1ST EDITION

Building Resilient Architectures on AWS

A practical guide to architecting cost-efficient, resilient solutions in AWS

AJIT PUTHIYAVETTLE
IMAYA KUMAR JAGANNATHAN
RODRIGUE KOFFI

Building Resilient Architectures on AWS

A practical guide to architecting cost-efficient, resilient solutions in AWS

Ajit Puthiyavettle

Imaya Kumar Jagannathan

Rodrigue Koffi

Building Resilient Architectures on AWS

The authors acknowledges the use of cutting-edge AI, such as Large Language Models, with the sole aim to do research and enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for readers. It's important to note that the content and ideas has been crafted by the authors and edited by a professional publishing team.

Group Product Manager: Preet Ahuja

Publishing Product Manager: Suwarna Patil

Book Project Manager: Ashwin Kharwa

Senior Editor: Mohd Hammad

Technical Editor: Irfa Ansari

Copy Editor: Safis Editing

Proofreader: Mohd Hammad

Indexer: Rekha Nair

Production Designer: Gokul Raj S.T

DevRel Marketing Coordinator: Rohan Dobhal

First published: December 2024

Production reference: 1081124

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

ISBN 978-1-83588-710-3

www.packtpub.com

To my father, Narayanan, and my mother, Chandramathi, for their boundless love, sacrifices, and embodiment of resilience that have forever shaped my life. To my beloved wife, Deepa, my steadfast partner, whose unwavering support and companionship have been the cornerstone of our shared journey. To my treasured children, Ayush and Tejas, the lights of my life, whose presence fills my heart with immeasurable joy and pride.

– Ajit Puthiyavettle

To my family - Manimekala, my dear wife, and our lovely children, Kavin and Kayal, who found a unique way to motivate me by regularly asking with curiosity and pride, "Are you there yet?" My fur baby, DJ, for his licks, walks, and pure love. Appa and Amma, for teaching me how to be tenacious and kind when the going gets tough. My brother, sister-in-law, and their two little boys, who gave me great support by adding to the fun each time they visited. This journey would not have been the same without your collective love and encouragement.

– Imaya Kumar Jagannathan

To my late mother, who left this world too soon, but whose love and invaluable life lessons remain engraved in my heart forever. To my father, for all his sacrifices and for being a true definition of resilience. To my sweet Catherine, my soulmate and life partner, who always believes in me and supports me. To my daughter, Alma, my priceless treasure who reminds me of the true essence of joy. To the Anéyé family, whose sheltering, love, and support have been an anchor through my life path. To my dear "Tantie" Fanda, for being like a mother to me.

–Rodrigue Koffi

Contributors

About the authors

Ajit Puthiyavettle is an accomplished Principal Solutions Architect with a proven track record of over a decade in architecting innovative solutions that drive business success. His expertise lies in collaborating closely with enterprise clients across diverse sectors, including financial services, healthcare, and life sciences, crafting tailored solutions that align with their strategic objectives. His deep understanding of industry trends, coupled with his technical acumen, allows him to architect solutions that not only meet current requirements but also anticipate future needs. He has presented at various public conferences, including AWS re:Invent, AWS re:Inforce, and AWS Summits, as well as having authored various blogs, workshops, videos, and so on.

Imaya Kumar Jagannathan is an expert architect, technical leader, speaker, and author with over 22 years of rich experience in the technology industry, specializing in designing and building complex, internet-scale applications. He has worked in highly impactful roles at various large organizations and is passionate about creating solutions that prioritize customers and make a meaningful difference to businesses. He is passionate about coaching and mentoring others and finds happiness in witnessing their success. He has presented at several top conferences, including AWS re:Invent, and has authored dozens of articles, blogs, videos, and live events. When not busy at work, he spends time learning about aviation.

Rodrigue Koffi is a technical leader, public speaker, and software engineer at heart with over a decade of experience. He is an author of multiple articles and whitepapers, and a speaker at top industry conferences. Currently working at AWS, he helps customers across multiple industries achieve their resilience goals through observability. He has held various roles, from software engineering to consulting and leading site reliability engineering teams. Rodrigue is passionate about high-scale and distributed systems, continuously expanding his knowledge in these domains. Outside of work, he finds joy in swimming and spending quality time with his family.

About the reviewers

Divyajeet Singh is a seasoned professional in the cloud computing space with extensive experience across major tech giants. Currently serving as a senior solutions architect at AWS, he has previously held similar roles at Google and Microsoft, solidifying his expertise in cloud technologies. Divyajeet thrives on challenges, viewing them as opportunities for growth and learning.

Beyond his professional pursuits, Divyajeet nurtures a keen interest in CAD and CAM. He finds joy in bringing ideas to life through CNC machines during his spare time, showcasing his passion for innovation beyond the digital realm. Divyajeet values work-life balance, enjoying travel and quality time with family and friends.

I extend my heartfelt gratitude to my parents, my wife, and my 5-year-old son, for their unwavering support. I also want to express my appreciation to my mentors and colleagues for the invaluable insights and collaboration throughout my professional journey.

Pranit Raje is a Solutions Architect with AWS India, bringing over six years of experience across various AWS teams. He excels in DevOps, automation, containers, and operational excellence through IaC, CI/CD, and DevSecOps practices. Pranit has played a key role in proposing, advising, consulting, and delivering critical technical solutions for diverse customers. He has authored blogs, technical articles, and workshops, both internally and externally. In his spare time, Pranit enjoys networking with like-minded professionals through technical conferences and meetups.

I am grateful to my family, friends, colleagues, and managers for their constant support and the valuable lessons I've learned from them. I feel fortunate to work in this field alongside such helpful individuals who challenge me to improve and keep learning. Special thanks to my family for their unwavering support and patience with my busy schedule.

