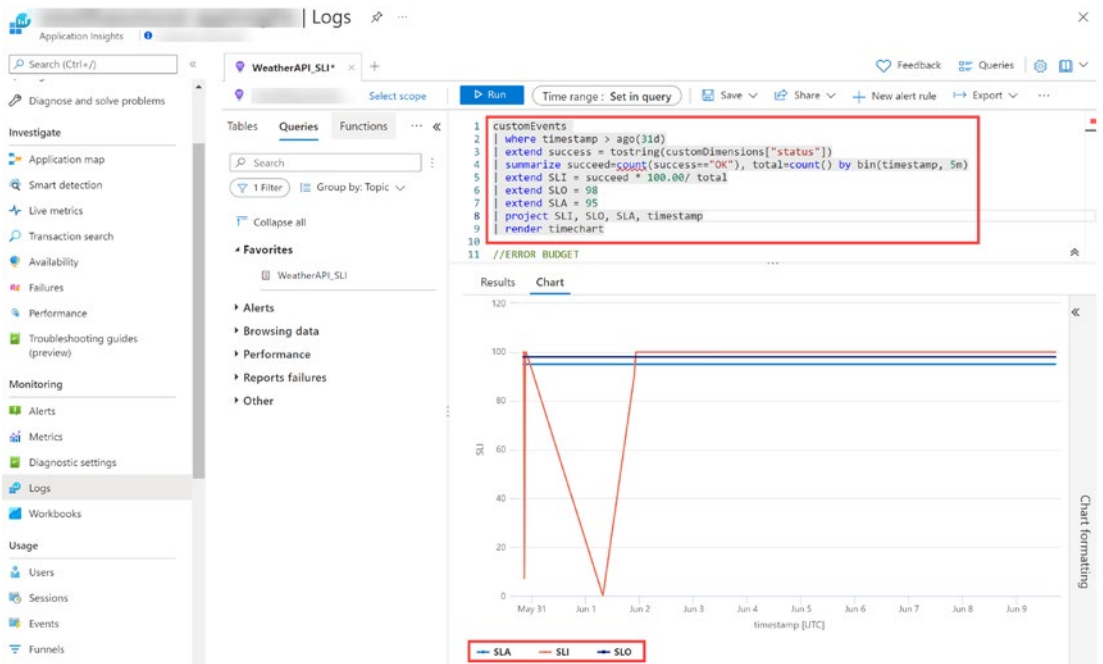
Figure 6-31. RoleName in Application Map

The website hosted in Azure Container App offers a tab to give the user weather forecast information. In the ApiHelper.cs file ([https://github.com/unaihuete-org/SRE\\_with\\_Azure/blob/master/src/ApiHelper.cs](https://github.com/unaihuete-org/SRE_with_Azure/blob/master/src/ApiHelper.cs)), a `TrackEvent()` telemetry is created to track the status of the calls made to this service, providing global position (latitude/altitude), response code, and API key used on the weatherForecast API backend call.

For the service, as an SRE for the demo, the following have been defined:

- **SLI:** Requests made to the weather forecast feature (using the `TrackEvent` mentioned before) with successful response for the user during the last 31 days.
- **SLO:** 98% of those calls should be successful during the last 31 days.
- **SLA:** 95%. Your contract agreement is normally lower than SLO to have some margin when things go wrong.
- **Total error budget:** 2% of monthly calls.

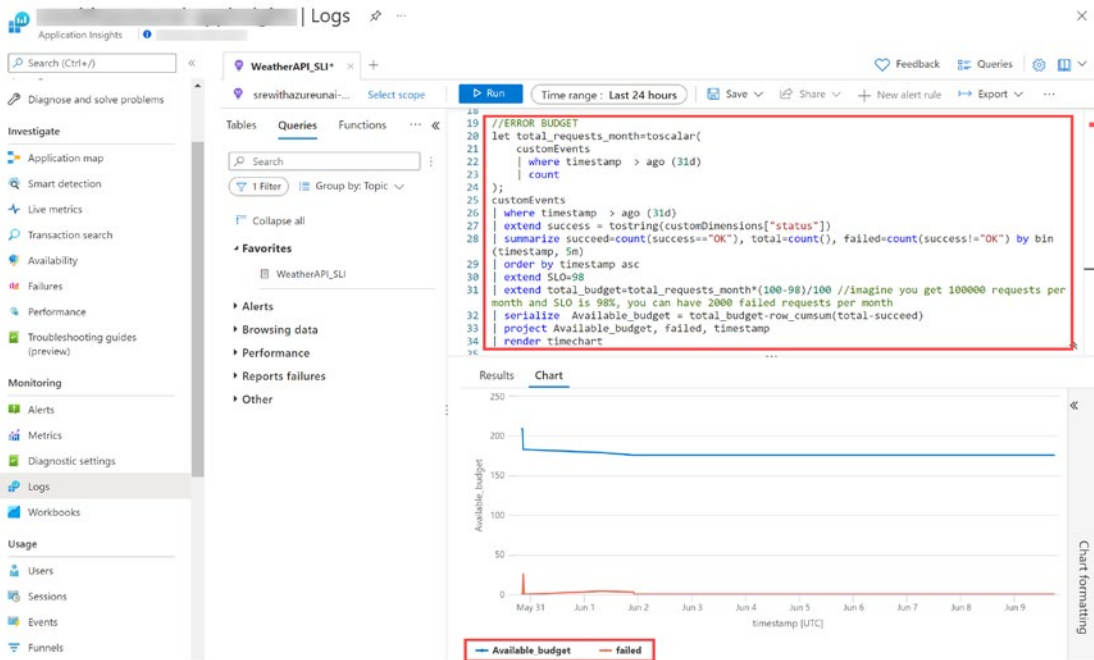
The Kusto query shown in Figure 6-32 is used to show all these metrics in a chart, as you can see there was a heavy downtime for the service at the beginning of the evaluation period.



**Figure 6-32.** Kusto query for SLI/SLO/SLA

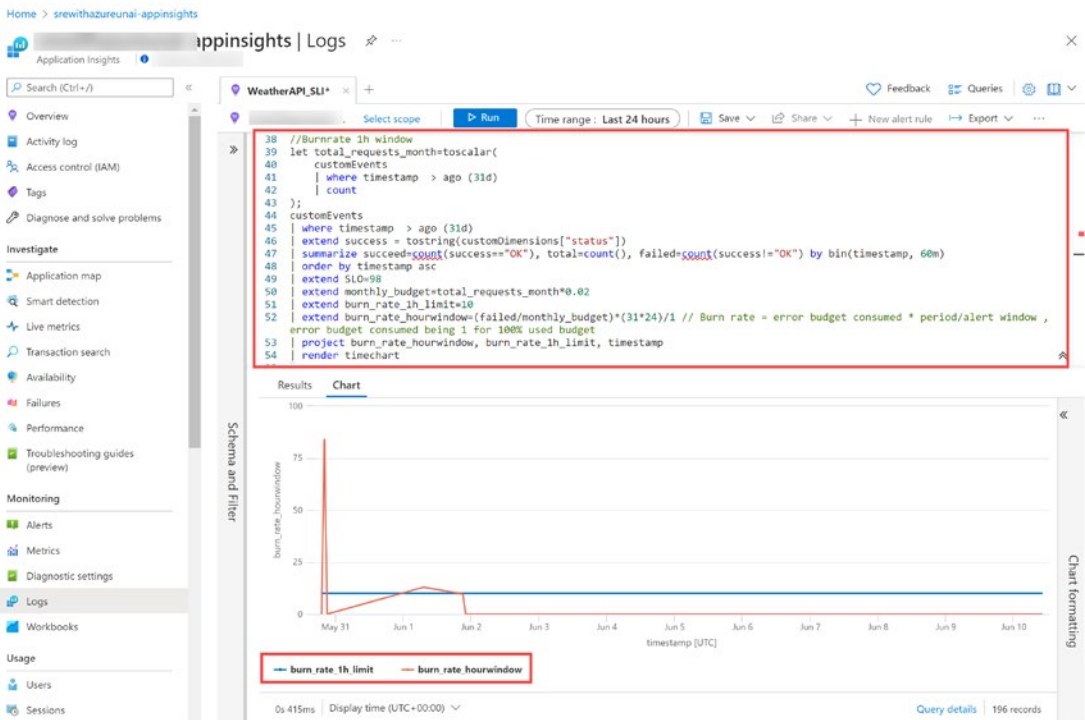
The **error budget** can also be measured by calculating first the total amount of calls made during the last 31 days and see how it decreases with every failed request (see Figure 6-33).





**Figure 6-33.** Error budget Kusto query

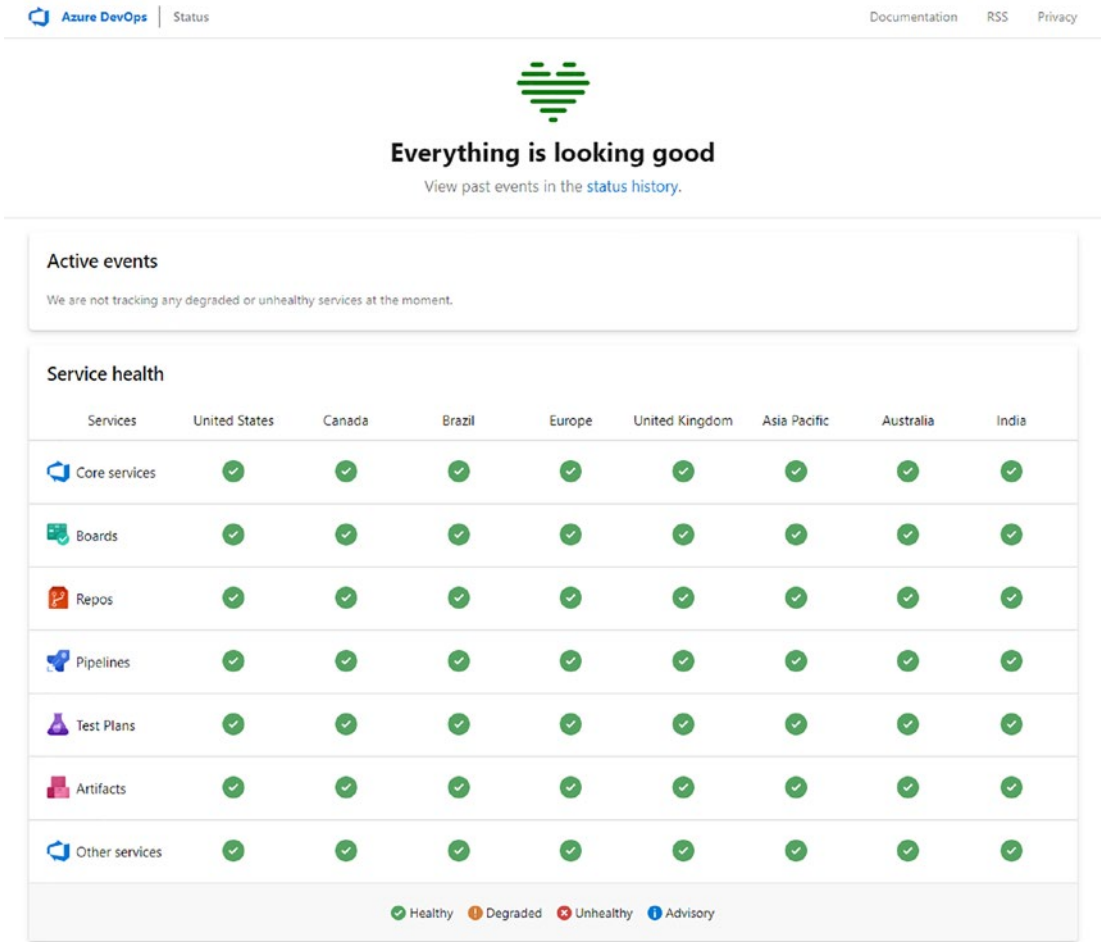
In order to react on time, you could measure **burn rate** in different time windows (to avoid noise); for example, track those burn rates above 10× in a 1-hour window (**fast burn alerts**, budget would be consumed in a day) and 2× in a daily window (**slow burn alert**). The query in Figure 6-34 shows how to track the 1-hour window burn rate.



**Figure 6-34.** Burn rate (1-hour window) Kusto Query

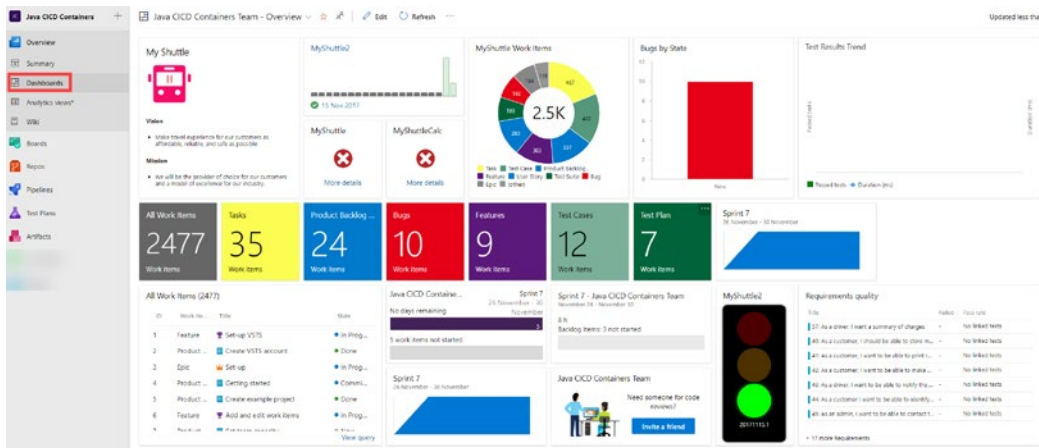
## Azure DevOps

Similar to the Azure Status website, Azure DevOps also offers status information for the different global instances of the service, link here: <https://status.dev.azure.com/>. When working with Azure DevOps, you may want to monitor the status of the platform itself when an unusual behavior is perceived (see Figure 6-35). It is recommended to get subscribed for notifications.



**Figure 6-35.** Azure DevOps status page

It would also be interesting to create your own Azure DevOps Dashboards in order to monitor the activity of your projects, including planning, repository, or pipeline/ tests activity (see Figure 6-36). Azure DevOps offers an extensive list of widgets you can include: <https://docs.microsoft.com/en-us/azure/devops/report/dashboards/widget-catalog?view=azure-devops>.

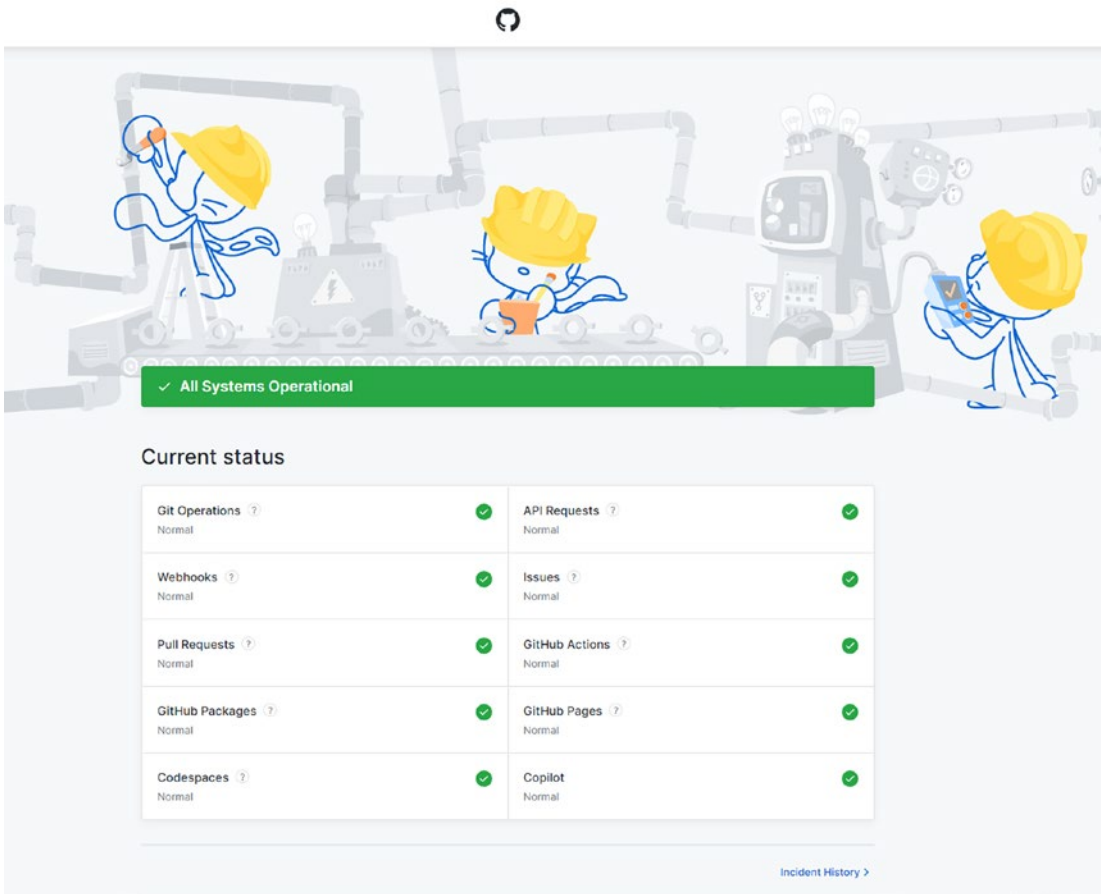


**Figure 6-36.** Azure DevOps Dashboard

For more advanced reports that can be externalized to tools like Power BI, you should take a look at the Analytics service offered by Azure DevOps. Analytics is the reporting platform offering historical data from your ADO Projects; it includes more advanced integrated reports on the tools and data externalizing options using Power BI/OData queries. In the following docs you can find the data (and options) available through the service: <https://docs.microsoft.com/en-us/azure/devops/report/powerbi/data-available-in-analytics?view=azure-devops>

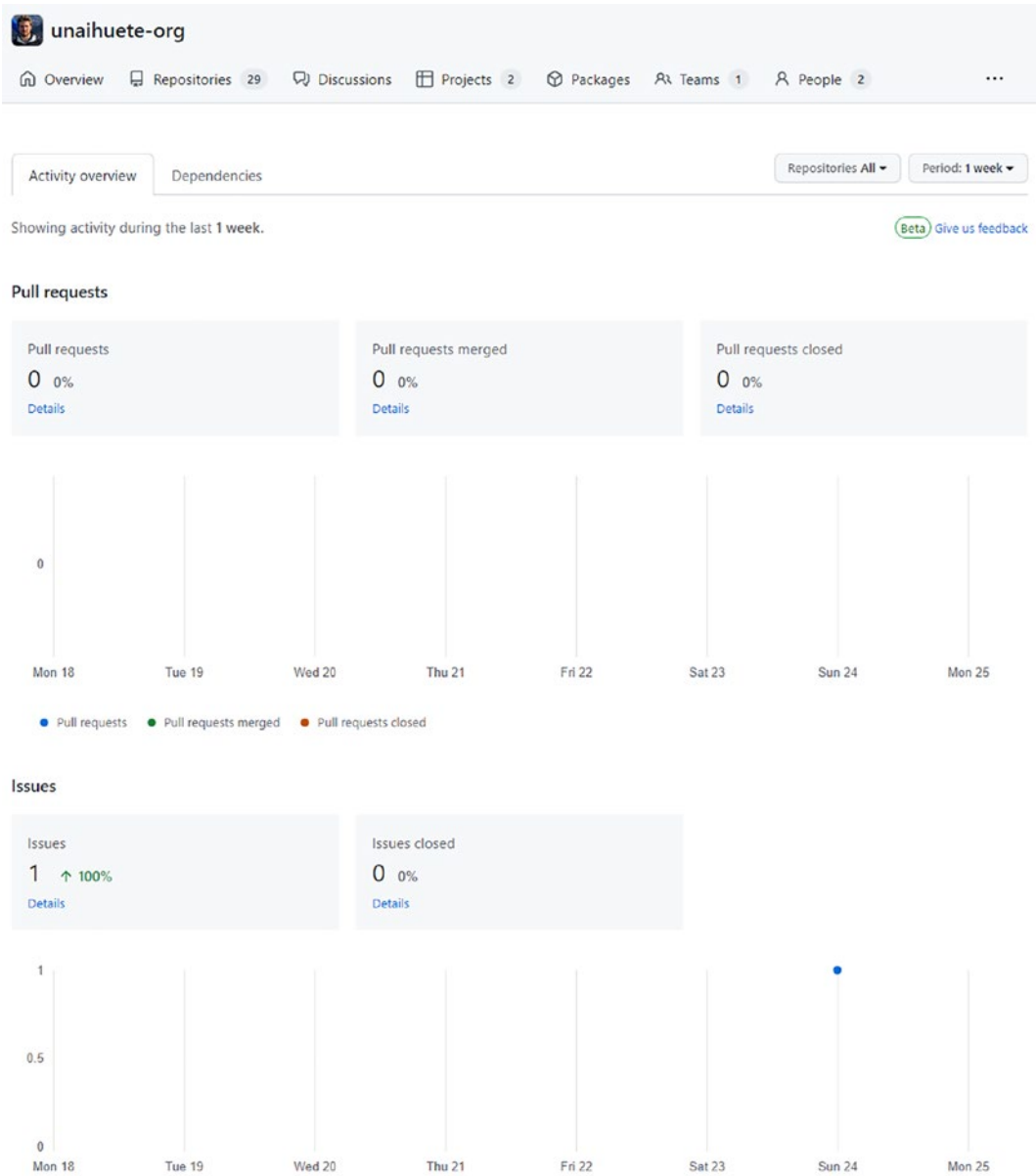
## GitHub

Like previous platforms, GitHub also offers a status website for the offered services on the following link: [www.githubstatus.com](http://www.githubstatus.com) (see Figure 6-37). You should also get subscribed for notifications when using this DevOps platform.



**Figure 6-37.** *GitHub status page*

GitHub also offers some reporting with the **Insights** service located on both repository and organization levels (see Figure 6-38). Compared to Azure DevOps Dashboards and analytics, you will see it lacks some GitHub Actions reporting features (at least for now). You can always externalize information using REST API and create your customized reporting solution if needed. GitHub Insights mainly shows information related to repository planning (issues), contribution (git repository), and dependency management data (used dependencies and security advisories).



**Figure 6-38.** GitHub organization Insights

## Summary

In this chapter, you were introduced to the base practice mentioned on Dickerson's hierarchy of reliability: monitoring.

The chapter was focused on explaining the basic concepts of Monitoring and Observability and the importance of Operational Awareness on the complex solutions organizations create nowadays. Before covering Azure monitoring technologies, it reminded about concepts seen in Chapter 2, like SLI/SLO/SLA and the idea of error budget/burn rate, as these metrics are critical to an SRE.

This chapter focused on explaining the Azure technology stack available to meet your observability goals and helping you track the previously mentioned indicators.

The next chapter will focus on showing an efficient way to handle incidents and explaining how to learn from previous experiences using blameless postmortems.

## CHAPTER 7

# Efficiently Handle Incident Response and Blameless Postmortems

In this chapter, you will be introduced to incident response and blameless postmortems practices, two of the pillars mentioned in Dickerson's hierarchy of reliability.

Incidents cannot be avoided; there is no 100% reliable system, so your focus should be on reducing the possible negative impact generated by these events.

By the end of this chapter, you should be able to do the following:

- ✓ Understand incident response (definition, life cycle/phases, and roles)
- ✓ Improve communication practices and implement ChatOps using Microsoft tools
- ✓ Learn how to detect incidents with Azure
- ✓ Learn how to diagnose incidents with Azure
- ✓ Learn how to remediate incidents with Azure
- ✓ Understand blameless postmortems

## Incident Response (IR)

As explained in previous chapters, organizations are designing architectures and automated processes to make sure no issues will arise in their running environments. Our CI/CD processes include branching strategies, pull requests, a great variety of



testing workloads, and modern deployment practices with that objective in mind. Nevertheless, live site incidents will always keep on happening, preventing all incidents is unrealistic, and thinking “it will never happen to me” will not help.

Changing the mindset is the key for improvement. Organizations need to embrace a culture where engineers think breaches/incidents will happen and act upon that. Thinking “things will never go wrong” just leaves you unprepared for those situations.

Our main objective from previous chapters was to reduce the amount of burden related to the issues. Incident response phase will define how we tackle live site issues. Each organization will need to define a framework with **structured guidance that will help IT staff stop, contain, and recover from incidents in an efficient way.**

This chapter will focus on showing the best practices adopted by companies for managing incidents. But what is an incident?