Table of Contents

9

Using Containers to Improve Resiliency 171

10

Resilient Architectures Across Regions 191

Part 3: Validating Your Architecture for Resiliency

11

Examples of Resilient Architecture 211

12

Observability, Auditing, and Continuous Improvement 229

13

Performing Chaos Engineering Testing 249

14

Disaster Recovery Planning and Testing 267

15

Finalize Building Resilient Architecture Using AWS Resilience Services 283

Preface

Cloud resilience is a critical aspect of modern IT infrastructure, referring to a system's ability to withstand, adapt to, and rapidly recover from disruptions while maintaining continuous operations. In today's digital landscape, where businesses rely heavily on cloud-based services, ensuring resilience is paramount to safeguarding against potential losses in revenue, productivity, and reputation.

Amazon Web Services (**AWS**) has established itself as a leading cloud service provider, offering a highly resilient infrastructure that sets the industry standard. AWS's approach to resilience is multifaceted, encompassing both the physical infrastructure and the services it provides.

At the core of AWS's resilient architecture is its global network of data centers, strategically located in multiple geographic regions worldwide. Each region is further divided into **Availability Zones** (**AZs**), which are physically separate data centers with independent power, cooling, and networking. This design inherently provides redundancy and fault tolerance, allowing applications to remain operational even if one or more AZs experience issues.

AWS's infrastructure is built with redundancy at every level, from networking equipment to storage systems. The systems are designed to automatically detect failures and initiate recovery processes, often without any manual intervention. This self-healing capability minimizes downtime and ensures high availability for customer applications.

Beyond the physical infrastructure, AWS offers a comprehensive suite of services and tools specifically designed to enhance resilience. For instance, AWS Resilience Hub helps customers assess and improve their application resilience by providing recommendations based on AWS best practices. AWS Fault Injection Simulator allows organizations to perform controlled chaos engineering experiments, helping them identify and address potential weaknesses in their systems before they manifest in production.

AWS also provides robust data replication and backup services, enabling customers to implement comprehensive disaster recovery strategies. Services such as Amazon S3 offer 99.999999999% durability, ensuring data remains safe and accessible even in the face of multiple simultaneous failures.

Furthermore, AWS's commitment to continuous improvement and innovation means it is constantly enhancing its resilience capabilities. It regularly publishes detailed post-mortems of any service disruptions, demonstrating transparency and a commitment to learning from incidents.

By leveraging AWS's resilient infrastructure and services, organizations can build applications that not only withstand failures but also adapt and scale in response to changing conditions. This level of resilience is crucial in today's fast-paced, always-on digital economy, where even brief outages can have significant consequences.

Who this book is for

This book is for cloud architects, developers, DevOps/SRE engineers, and executive decision makers, or in fact for anyone who has any capacity to influence decision making with respect to building resilient applications on AWS.

Specifically, these are the personas that this book targets:

- **Cloud architects**: Deep technical experts who make important decisions when it comes to designing application and infrastructure architecture, this book will help them learn about the rich services and features on AWS that they can leverage to put out a fault-tolerant system.
- **Developers and DevOps/SRE**: These personas will learn why it is important for them to focus on writing efficient application code that takes advantage of the reliable infrastructure offered by AWS, and how they can build automation for continuous management and monitoring of their application performance and health.
- **Executive decision makers**: You will learn about the value of leveraging an advanced, reliable infrastructure that offers limitless possibilities to build workloads that allow your teams to iterate quickly and add customer value in a highly competitive business environment.

What this book covers

Chapter 1, *Understanding Resilience Concepts*, introduces the concept of resilience by drawing parallels with the aviation industry. It covers achieving resilient architecture using AWS infrastructure, fault-tolerant design best practices, and the shared responsibility model. You'll learn about potential failure points and understand why maintaining resilience is an ongoing process crucial for a robust infrastructure.

Chapter 2, *Implementing Resilient Compute and Auto Scaling*, covers resilient compute and auto scaling solutions on AWS, focusing on failure-resistant system design, redundancy, and fault tolerance. It explores AWS Auto Scaling, cost-saving strategies, and the importance of monitoring. Key topics include multi-Availability Zone deployments, stateless architectures, and extending resilience to containers and serverless architectures.

Chapter 3, *Securing and Backing Up Critical Data*, covers data security and resilience strategies on AWS. It explores access control, layered backup strategies, multi-Region models for improved availability, automated recovery mechanisms, and disaster recovery best practices. You'll learn how to design highly resilient and available systems using various AWS services.

Chapter 4, *Orchestrating Graceful Degradation*, explores the design principle of graceful degradation, explaining why it is critical to prevent your systems from facing catastrophic failures. You will also learn about different strategies to contain outages and how you can streamline the recovery process efficiently.

Chapter 5, *Exploring the AWS Shared Responsibility Model*, provides an introduction to what shared responsibilities between AWS and customers look like and the different roles these parties play in designing and operating a resilient infrastructure.

Chapter 6, *Learning AWS Well-Architected Principles for Resiliency*, explores the critical pillars of the AWS Well-Architected Framework through the lens of building resilient cloud solutions. This includes the operational excellence, reliability, security, efficiency, and cost optimization pillars. You'll learn how to use AWS services to reduce heavy lifting, automate deployments, improve operational procedures, and secure your applications.

Chapter 7, *Architecting Fault-Tolerant Applications*, discusses architectural patterns and best practices for building fault-tolerant, highly available applications on AWS. You will learn about redundancy, loose coupling, graceful degradation, fault isolation concepts, and how important it is to build the right architecture to take the best from what AWS offers.

Chapter 8, *Resiliency Considerations for Serverless Applications*, helps you understand the advantages and strategies for building serverless-based applications. The chapter covers their impact on improving resilience. You will learn about idempotency, asynchronous transactions, error handling, and testing and deployment strategies.

Chapter 9, *Using Containers to Improve Resiliency*, focuses on ways container-based applications help with greater resilience. In this chapter, you will learn how to build, deploy, and operate containers using AWS services. You will gain an understanding of immutable deployments, scaling and security for containers, and specific considerations compared to traditional virtual machines.

Chapter 10, *Resilient Architectures Across Regions*, gives insights into running applications across multiple regions. It covers active/passive, active/active, and cell-based architectures, with a focus on high availability. You will learn about the pros and cons of each deployment model and what considerations to take for multi-Region deployments.

Chapter 11, *Examples of Resilient Architecture*, delves into architectural patterns for building resilient systems, ensuring reliability and availability through fault-tolerant designs, leveraging practical examples and real-world scenarios.

Chapter 12, *Observability, Auditing, and Continuous Improvement*, focuses on designing for observability, covering essential monitoring and auditing techniques to proactively identify issues and maintain system health for resilient applications.

Chapter 13, *Performing Chaos Engineering Testing*, explores chaos engineering principles, introducing controlled fault injection tests to proactively identify vulnerabilities and validate resilience mechanisms, enabling more robust system development.

Chapter 14, *Disaster Recovery Planning and Testing*, focuses on **Disaster Recovery Planning** (**DRP**), outlining procedures for crafting an effective plan, incorporating testing strategies to identify vulnerabilities, and enabling organizations to recover quickly and maintain business continuity during disruptive events.

Chapter 15, *Finalize Building Resilient Architecture Using AWS Resilience Services*, delves into AWS Cloud Resilience, exploring tools and capabilities for fault injection, chaos engineering, disaster recovery, and backup solutions to build highly reliable and available applications on the AWS cloud.

To get the most out of this book

Some chapters invite you to try hands-on instructions. These chapters will have a *Technical requirements* section for more details about the prerequisites. The main requirement is having access to a non-production AWS account to follow along.

Software/hardware covered in the book	Operating system requirements
AWS Command Line Interface (AWS CLI)	Windows, macOS, or Linux
Git	

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

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Building-Resilient-Architectures-on-AWS`. If there's an update to the code, it will be updated in the GitHub repository.

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

Conventions used

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

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In this example, `RedrivePolicy` specifies that messages that fail to be processed after three attempts (`maxReceiveCount: 3`) will be moved to the `my-dead-letter-queue` SQS queue."

A block of code is set as follows:

```
  DeadLetterQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: my-dead-letter-queue
```