There is no agreed definition for it. We could define it as a service disruption that impacts your customers. Incidents could be security-focused (like denial-of-service attacks) or performance-related ones (e.g., your VM not being able to handle enough load of requests).

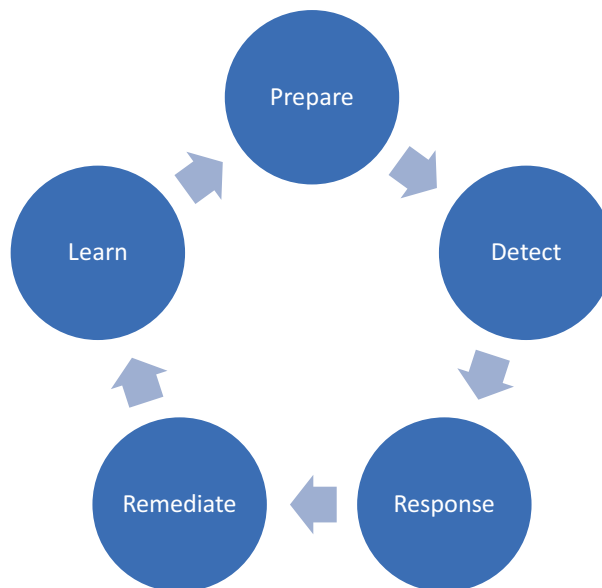
There are a variety of incident frameworks we can find to help define your incident response (IR) guidance. Let’s take a look at the main characteristics of frameworks/templates defined by some public IT organizations.

As Table 7-1 shows, most frameworks define similar logical steps:

**Table 7-1.** *Incident response frameworks*

<b>NIST</b>	<b>ISO</b>	<b>SANS</b>
Preparation	Prepare	Preparation
Detection and analysis	Identify	Identification
Containment, eradication, and recovery	Assess	Containment
	Respond	Eradication
		Recovery
Postincident activity	Learn	Lessons learned

- **Preparation/readiness:** This will be the related prevention activity (resilient architectures, automated processes, knowledge transfer, and proactive monitoring).
- **Detect/identify:** Getting notified about the issue and understanding the nature of the problem.
- **Resolving phase:** This phase is different between frameworks, but it is mainly focused on the reactive activity to resolve the issue. This will probably be the longest and most complex phase of the process.
  - **Response:** First, we focus on containing the issue, reducing the impact surface.
  - **Remediation:** Then, we work on fixing/eradicating:
    - Testing/validating that the issue is fixed.
- **Postincident/learning/analysis:** This will be the focus of the second section of the chapter (blameless postmortems). Every issue has to be taken as an opportunity to learn (and avoid similar problems in the future).



*Figure 7-1. Incident life cycle*

## Incident Response Pillars

How fast our teams will be able to execute the phases mentioned will define the ability to recover from unwanted scenarios. The best SRE teams adhere to a strict incident response process. For example, Google's incident response system is based on the Incident Command System (ICS), established in 1968 by firefighters to deal with wildfires.

The basic principles for an incident response process are composed of

- Chain of command (or clear line of command).
- Clear roles and responsibilities.
- Documentation: Recording debugging/mitigation/collaboration activities is the **key for robust postmortems and feedback to improve our process!**
- Incident identified and remediated ASAP.

SRE teams will try to catch the most common incidents with Alerts and use automation to fix recurrent issues. For example, recurring incidents could be automatically detected with Azure Alert rules, runbooks could be triggered in Azure Automation to fix an issue (restart the service, scale it, change settings, etc.), and notifications could be sent to an SRE just to validate that the issue is fixed.

Let's take a look at the roles that need to be defined.

## Roles

Roles bring clarity to the chaotic situations incidents could create. You will clearly define the responsibility of your team members. Different roles can be found across organizations, but these could be commonly found:

- **First/second responder:**
  - **First:** The on-call engineer is notified.
  - **Second:** Backup for the first engineer (if unavailable or more people needed).

- **Scribe:** The least technical role as the main responsibility will be to document the incident: tracking events, activities, decisions taken, important information, etc. Capturing the chronology of the information will help avoid duplicated work and bring clarity to the mitigation process.
- **Communication lead/coordinator/manager:** The “public face” of the incident response team. It will keep customers informed about the status of the incident in a periodic manner. It shares information not only with customers but also with those stakeholders not involved actively in the incident (e.g., sales, customer support, marketing, etc.). It works closely with the incident commander to share customer impact and share updates with customers. It needs to have access to resolution plan information to correctly define communication strategy. It may also share updates on social media or other communication channels (product website, LinkedIn, Twitter, etc.).
- **Technical lead/subject matter expert (SME):** The owner or technical expert familiar with the investigated systems. First/second responders will escalate problems to them if help is needed. Keep a list of SMEs related to your solutions areas.
- **Incident commander/manager:** Responsible for driving the resolution and activities during the incident. They coordinate all efforts for incident resolution, “what is happening,” and “who’s doing what”; it helps engineers stay focused.

During the **preparation** phase mentioned previously, incident commanders will need to define the best communication channels for all the involved stakeholders. Throughout the **resolving** phase, they gather information and manage team members. An incident commander usually does not have the technical skills to fix issues; it collects feedback from SMEs and manages pending work. It could be said they act as Program Manager/Owner on an Agile team. Finally, they will be responsible for driving improvements identified during the **postincident/postmortem phase**.

## On-Call/Rotations

Now that responsibilities/roles have been defined, you need to set up a schedule to describe the shifts for your team members. There are many factors to take into account when defining an on-call schedule: size of the team/organization, global distribution, service ownership, and availability.

An efficient on-call strategy will strive for **balance**; coverage has to be provided to critical services, but never at the expense of the engineer's health. For example, Google places a 50% cap on the "ops" activities (on-call, tickets, manual tasks, etc.) for all SREs (<https://sre.google/sre-book/introduction/>).

These are some of the main methods for effective/sustainable rotation schedules:

- **Follow the sun shift:** Define on-call shifts based on normal working hours (used by global organizations).
- **Primary/secondary:** Define a solution where notifications will be sent to a "backup" responder in case the primary one does not acknowledge.
- **Everyone's responsibility:** One of the biggest mistakes is just involving the OPS team. Everyone should participate and "take the pager." Developers should participate when the application layer is involved. "**Sharing the pain**" will also motivate DEV teams to improve monitoring and reduce "toil."
- **Allow flexible schedules:** Key for work-life balance. Clear schedules defined weeks ahead and give options for changes (personal emergencies, same as incidents, cannot be planned).
- **Avoid meaningless alerts/reduce toil:** Nobody likes to be woken up at 2 a.m. for fixing the same issue you have seen during the last months. **Automation** will be the answer to deal with repetitive/meaningless issues.

All of the points discussed will help your organization perform better and increase not only employee retention but also customer satisfaction.

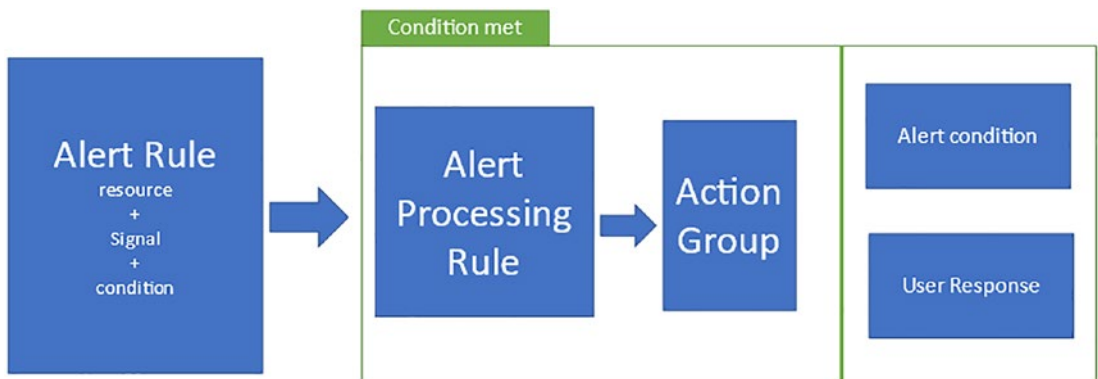
## Incident Tracking/Detection

In order to start working on an incident, it is obvious you need to be able to detect it. It determines the first phase of an incident response process.

For incident detection, you will mainly rely on your monitoring tools/process (and the ones offered by the cloud provider) to get notified about an unexpected behavior. The book has a full chapter focused on monitoring Azure solutions. But just as a reminder, an Azure Alert can be created for the following signal types:

- Metrics: Most stored by default for 93 days
- Log queries (KQL queries): Any information collected using Azure Monitor Logs and Application Insights
- Activity log events
- Azure platform health (planned maintenance, service issues, and health/security advisories): Azure Service Health (<https://docs.microsoft.com/en-us/azure/service-health/service-health-overview>)
- Availability test (run using Application Insights)

Remember, an Alert on Azure is composed of the elements shown in Figure 7-2.



**Figure 7-2.** Azure Alert structure

Review all components from Chapter 6 if needed (previously explained).

For example, you are using a Cosmos DB as part of your solution. The Cosmos DB database has been set up with a provisioned throughput of 400 RU/s (no autoscaling chosen), “Request Units (RU)” being the normalized metric for CPU/IOPS and memory

performance (<https://docs.microsoft.com/en-us/azure/cosmos-db/request-units>). If the requests sent to the Cosmos DB require more computing power than established by RUs, your Cosmos DB will respond with an HTTP response of 429. Let's create an alert based on this metric:

1. From the Cosmos DB resource **Alerts** ➤ **Create Alert** and on the **Condition** tab, select the **Total Request** signal and apply the dimension **Status Code = 429**. Define a threshold of **Count Greater than 0** as the criterion to get alerted, as shown in Figure 7-3.

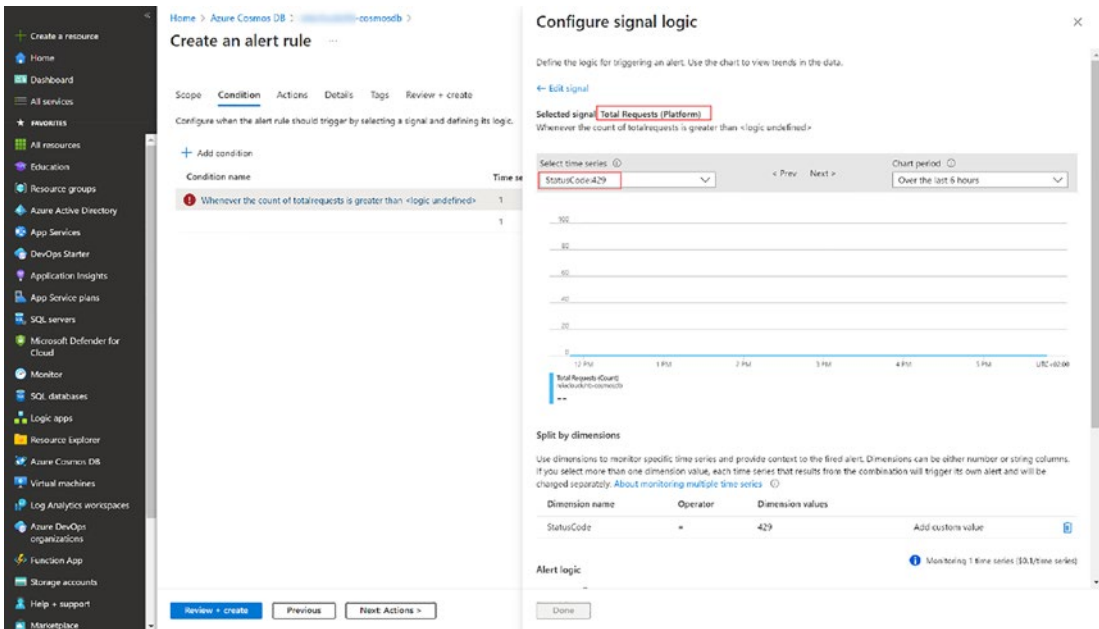


Figure 7-3. Cosmos DB alert

2. On the **Actions** tab, create an Action Group, choosing from the available options mentioned above. For example, you could
  - a. Send an email notification
  - b. Trigger a Logic App to create a conversation channel on Slack/Teams
  - c. **Fix the incident automatically** by executing Azure Automation Runbooks using the Azure PowerShell commands (<https://docs.microsoft.com/en-us/powershell/module/az.cosmosdb/update-azcosmosdbsqldatabase-throughput?view=azps-7.5.0>).

All of the discussed options work for anomalies detected either by the platform or monitoring tools used. We could categorize the detected anomalies in three big groups:

- **Proactive alerts:** The platform or monitoring tool could alert us about potential issues: maintenance windows, service degradation, long response times, etc. For example, Application Insights is able to provide proactive alerts (called **Smart Detection**) based on machine learning rules.
- **Reactive alerts:** Most alert rules are defined by engineers; in general, we get notified once the incident has happened.
- **Not tracked alerts/customer notified:** Alerts notified by our users that for some reason our monitoring tools did not track. These ones could be more difficult to deal with, as you may miss clear information for response/remediation.

During the phase of detection, you don't only want to get notified, you would like to assess the incident to get critical information for the next phases:

- Incident firing time ► capture information based on the timeline to determine cause.
- Impacted users/stakeholders.
- On-call engineer getting notified.
- Is the incident tracked in our tracking software (e.g., GitHub/Azure DevOps/ServiceNow)?
- Related technology ► Do we need the help of subject matter experts?

The next section will focus on creating a standardized approach for clear collaboration/communication and incident tracking system.



## Communication and ChatOps

Having a clear communication strategy will be critical during the incident response life cycle (see Figure 7-1):

- **Prepare:** Getting new team members ready and learning from previous outages will be extremely important. A strategy to share response plan updates, procedure changes, and lessons learned needs to be defined.
- **Detect:** Communicate detected issues to the proper stakeholders.
- **Response:** Communicate details about diagnosis, steps to proceed, and distribution of tasks.
- **Remediation:** Openly communicate how/when service was restored.
- **Analysis:** Communicate lessons learned and follow up actions.

Take a look at the historical incident examples from the Azure Status website (<https://status.azure.com/en-gb/status/history/>). A summary of incidents is provided, including the following points for each of them:

- Summary of impact ► Detected details
- Root cause ► Response/detect
- Mitigation ► Remediation
- Next steps ► Analysis/Follow-up
- [EXTRA] Provide feedback: Asking affected customers how incident communication experience could be improved

It should be obvious by now that communication is a critical pillar for incident response. These are aspects every clear communication strategy should include:

- Use a centralized approach to keep the information accessible to everyone in need of it. For example, you could use Azure DevOps work items or GitHub Issues as the tracking software.
  - Engineers will be able to search previously handled incidents for getting ideas.

- Document everything!
  - On early stages, SREs depend on the expertise of SMEs. Knowledge shared verbally has the risk of getting “lost” and needing to “relearn” it.
  - It will help with onboarding new members.
  - Don’t trust you will remember actions/steps taken; human’s short-term memory is not reliable enough.
- Use tools to improve the effectiveness of communication, like dedicated channels for people working on an incident.

You will want to automate these repeatable management/creation processes as much as possible.

## ChatOps

ChatOps is a well-known concept nowadays. It is a model that defines collaboration workflows by connecting tools, people, and processes by using automation. As part of an incident management framework, it brings the following benefits:

- Helps centralizing communication about:
  - Detection
  - Progress
  - Remediation
- Breaks silos: Increases visibility and awareness. Everyone has access to the same information.
- Helps on asynchronous conversation.
- Helps documenting everything (conversations, actions, logs, traces, etc.), which clearly helps during postmortem and future incident resolution.

More mature teams will make use of artificial intelligence and customized bots to incentivize collaboration and improve remediation time. The demo at the end of the section will give some ideas of how to implement such solutions. Some examples of advanced ChatOps practices could be the following:

- Automatically create collaboration environment for your IT staff and an incident status tracking item on a planning tool.
  - For example: Every time an incident is detected, by using solutions like Logic Apps/Azure Functions, you can create a dedicated channel for discussion in Microsoft Teams and create a work item in Azure DevOps to manage status and incident owners.
- Create bidirectional integrations with ITSM tools like ServiceNow.
- Use machine learning and AI services (like Bots or Cognitive Services) for creating a rich knowledge base/searching experience that may help remediating future incidents (linking to similar issues resolved in the past).
  - For example, we could import all collected documentation (teams' conversations, work item comments, screenshots, etc.) into Azure Search (<https://docs.microsoft.com/en-us/azure/search/search-what-is-azure-search>) for creating the AI-based searching experience (knowledge base).

## Eradication/Remediation

Once the thread has been identified and proper channels/collaborations are set, you need to start the remediation activities. Incident responders may need to escalate their concern to SMEs in case they get stuck during this phase. You should have a clear idea of the SMEs you can rely on, based on expertise needed, to have a smooth remediation phase.

This phase could be divided into two subphases:

- **Respond:** Activities to try to contain the thread during mitigation and avoid further consequences on the system if possible. As an example, many DevOps companies nowadays make use of **feature flags** ([www.martinfowler.com/articles/feature-toggles.html](http://www.martinfowler.com/articles/feature-toggles.html)) to control the

runtime behavior of capabilities offered. If a new payment system by an e-commerce website starts crashing, by using Feature Flags, you could disable it, direct users to the previous payment solution, and work carefully on fixing code until the next version is tested/released.

- **Eradicate:** Hopefully, the incident has been contained, and now you can work on a permanent fix for the issue.

It is worth repeating that documenting every action will be critical for successful postmortem activities and creating a knowledge base for future incidents.

Stakeholders should also be periodically updated by the communication lead/manager by using the appropriate channels (external/internal websites, social media, or other communication tools):

- What do you know?
- Containing/eradicating activities in progress
- Timing for the next update

Theory is great, but what about practice? How can you remediate issues using Azure-provided tools? Let's discuss some of the options offered. Many of these tools have been explained in detail in the previous chapter.

### **Azure Service Health**

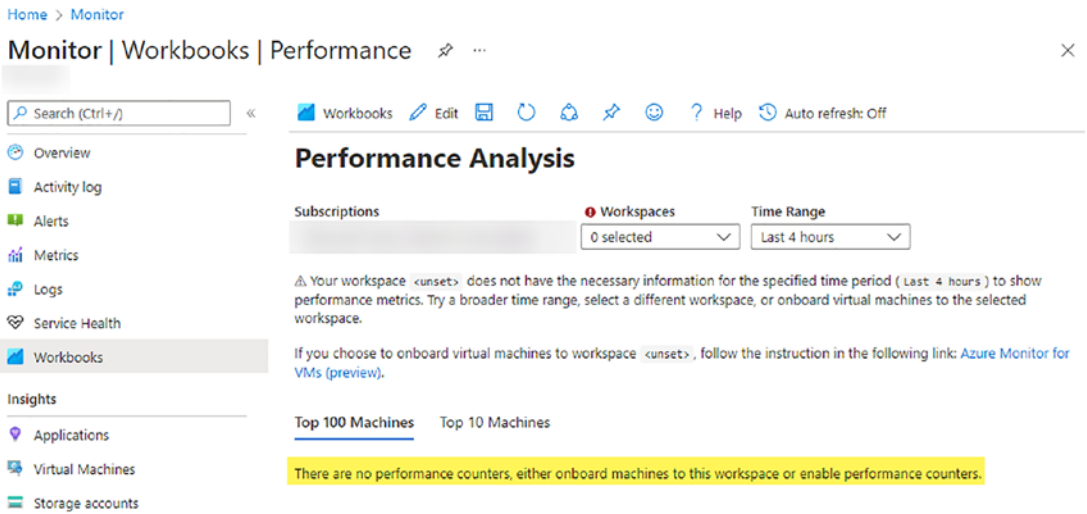
Azure Service Health will help you identify if the issue is related to your system design (code/setup) or to the cloud-offered service. As mentioned before, alerts can be set to reduce time to detect this kind of incident.

### **Azure Monitor Logs and KQL**

As mentioned in the previous chapter, Azure Monitor Logs lets us collect logs offered by different Azure solution layers (subscription activity, resource logs, application logs, etc.). Logs are mostly collected by using the **Diagnostic Settings** tab of Azure services and enabling offered log export (which depends on each service) to a **Log Analytics workspace**.

Once data is collected, we can use Kusto Query Language (KQL) queries from the **Logs** tab (see Figure 7-4) of Azure Monitor or individual services. It can be used to analyze information, present it in charts, or even create alert rules based on results.





**Figure 7-5.** Empty workbook

### Azure Monitor Insights (or Insights)

Azure offers some predefined reports from the **Insights** tab (see Figure 7-6), also offered in some individual service windows like Cosmos DB or virtual machines. Most Insights reports are based on workbooks (which can be customized).

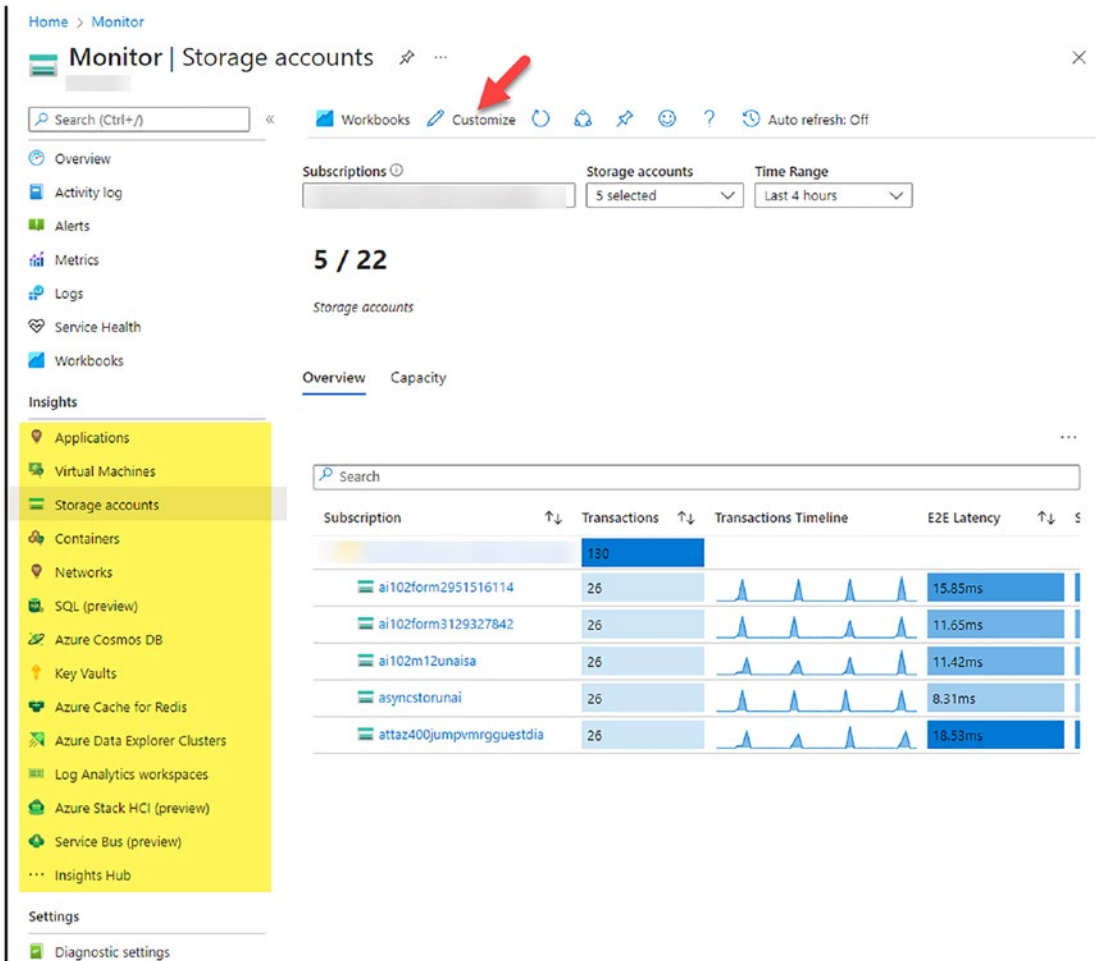


Figure 7-6. Insights reports

### Application Insights

Application Insights, as part of the Azure Monitor offering, is the tool offered by Azure for Application Performance Management (APM). As explained in the previous chapter, it is a tool that will help you have a 360-degree view of your solutions and be able to not only react to detected incidents, but also take a proactive approach by analyzing the behavior of your services. For incident response, these are some of the capabilities offered:

- **Application Maps:** Understand the impacted areas.
- **Failures/Snapshot debugger:** Deep dive into Server/Client Issues and collect debug snapshots when exceptions happen to help in diagnosing production issues.
- **Alerts:** Create alerts based on collected telemetry.
- **Smart Detection:** Azure will provide alerts for potential performance issues and anomalies based on the “normal” (historical) data for your solution (machine learning–powered alerts).
- **Availability tests:** Test your solution periodically from different global points.
- **Logs:** Query collected telemetry using KQL and create customized charts/alerts.
- **Workbook/Troubleshooting guides:** Create workbooks to help your engineer analyze the running environment. **Troubleshooting guides** can be created based on workbook capabilities (in preview).
- **Usage:** Analyze user experience and incident-related impact with the given user experience reports (see Figure 7-7).
- **Performance/Profiler:** Analyze solution performance and identify “slowest” code paths using the **Profiler**.