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

```
$ aws arc-zonal-shift start-zonal-shift \
--resource-identifier arn:aws:elasticloadbalacing:...
--away-from euw1-az2
--comment "possible issue isolated to AZ2"
--expires-in 12h
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Under **Metrics**, select the **RDS** namespace."

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

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Share Your Thoughts

Once you've read *Building Resilient Architectures on AWS*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

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

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

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

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

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

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83588-710-3`

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

Part 1: Setting the Stage – Learning the Basics of Designing Resilient Architectures

This part emphasizes the importance of resilient architecture design and explores different areas for architects to pay attention to. This part also discusses the responsibilities that the user and AWS share in designing and maintaining resilient environments.

This part has the following chapters:

- *Chapter 1*, *Understanding Resilience Concepts*
- *Chapter 2*, *Implementing Resilient Compute and Auto Scaling*
- *Chapter 3*, *Securing and Backing Up Critical Data*
- *Chapter 4*, *Orchestrating Graceful Degradation*
- *Chapter 5*, *Exploring the AWS Shared Responsibility Model*

1
Understanding Resilience Concepts

Being able to build a fully resilient application environment is a daunting task for many of us. Simply defining what resilience even means varies based on several factors. For example, simply being able to quickly reboot a standalone server when an application is running out of memory might be considered as having a resilient environment if that is what you are looking for. However, when the term resilient is brought up, the typical expectation is that we are aiming to build an architecture that is resilient to all internal and external influencing factors and will exhibit high availability, fault tolerance, and scalability under all circumstances.

In this book, we will help provide clarity about resiliency, dive deep into what resilient architecture looks like, and explore the measures you can take to build a resilient design that serves the resilience goals you have set for your applications. The book will primarily focus on services and features available on **Amazon Web Services** (**AWS**) and will also leverage best-practice guidelines that AWS has put together for customers in this regard.

In this chapter, we will first try to understand what resilience means. We will attempt to explain the concept through the eyes of the commercial aviation industry where resilience is of utmost importance in every single aspect of the design for building aircraft.

In this chapter, we will cover the following topics:

- Demystifying resilience
- Facing the cloud's storm
- The continuous resilience journey

Demystifying resilience

Resilient infrastructure architecture is a design that can withstand and recover from disruptions quickly. It is built on the principle of redundancy, with multiple layers of protection that can be activated in the event of a failure. This approach helps to ensure that critical systems are always available, even in the face of major disruptions.

One of the famous quotes from Werner Vogels (CTO, Amazon) is –

"Everything fails all the time."

With time, failure is imminent. No matter how well you design your infrastructure, something is going to give up and fail at some point. The idea behind resiliency is to be intentional about understanding this reality and building systems that are able to recover from failures quickly without causing catastrophic business disruptions that cause huge financial impact.

The concept of resiliency is best understood when we examine the commercial aviation industry. Almost all parts and functions of a commercial aircraft have multiple redundant parts and methods of operation with isolation and fault tolerance built in. Whether it is hydraulics, avionics, power generation, fuel control, or the physical strengthening of the fuselage, multiple layers of redundancies are in place to prevent single-point failures from causing catastrophic accidents.

Modern aircraft are very big and they heavily rely on hydraulic systems to operate different mechanical parts of the plane, such as the ailerons, elevator, rudder, slats, and flaps that are required to control the aircraft's balance, elevation, yaw, lift, and so on. They also have electronic flight instrumentation systems, which are part of the critical components in flying commercial aircraft. These critical systems are mainly powered using the plane's engines and the expectation is that there will never be a situation where these systems do not receive the required electrical power to function.

Aircraft generally have a primary and secondary mechanism, as well as a backup mechanism for all functionalities put in place that is only used in situations where both the primary and secondary mechanisms have failed to provide the required functionality.

One of the most famous incidents in aviation is the *Gimli Glider* story (`https://simpleflying.com/gimli-glider/`). On July 23, 1983, a Boeing 767 ran out of fuel while flying over Canada. The pilots were able to glide the plane to an emergency landing at Gimli, a former military base. 35,000 ft, while cruising at an altitude of 35,000 ft, ran out of fuel completely due to a misunderstanding on how to calculate fuel for the aircraft at a time when Canada was converting from the imperial system to the metric system.

The pilot used the old imperial system and filled far less fuel than what was necessary to complete the flight. As the plane ran out of fuel completely, the engines, which are the primary source of power, shut down, along with the secondary **Alternate Power Unit** (**APU**), which also did not function. As this happened, the plane's electronic flight instrumentation system did not work due to lack of power, and the pilots were left with only a few basic battery-powered emergency flight instruments. The 767 was

the first twin-engine wide-bodied plane built by Boeing, and due to its large control surface, it was set up with hydraulic systems to fly the plane because simply using muscle power alone is not feasible to operate a jumbo jet of that size. Without power, the hydraulic systems were not operational as well, which meant that the pilots had to resort to the secondary option of using muscle power to control the plane unless they somehow found a way to power the hydraulic systems.

Therefore, the pilots deployed the backup power source called the **Ram Air Turbine** (**RAT**), which is a small wind turbine fan that can generate power from the airstream generated through the speed of the aircraft. With the help of the power generated from the backup source, the pilots were able to use the hydraulics and eventually glide the plane to successfully land on an abandoned runway at Gimli air station used by the **Royal Canadian Air Force** (**RCAF**) in Manitoba without any lives being lost. The aircraft went through some non-critical damages in the frontal portion of the fuselage due to the nose wheel not locking in position on touchdown.