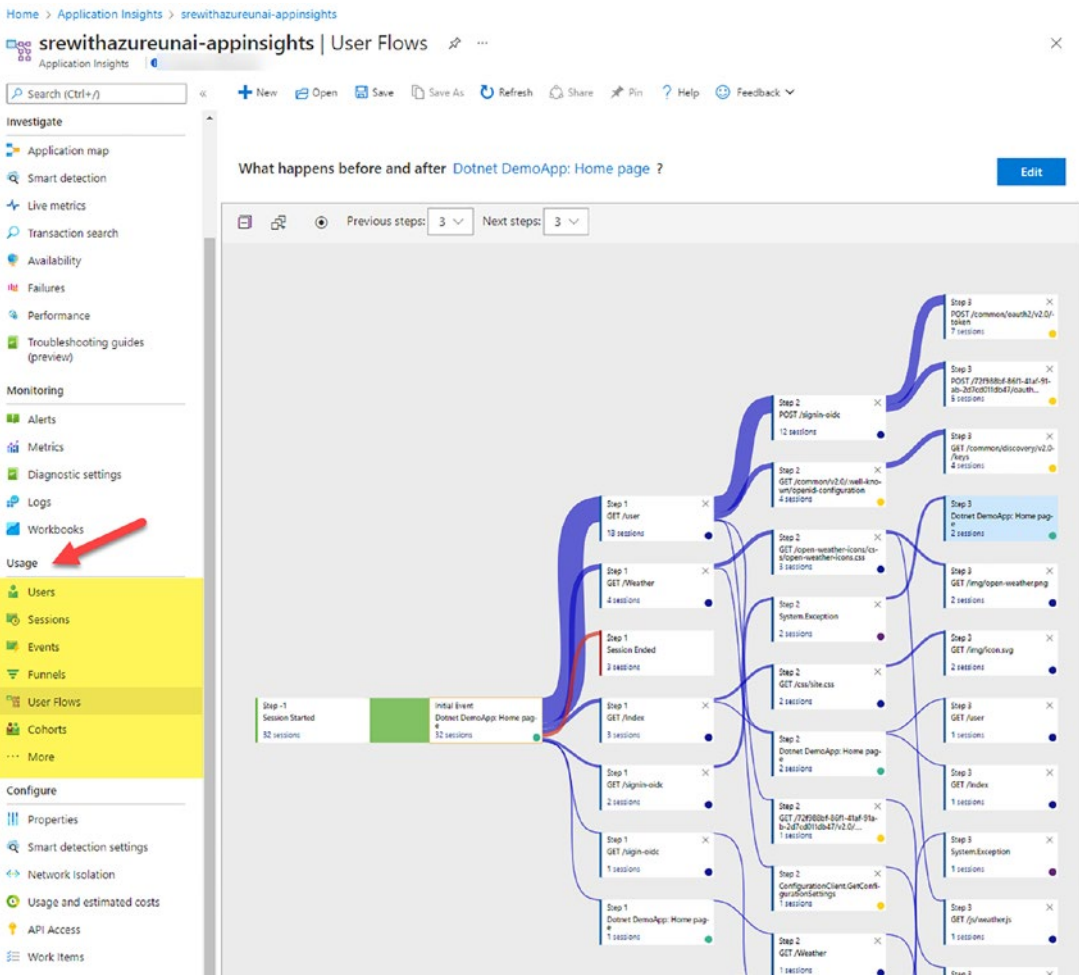


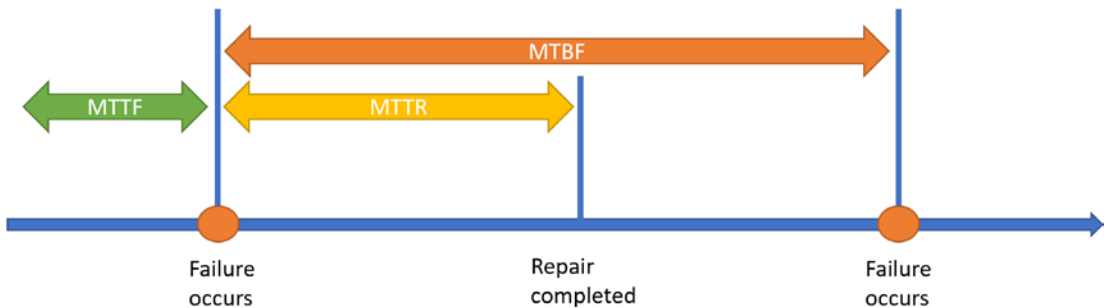
Figure 7-7. Application Insights Usage reports

As you can see, some tools are prepared for a reactive approach (once the incident has happened); some others like **Smart Detection** aim to help you before the incident has happened (proactive approach). Maximizing efforts on proactive work will reduce production-raised incidents.

## Measuring Performance

Previously, in Chapter 2, we covered some of the metrics used by SRE teams to track the maturity of the organization. Some of those metrics were (see Figure 7-8)

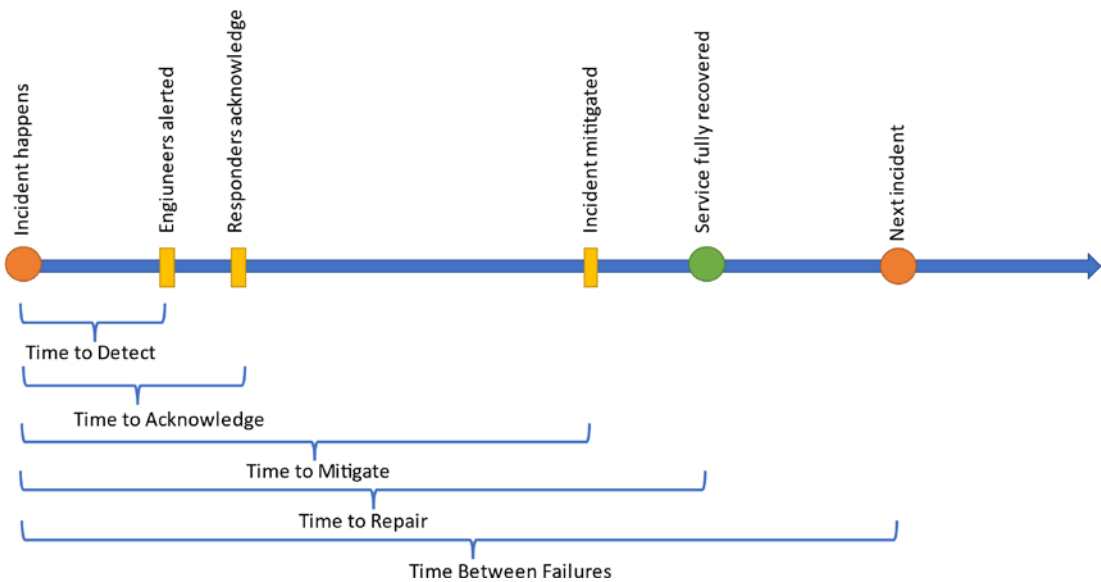
- Mean Time to Failure (MTTF)
- Mean Time to Repair (MTTR)
- Mean Time Between Failures (MTBF)



**Figure 7-8.** SRE metrics

Organizations will use different metrics to track their status. Sometimes, terms and metrics will overlap. For example, the “time to repair” is also called “time to remediate,” “time to restore,” or “to recover.” They all refer to the time taken to bring services back to an operational state defined by our Service-Level Objectives.

Other metrics such as “mean time to detect (MTTD)” (or discover), “mean time to notify,” and “mean time to acknowledge” could be considered phases of the previously mentioned MTTR (e.g., MTTD is part of the calculation for MTTR), as MTTR covers all phases from incident happening to full mitigation (see Figure 7-9). It could be worth monitoring to track improvement in detection and notification practices.



**Figure 7-9.** Incident metrics

Defining a set of Objective and Key Results (OKRs: <https://en.wikipedia.org/wiki/OKR>) based on tracked data will help your organization teams align and work together toward the vision of the company.

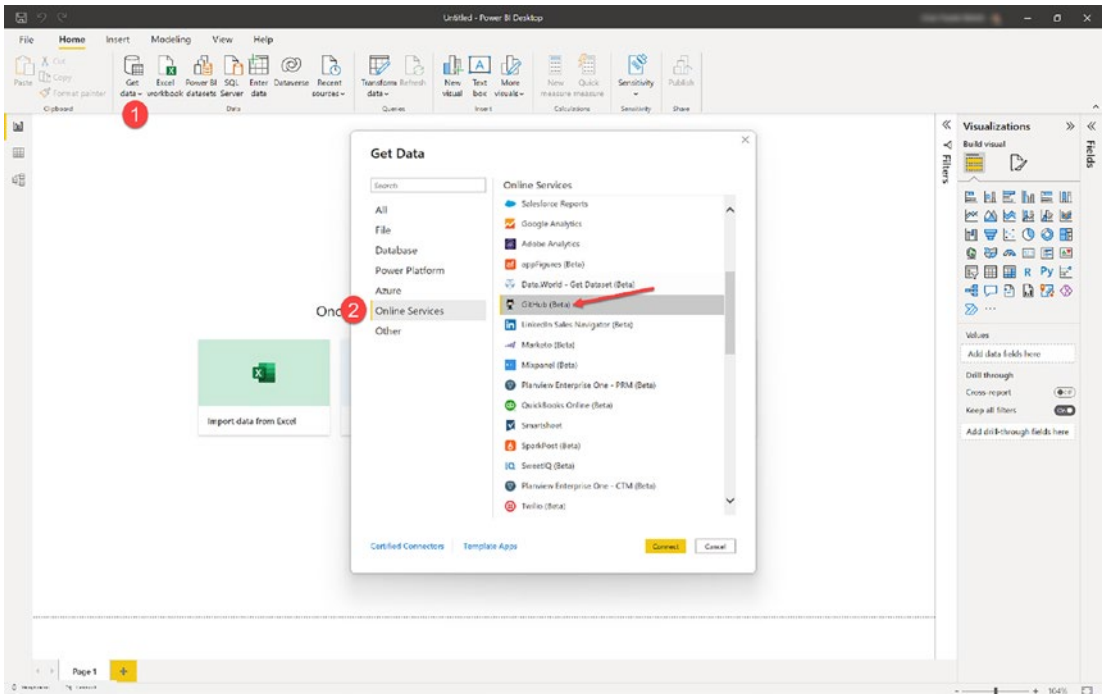
OKRs should not be only service focused, customer satisfaction and employee morale will need to be taken into account too. Check the example of OKRs defined by the Power BI SRE team: <https://docs.microsoft.com/en-us/power-bi/enterprise/service-admin-site-reliability-engineering-model#measuring-success-through-objective-key-results-okrs>.

Which options do you have for tracking these metrics with Microsoft tools? It will depend on the tools involved. For example, if you are using Azure Monitor alerts together with GitHub Issues for incident tracking (as shown in a later demo), you could use Power BI to collect and measure the mentioned metrics:

- **Mean Time to Repair:** Measure time difference since Alert was fired (Azure Monitor alerts property) until GitHub Issues gets closed.
- **Mean Time Between Failures:** Measure the mean time between consecutive GitHub Issues creation times.

- **Other Metrics:** These will be affected by the metadata exposing/collecting capabilities of the tools being used. For example, GitHub Issues does not provide much historical information on assigned users and state changes compared to Azure DevOps.

Check the GitHub (Beta) integration for Power BI Desktop to connect to the dataset of a repository and easily create reports for mentioned metrics (see Figure 7-10 and Figure 7-11).



**Figure 7-10.** GitHub integration for Power BI

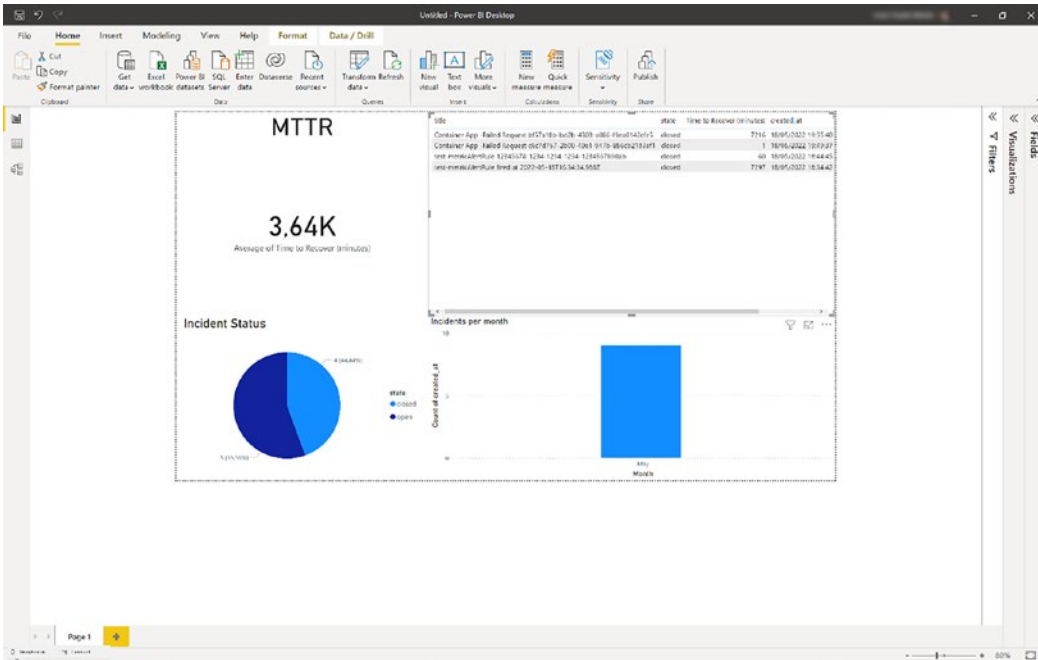


Figure 7-11. Power BI simple report

## [DEMO] Incident Response

The following demo will show a practical way of applying the concepts covered in the last section. It looks as shown in Figure 7-12.

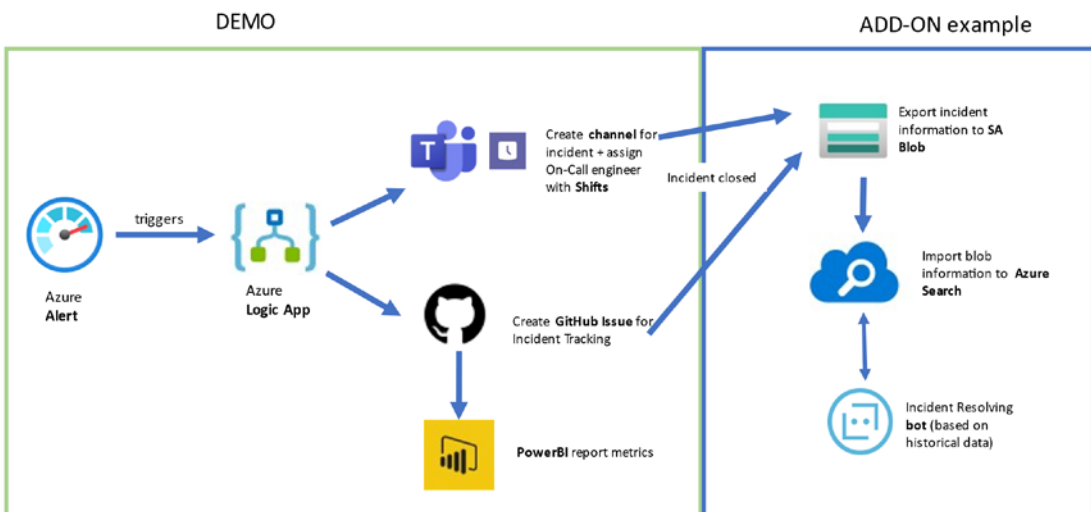


Figure 7-12. Demo architecture

The left side of the demo will be shown, whereas the right one shows an example of how incident response could be taken to the next level:

- **Demo:** Every time an Azure alert criterion is met, it triggers a **Logic App** (<https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-overview>) using alert **Action Groups**. The Logic App runs the following workflow (shown in Figure 7-13). A used Logic App design can be found at <https://github.com/unaihuete-org/SREwithAzure-IncidentResponse/blob/main/logic-app.json>.
  - Create a Microsoft Teams channel for discussion (see Figure 7-14).
    - Post an **Adaptive Card** ([www.adaptivecards.io/designer/](http://www.adaptivecards.io/designer/)) with details about the incident.
    - Using **Microsoft Teams Shifts** (<https://docs.microsoft.com/en-us/microsoftteams/expand-teams-across-your-org/shifts-for-teams-landing-page>), we define an On-Call schedule, and Logic App notifies the on-call engineer with a mention in a channel message (see Figure 7-15).
  - Create a **GitHub Issue** for incident tracking.
  - **Power BI** reports for data extracted from the GitHub Issues incident tracking solution.
- **ADD-ON (not implemented):** It shows a possible add-on to the shown architecture. Once the incident is resolved, all the activity information (channel posts, files, issue tracking activity, etc.) could be collected in the following way for advanced analysis:
  - Export incident data (Teams channel + GitHub Issues) to **Azure Storage Account blob**. Another Logic App, GitHub Actions, or other automation tool could be used, together with REST API calls to export information from the mentioned tools. **Microsoft Graph API** (<https://docs.microsoft.com/en-us/graph/use-the-api>) is a great source to interact and get information with Microsoft services like Teams, Outlook, OneDrive, etc.
  - Use **Azure Cognitive Search** (<https://docs.microsoft.com/en-us/azure/search/search-what-is-azure-search>), a cloud search service that is able to ingest data, enrich it using cognitive

services (e.g., extract data from PDF files or screenshots you may have used during incident resolution), index it, and offer a powerful searching experience.

- Create an **Azure Bot** (<https://docs.microsoft.com/en-us/azure/bot-service/?view=azure-bot-service-4.0>) that helps you in resolving incidents by looking at historical data indexed on Azure Search.

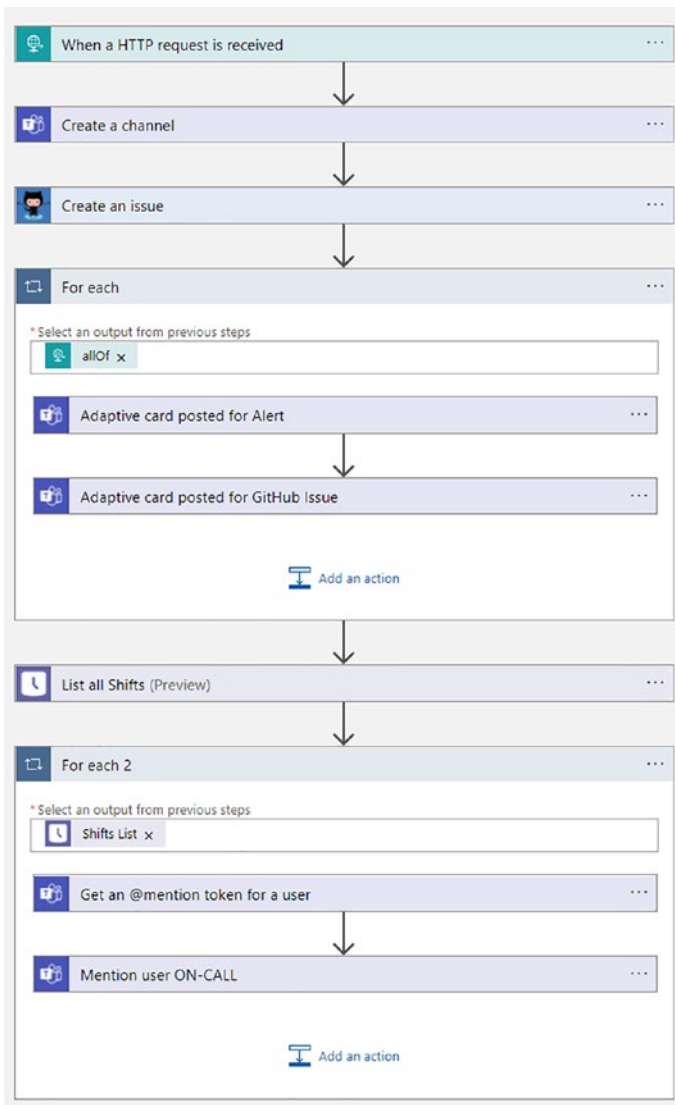
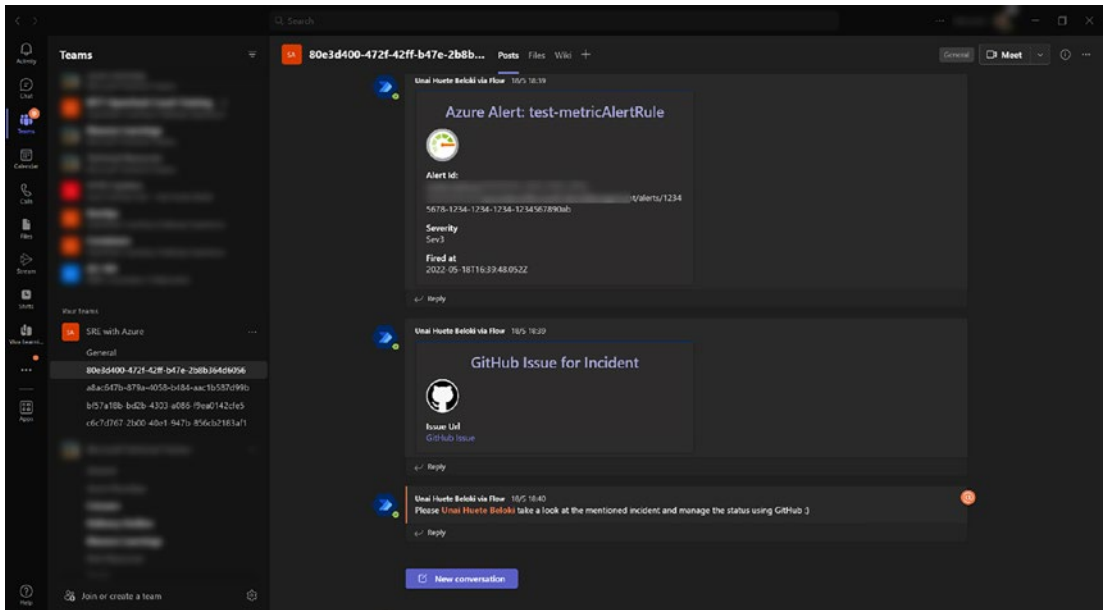


Figure 7-13. Incident response Logic App



**Figure 7-14.** Microsoft Teams channel and posted messages



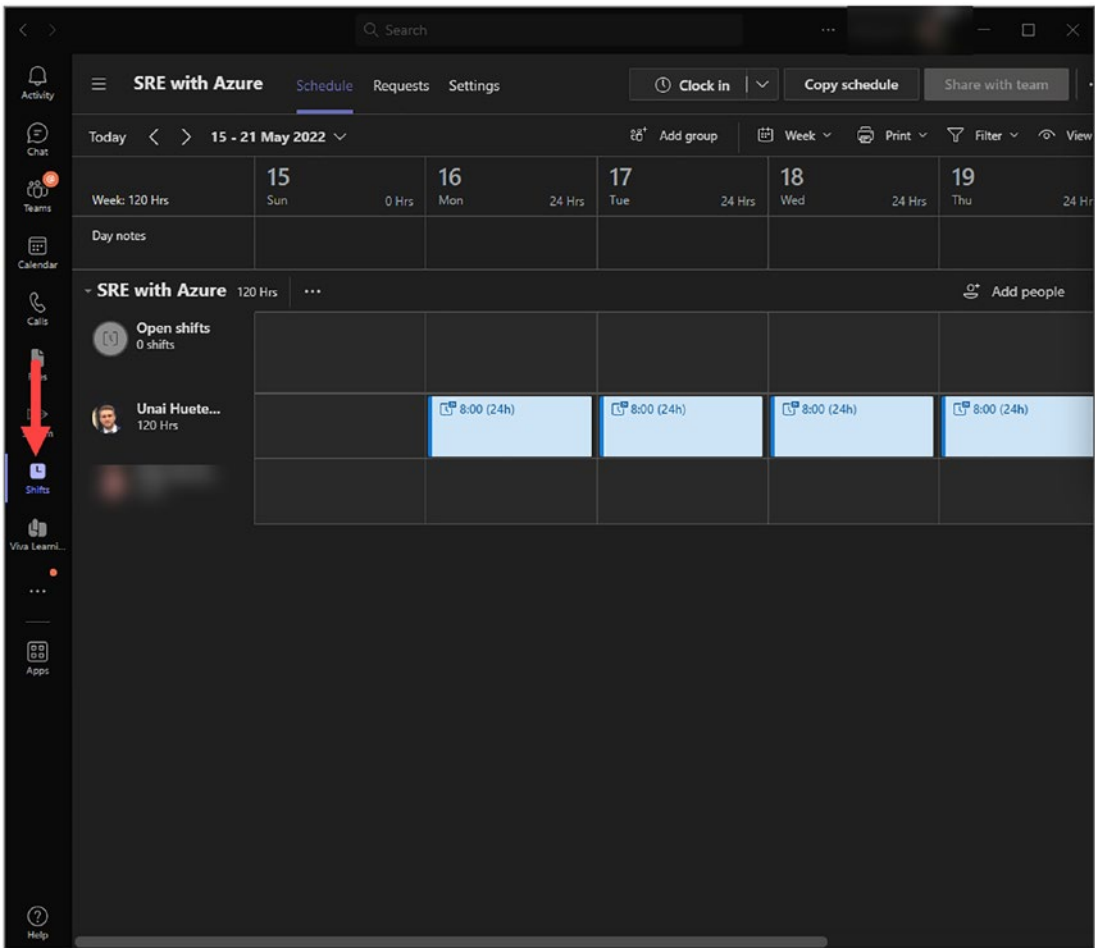


Figure 7-15. Microsoft Teams Shifts

## Blameless Postmortems

The previous part of the chapter was focused on reducing the impact of incidents by applying automation practices that will help you shorten the timeline of the issue. You need to be ready for it as incidents are inevitable.