The aircraft was subsequently repaired and placed back into service until its retirement in 2008. This example illustrates how redundant, fault-tolerant systems can help mitigate catastrophic disasters by providing alternative options when challenging situations arise.

In this section, we will talk about what cloud resilience is and how you can go about building resilient architectures in the cloud. We will explore a variety of technical areas to consider, learn how we can work coherently with the **Cloud Service Providers** (**CSPs**), and leverage the underlying infrastructure in building resilient systems.

Cloud resilience

Cloud resilience, a critical facet of contemporary cloud computing architectures, is critical in ensuring the reliability and availability of cloud-based applications and services. In cloud computing, resilience denotes the capacity of a system to withstand and recuperate from failures, disruptions, or unforeseen events without compromising its functionality or performance.

Mission-critical systems are required to be resilient to both external and internal factors. Just hosting your applications on a well-established CSP does not automatically provide resiliency. It is essential to plan for different aspects of infrastructure and software design to ensure your application can withstand unexpected turmoil and disruptions.

In traditional infrastructure, such as self-hosted server environments, private data centers, or colocation hosting environments, you were fully responsible for building resilience at all layers of the infrastructure. This includes procuring the right hardware; ensuring the availability of an uninterrupted power supply, sufficient cooling mechanisms, reliable high-speed network connectivity, strong physical security; and continuous monitoring of hardware failures and replacements.

In public cloud environments, there is a great level of resilience built in by default simply because public CSPs such as AWS, Microsoft Azure, and Google Cloud Platform operate at a very large scale to support a highly efficient multi-tenant environment that allows them to provide infrastructure services to host customers that operate workloads at internet scale for a variety of use cases, such as financial services, healthcare, media streaming, and artificial intelligence.

These companies employ highly skilled technical personnel, perform regular maintenance of equipment, replenish hardware frequently, are able to procure high-quality materials and parts and secure the premises with world-class security protocols, which can be unattainable for companies whose core business is not in operating a modern and efficient large-scale data center. Cloud services differ in the mechanisms they provide for handling resiliency. Customers play an active role in establishing resiliency for the workloads they host on the cloud. While each cloud operator has their own methodology for this, AWS uses a **shared responsibility model**. We will mainly focus on AWS's approach to be in line with the goals of the book.

Shared Responsibility Model

In cloud environments, resilience is a shared responsibility between the service provider and the customer using the environment.

The CSP is responsible for the resilience of the underlying hardware, network equipment power supply, network, air conditioning, physical security, and so on; the customer is responsible for designing the application architecture in such a way that their application is resilient to other unforeseen disruptions, such as a cyberattack affecting the application performance, database performance and stability issues, API design failures due to load-balancing scenarios, sudden increase in traffic requests, or dealing with failures occurring in the underlying infrastructure that were not mitigated by the hedges put forth by the CSP.

The following diagram shows the different responsibility areas that the customer and the CSP own:

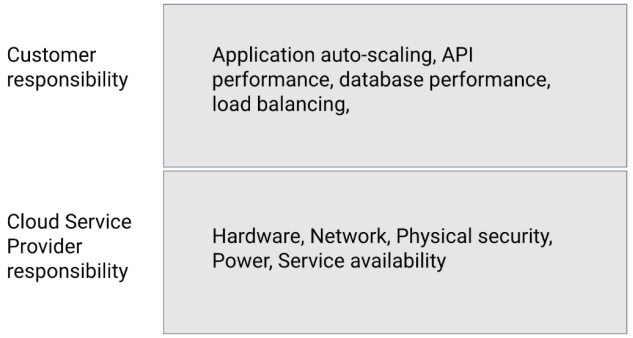

Figure 1.1 – Customer and CSP responsibility matrix

It is important to note that the line between the CSP and customer responsibilities moves depending on the service. For fully managed services, which are also called **Software as a Service** (**SaaS**), scaling and performance fall into the responsibility of the CSP, and for self-managed services that are hosted on **Infrastructure as a Service** (**IaaS**) environments, scaling and performance fall into the responsibility of the customer. You will learn about the customer versus CSP responsibilities in detail in *Chapter 5*. In order to guide customers to follow best practices in using the cloud platform,

AWS has devised a robust framework called the **AWS Well-Architected Framework**, which covers a range of topics that offer prescriptive guidance. The AWS Well-Architected Framework describes key concepts, design principles, and architectural best practices for designing and running workloads in the cloud. It provides a consistent approach to evaluating and improving your cloud architecture across the following six key pillars:

- **Operational excellence**: `https://docs.aws.amazon.com/wellarchitected/latest/framework/operational-excellence.html`
- **Security**: `https://docs.aws.amazon.com/wellarchitected/latest/framework/security.html`
- **Reliability**: `https://docs.aws.amazon.com/wellarchitected/latest/framework/reliability.html`
- **Performance efficiency**: `https://docs.aws.amazon.com/wellarchitected/latest/framework/performance-efficiency.html`
- **Cost optimization**: `https://docs.aws.amazon.com/wellarchitected/latest/framework/cost-optimization.html`
- **Sustainability**: `https://docs.aws.amazon.com/wellarchitected/latest/framework/sustainability.html`

While all six pillars are critical and need to be given detailed consideration when designing applications, the **Reliability** and **Operational excellence** pillars cover plenty of ground specifically about resiliency. One of the important takeaways is that while AWS is responsible for the resiliency *of* the cloud, the customer is responsible for resiliency *in* the cloud. What this means is that if you are using AWS to run your workload, it is your responsibility to ensure that the design put forth has the necessary redundancy and guardrails in place to be resilient, guided by the Well-Architected Framework.

Why resiliency?

Do we really need to build resilient applications? What exactly are we trying to achieve by building highly resilient infrastructure and applications? These are very important questions that need to be answered well before you sit down and design an application architecture.

The reality is that you may not really need to build an application that is always available and scales infinitely. While we will discuss this topic in detail in the next chapter, I want you to understand that adding redundancy will incur higher costs. There are no two ways about this, as compute, storage, and network contribute to additional costs based on usage. The question is, what trade-offs are you ready to make?

Do you want to reduce your costs in the short term, risk customer satisfaction, or take a long-term vision and prioritize customer experience? The answers will vary based on the application's business use case in question.

In today's environment, whether you are building an internal-facing business application for a large enterprise or you work at a start-up that is building a new mobile app for a young audience using cutting-edge technologies, the expectations are the same in terms of application performance, usability, availability, and the experience. End users are used to their experiences on their mobile devices, which typically provide them with instant responses, pretty graphics, and overall delight. When the same users log on to their business applications, they have the same expectations as their personal experience. An unreliable application will simply not sell and will result in poor customer satisfaction rates, business loss, and reduced user productivity, which can impact business revenue.