Since changes are constant, complex systems will never be 100% reliable. Incidents are mostly perceived as a negative event; nobody likes to be in a stressful situation where an issue is affecting your running business. Nevertheless, you should also focus on the **learning** opportunity given by those incidents, as they expose unknown vulnerabilities and give you an opportunity to prevent future recurrences. They can be seen as an opportunity to improve our systems.

Blameless postmortem is a process composed of the following ideas and cultural shift based on honesty, learning, and accountability:

- Building a **timeline of incident remediation** actions and collection of all documentation is critical for understanding root causes and thus creating a successful learning process (using techniques covered in the “Incident Response (IR)” section).
  - Alert details
  - On-call engineers involved
  - Incident tracking history and data, based on tools like Azure DevOps work items and GitHub Actions
  - Discussions, based on tools like Microsoft Teams and Slack
- **No “pointing fingers.”** There is **no one to blame**; avoid the human natural tendency of looking for guilty engineers; it is assumed that staff acted with the best intentions possible (based on what they knew, skills, and abilities). **Focus on the system, not the person.** This is the most difficult change for an organization, but it has the biggest impact. Remember the Westrum organizational culture mentioned in Chapter 1 (see Table 7-2). According to DevOps Research and Assessment (DORA), organizational culture really affects information flow. This idea could be applied to SRE teams too.

*Table 7-2. Westrum organizational culture*

<b>Pathological (Power Oriented)</b>	<b>Bureaucratic (Rule Oriented)</b>	<b>Generative (Performance Oriented)</b>
Low cooperation	Modest cooperation	High cooperation
Messengers shot	Messengers neglected	Messengers trained
Responsibilities shirked	Narrow responsibilities	Risks are shared
Bridging discouraged	Bridging tolerated	Bridging encouraged
Failure ► scapegoating	Failure ► justice	Failure ► inquiry
Novelty crushed	Novelty problems	Novelty implemented

From the table, it can be concluded that **Generative** culture environments will have many benefits related to the postmortem activity:

- **High cooperation/bridging** ► better collaboration, breaking silos, cross-functional teams.
- **Messengers trained/failure leads to inquiry** ► employees will share potential issues as soon as they identify them; messenger is not punished; remove blame; failures lead to questions.
- **Risks are shared** ► Quality, availability, reliability, and security are everyone's responsibilities.

As you can see, “blame” is harmful to the business. It discourages engineers from speaking up when potential issues are identified due to fear of impacting them negatively. As a result, silence increases Mean Time to Acknowledge and Resolve incidents, increasing the impact. Google found ([www.inc.com/justin-bariso/after-years-of-research-google-discovered-secret-weapon-to-building-a-great-team-its-a-lesson-in-emotional-intelligence.html](http://www.inc.com/justin-bariso/after-years-of-research-google-discovered-secret-weapon-to-building-a-great-team-its-a-lesson-in-emotional-intelligence.html)) that high-performance teams with strong psychological safety had many benefits: for example, more conversation turn-taking. If everyone shares perspective, collective intelligence increases. Furthermore, Margaret Heffernan discusses in this TED talk ([www.ted.com/talks/margaret\\_heffernan\\_forget\\_the\\_pecking\\_order\\_at\\_work?language=en](http://www.ted.com/talks/margaret_heffernan_forget_the_pecking_order_at_work?language=en)) an experiment around productive teams being the ones where people participated equally and diversity encouraging a culture of helpfulness.

During the blameless conversation, questions should be framed so that focus is on the technology and not on the engineers (avoid indirect “point fingers” behavior). For example:

- “Who made the change that modified the database schema?”
- Instead, “what was the reason to modify the database schema that led to the failure?”

Learnings will be captured and shared across the organization using tools like Azure DevOps Wiki and GitHub Wikis as a centralized repository for learning/best practices. The postmortem could also lead to **follow-up actions** focused on avoiding similar incidents in the future.

## Best Practices/Tips

The following tips could be taken into account for a successful postmortem process:

1. Owner/roles: The incident commander/manager (defined in roles previously) will take care of driving the discussion and providing the collected data/feedback.
2. Not all incidents require a postmortem activity. Only high-severity incidents may need it or incidents that took more time than expected (**define a threshold**).
3. Ask proper questions. **Remember to ask questions with a focus on the system**, not on the person. Asking “how”/“what” better than “why” (which tends to judge). Some organizations like to drive Root Cause Analysis (RCA) activities using the **5 Whys technique** (<https://sixsigmastudyguide.com/5-whys/#:~:text=The%205%20Whys%20is%20a%20basic%20root%20cause,identify%20the%20root%20cause%20and%20then%20eliminating%20it>); be careful as “why” questions tend to indirectly “finger point.”
4. Be careful with the language used in conversations. **Normative** language (e.g., “unsatisfactory” or “irresponsibly”) often is used for judgment.
5. Don’t focus only on things that went wrong; **also focus on things that went right**; that could be used on consequent incidents (collaborations, actions, decisions, etc.).
6. Short meeting of around 60–90 minutes.
7. Celebrate learnings.

## Summary

In this chapter, you were introduced to two main practices mentioned in Dickerson's hierarchy of reliability: incident response and blameless postmortems.

Every organization will need to define an incident response framework that will guide the staff through the incident remediation process. The chapter defined the phases, activities, and roles that should be included in the incident response journey. No system is 100% reliable to define, and perfecting the practices mentioned will help you decrease the impact of these issues.

As incidents will always keep appearing, let's use them as an opportunity for learning and improving our existing solutions. Blameless postmortems will focus on finding the root cause of incidents by driving conversation based on data collected during the incident response process and creating a culture where engineers are not blamed for their actions.

The next chapter will focus on showing interesting tools provided by Azure to proactively test the resiliency of your solutions, getting prepared for real-life incident scenarios.

## CHAPTER 8

# Azure Chaos Studio (Preview) and Azure Load Testing (Preview)

Although this is the last chapter in the book, it doesn't mean it's less important. Actually, it's the last one because it didn't exist in the initial outline (at least not with this topic) and only got added because both services were announced at Microsoft Ignite in November 2021 as public preview, where I was so excited I could finally talk about it and somehow convinced Apress to dedicate another chapter for it.

Keep in mind this chapter is written on the Azure Chaos Studio and Azure Load Testing preview bits as available at the time of writing, which could mean some parts (or a lot of parts) might have changed by the time you go through them. Although the logical concepts of Chaos Engineering (which I'll talk about first) will remain valid.

- Understand Chaos Engineering
- Configure Azure Chaos Studio service
- Define Azure Chaos Studio chaos experiments
- Run chaos experiments against target resources
- Understand Load/Performance testing
- Configure/run Azure Load Testing service

## Intro to Chaos Engineering

*Chaos Engineering is the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production.*

Source: <https://principlesofchaos.org/>

Chaos Engineering is all about experimenting – typically against production-running systems – to identify and find loopholes, pitfalls if you want, in the way the system is running, which makes the system less reliable.

The more loopholes we can identify up front, the more confidence we can have in the system's reliability. By introducing a series of event simulations, whether based on real incidents happened earlier or based on simulated outages that could happen, we target our workloads and learn from its impact.

An easy example is CPU pressure.

Imagine a workload is running fine for months, with an average CPU load that's keeping the system running healthy. Suddenly, a CPU spike occurs and crashes the application. Apart from troubleshooting the root cause of the CPU spike, probably a task for engineering or development teams, it might be equally relevant to find out why the system reacted with a crash of the application. Even more so, if we could have simulated a CPU spike happening, our engineering and development teams could have focused on mitigating the problem by releasing a fix, updating the architecture to an even more fault-tolerant setup.

Don't get me wrong though, as Chaos Engineering is a lot more than injecting outage triggers to bring production environments to their knees. There's a lot more complexity involved, especially since an outage is typically not caused by one single failure, but more of a series of incidents.

Reusing the CPU pressure example, one could consider a scenario where CPU is spiking, because of a latency in database operations, putting a calculation or database update on hold. Or maybe there is a network connectivity issue, by which an operation cannot be written to the database back end, causing so many retry operations, which spikes CPU. So instead of just "simulating" the CPU spike, it is also important to capture *all possible side effects* that could cause a CPU pressure.

Which – to me – also explains why it's called an engineering discipline, as there is quite some engineering involved in all the interactions across different systems, components, and workloads.

Now, you might think that Chaos Engineering is the next big thing (maybe even coming after SRE and DevOps?), but yet too revolutionary for your cloud environments. But nothing is more wrong.

In fact, Chaos Engineering has been around for more than ten years already, initiated by software engineers from Netflix around 2008, when they started migrating from on-premises data centers to public cloud data centers. While there are a lot of similarities between managing your own data center and using public cloud, there are also big differences. It was mainly those differences that forced Netflix's engineers to create services architectures with higher resiliency.

## Chaos Monkey

Going through the testing related to this cloud migration resulted in the creation of an internally developed Chaos Orchestration tool around 2010, branded Chaos Monkey, which was publicized as an open source product in 2012. More information on the tool and how to use it is available on GitHub (see Figure 8-1): <https://github.com/netflix/chaosmonkey>.



**Figure 8-1.** Chaos Monkey

From the README.md file:

*Chaos Monkey randomly terminates virtual machine instances and containers that run inside of your production environment. Exposing engineers to failures more frequently incentivizes them to build resilient services.*



*Chaos Monkey is an example of a tool that follows the Principles of Chaos Engineering.*

### *Requirements*

*This version of Chaos Monkey is fully integrated with Spinnaker (<https://spinnaker.io>), the continuous delivery platform that we use at Netflix. You must be managing your apps with Spinnaker to use Chaos Monkey to terminate instances.*

*Chaos Monkey should work with any back end that Spinnaker supports (AWS, Google Compute Engine, Azure, Kubernetes, Cloud Foundry). It has been tested with AWS, GCE, and Kubernetes.*

Netflix designed Chaos Monkey to allow them to validate the stability of their production-running workloads (the Streaming Service we all use), which was running on Amazon Web Services. The main purpose of Chaos Monkey was detecting how their systems would respond to critical components being taken down. By intentionally shutting down workloads, it would become clear what weaknesses are present in the total topology and allow the engineering teams to work toward mitigation.

## **Principles of Chaos (Engineering)**

While it wasn't just developed for it, I am sure I can say that offering Chaos Monkey as an open source solution definitely helped in boosting the message around Chaos Engineering, as defined by the *Principles of Chaos* ([www.principlesofchaos.org](http://www.principlesofchaos.org)).

Without literally repeating what they define as Chaos Engineering, it boils down to this:

- Traditional testing of failures and outages is typically focused around one single scenario, for example, a CPU spike as mentioned earlier; for production-running workloads, an incident is hardly ever related to a single cause, but more a sequence of events. Chaos Engineering tends to focus on realizing this sequence of actions against systems. The more complex the simulations, the more confidence you get in the reliability of your systems.

- Weaknesses and shortcomings in systems and running workloads, regardless if the architecture is traditional virtual machines, more complex containerized architectures, or serverless and microservices oriented, should be identified before they manifest while running as a mission- or business-critical production workload. The starting point of Chaos Monkey as an example orchestrator was mainly virtual machine based. You can probably come up with a long list of typical events that might hammer a virtual machine’s uptime, such as storage latency or outage, disk crash, operating system faults, CPU pressure, systems going down, and systems constantly rebooting. But moving on to containerized workloads makes this process much harder. Apart from the underlying hypervisor, you also need to identify the correlation across containers, network security, storage integration, identifying how, for example, Kubernetes PODs across different nodes in a cluster are behaving, how and when are PODs restarted, and more. Ultimately you may end up in the most “uncontrollable” scenarios, microservices and serverless. If you don’t manage the underlying platform, you don’t manage the network or storage interaction, almost every layer is managed for you, there might not be too much for you to control, but you are still responsible for making sure the microservice (Azure Functions, AWS Lambda) is running. Maybe just because of this, it might be much harder to actually manage and validate the uptime.
- **Chaos Engineering is by design proactive**, which means you tend to identify weaknesses long before they – potentially – happen. First of all, to find the weaknesses themselves and also to be better prepared for when they happen. This might sound weird, as it should be possible to mitigate the issue before it arises, specifically thanks to the drill exercises pushed through before. But after working in the IT industry for 25 years, I’m sure it is quite impossible to *plan for everything*, especially in a public cloud environment, where you don’t manage the full stack of the architecture workload.

## Azure Chaos Studio

Chaos Monkey is a great tool, and although it is heavily integrating and relying on Spinnaker, it also makes it platform and cloud agnostic, supporting Amazon AWS, Google GCP, Microsoft Azure, and also Kubernetes on-premises or in a cloud scenario for example.

Since this is a book on achieving SRE with Azure, my initial intention for this chapter was writing about Chaos Monkey (and Chaos Mesh for Kubernetes) and using it against an Azure reference architecture scenario, as discussed in a previous chapter.

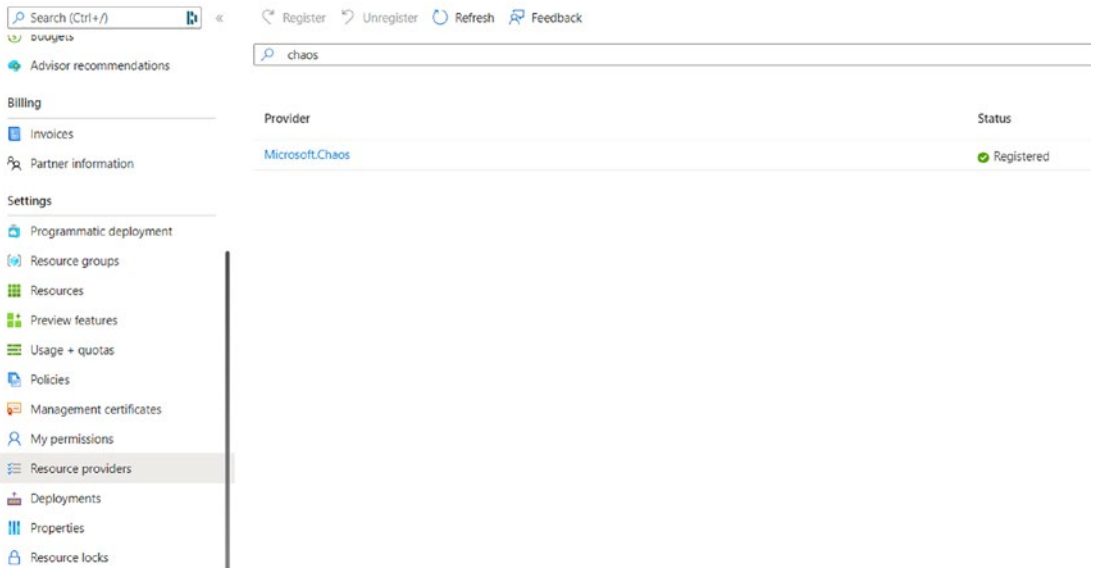
However, at Microsoft Ignite in November 2021, the great news came that Azure Chaos Studio would go into public preview, which means it is now available for anyone to test from that date onward. (Note: I've been looking into Azure Chaos Studio for a few months already as a Microsoft employee but wasn't allowed to talk or share anything publicly.)

That's where now the remaining part of this chapter will be dedicated to Azure Chaos Studio.

## Azure Chaos Studio Architecture

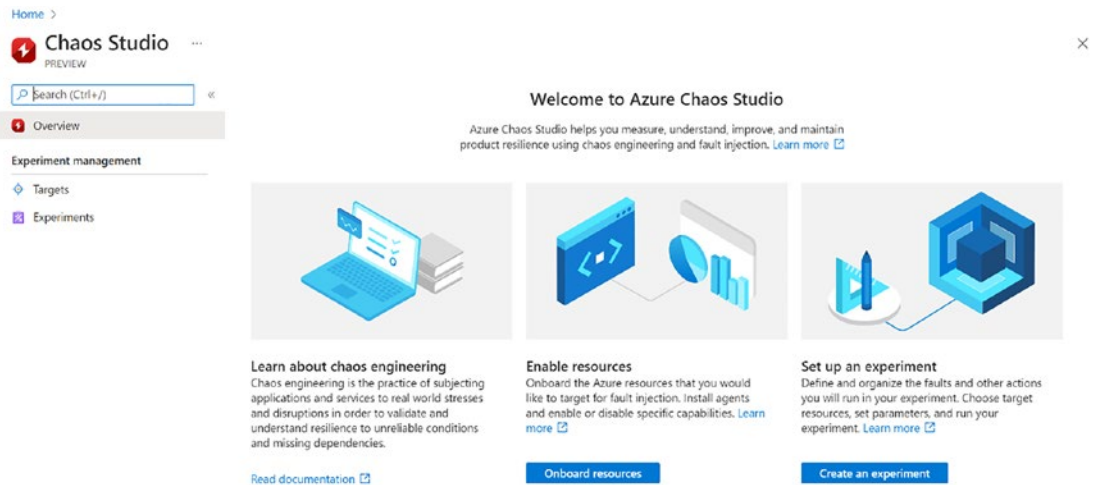
Azure Chaos Studio is provided as a service, which means you don't have to deploy your own infrastructure first to get it up and running. Under the hood, it is based on the open source Chaos Mesh ([A Powerful Chaos Engineering Platform for Kubernetes | Chaos Mesh \(chaos-mesh.org\)](#)) solution, which (typically) runs as a Kubernetes-based architecture. But again, none of that is important for you to know about, as it is all running behind the curtains.

1. The first thing you need to check is to make sure the “Microsoft. Chaos” Azure Resource Provider is enabled (registered) in your subscription. To do that, open your Azure portal, and search for Subscriptions.
2. Select the subscription in which you want to enable Azure Chaos Studio. From within the detailed blade, select “Resource Providers” under the Settings pane, and search for “Chaos”, as shown in the screenshot. Select “Microsoft.Chaos” and click “Register” in the top menu; give it a few minutes, until the Status column shows “Registered” (see [Figure 8-2](#)).



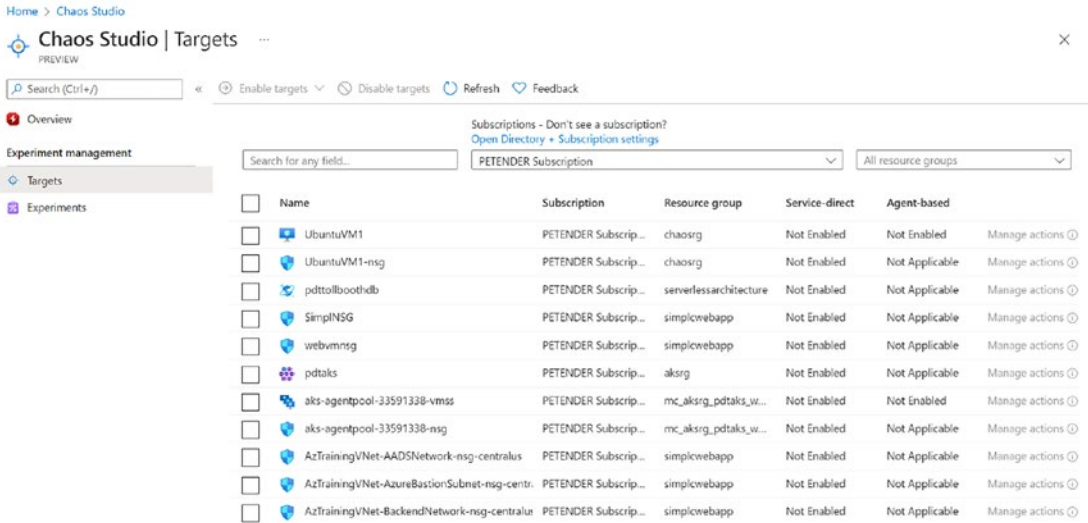
**Figure 8-2.** Register *Microsoft.Chaos*

3. You are now ready to start using Azure Chaos Studio in your subscription.
4. From your Azure portal, search for Chaos Studio (see Figure 8-3).



**Figure 8-3.** Azure Chaos Studio

- Next, enable resources for Chaos testing by selecting “Onboard Resources”. This brings you to the Targets section of the blade (see Figure 8-4).



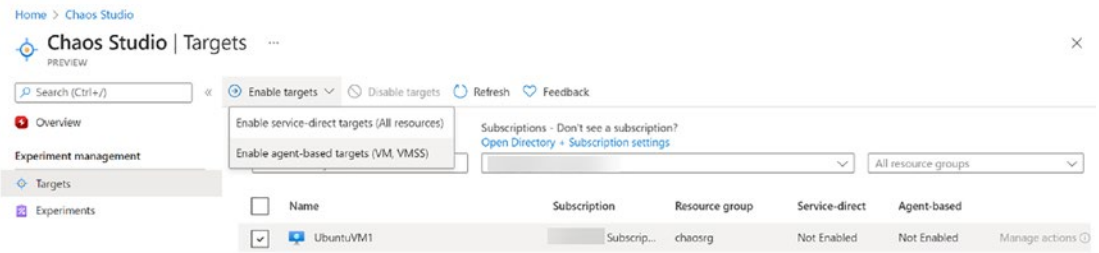
**Figure 8-4.** Target section

- Here, you can filter for specific subscriptions or specific Resource Groups (or both), where next, you need to select the Azure Resource(s) you want to use as a target.

The remaining part of this chapter walks you through two common scenarios: how to integrate Chaos Engineering with Chaos Studio for an Azure Virtual Machine (agent based) and how to use it for an Azure Kubernetes Service (AKS) cluster.

## Onboarding an Azure VM to Chaos Studio

- In this example, I’m going to target an UbuntuVM1 machine, selecting it, which unlocks the “Enable Targets” menu option (see Figure 8-5).



**Figure 8-5.** Enable targets

2. Here, I can choose between service-direct targets and agent-based targets. Since it's a virtual machine, the agent based would be recommended.
3. This redirects you to the “Enable Agent-based targets” blade (see Figure 8-6), where you need to provide additional configuration parameters, one being a User-Managed Identity, which is establishing the authentication integration between the target resource and the Chaos Studio, as well as an Application Insights resource, which provides the observability and monitoring aspects of running Chaos Studio.