At this point, it is a good idea to summarize what we have learned so far. We now have an idea of what cloud resilience means, and through the AWS Shared Responsibility Model, we understood what areas users of the AWS cloud platform have to build resilient workloads. We also got insights into how redundancy plays an important role in improving resiliency for workloads.

In the following section, we will look at other foundational principles that are critical to establishing and maintaining a resilient infrastructure.

Resilient foundations

AWS offers a variety of levers for you to pull in order to design resilient applications. In fact, building a *fully resilient* application that can withstand all imaginable disaster scenarios is somewhat unattainable in most cases. The emphasis should be on building strong mitigation tactics that will help with recovering from disasters quickly.

The following diagram shows the **Swiss cheese model**. This is a well-known model that demonstrates that there can be unique, unknown situations that can pass through a series of barriers that were put in place to prevent them from occurring.

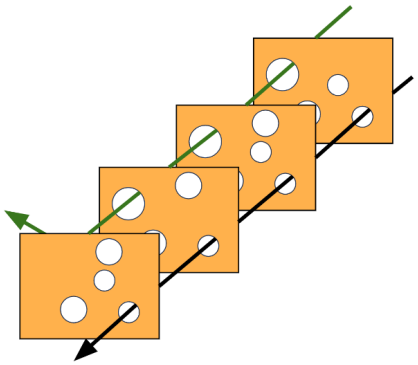

Figure 1.2 – Swiss cheese model showing the possibility of a disaster striking despite taking preventive measures

Resilient architectures should consider the following important factors:

- **Redundancy**: This is about provisioning parallel infrastructure in addition to what is minimally required so that a single point of failure is avoided.
- **Fault tolerance**: This is the ability of a system to recover from a system failure. This is commonly achieved through system mirroring, application logic, and configuration.
- **Isolation**: This is about choosing to build the application on physically isolated infrastructure in order to prevent an issue from cascading into affecting the entire application.
- **Automation**: This is the most important piece of the resiliency puzzle. Anything that can be automated should be automated. Whether it is infrastructure provisioning, application deployment, or auto scaling, automation is the key to recovering from inevitable disaster situations quickly and getting back to normalcy.
- **Monitoring**: This is about constantly being in the know of system performance in order to identify anomalies and take necessary actions. To build resilient applications, you need to employ a strong monitoring solution combined with a disciplined process.
- **Security**: This is about designing the infrastructure and applications by following industry-standard security best practices to ensure issues such as cyberattacks and code vulnerabilities do not cause issues.

Recovering from disasters quickly is a key element of building resilient architectures. This means capturing key metrics such as the following:

- **Mean Time to Recovery** (**MTTR**), which is a calculation of the amount of time needed to recover from a failure after it happens.
- **Recovery Time Objective** (**RTO**), which is the maximum amount of time that a business can afford to be without a critical service or system before it starts to incur unacceptable losses.
- **Recovery Point Objective** (**RPO**), which is the maximum tolerable amount of data loss that an organization is willing to accept after a disaster.

These metrics are critical to making scientific, informed decisions about building resilient architectures. We will be using the Gimli Glider incident along with aircraft design principles to dive deeper into the preceding factors.

Building redundancy in the infrastructure

As discussed earlier, building redundant architecture does involve higher costs, and the costs can vary depending on the level of resiliency you want to build into the application. For example, most business-critical applications that need to have higher availability can deploy an architecture that incurs multiple redundant layers in order to provide higher **Service-Level Availability** (**SLA**), which is measured in percentages. An SLA is a contract between a service provider and a customer that defines the level of service that the customer can expect. The SLA is calculated as the percentage

of time a service is available to users. It is calculated by taking the total amount of time the service is available and dividing it by the total amount of time the service should have been available. An application that claims to have a 99.99% SLA is expected to only have a downtime of 4 minutes and 23 seconds per month and 52 minutes and 36 seconds per year. Learn more about SLAs and how they are calculated from the official AWS documentation here: `https://aws.amazon.com/what-is/service-level-agreement/`.

In order to ensure you maintain the committed SLA, you need to ensure you continuously track the **Service-Level Indicators** (**SLIs**) that inform you about the SLA performance. SLIs are essentially metrics that measure the performance of a service. They are used to track the reliability, availability, and performance of a service, and to identify areas where improvements can be made.

SLIs are typically defined in terms of the following characteristics:

- **Metric**: The specific metric that is being measured
- **Unit**: The unit of measurement for the metric
- **Threshold**: The acceptable value for the metric
- **Frequency**: The frequency with which the metric is measured

SLIs are used in conjunction with **Service-Level Objectives** (**SLOs**) to define the level of service that a customer can expect. SLOs are specific, measurable objectives that define the expected service level, such as availability, response time, or error rate.

SLOs are typically defined in terms of the following characteristics:

- **Target**: The desired value for the metric
- **Tolerance**: The acceptable range of values for the metric

SLIs and SLOs are used together to create an SLA. Here are some examples of SLIs:

- **Availability**: The percentage of time that a service is available to users
- **Latency**: The average time it takes for a request to be processed by a service
- **Throughput**: The number of requests that a service can process per second
- **Error rate**: The percentage of requests that result in an error

SLIs are an important tool for managing the performance of a service. They can be used to identify areas where improvements can be made, and also to track the progress of those improvements.

Going back to the Gimli Glider example, the only way a large wide-bodied plane such as the Boeing 767 could have landed successfully that way was due to the redundant systems built in. It had several alternate sources to power the airplane's hydraulic systems. The pilots also had access to alternate manual controls to operate the plane even in the absence of the hydraulic systems; it would have been a very tiring process to do so, though.

Using fault tolerance to build resiliency

To enable your systems to handle failures, you need to include mechanisms for the system to recover from issues without much delay. This can either be through directing operations to a redundant system or simply by rebooting automatically without causing cascading effects.

Deploying application servers behind a **load balancer**, creating logic in applications to use alternate paths, and using configurations to change functionality are some of the strategies typically deployed to improve fault tolerance.

You can relate this to how the Boeing 767 plane continued flying despite technical components failing unexpectedly. This is one of the key contributing factors to why the Gimli Glider incident did not cost any lives.

Decoupling systems through isolation principles

One of the critical components of building reliable systems is to identify dependencies between different systems and introduce *guardrails* such that the *blast radius* is as minimal as possible. The **12 Factor App** principles (visit this page to learn all about it: `https://12factor.net/`) put forth by Heroku is a set of ideas that help design microservice-based architectures that encourage isolation as one of the key principles.

Applications that are loosely coupled are not only easy to scale horizontally but also do not cause upstream applications to fail when they are facing downtime. In an ideal world, an application/microservice that fails should not cause any problems to either downstream or upstream systems. Take a look at the following figure. Microservices A and B are deployed in isolated environments where they do not share tight dependencies, causing one's failure to impact the other.

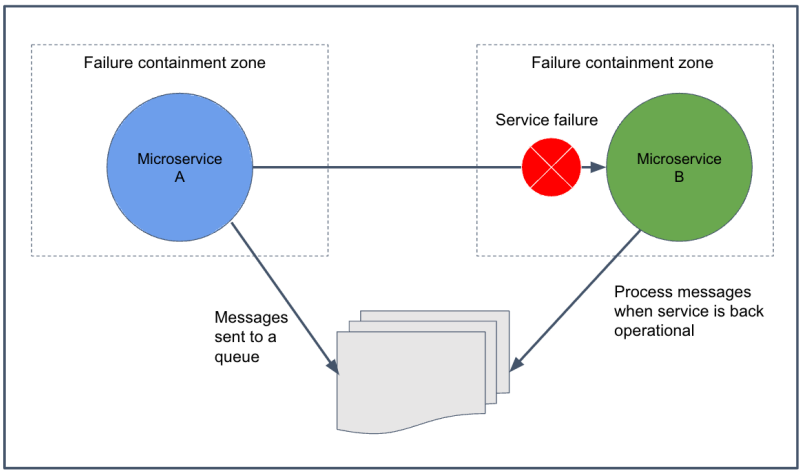

Figure 1.3 – Reduced blast radius due to loosely coupled architecture design pattern

The failure of microservice B is contained within its own operational zone and its outage doesn't impact the functions of microservice A. Microservice A continues to function and sends the messages to an alternate destination sensing microservice B's outage. This exemplifies both isolation and fault tolerance principles in action to provide higher resilience.

Connecting this back to the Gimli Glider incident, commercial airplanes are designed in such a way that a failure in one part of the plane does not cause a cascading effect on other parts, resulting in a total failure. In that plane, despite the engines stopping working due to lack of fuel and the APU failing as well, the RAM turbine system kicked in to provide the required power to key systems, which the pilots used to navigate the plane to a safe landing.

Using automation to improve disaster recovery time

An important part of having a better RTO is automating the entire application development and deployment process, including infrastructure creation, upgrades, and auto scaling. The *shift-left* movement is aimed toward automating everything that can possibly be automated. AWS offers a rich set of options to automate your infrastructure and applications regardless of the compute service being used.

One of the simplest forms of automation you can think about is setting up AWS Auto Scaling, which supports Amazon EC2, Amazon ECS, Amazon DynamoDB, and Amazon Aurora services, as of early 2024. This is an easy-to-use feature that can come in handy to scale your resources on-demand based on specific parameters so your users are not hit with application performance degradation issues.

It is recommended to programmatically configure and deploy infrastructure using AWS **Cloud Development Kit** (**CDK**) or AWS CloudFormation. Using AWS CodePipeline can help you create a **CI/CD** (short for **continuous integration and continuous deployment**) process for infrastructure using Git-based source control and configurable deployment processes. This process is often referred to as **infrastructure as code** (**IaC**).

You can easily relate this to the Gimli Glider incident. In high-stress scenarios, pilots should adhere to the fundamental principles of *aviate, navigate, and communicate*, in that order. The Boeing 767 aircraft did not wait for manual pilot input to deploy the RAM turbine, which provided power to the aircraft. The designers of the aircraft were fully aware that pilots in command could encounter stressful circumstances. Thus, it was critical to implement automation to help alleviate pilot stress and facilitate a timely resolution of the situation. Similarly, as architects and developers of software systems, we should ensure that automation is employed wherever necessary, such as auto scaling, deployment, and alerting, to make managing large-scale systems easy.

Monitoring systems to track system health continuously

Setting up proper monitoring of the environment is highly critical to provide resiliency. In order to address a problem, the first step is to understand what normalcy looks like. Continuously collecting signals such as metrics, logs, and traces from the environment can give you information about the performance and health of the infrastructure and workloads.

Create dashboards for various applications and infrastructure for visualizing the information, while also setting up alerts to notify when something goes wrong. Amazon CloudWatch offers rich dashboarding and alerting features to satisfy these needs. Using CloudWatch Anomaly Detection, you can set alarms to go off only when a specific metric value goes outside a certain range dynamically without you setting a hard limit yourself.

Oftentimes, monitoring is an afterthought and not paid the required attention when new workloads are being designed or when workloads are migrated from one environment to another. This leads to several problems during the operational phase, such as lack of visibility, higher costs, increased application downtime, and overall operational inefficiency.

A monitoring strategy should be set along with the architecture design, considering input from business and technical teams. A committed SLA should be one of the main drivers in making decisions on what signals to collect, what information should be visualized on the dashboard, how alerts should be set up for anomalies, and so on.

Anchoring this back to the Gimli Glider incident, imagine the pilots not having visibility into the key metrics of the plane while trying to fly in this stressful situation. The plane designers designed the systems so that the pilots could get continuous insights into the key details required to aid situational awareness. As soon as power came back on, the cockpit displays were one of the first systems to come to life.

So far, we have explored the concept of resilience and its relevance in infrastructure design. We have delved into the significance of redundancy and fault tolerance in ensuring that systems can withstand and recover from disruptions. Furthermore, we have engaged in discussions surrounding the shared responsibility model within AWS cloud environments, underscoring the joint responsibilities of customers and AWS in achieving resilience. Additionally, we have examined the pertinence of resilient applications in today's digital landscape, along with the critical factors to consider when designing resilient architectures. Moreover, we have gained insights into key tracking mechanisms such as MTTR, RTO, and RPO, which are instrumental in measuring and enhancing resilience.

In the subsequent section, we will examine several noteworthy design considerations applicable to the construction of highly available applications on AWS. We will explore methods to effectively utilize some of the inherent infrastructure capabilities within the AWS environment in architectural design. Furthermore, we will address several frequently overlooked principles that are crucial for maintaining a dependable environment.

Facing the cloud's storms