[Home](#) > [Chaos Studio](#) >

## Enable agent targets ...

PREVIEW

### Managed Identities

The Chaos Studio agent uses a user-assigned managed identity on a virtual machine or virtual machine scale set to authenticate to the Chaos Studio service. You can use a single user-assigned managed identity for multiple virtual machines or virtual machine scale sets. When the agent is enabled, the selected managed identity will be applied to the selected VMs and the Chaos Studio VM Extension will be provisioned to use that identity.

Subscription \* ⓘ

Managed identity \*

**i** If you do not have a managed identity, first create one in the Managed Identities service in the portal.

### Application Insights

The Chaos Studio agent can optionally send diagnostic information about fault execution and agent health to an existing Application Insights account.

Send diagnostic data to Application Insights

Enabled

Disabled

To get started, choose an Application Insights account, or if you already know the instrumentation key for your account you can directly specify it below.

**Figure 8-6.** *Enable agent targets*

- Let's start with creating the User-Managed Identity first. From the Azure portal, search for Managed Identities, and create a new one.
- Define the Resource Group, Region (Make sure the Region here is the same as where your target resources are running), and unique name for the User-Managed Identity and confirm by clicking the Review + Create button (see Figure 8-7).

Home &gt; Managed Identities &gt;

## Create User Assigned Managed Identity

Basics Tags Review + create

### Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ

Resource group \* ⓘ  [Create new](#)

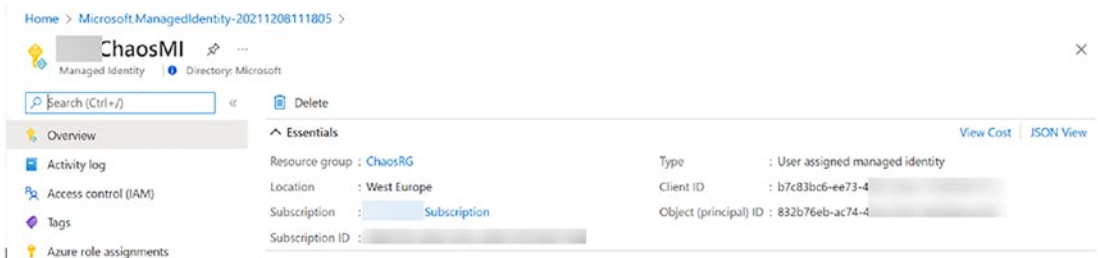
### Instance details

Region \* ⓘ

Name \* ⓘ

**Figure 8-7.** User Assigned Managed Identity

- Wait for the Managed Identity resource to get created (see Figure 8-8).

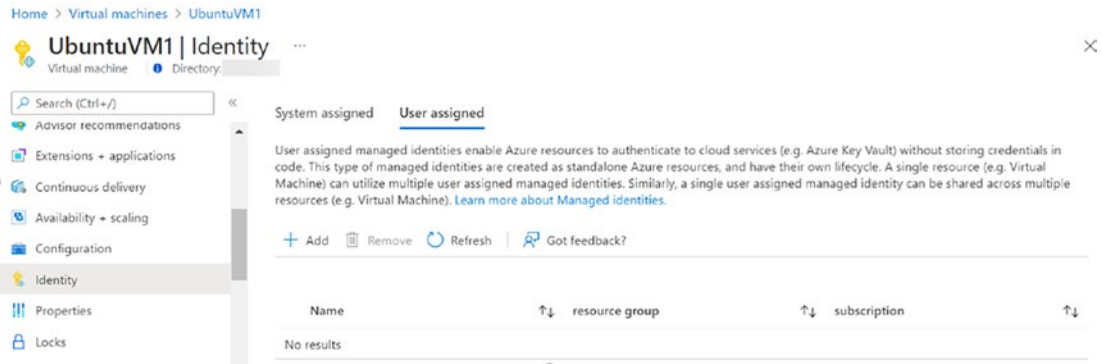


**Figure 8-8.** User MI created

- Before Chaos Studio can target an Azure Resource, it should get the User-Managed Identity assigned to it. Let's do that for our sample UbuntuVM1 Virtual Machine. From the Azure portal, browse to Virtual Machines and select the UbuntuVM1 resource.

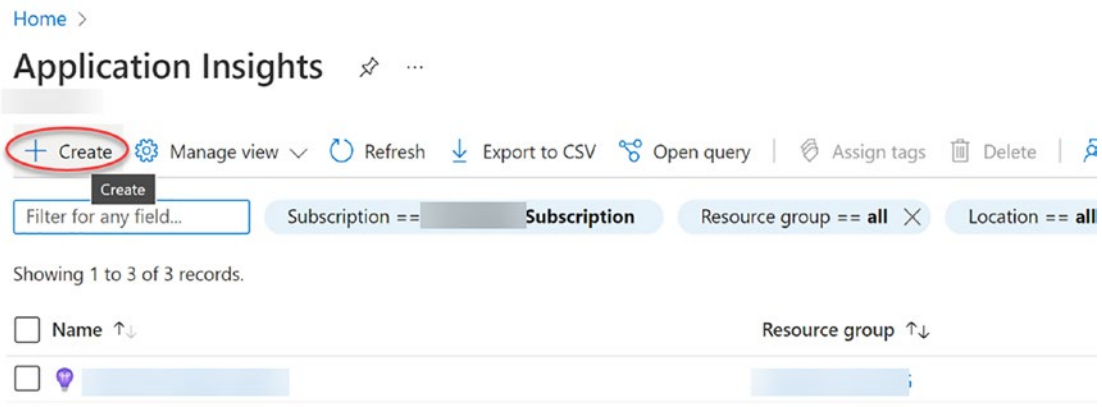


8. From the UbuntuVM1 blade, find Identity under Settings, and navigate to “User Assigned” (see Figure 8-9).



**Figure 8-9.** Assigning MI to VM

9. Under User Assigned, click Add, and select the User Assigned Managed Identity you created earlier.
10. Click Add to confirm the changes.
11. From the Azure portal search, look for Application Insights. Create a new instance (see Figure 8-10).



**Figure 8-10.** Create Application Insights

- Provide the necessary information for Azure Resource Group, Application Insights Service Name, the Azure Region where you want to get the service deployed (Note: this should be the same region as where your target resources are running), as well as the Log Analytics Workspace you want to use to store the log information (see Figure 8-11).

[Home](#) > [Application Insights](#) >

## Application Insights ⋮

Monitor web app performance and usage

**basics** [iags](#) [review + create](#)

Create an Application Insights resource to monitor your live web application. With Application Insights, you have full observability into your application across all components and dependencies of your complex distributed architecture. It includes powerful analytics tools to help you diagnose issues and to understand what users actually do with your app. It's designed to help you continuously improve performance and usability. It works for apps on a wide variety of platforms including .NET, Node.js and Java EE, hosted on-premises, hybrid, or any public cloud. [Learn More](#)

### PROJECT DETAILS

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ  ✓

Resource Group \* ⓘ  ✓  
[Create new](#)

### INSTANCE DETAILS

Name \* ⓘ  ✓

Region \* ⓘ  ✓

Resource Mode \* ⓘ  Classic  **Workspace-based**

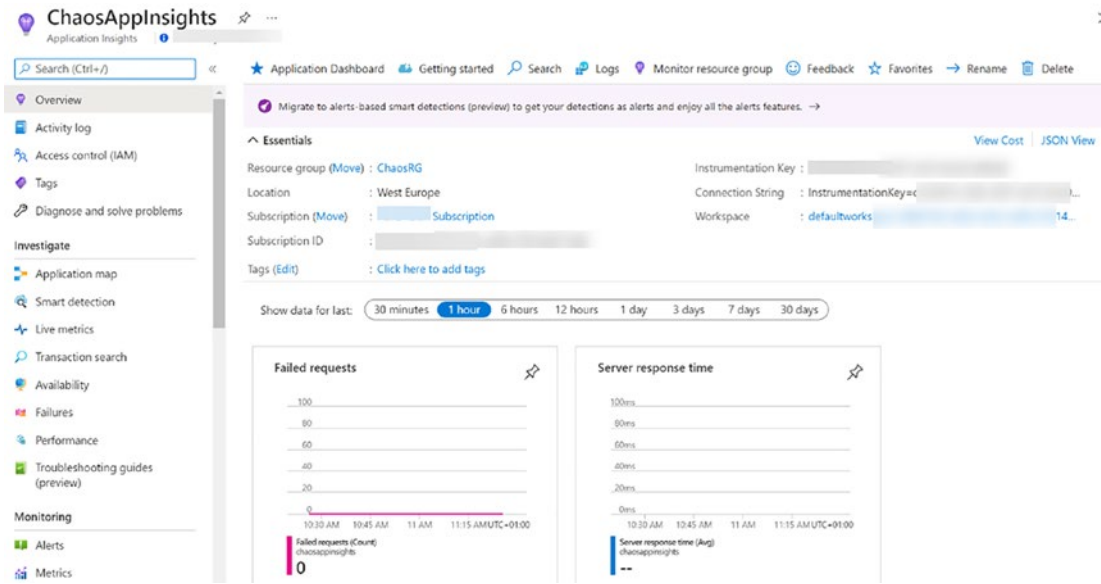
### WORKSPACE DETAILS

Subscription \* ⓘ  ✓

\*Log Analytics Workspace ⓘ  ✓

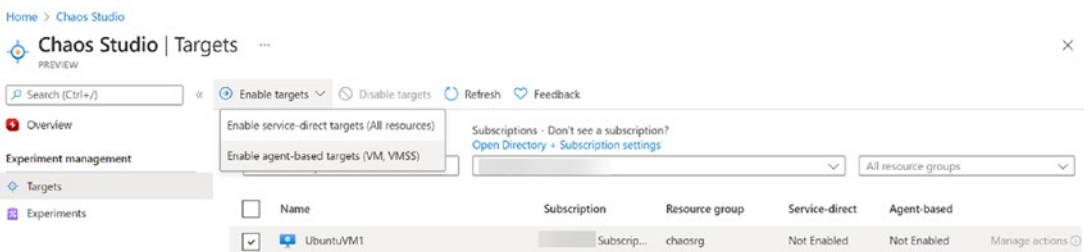
**Figure 8-11.** *Application Insights settings*

- Confirm the creation of this resource by clicking “Review and Create” and wait for the service to get deployed (see Figure 8-12).



**Figure 8-12.** Application Insights created

- This completes the setup of the Azure Chaos Studio prerequisites; let’s head back to the service and continue the configuration of our target resource. From the Azure portal, search for Chaos Studio again, and repeat the steps from before where you select your target, and enable it for agent-based target scenario (see Figure 8-13).



**Figure 8-13.** Enable Chaos Studio Targets

- Once redirected to the “Enable agent-based targets” page, complete the settings for both Managed Identity and Application Insights, reflecting the services you created in the previous steps (see Figure 8-14).

[Home](#) > [Chaos Studio](#) >

## Enable agent targets ...

PREVIEW

### Managed Identities

The Chaos Studio agent uses a user-assigned managed identity on a virtual machine or virtual machine scale set to authenticate to the Chaos Studio service. You can use a single user-assigned managed identity for multiple virtual machines or virtual machine scale sets. When the agent is enabled, the selected managed identity will be applied to the selected VMs and the Chaos Studio VM Extension will be provisioned to use that identity.

Subscription \* ⓘ

Managed identity \*

**i** If you do not have a managed identity, first create one in the Managed Identities service in the portal.

### Application Insights

The Chaos Studio agent can optionally send diagnostic information about fault execution and agent health to an existing Application Insights account.

Send diagnostic data to Application Insights

- Enabled
- Disabled

To get started, choose an Application Insights account, or if you already know the instrumentation key for your account you can directly specify it below.

Application Insights account

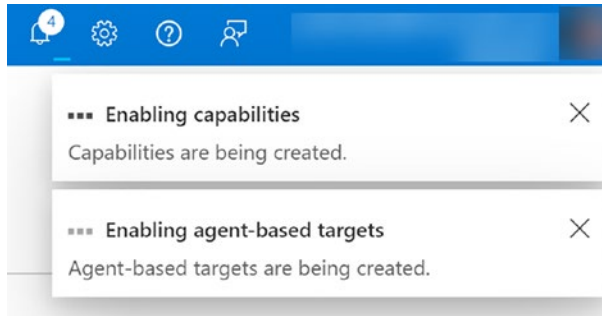
Instrumentation key \*

[Review + Enable](#)[Cancel](#)

**Figure 8-14.** Enable targets

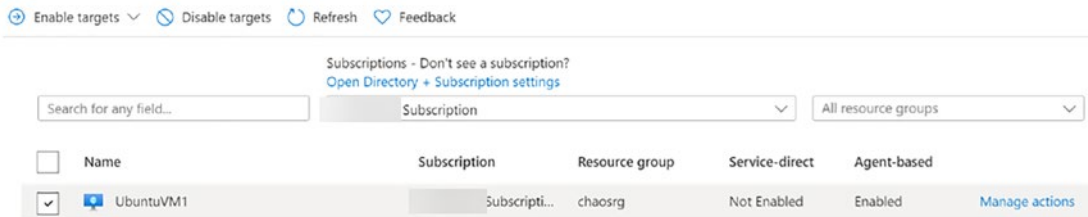
- Click “Review + Enable” and confirm once more in the next blade.

- From the notification pane (upper right in the Azure portal), wait for the updated notification messages, confirming the agent got deployed successfully (see Figure 8-15).



**Figure 8-15.** *Wait for agent deployment*

- Switching back to the Chaos Studio blade in the Azure portal, it will also show a status of “Enabled” for “Agent-based” for the specific virtual machine selected before (see Figure 8-16).



**Figure 8-16.** *Agent enabled*

- Notice the “Manage actions” for the selected virtual machine. This is where you find the different predefined agent-based Azure Chaos actions available (see Figure 8-17).

## Manage actions

PREVIEW

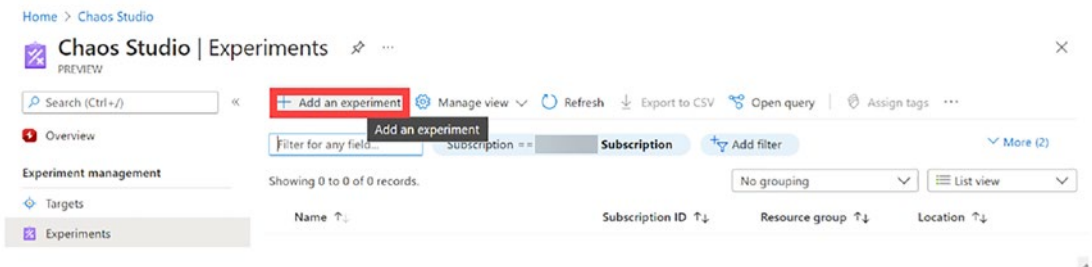
Capabilities can only be modified for enabled targets. Return to the Targets page and enable/disable [agent-based or service-direct] capabilities.

### Agent-based capabilities

<input checked="" type="checkbox"/> Capability	Description
<input checked="" type="checkbox"/> CPU Pressure	
<input checked="" type="checkbox"/> Physical Memory Pressure	
<input checked="" type="checkbox"/> Virtual Memory Pressure (Linux)	
<input checked="" type="checkbox"/> Disk I/O Pressure (Windows)	
<input checked="" type="checkbox"/> Disk I/O Pressure (Linux)	
<input checked="" type="checkbox"/> Stop Service	

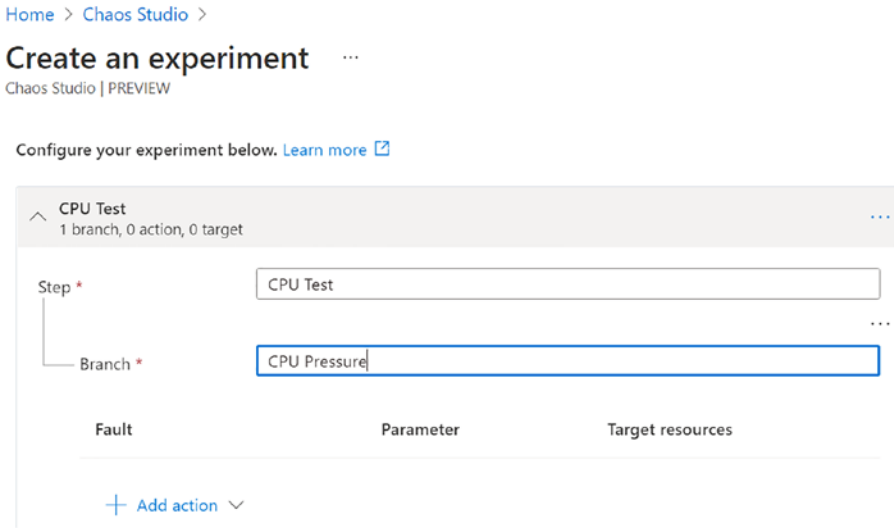
**Figure 8-17.** Manage actions

- From the Chaos Studio blade, select “Experiments” and create a new experiment (see Figure 8-18).



**Figure 8-18.** Add an experiment

- Complete the necessary parameters allowing you to create a new Experiment Resource, define the Resource Group, specify a name for the Experiment (e.g., CPUStressTest), and define the Azure Region (Note: the region should be the same as where the target resource is running).
- Click Next to open the Experiment Designer (see Figure 8-19).



**Figure 8-19.** *Experiment Designer*

23. Provide a descriptive name for Step 1, for example, CPU Test; specify a descriptive name for Branch 1, for example, CPU Pressure. Next, click “Add Action” and select “Add Fault” from the selection list. This opens the “Add Fault” blade, where you need to specify some parameters for this fault simulation. First, click the Faults drop-down list, and select “CPU Pressure”. This test will simulate a high CPU spike for the Ubuntu VM in our example. Next, set the Duration parameter to ten minutes, and move the Pressure Level to 99% (see Figure 8-20).

## Add fault

PREVIEW

### Faults

A fault is a failure that Chaos Studio will inject into your application or infrastructure. Select which fault you would like to add to your experiment. For more information and sample values for each fault, [visit the fault library](#). [↗](#)

CPU Pressure ▼

### Parameters

Parameters allow you to customize the impact of a fault.

Duration (minutes)

10 ▲▼

pressureLevel

99

virtualMachineScaleSetInstances ⓘ

**Figure 8-20.** CPU pressure fault

24. Click “Next” to specify the Target VM for this action and fault simulation. Here, select the Ubuntu VM you used earlier (see [Figure 8-21](#)).

## Add fault

PREVIEW

Fault details  Target resources

ⓘ A chaos target resource is an Azure resource against which you will run a fault. A resource must first be set up as a chaos target before it can be selected here. [Learn more](#)

Subscriptions - Don't see a subscription? [Open Directory + Subscription settings](#)

PETENDER Subscription ▼

<input checked="" type="checkbox"/>	Resource name	Subscription	Resource group
<input checked="" type="checkbox"/>	 UbuntuVM1		chaosrg

**Figure 8-21.** Select the target



- 25. Confirm the creation of the experiment by clicking “Review and Create” (see Figure 8-22).

Home > Chaos Studio >

## Create an experiment

Chaos Studio | PREVIEW

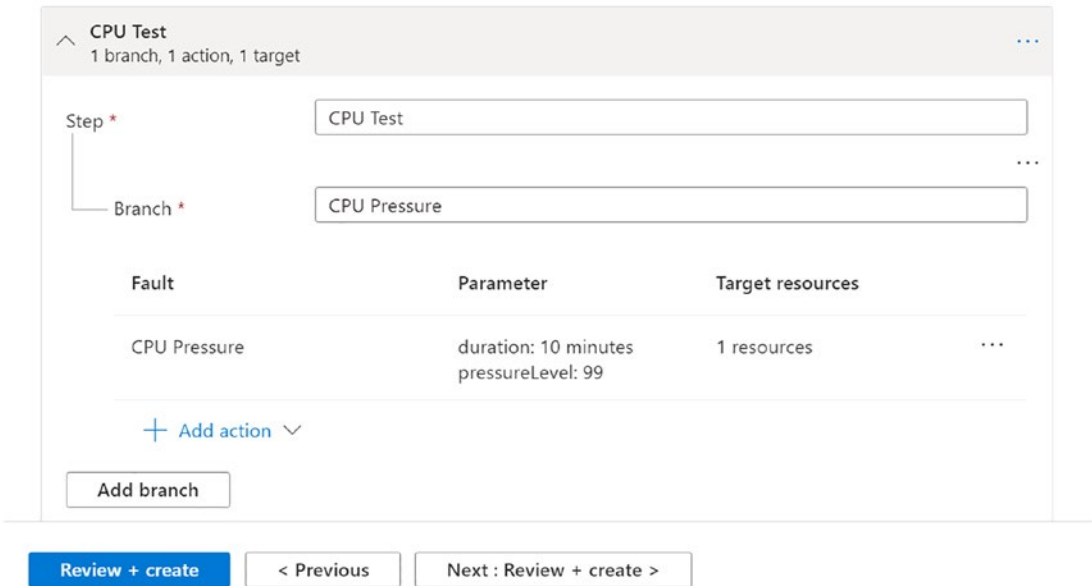


Figure 8-22. Confirm the experiment

- 26. After a few minutes, the newly created experiment shows up in the list of Chaos Experiments (see Figure 8-23).

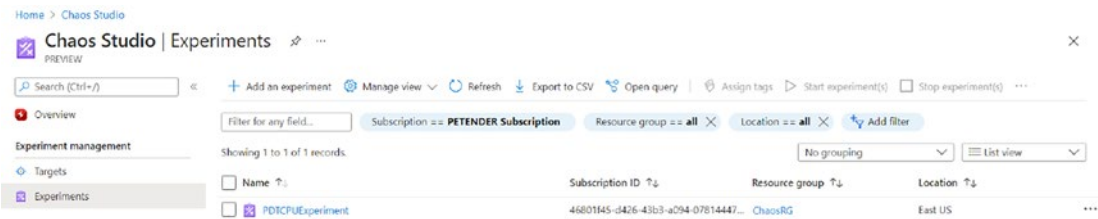
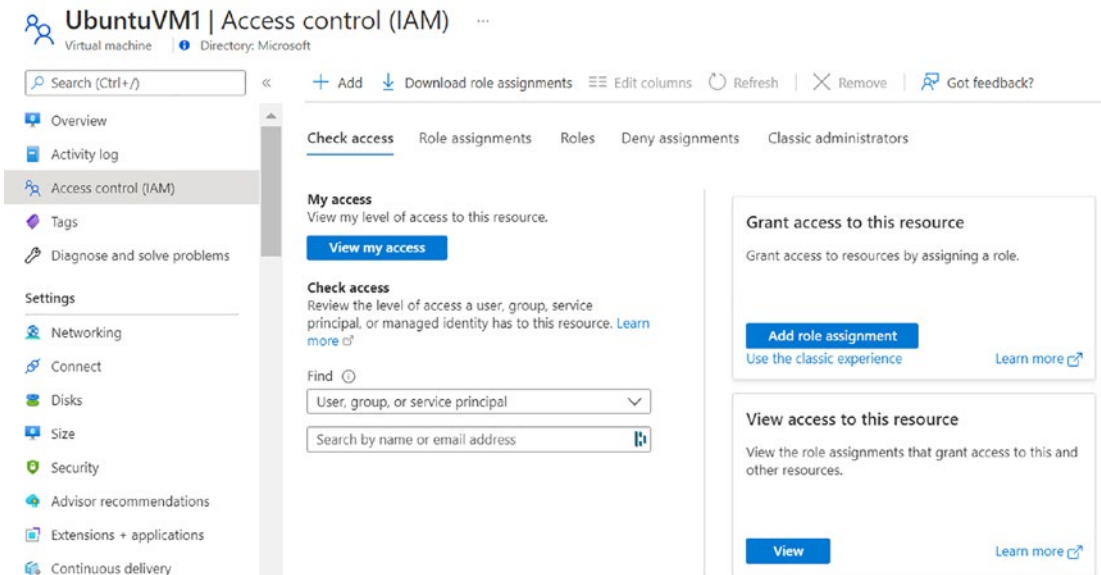


Figure 8-23. Experiments tab

27. In order to allow the Chaos Experiment to interact with the target Resource, it needs Reader permissions from a Role-Based Access Control (RBAC) level. In real life, I would recommend locking this down to the individual resource level or resource group level or any suitable level in your organization. To make this happen, select the level you want (I'm going for that individual UbuntuVM1 resource), and select IAM (Access Control) from the Settings blade (see Figure 8-24).



**Figure 8-24.** Access Control

28. Click “+Add” from the top menu and select “Add Role Assignment”. From the list of Roles, select “Reader” (see Figure 8-25).

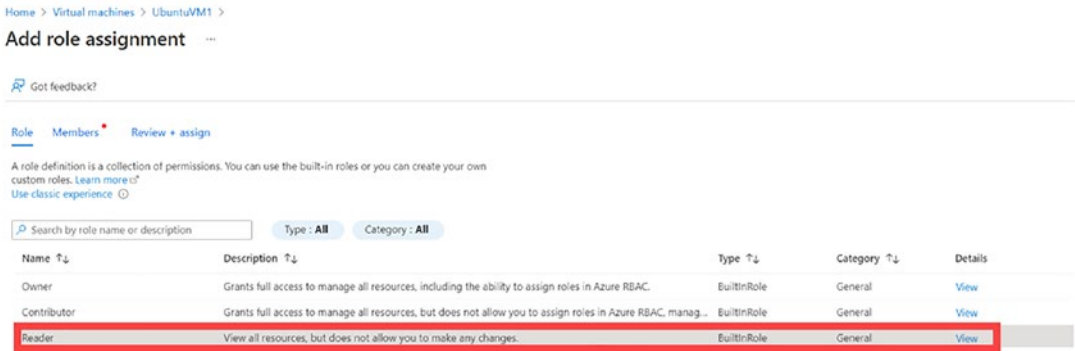


Figure 8-25. Reader role

- 29. Click “Add Members”, and search for the name of the Chaos Experiment you created before. In my case, this was PDTCPUExperiment (see Figure 8-26).

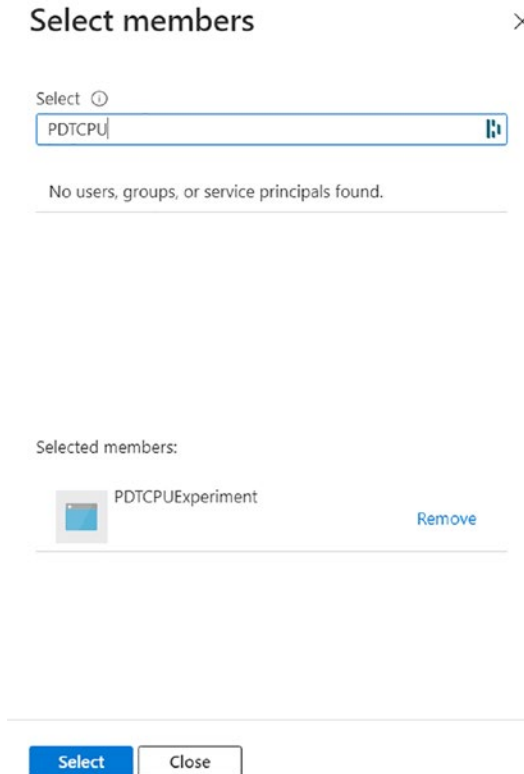
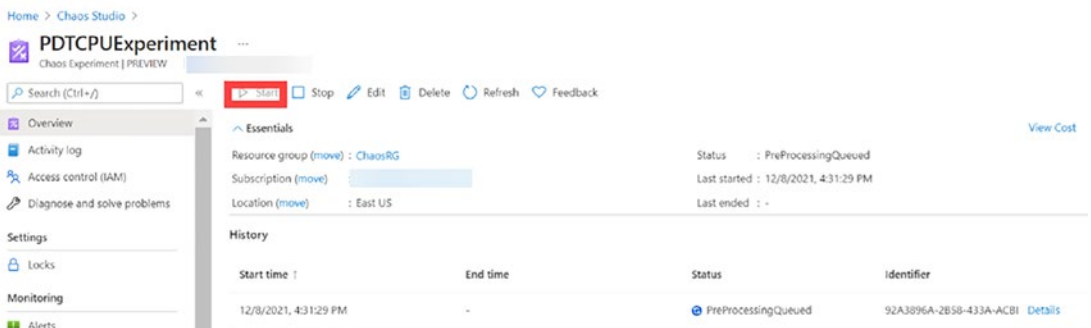


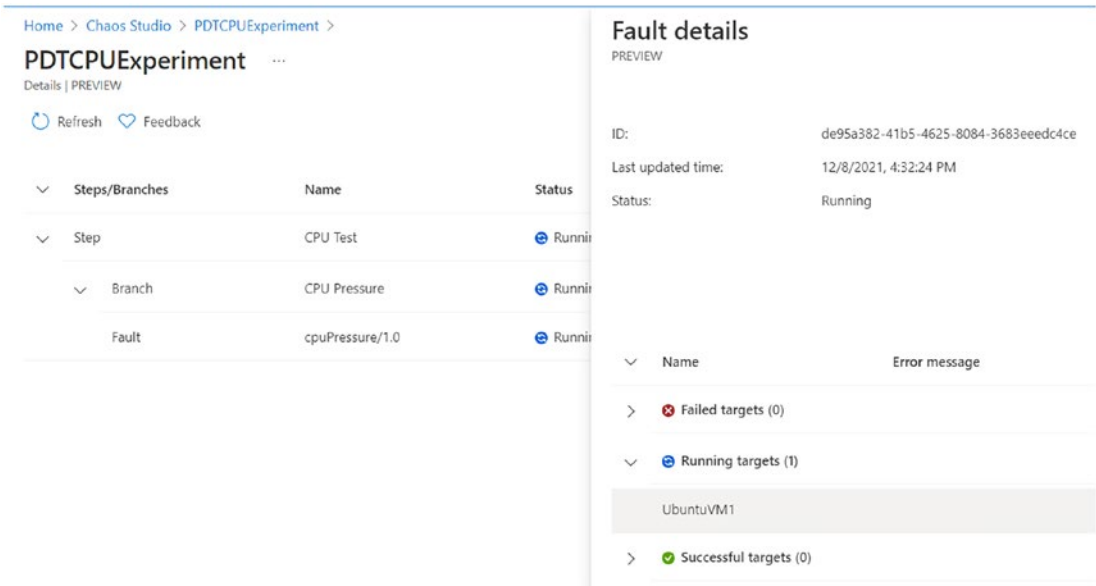
Figure 8-26. Chaos experiment added to Reader

30. Confirm by clicking “Select”, followed by “Review and Assign”. Wait for this RBAC permission to get assigned to the Chaos Experiment.
31. Return to Chaos Studio, select Experiments, and open the experiment created earlier.
32. From the top menu, click “Start” to kick off the CPU Pressure Chaos Experiment (see Figure 8-27).



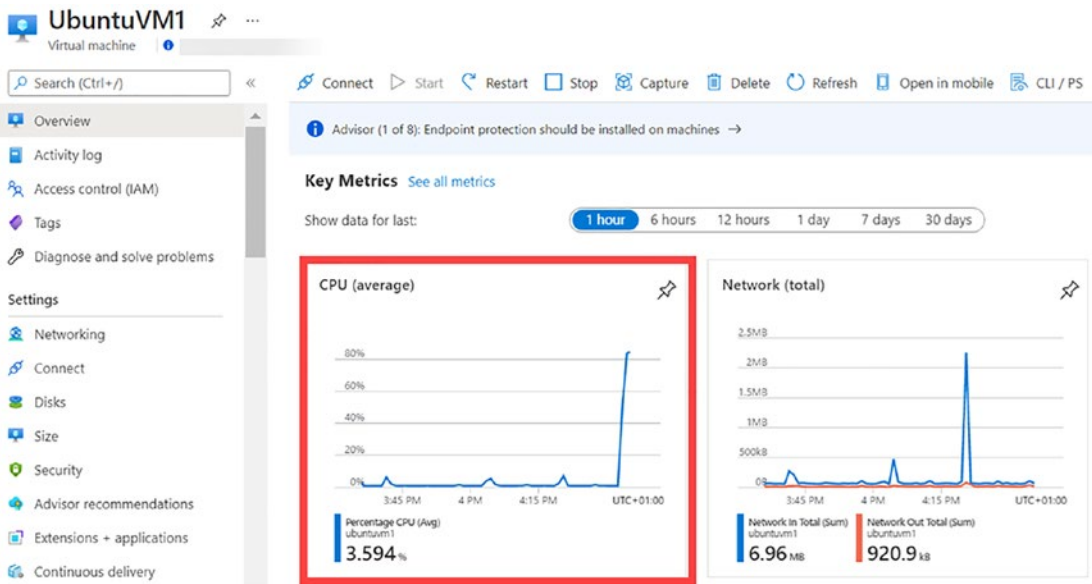
**Figure 8-27.** Start the experiment

33. Validate the status throughout the process. Once the status switched to “Running”, click on the “Details” link to the right of the item line. This will show you more real-time details about the running Chaos Experiment (see Figure 8-28).



**Figure 8-28.** Experiment details

34. You can also validate the CPU load from the Virtual Machine Monitoring itself. Navigate to the virtual machine, open its details blade, and navigate to Monitoring. Here, notice the CPU (average) load in the first graph (see Figure 8-29).



**Figure 8-29.** VM Overview tab

35. And when clicking on the graph, it will open a more detailed “Metrics view” for this resource.
36. As you can see from the chart (see Figure 8-30), once the Experiment kicked off, the CPU was gradually, but quite fast, moving up to (close to) 99% and running at this level for around ten minutes based on the parameters defined for the fault action. Once the test is complete, you can also validate this from the same charts (see Figure 8-31).

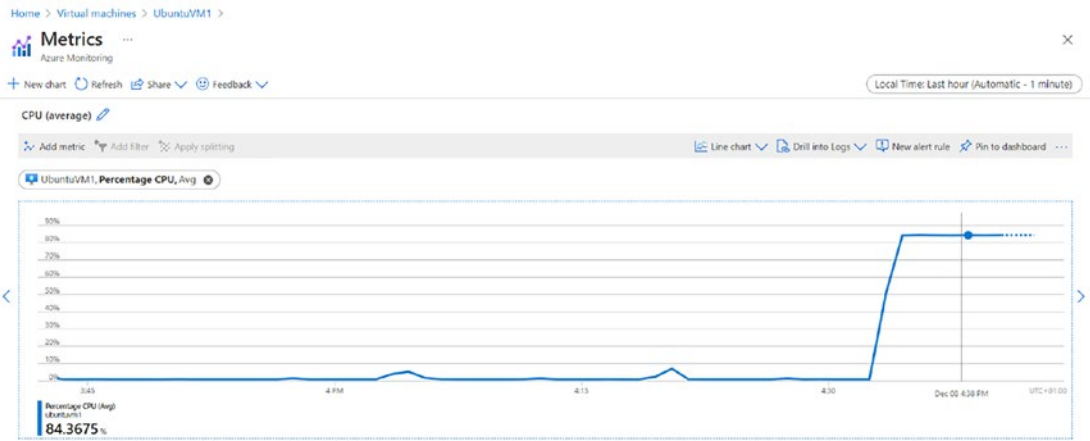


Figure 8-30. Metric Explorer for VM

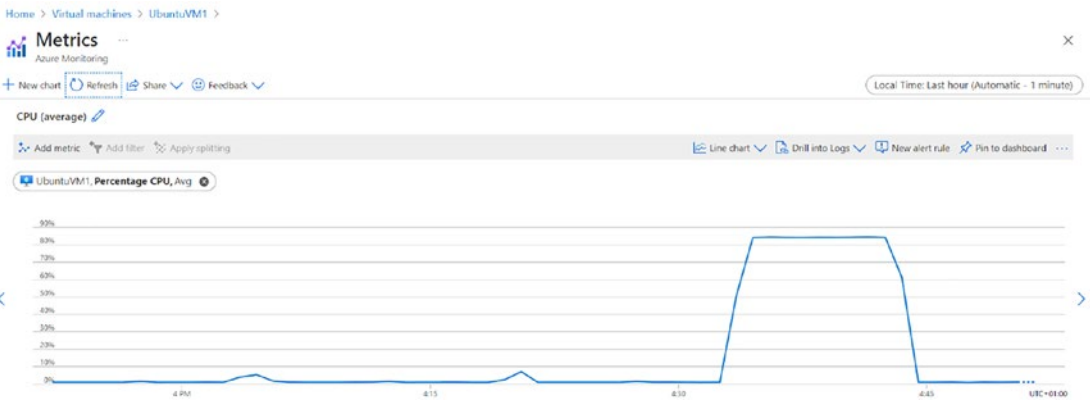


Figure 8-31. Test finished

37. Also from the Azure Chaos Studio details, you can validate the status of this experiment (see Figure 8-32).

Steps/Branches	Name	Status
Step	CPU Test	Completed
Branch	CPU Pressure	Completed
Fault	cpuPressure/1.0	Completed

**Figure 8-32.** Experiment details

38. This completes the configuration and walk-through of deploying Azure Chaos Studio, allocating the correct identities, required prerequisites such as User-Managed Identity and Application Insights resources, as well as how to create and run a Chaos Experiment.

---

**Note** There are a lot more fault scenarios available, and the Chaos Studio team is heavily working on even more real-life-based fault scenarios to integrate them in the solution. I relied on the most common scenario discussed during preview demos from the Product Group for now, but hopefully, by the time you are reading this book, it will be an impressive, useful list of faults for different Azure resources and services scenarios.

---

## Onboarding an AKS Cluster to Chaos Studio

In this next example, I'll detail how to integrate Chaos Engineering fault injection to validate an Azure Kubernetes Cluster with Linux nodes scenario.

If you don't have an AKS deployed yet, feel free to use this Microsoft QuickStart article as a guidance:

<https://docs.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-rm-template?tabs=azure-cli>



1. One of the main differences between the previous virtual machine scenario and Azure Kubernetes Service, is that you need to install ***Chaos Mesh faults*** into the cluster resource before you can interact with it from within Azure Chaos Studio. While it is possible to run this remotely using Azure CLI from your admin workstation, it might be easiest to run this from within Azure Cloud Shell.
2. With a deployed Kubernetes solution, open Azure Cloud Shell and select the Bash option. Establish connectivity to the AKS cluster by running the following command:

```
az aks get-credentials -g $RESOURCE_GROUP -n $CLUSTER_NAME
```

replacing the variables \$RESOURCE\_GROUP and \$CLUSTER\_NAME with the actual values of the deployed ASK environment (see Figure 8-33).



**Figure 8-33.** Connect to the cluster

3. In my example, I ran this command earlier, but it is fine to overwrite, to make sure you get the latest credentials information.

Next, you need to install the Chaos Mesh package, which is easiest using Kubernetes Helm package manager. Since Helm is already part of Azure Cloud Shell, you can go ahead and run the following commands:

```
helm repo add chaos-mesh https://charts.chaos-mesh.org
helm repo update
kubectl create ns chaos-testing
```

```
helm install chaos-mesh chaos-mesh/chaos-mesh --namespace=chaos-testing --set chaosDaemon.runtime=containerd --set chaosDaemon.socketPath=/run/containerd/containerd.sock
```

resulting in the following output (see Figure 8-34).

```
Bash
r@Azure:~$ helm repo add chaos-mesh https://charts.chaos-mesh.org
"chaos-mesh" already exists with the same configuration, skipping
r@Azure:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "chaos-mesh" chart repository
Update Complete. ✨ Happy Helming! ✨
r@Azure:~$ kubectl create ns chaos-testing
namespace/chaos-testing created
r@Azure:~$ helm install chaos-mesh chaos-mesh/chaos-mesh --namespace=chaos-testing --set chaosDaemon.runtime=containerd --set chaosDaemon.socketPath=/run/containerd/containerd.sock
NAME: chaos-mesh
LAST DEPLOYED: Mon Jun 27 20:20:45 2022
NAMESPACE: chaos-testing
STATUS: deployed
REVISION: 1
TEST SUITE: none
```

**Figure 8-34.** Install Chaos Mesh using Helm

- Next, validate that the Chaos Mesh PODs are successfully running by executing the following command:

```
kubectl get pods --namespace chaos-testing -l app.kubernetes.io/instance=chaos-mesh
```

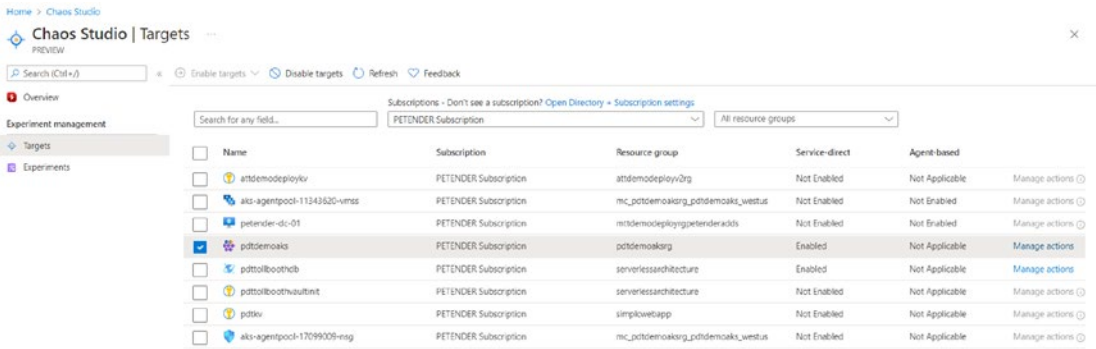
```
Bash
r@Azure:~$ kubectl get pods --namespace chaos-testing -l app.kubernetes.io/instance=chaos-mesh
```

NAME	READY	STATUS	RESTARTS	AGE
chaos-controller-manager-65d888c78f-gsxdj	1/1	Running	0	66s
chaos-controller-manager-65d888c78f-hftdq	1/1	Running	0	66s
chaos-controller-manager-65d888c78f-sd5s2	1/1	Running	0	66s
chaos-daemon-2bhqm	1/1	Running	0	66s
chaos-daemon-nbr94	1/1	Running	0	66s
chaos-daemon-xsxn4	1/1	Running	0	66s
chaos-dashboard-84ffc4bb9-r5l5q	1/1	Running	0	66s

**Figure 8-35.** Confirm Chaos Mesh

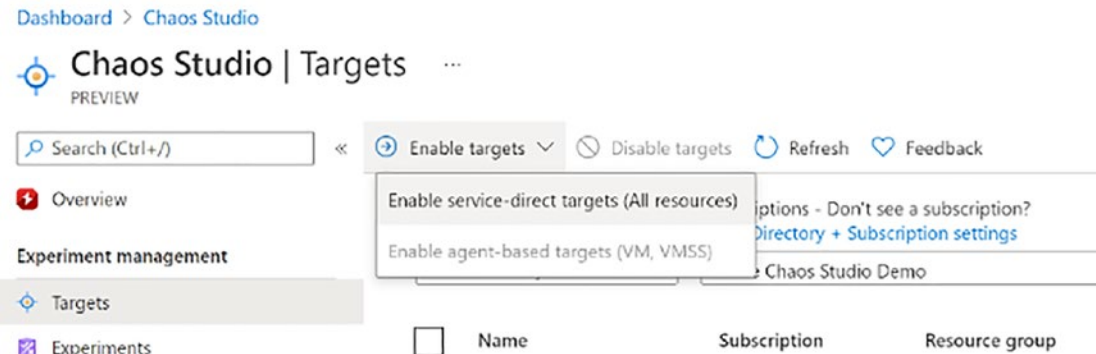
Your AKS cluster is now ready for Azure Chaos Studio integration (see Figure 8-35).

- From the Azure portal, browse to Chaos Studio (Preview), and select Targets. Identify your AKS resource (see Figure 8-36).



**Figure 8-36.** Select the AKS target

- Notice the Agent-based option is not applicable in this case, since we’re directly connected to the service-layer of the Kubernetes cluster.
- In the upper menu, select “Enable Targets”/Enable Service-Direct targets (All Resources).



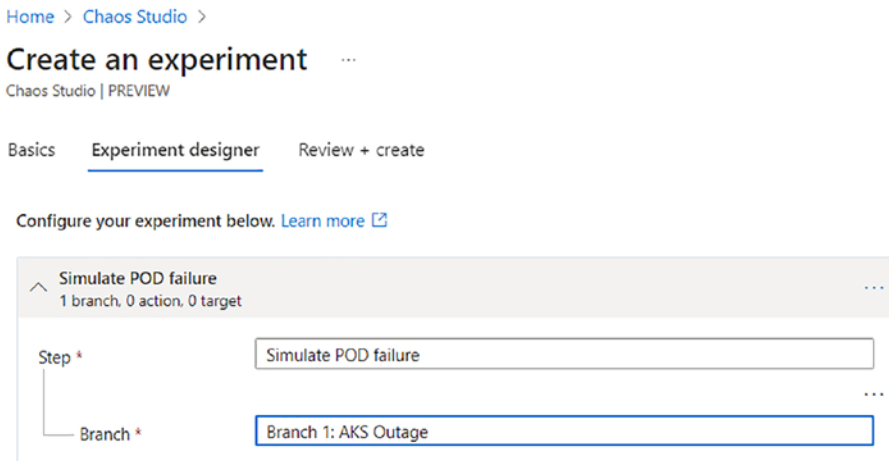
- Wait for the notification regarding this component installation.

Once this is done, you can continue by specifying the Chaos Studio experiment.

- From Chaos Studio, browse to Experiments. Click “Create Experiment”. Complete the Azure required settings for Subscription, Resource Group, Name for the experiment, and the location where you want to create this. Since each experiment is nothing more than a stand-alone Azure Resource object, it should feel familiar to creating similar objects.

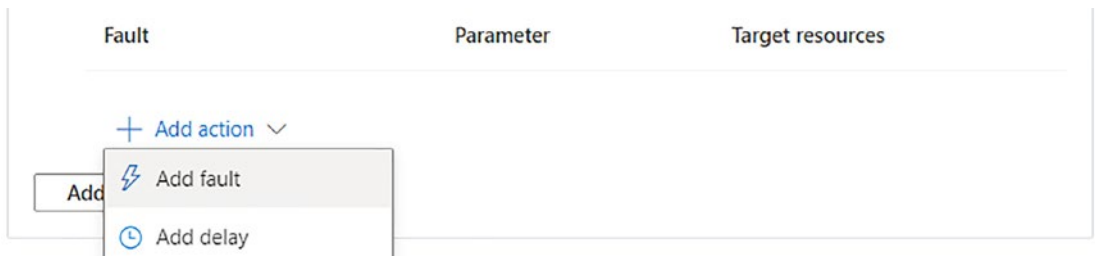
2. Click “Next” to navigate to the Experiment Designer. This is where you define the actual chaos testing (fault injection) you want to run as part of your Chaos Engineering. As mentioned at the start of the chapter, each fault is following a hierarchy of Steps and Branches, under which you define one or more fault scenarios (Actions).

Provide a descriptive name for the Step (e.g., Simulate POD Failure) as well as for the Branch (e.g., Branch 1: AKS Outage) (see Figure 8-37).



**Figure 8-37.** Experiment Designer

3. Under the Fault section, click “+Add Action”/Add Fault (see Figure 8-38).



**Figure 8-38.** Add fault

- From the list of faults available, select “AKS Chaos Mesh Pod Chaos”, allowing to simulate a POD failure within the AKS cluster. Specify the duration in minutes, and complete the jsonSpec field.

This is where I’m hoping the GA version of the product will make this a bit easier, as for now you need to *translate* the Chaos Mesh YAML syntax into a JSON syntax. Copy the following snippet of YAML (common from the Chaos Mesh docs, stripped off the useful information):

```
action: pod-failure
mode : all
duration: '600s'
selector:
  namespaces:
    - default
```

which should look like this in a JSON format:

```
{
  "action": "pod-failure",
  "mode": "all",
  "duration": "600s",
  "selector": {
    "namespaces": [
      "default"
    ]
  }
}
```

*The resulting portal blade looks like the one shown in Figure 8-39.*

## Add fault

PREVIEW

Fault details
  Target resources

### Faults

A fault is a failure that Chaos Studio will inject into your application or infrastructure. Select which fault you would like to add to your experiment. For more information and sample values for each fault, [visit the fault library](#). [↗](#)

AKS Chaos Mesh Pod Chaos

### Parameters

Parameters allow you to customize the impact of a fault.

Duration (minutes)

10

jsonSpec \*

```
{
  "action": "pod-failure",
  "mode": "all",
  "duration": "600s",
  "selector": {
    "namespaces": [
      "default"
    ]
  }
}
```

**Figure 8-39.** Fault description

5. Click “Next: Target Resources” to specify the AKS Resource (see [Figure 8-40](#)).

## Add fault

PREVIEW

- Fault details
- Target resources

ⓘ A chaos target resource is an Azure resource against which you will run a fault. A resource must first be set up as a chaos target before it can be selected here. [Learn more](#)

Subscriptions - Don't see a subscription? [Open Directory + Subscription settings](#)

PETENDER Subscription

<input checked="" type="checkbox"/>	Resource name	Subscription	Resource group
<input checked="" type="checkbox"/>	 emoaks		moaksg

**Figure 8-40.** Select the target

- You are now returning to the “Create Experiment” blade, from where you can confirm the creation. Click the “Create” button to do this (see Figure 8-41).

Home > Chaos Studio >

## Create an experiment

Chaos Studio | PREVIEW

Basics Experiment designer Review + create

Configure your experiment below. [Learn more](#)

Simulate POD failure  
1 branch, 1 action, 1 target

Step \*

Branch \*

Fault	Parameter	Target resources
AKS Chaos Mesh Pod Chaos	duration: 10 minutes jsonSpec: { "action": "po...	1 resources

[+ Add action](#)

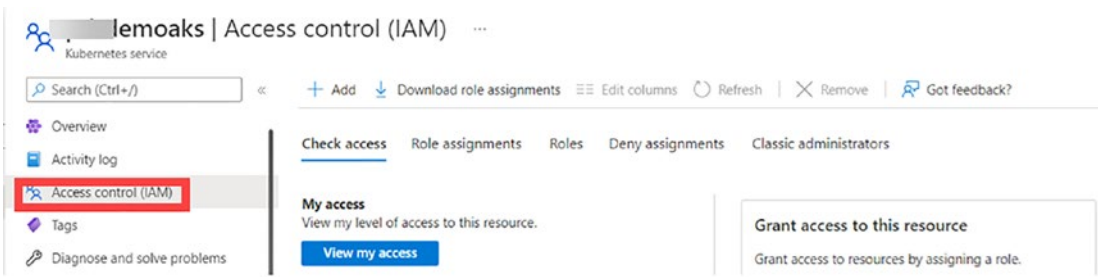
Provide feedback  
[Did you find what you needed? Let us know how it went.](#)

---

**Figure 8-41.** Create an experiment

7. If you went through the previous UbuntuVM scenario, you already know that this experiment will be linked to a newly created System-Assigned Managed Identity. To allow Chaos Studio to interact with your AKS cluster, you need to specify the correct permissions (IAM) on the AKS cluster for this System-Assigned Managed Identity (see Figure 8-42).

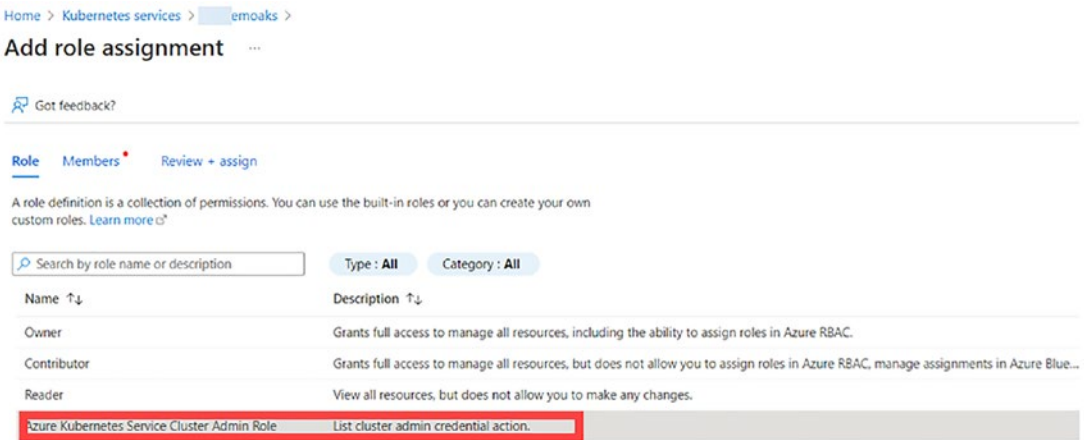




**Figure 8-42.** AKS cluster’s access control

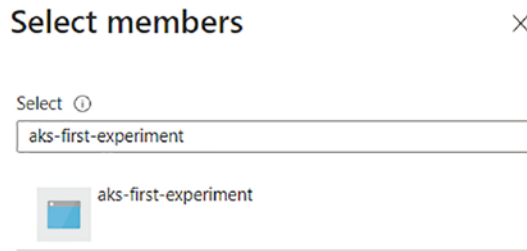
*To do this, navigate to the AKS Service resource in the Azure portal, and select “Access Control (IAM)”.*

8. Click “+Add/Add Role Assignment”. In the list of Roles, search for “Azure Kubernetes Service Cluster Admin Role” and select it (see Figure 8-43). Click “Next”.



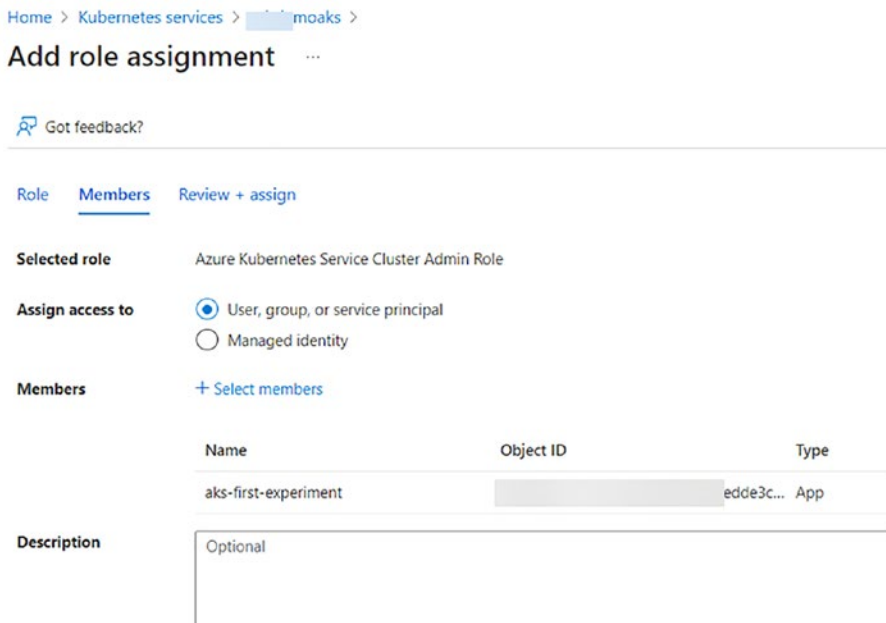
**Figure 8-43.** Select the role

9. Click the “+Members” section in the next screen, and search for the name of the experiment you used (aks-first-experiment in my example) (see Figure 8-44).



**Figure 8-44.** Select experiment system MI

- The configuration of the Role-Based Access looks like the one shown in Figure 8-45.



**Figure 8-45.** Role assignment

- Confirm the creation by clicking the “Review + Create” button. Wait for the permissions to get defined. Once complete, let’s trigger an experiment.
- Navigate back to Azure Chaos Studio and select “Experiments”. Select the experiment you just created (see Figure 8-46) and click “Start Experiment” and confirm with Yes.

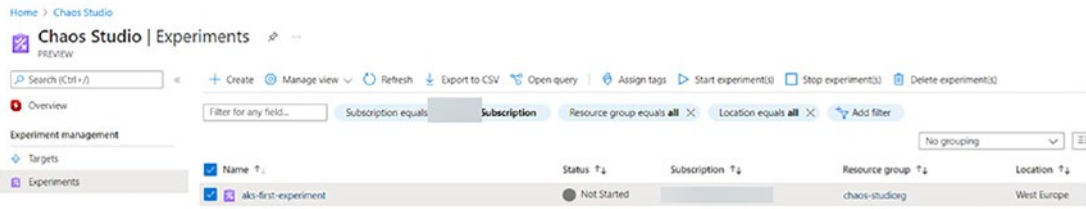


Figure 8-46. Select the experiment

13. Click on the Name of the Experiment, to see a more detailed view of the process (see Figure 8-47).

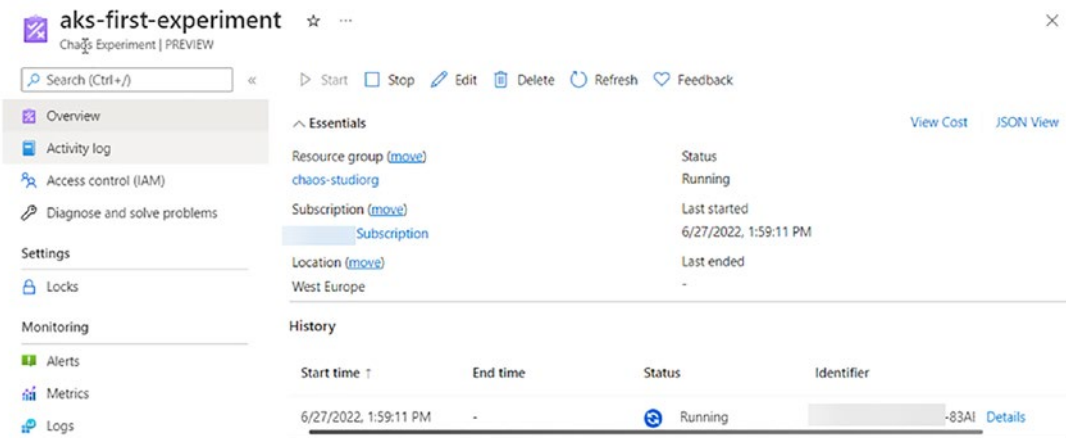


Figure 8-47. Experiment details

14. Wait for the process to complete or click “Details” to validate more information regarding the ongoing process.
15. This completes the process of how to set up Chaos Studio for an AKS Kubernetes Services Cluster and run a POD failure simulation.

I would encourage you to continue experimenting and playing with several of the other scenarios available for both Azure Virtual Machines and AKS, as well as for other Azure resources such as Azure Key Vault secret store, or injecting faults into an Azure Network Security Group.

Where this was the original end of the Chaos Engineering topic with Azure Chaos Studio (Preview), as mentioned earlier, Microsoft released another SRE-minded tool allowing for load/performance testing. While not necessarily limited to SRE, the tool is bringing in so many SRE-related features; we decided to add another quick intro on this Preview service in this chapter. Let's check it out.

## Load/Performance Testing

Load tests, often called (or related to) performance tests, study the behavior of your solutions under a certain load of virtual users, analyzing metrics like response times or how architecture adapts to load (autoscaling).

Remember from Chapter 3, **performance testing** could be categorized in two groups:

- **Load testing** is mainly used for triggering scalability under certain expected user loads (try to simulate possible scenarios).
- **Stress testing** is often used to validate the maximum load an application or system can withstand before it breaks (tries to find the system's limits).

Historically, Microsoft has provided tools for performance and load testing with Visual Studio products. However, Visual Studio 2019 is the last version that fully supports that feature. Together with Azure DevOps, this toolset used to provide a cloud-based, CI/CD-focused, load-testing offering until March 2020 : <https://devblogs.microsoft.com/devops/cloud-based-load-testing-service-eol/>. The product team started recommending switching to the community popular open source Apache JMeter (<https://jmeter.apache.org/>) to design performance tests. Running highly scalable Apache JMeter tests is not easy regarding infrastructure setup; people started using Kubernetes-based architectures to create highly scalable tests (example: <https://techcommunity.microsoft.com/t5/azure-global/scalable-apache-jmeter-test-framework-using-azure-kubernetes/ba-p/1197379>).

Due to this complexity, Azure decided to release the following service, an easy to define/configure/execute cloud-based load-testing offering.

## Azure Load Testing

Additionally, at Microsoft Ignite in November 2021 (still in Preview), Microsoft also announced a new fully managed Azure Service for load-testing scenarios. It enables engineers to run high-scale simulations based on JMeter scripts to understand performance bottlenecks of your solutions. Bringing many advantages like

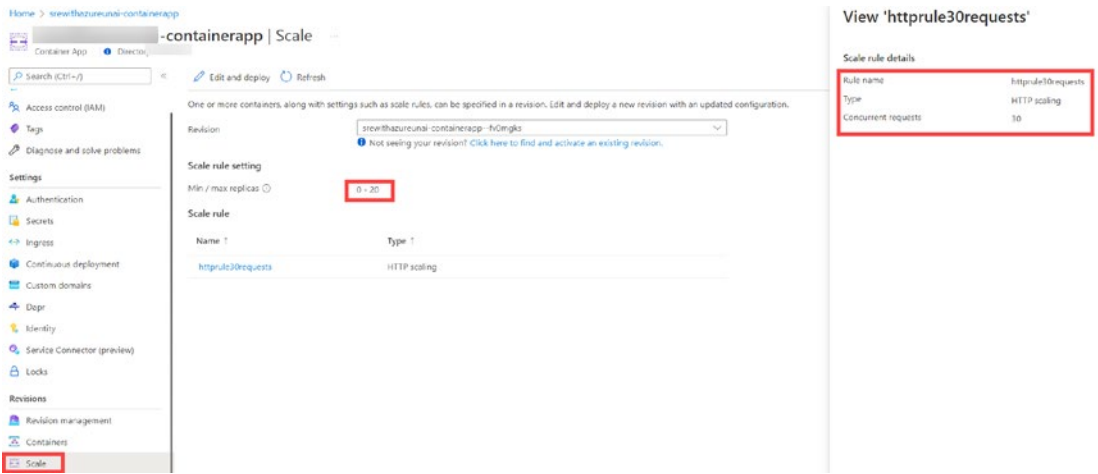
- No complex testing infrastructure to care about
- View of client/server metrics for predefined resources
- Easy integration for CI/CD execution on Azure DevOps or GitHub Actions pipelines

Easy load tests can be created by just providing URL to use or uploading more advanced JMeter scripts. Let's review the tool based on a demo scenario.

### Azure Load Testing for Azure Container App

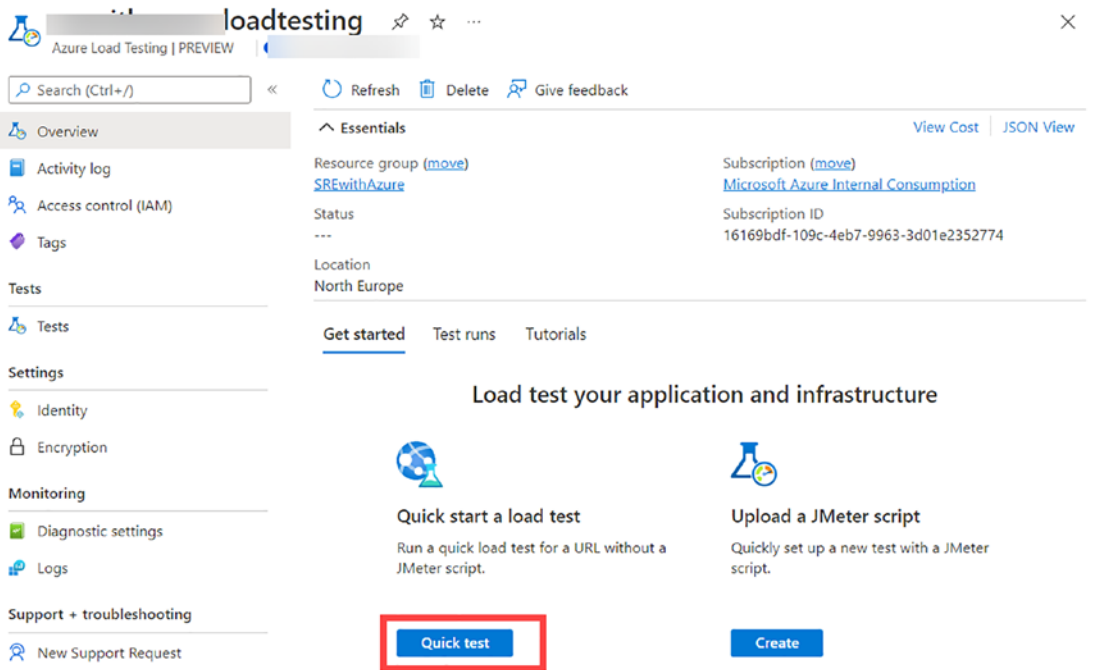
The scenario will be based on the solution used across previous chapters: .NET 6 containerized solution running on Azure Container Apps. As mentioned in previous chapters, Azure Container Apps lets you avoid Kubernetes management/configuration complexities by providing an easy way to apply those Kubernetes-related features. This demo will use autoscaling features offered to scale our container app from 0 to 20 replicas based on HTTP requests handled (many other rules based on KEDA are supported). KEDA is a Kubernetes-based Event-Driven Autoscaler and is a simple lightweight component that can be added to your cluster (<https://keda.sh/>). Azure Container Apps already includes it.

As shown in Figure 8-48, the container app revision has an attached autoscale rule based on HTTP requests. Each replica can handle up to 30 requests at the same time, scaling from 0 to 20 instances based on load.



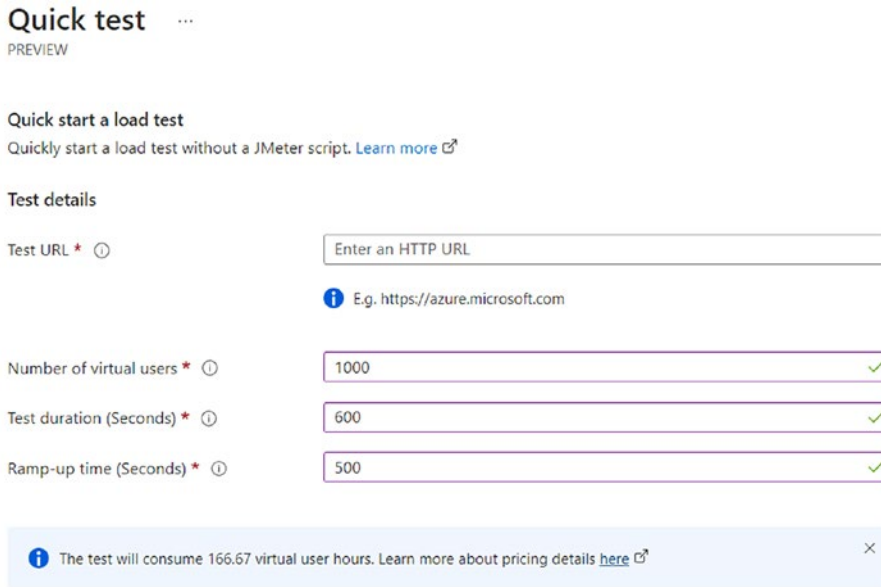
**Figure 8-48.** HTTP scale rule

On Azure Load Testing (Preview), create a resource, and on the Overview tab, create a Quick Test (URL based instead of JMeter for the demo) (see Figure 8-49).



**Figure 8-49.** Create Quick Test

For the test details (see Figure 8-50), the demo will be testing 1000 virtual users (on the back end, there is a maximum of 250 per test engine, so 4 will be used), running the test for 600 seconds and ramping up the user amount slowly during the first 500 seconds (0 to 1000 users increasing second by second).



**Figure 8-50.** Load test details

Clicking on the **Test** tab and created test, it will take you to the settings of your testing scenario, being able to modify extra settings or see historical execution. Clicking on **Configure ► Tests**, it will show the JMeter file created by the service and parameters defined for it (see Figure 8-51).


## Edit test ...


PREVIEW

Basics Test plan Parameters Load Test criteria Monitoring

### Script, data and configuration files

Please upload one JMeter script. Uploading data and configuration files is optional. In case your test plan uses a 'user.properties' file, please specify it using the File relevance dropdown.

Choose files \* ⓘ  

File name	Size	Status	File relevance	Progress
<a href="#">quick_test.jmx</a>		 Completed		100% <span>⋮</span>

**Figure 8-51.** Test plan

See how for 1000 users, a maximum of 250 is defined per test engine, needing a total 4 machines in the backend (see Figure 8-52 and Figure 8-53). Secrets can be stored in Key Vault, and managed identities can be used to provide secret values to your JMeter scripts.



# Edit test ...

PREVIEW

- Basics
- Test plan
- Parameters**
- Load
- Test criteria
- Monitoring

## Environment variables

These are non-sensitive parameters exposed as environment variables for access by the load test engine at runtime. [Learn more](#)

Name ⓘ	Value ⓘ	
domain	srewithazureunai-containerapp.yellowsmoke-98e...	
protocol	https	
path	weather	
threads_per_engine	250	
ramp_up_time	500	
duration_in_sec	600	
<input type="text" value="Enter name"/>	<input type="text" value="Enter value"/>	

## Secrets

Secrets are sensitive parameters that are required for running the load test. These parameters are passed to the load test engine in a secure way and are not exposed anywhere. Secrets should be stored in Azure Key Vault, and the secret identifier should be provided as the value. [Learn more](#)

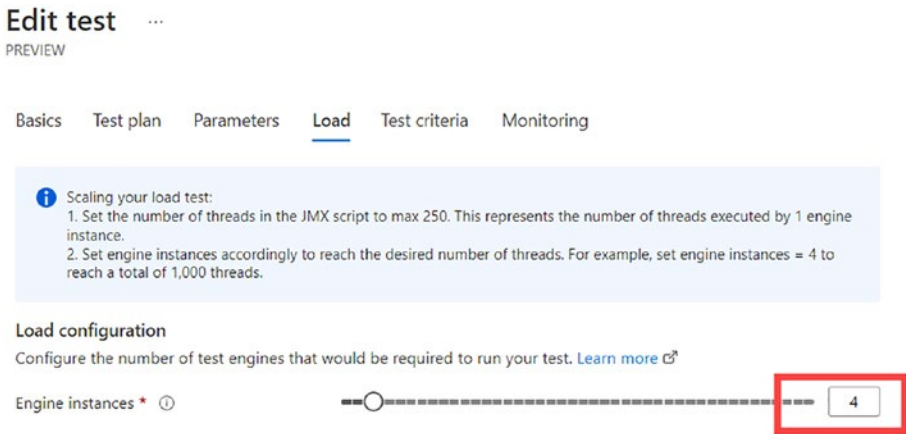
Name ⓘ	Value ⓘ
<input type="text" value="Enter name"/>	<input type="text" value="Enter value"/>

## Key Vault reference identity

Configure the test to use this identity to access Azure Key Vault for all secret references. By default this is system-assigned identity.

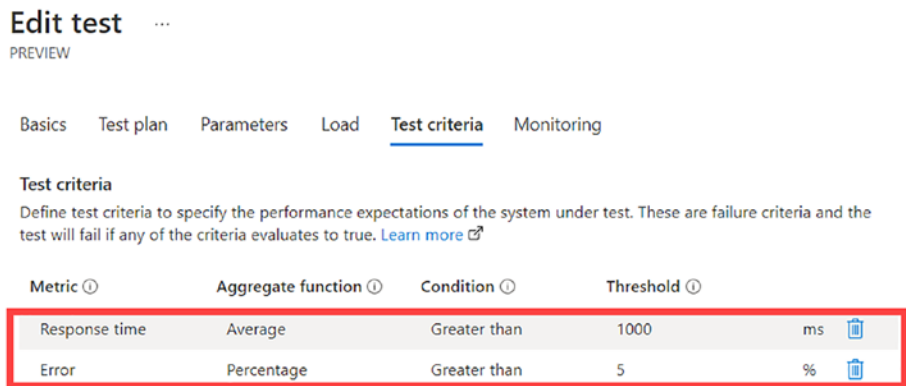
- Identity type \*
- System assigned identity
  - User-assigned identity

**Figure 8-52.** Parameters



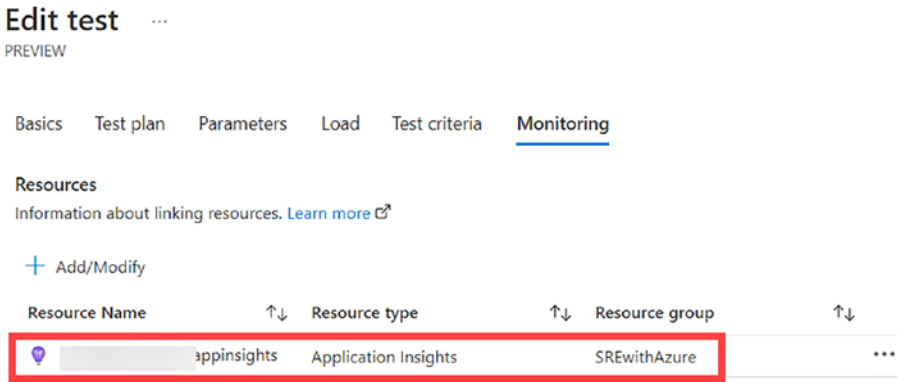
**Figure 8-53.** Engine instances

Test criteria can also be defined for evaluating that the conditions are satisfying your expected metrics (useful to break CD pipelines in case they do not) (see Figure 8-54).



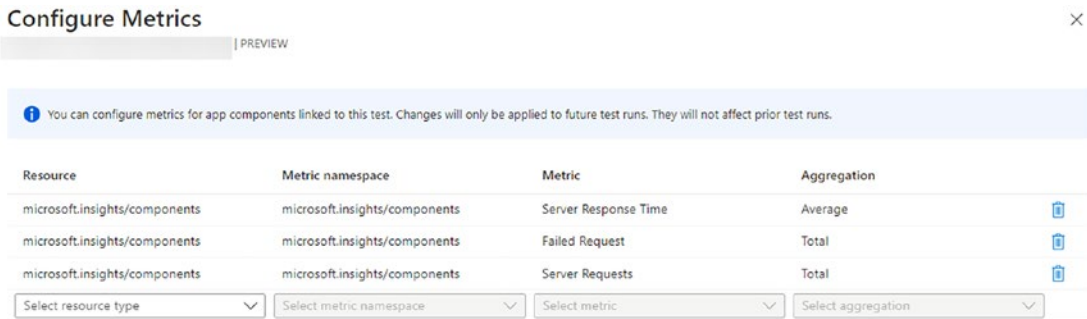
**Figure 8-54.** Test criteria

Finally, the testing view offers an option to link related resources so that metrics/logs from those resources can be shown on the load-testing report. In this case, Azure Container Apps is not a supported resource yet, but you can link it to the Application Insights resource used to monitor it (see Figure 8-55).



**Figure 8-55. Monitoring**

Back on the test page, clicking on **Configure** ➤ **Metrics** (see Figure 8-56) will give you options to include metrics (related to resources attached previously) on the execution report.



**Figure 8-56. Metrics**

Once the test has run, you can review the execution report given by the tool (see Figures 8-57 and 8-58). The report shows the test criteria results, client-side charts, and server-side charts. You can see how our Container Apps solution has been able to incrementally handle up to 1000 virtual users; even if the server response has been slightly affected, no failures have occurred.

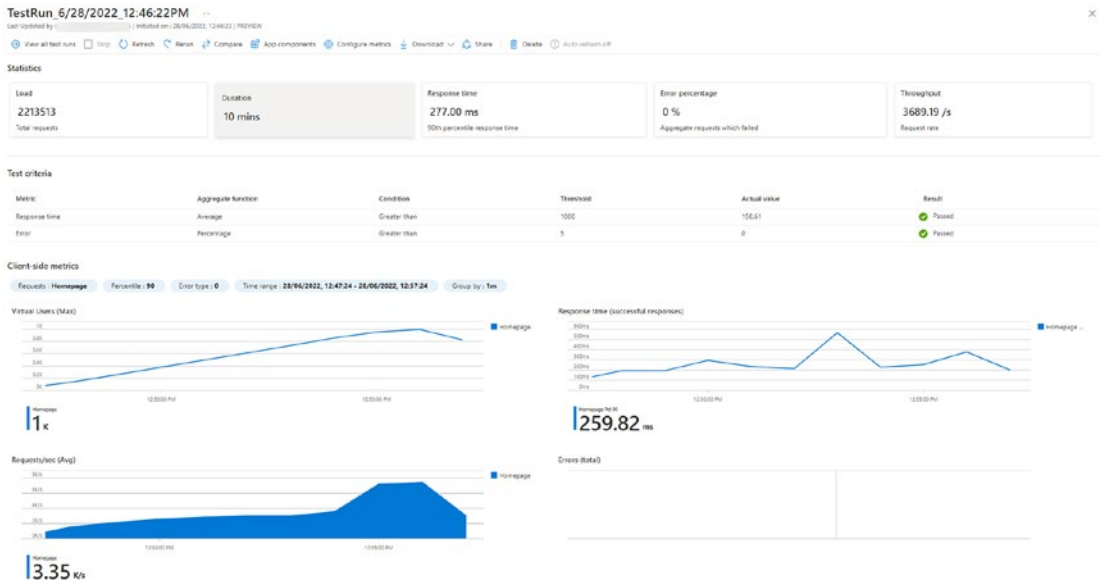


Figure 8-57. Execution report part 1

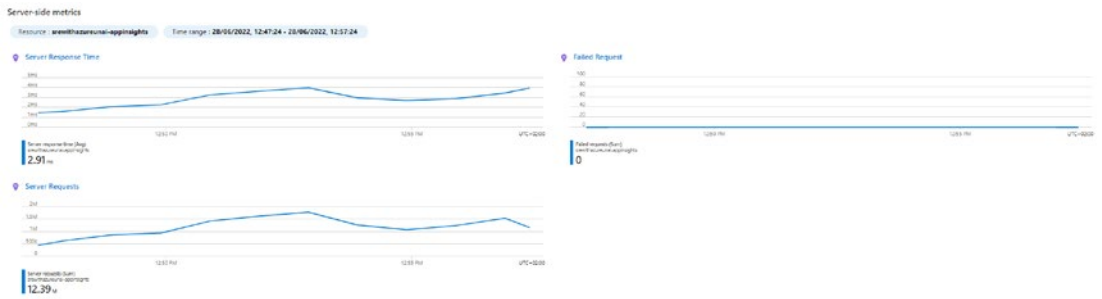


Figure 8-58. Execution report part 2

As Azure Container Apps is not officially supported yet for metric tracking, if you switch to the Metric Explorer tab of your Azure Container Apps (Log Analytics queries could also be used), you will see how the solution has scaled the number of replicas during the load-test execution to handle load (see Figure 8-59).

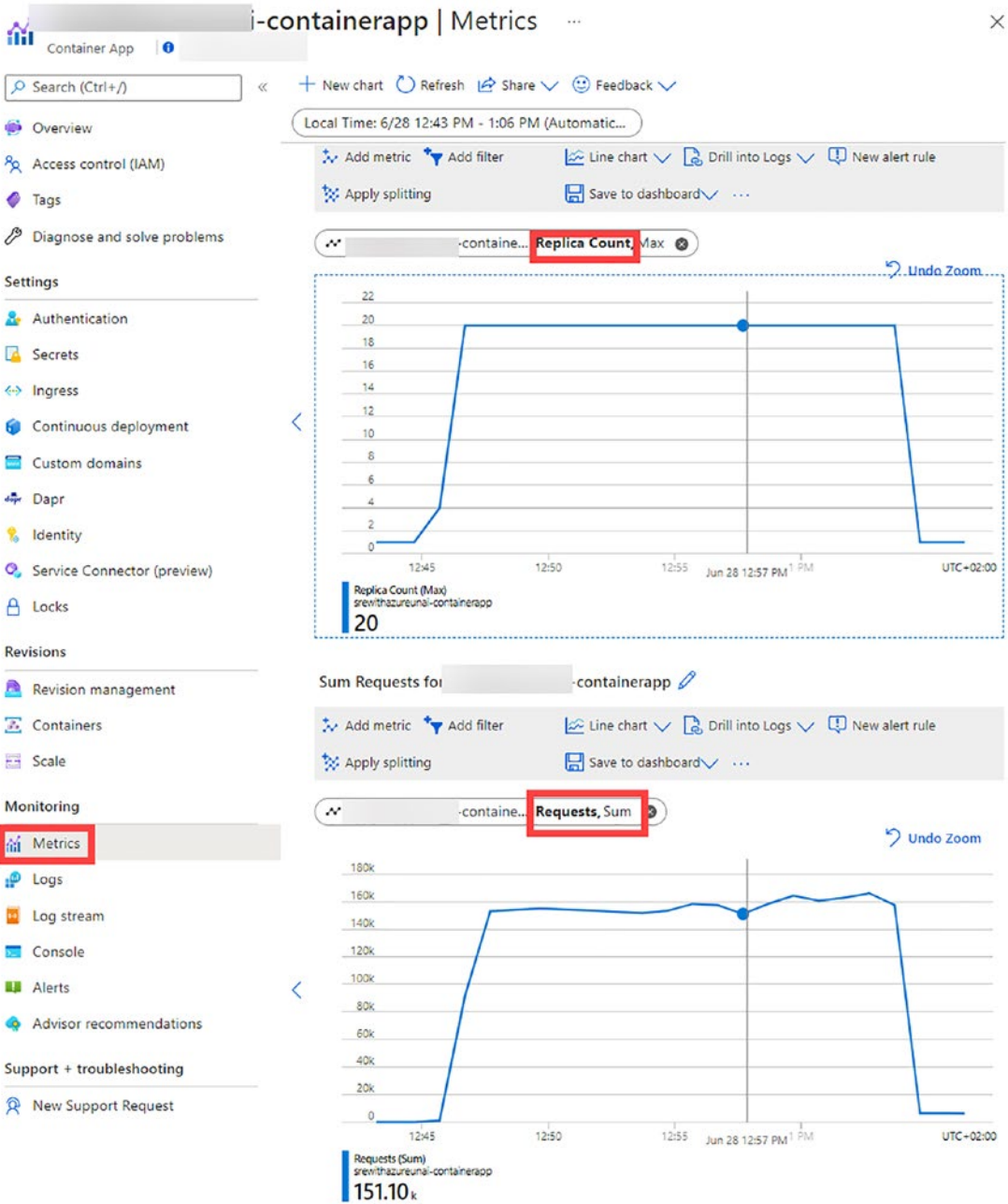


Figure 8-59. Azure Container Apps Metrics Explorer

## Summary

In this chapter, you were introduced to the concept of Chaos Engineering, allowing organizations and their SRE teams to run simulated faults against production-running resources. By integrating chaos testing, it allows you to identify any crucial outages when incidents are occurring and prepare your teams to mitigate them. Overall, the more faults are identified for your workloads, the more reliable they will be.

Chaos Engineering was somewhat founded by the Netflix Engineering team, who moved their experience into an (now) open source tool, Chaos Monkey.

The concepts of Chaos Engineering are now also available in Azure, thanks to the (currently in preview) service, Azure Chaos Studio. This module tried to explain and walk you through its reason of existence, as well as guide you through a step-by-step demonstration on how to enable Chaos Experiments such as simulating a high CPU pressure against an Azure Ubuntu Virtual Machine.

Additionally, the chapter included a quick introduction to load/performance testing, focusing on the recently released Azure Load Testing offering (in preview), using a demo scenario to show its capabilities.

# Index

## A

- Application Performance Management (APM), 167, 208
- Automation
  - guest configuration, 104–106
  - overview, 91
  - reliability, 92
  - SRE process, 92, 93
  - testing/releasing, 92
- Azure Container Registry (ACR), 88, 128, 134
- Azure Kubernetes (AKS) cluster
  - access control, 258
  - Chaos Studio experiment, 252
  - command, 250
  - configuration, 259
  - connection, 250
  - detailed view, 260
  - experiment creation, 257
  - experiment designer, 253
  - fault description, 255
  - fault section, 253
  - helm package manager, 250, 251
  - JSON format, 254
  - members section, 258, 259
  - mesh testing, 251
  - navigation, 259
  - role selection, 258
  - service-direct targets, 252
  - target selection, 251, 252, 256
  - websites, 249
- Azure Kubernetes Service (AKS), 54, 55, 57, 72, 87, 230, 249–255, 257, 258, 260

## B

- Blue-green deployment, 22, 60, 123–124, 138, 139

## C

- Canary deployment, 126
- Chaos engineering
  - AKS cluster, 249–261
  - Chaos Monkey, 225, 226
  - CPU load testing, 224
  - definition, 224
  - design proactive, 227
  - disadvantages, 227
  - Netflix, 226
  - principles, 226, 227
  - studio
    - architecture, 228
    - definition, 228
    - target section, 230
    - VM onboarding process, 230
  - traditional testing, 226
  - virtual machine process
    - access control, 243
    - agent-based targets page, 236, 238
    - agent targets, 231, 232
    - application insights, 234–236
    - CPU experiment, 245
    - CPU pressure fault, 241
    - details, 246, 249
    - experiment designer, 239, 240
    - fault simulation, 241

## INDEX

### Chaos engineering (*cont.*)

- manage actions, 238, 239
- managed identity resource, 233
- members selection, 244
- menu option, 230, 231
- metric explorer, 248
- notification pane, 238
- reader role, 243, 244
- review/creation, 242
- targets, 236
- testing process, 248
- UbuntuVM1 blade, 234
- user-managed identity, 232, 233
- VM overview tab, 247

### Configuration as Code (CaC), 56, 59, 104

### Continuous integration/continuous

- delivery (CI/CD), 56
- automation (*see* DevOps)
- deployment, 94, 95
- version control/CI, 93, 94

## D

### Deployment strategies

- A/B testing, 127
- app configuration, 125
- blue-green, 123, 124
- canary, 126
- dark launching, 127
- definition, 122
- feature flags, 124, 125
- GitHub actions
  - access control, 131
  - app configuration, 129, 130, 141
  - blue-green, 139
  - CD workflow, 134–139
  - CI workflow, 131–133
  - container applications, 128, 139

### demo architecture, 127, 128

- jobs, 134
- key vault, 129, 130
- policies, 131
- principals, 130
- trigger/variables, 134
- ring-based deployment, 126
- rolling deployment, 122, 123

### Desired State Configuration (DSC), 104–106

### DevOps, 5–8

- code configuration, 104–106
- continuous integration, 93, 94
- definition, 93
- delivery/deployment, 94, 95
- demo project
  - ADO environments, 112, 113
  - Cloud Shell, 110
  - component, 110
  - pipeline stages, 113, 114
  - properties, 110
  - resource group, 118
  - service connection, 111
  - variable groups, 112
  - WebApp deployment, 119
  - YAML pipeline, 110, 113–118
- deployment (*see* Deployment strategies)
- GitHub actions, 119–122
- Infrastructure as Code (IaC), 98–103
- monitoring/operation, 97
- monitoring strategy, 188
- pipelines
  - agent types, 106, 107
  - artifacts, 108
  - authenticated connections, 109
  - checks and approvals, 109
  - components, 106–119



- generic components, 108
  - pricing model, 109
  - stages and jobs, 108
  - structure, 107
  - system variables, 108
  - triggers, 108
    - YAML and classic types, 107
  - rugged DevOps/DevSecOps, 96–98
  - secure, 97
  - shift-left testing, 96
  - static analyzers, 97
  - thread modelling tool, 97
  - WAF reliability, 56, 57
- E**
- Effective availability, 23
- F**
- Fault injection testing, 61
- G, H**
- GitHub
  - monitoring
    - insights service, 190
    - organization, 191
    - status page, 189, 190
- Grafana, 55, 154, 158–160
- I, J**
- Incident response (IR)
  - blameless postmortems
    - automation practices, 218
    - benefits, 220
    - ideas and cultural shift, 219
    - learning opportunity, 218
    - organizational culture, 219
    - postmortem process, 221
  - ChatOps, 203, 204
  - communication, 202, 203
  - definition, 193, 194
  - demo reports
    - add-on, 215
    - alert criterion, 215
    - architecture, 214
    - channel and posted messages, 217
    - cognitive services, 215
    - historical data, 216
    - logic app, 216
    - shifts, 218
  - detection/identification, 195
  - eradication/remediation
    - application insights, 208
    - capabilities, 208
    - communication, 205
    - empty workbook, 207
    - insights reports, 208
    - monitor logs, 206
    - smart detection, 210
    - subphases, 204
    - usage reports, 210
    - workbooks, 206
  - frameworks, 194
  - life cycle, 195
  - objectives, 194
  - performance, measurement
    - detection/notification, 211
    - GitHub integration, 213
    - mean time to detect (MTTD), 211
    - metrics, 211
    - Objective and Key Results, 212
    - Power BI, 212, 214
  - postincident/learning/analysis, 195
  - preparation/readiness, 195

## INDEX

### Incident response (IR) (*cont.*)

- resolving phase, 195
- response process
  - commander/manager, 197
  - on-call schedule, 198
  - principles, 196
  - roles, 196, 197
  - rotation schedules, 198
  - technical lead/SME, 197
- tracking/detection
  - actions tab, 200
  - alert structure, 199
  - Cosmos DB alert, 200
  - definition, 199
  - metrics, 200
  - phases, 201
  - platform/monitoring tools, 201
  - types, 199

Infrastructure as a Service (IaaS), 42, 51, 54, 69, 75, 82–84

Infrastructure as Code (IaC), 56, 59, 104

- approaches, 98
- ARM template and Bicep, 99–103
- definition, 98–103
- templates, 99

## K

Kusto Query Language (KQL), 161, 163–165, 171, 174, 199, 205

## L

Load/performance testing

- advantages, 262
- container apps, 262, 270
- definition, 261
- details, 264
- engine instances, 267

execution report, 269  
HTTP scale rule, 263  
metrics, 268  
monitoring, 267, 268  
parameters, 266  
quick test creation, 263  
tess criteria, 267  
test plan, 265

Load testing, 60, 61, 261

Local Configuration Manager (LCM), 104

## M, N

Mean Time Between Failure (MTBF), 30, 32, 36–39, 211, 212

causes and consequences, 38, 39

Cisco field notice, 36, 37

definition, 36

formula, 36

server hardware, 37

unavailability metrics, 39

Mean Time to Failure (MTTF), 30–33, 39, 75, 211

Mean Time to Repair (MTTR), 33–35, 211, 212

Monitoring strategy

activity logs, 153

alerts

condition clears, 181

definition, 181

elements, 178

proactive approach, 178

processing rule, 181

signals, 178

static/dynamic process, 179

application insights

availability test, 172, 173

configuration, 176–178

- connection strings, 169
- customization, 176
- definition, 167
- e-commerce website/banking system, 176
- failures/performance tabs, 173
- features, 169
- instrumentation options/setup, 168–170
- live metrics stream, 171
- logs service, 174
- map, 169, 170
- properties/metrics, 176
- release annotations, 177
- smart detection, 170
- TelemetryInitializer, 177
- transaction search, 172, 173
- troubleshooting guides, 174
- usage section, 174, 175
- data sources, 151–153
- definition, 144
- demo files
  - application insights, 182
  - application map, 184
  - burn rate, 186, 187
  - definition, 184
  - error budget, 185, 186
  - Kusto query, 185
  - solution, 182
  - transaction details, 183
- DevOps, 187–189
- diagnostic setting, 152
- export subscription, 152
- GitHub, 189–191
- log analytics/application insights
  - concepts, 160–162
  - dashboard, 167
  - Kusto, 164, 165

- resource graph, 165–167
- storage account, 162
- user interface, 164
- workspace resource, 162–164
- logs/traces collection, 150
- observability, 148
- operational process
  - aspects, 145
  - error budget/burn rate, 147
  - hierarchy, 144
  - service health, 148–150
  - SLI/SLO/SLA, 145, 146
  - status website, 150
- overview, 143
- visualization, 153
  - dashboards, 153, 154
  - Grafana, 158, 159
  - insights section, 156, 157
  - metrics, 154, 155
  - Power BI, 160
  - storage account resource, 157
  - workbooks, 155, 156

## O

- Objective and Key Results (OKRs), 212
- Observability, 25, 39, 49, 54, 55, 57, 148, 150, 158
- OpenTelemetry, 168
- Open Web Application Security Project (OWASP), 97

## P, Q

- Performance testing, 61, 94, 261, *See also* Load/performance testing
- Platform as a Service (PaaS), 29, 42, 51, 54, 75, 79, 84–86

## INDEX

Policy guest configuration, 106  
Postmortems, 39–40, 196, 205, 220, 221

## R

Recovery Point Objective (RPO),  
    20, 62, 75, 85  
Recovery Time Objective (RTO), 20, 34,  
    62, 75, 82, 83, 85  
Reliability, 3, 18, 22–24, 41, 46–48, 70, 92,  
    97, 145, 146  
Resiliency, 49, 59, *See also* Testing  
    application  
    architecture, 70, 82  
    autoscaling, 78, 79  
    availability set, 72, 73  
    avoid affinity, 78  
    capacity planning, 70  
    components/services, 78  
    design application, 76–78  
    endpoint monitoring, 77  
    features, 69  
    IaaS-based scenario, 83, 84  
    idempotent task, 77  
    load balancers, 79–81  
    microservices, 86–88  
    PaaS architecture, 84–86  
    platform, 71, 72  
    portal helper, 81  
    queue/publisher/subscriber  
        pattern, 77  
    recovery metrics, 75  
    region pairs, 73  
    reliable service, 69  
    replication/redundancy, 82  
    requirements, 75–77  
    retry/circuit breaker pattern, 76  
    shared responsibility model, 75, 76

temporary/large-scale failures, 70  
testing, 89  
throttling, 77  
update/fault domains, 71, 72  
zone redundancy, 73

Ring-based deployment, 60, 126

Risk assessment  
    aggregate, 20  
    availability target, 19  
    definitions, 18  
    e-commerce business, 18  
    error budget, 22, 23  
    nonfunctional requirements, 20  
    reliability, 18  
    SRE metrics *vs.* developer  
        metrics, 22  
    tolerance level, 20, 21

Role-Based Access Control (RBAC),  
    154, 161, 243

Rolling deployment, 122, 123

## S

Self-healing services, 57  
Serverless resilient architecture, 86  
Service-Level Agreement (SLA), 7, 24,  
    53, 54, 146  
    components, 28  
    downtime, 27  
    load balance, 29  
    metrics, 29  
    queue, 28, 29  
    WebApp and MySQL, 28  
    website, 26–28  
    workload, 30  
Service-Level Indicator (SLI), 24–28, 74,  
    145–147, 184  
Service-Level Management

- blameless postmortems, 42
- capacity planning, 42
- definitions and acronyms, 17
- development process/user
  - experience, 42
- glossary, 18
- hierarchy, 41–43
- metrics, 24
  - contractual agreement, 26–28
  - indicator, 25, 26
  - objectives, 25, 26
- monitoring, 41, 43
- postmortems, 39, 40
- reliability, 23, 24
- risk assessment (*see* Risk assessment)
- test/release processes, 42
- toil, 40
- unavailability metrics
  - acronyms, 30
  - MTBF, 36–39
  - MTTE, 31–33
  - MTTR, 33–35
- Service-Level Objectives (SLOs), 22, 25, 26, 146
- Shift-left testing, 59, 91, 96
- Site Reliability Engineering (SRE)
  - automation, 7, 92, 93
  - challenges, 9
  - definition, 3
  - DevOps, 3–6
  - engineering process, 8
  - Google guidance, 8
  - history, 1
  - implementation, 9
  - job role perspective, 12–14
  - monitoring service, 11
  - operational tasks, 2
  - organization, 10

- overview, 1
- pathological, bureaucratic and
  - generative, 5
- service levels, 7
- westrum typology, 6
- workloads information, 9
- Software Composition Analyzers
  - (SCA), 97
- SQL Database, 51, 53, 75, 169
- Stress testing, 61, 261
- Subject matter expert (SME), 197, 201

## T, U

- Teasing misconception, 40
- Testing application
  - blue/green deployment, 60
  - Chaos/Load testing, 60
  - definition, 59
  - dev/test environment, 59
  - disaster/recovery, 61
  - fault injection, 61
  - performance testing, 61
  - reliability tests, 60
  - ring-based deployment, 60
  - shift left testing, 59

## V

- Virtual machines (VMs), 54–56, 61, 72, 73, 75, 82–84, 87, 104–106, 122, 177, 207, 227, 230–249

## W, X, Y, Z

- WebApp/App Service, 28, 29, 84, 85, 154, 155
- Web Application Firewall (WAF), 86

## INDEX

### Well-Architected Framework (WAF)

- assessments, 62, 63
  - descriptive name, 63
  - evaluation options, 64
  - questionnaire, 65
  - recommended actions, 67
  - results, 65–67
  - workload type, 64
- building blocks, 47
- business workloads, 46
- checklist, 58
- factors, 48
- learning process, 45
- linear flow, 47
- objectives, 46–48
- POC environment, 48

### reliability

- building block, 49
- cloud environments, 51
- concepts, 49
- data replication, 53
- DevOps/automation, 56, 57
- disaster recovery solutions, 51
- high-availability, 52–54
- hypervisor solutions, 50
- observability, 54, 55
- on-premises data centers, 50–52
- self-remediation, 57
- requirements, 58, 59
- service provider, 49
- technical details, 45
- testing application, 59–62