While AWS offers an environment that is generally regarded to offer high availability, security, and scalability compared to typical on-premise environments, it is up to you as the builder to design your architecture in a way that effectively makes use of necessary features to build an environment that can withstand unforeseen incidents that affect your application environment.

AWS offers a secure, scalable, and resilient environment for you to run your workloads. AWS has Regions all over the globe, with each Region having multiple **Availability Zones** (**AZs**) within. An **AWS Region** is a distinct geographical area where AWS clusters its data centers. Each Region is completely independent and isolated from other Regions, providing fault tolerance and high availability. They allow you to deploy your applications and store your data closer to your users for reduced latency and improved performance. In a Region, an AWS AZ is an isolated location that is shielded from failures in other AZs. It offers low-cost, low-latency network connectivity to other AZs in the same Region. Each AZ contains at least one data center, each with redundant power, networking, and connectivity, housed in separate physical facilities. These zones are designed to support the operation of scalable and fault-tolerant production applications and databases, which would be difficult to achieve with a single data center. Each AZ operates independently, so a failure in one won't affect others, ensuring high availability and fault tolerance for applications and databases. AWS customers can use multiple AZs within one Region to increase redundancy and reliability. Each AZ is isolated, with independent power, cooling, and networking, and when an entire AZ goes down, workloads can be failed over to another AZ in the same Region, a capability known as **multi-AZ redundancy**.

The following diagram shows how AWS Regions have redundancy and isolation in place to support high resilience and availability. If there happens to be an outage in an AZ, which in itself is very rare, it doesn't affect other AZs in the Region.

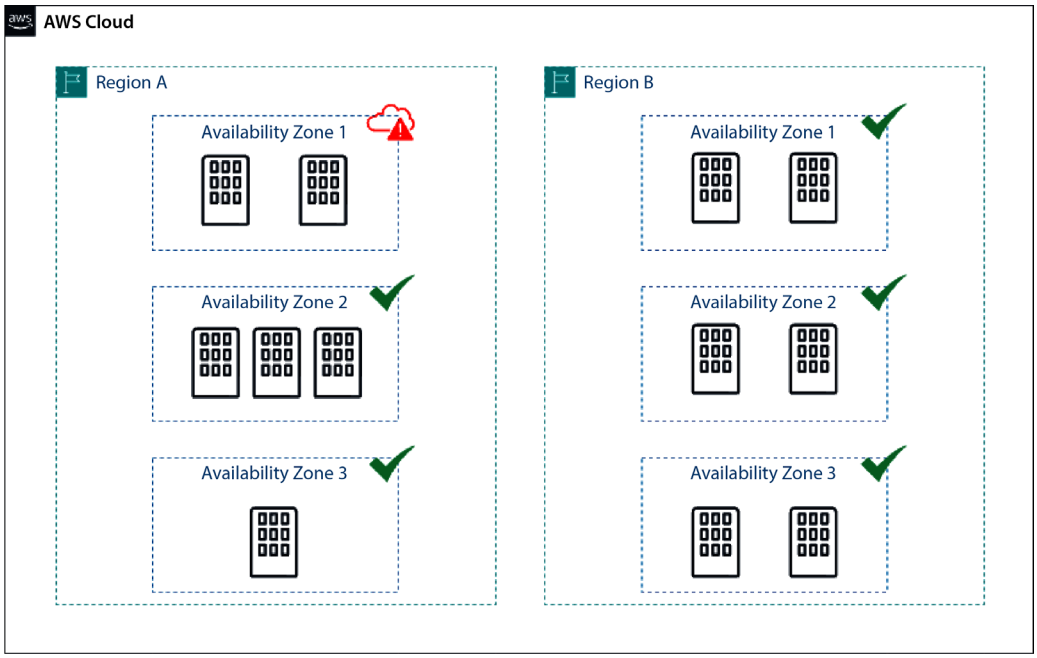

Figure 1.4 – Isolated AZs within an AWS Region

Just like other data center operation scenarios, AWS is not exempt from hardware, network, or power failures. However, such issues are overcome by AWS through proper monitoring and automation practices to address the problems quickly in such a way that customers do not experience any performance degradation or have the problem have a cascading effect on a large scale.

At this point, it is worth remembering the Shared Responsibility Model we discussed earlier in the chapter. As you can see here, AWS offers a very robust infrastructure for you to host your workloads. AWS takes ownership of keeping the underlying hardware and infrastructure healthy, performant, and secure. It is the customers' responsibility to design applications that leverage the infrastructure in a way that allows them to enjoy maximum benefits.

While AWS takes great care in keeping the infrastructure healthy, there can be situations where something such as a software bug, unforeseen hardware failure, or a natural disaster can impact service availability depending on the size of the impact. Due to built-in redundancy and isolation models, this is mostly restricted to a small blast radius and it rarely affects an entire AZ or a Region.

However, as users of AWS, it is always recommended to design applications that do not simply rely on resources available within a single AZ.

For example, consider a scenario where a data center inside an AZ goes down due to a natural disaster, such as a fire, flood, or earthquake. The application making use of the underlying infrastructure will also go down along with it. This is why it is important to design applications that leverage more than one AZ.

While we will learn more about this topic in the upcoming chapters, designing applications that leverage multiple AZs starts with creating **Virtual Private Cloud** (**VPC**) networks that have subnets spread across multiple AZs. The higher the number of AZs used, the better the availability is.

Here are some additional details about how to design applications that leverage multiple AZs:

- Use a load balancer to distribute traffic across multiple AZs. This will help to ensure that if one AZ goes down, traffic will still be routed to the other AZs.
- Store your data in multiple AZs. This will help to ensure that if one AZ goes down, your data will still be available in the other AZs.
- Use a **highly available database**. A high-availability database is a database that is designed to stay up and running even if one or more of its components fail.
- Test your application for availability. Make sure to test your application for availability to ensure that it can withstand the failure of one or more AZs.

By following these tips, you can design applications that are more resilient to failures and that can provide better availability for your users.

Software bugs and security threats

One obvious reason why an application is not resilient is due to performance issues introduced while programming the application. The customer using AWS is solely responsible for writing optimal code to deliver a good experience to the end users. Following programming best practices that leverage well-known design principles is key to writing good software.

Creating stateless, horizontally scalable microservice applications makes the entire software development lifecycle much easier compared to managing a large monolith application. The Microservice architecture allows you to achieve better resiliency by providing redundancy, isolation, and easier automation options.

Software developers can write secure code by following best practices and ISO standards. Some of the best practices include the following:

- **Using secure coding techniques**: This includes using secure coding languages and libraries, avoiding common security vulnerabilities, and implementing security controls such as input validation and output sanitization.
- **Testing code for security vulnerabilities**: This includes static code analysis, dynamic code analysis, and penetration testing.
- **Following ISO standards**: ISO 27001, ISO 27002, and ISO 27005 are all international standards that provide guidance on information security management.

The **International Organization for Standardization** (**ISO**) 27000 series of standards comprises ISO 27001, ISO 27002, and ISO 27005. These standards provide a comprehensive framework for **Information Security Management Systems** (**ISMSs**) and risk management. Here is a concise overview of each standard:

- **ISO 27001**: This is an internationally recognized standard that outlines the specifications for an ISMS. It presents a systematic approach to managing sensitive company information, guaranteeing its security. ISO 27001 assists organizations in establishing, implementing, maintaining, and continuously enhancing their ISMS.
- **ISO 27002**: This standard offers guidance and principles for establishing, implementing, maintaining, and enhancing information security management in an organization. It assists organizations in choosing controls as part of the ISMS implementation process. ISO 27002 is a complementary standard that facilitates the implementation of ISO 27001.
- **ISO 27005**: ISO 27005 serves as an international standard that guides organizations in conducting information security risk assessments in alignment with ISO 27001 requirements. Applicable across organizations of any size or industry, this standard aims to facilitate the effective implementation of comprehensive information security measures grounded in a risk management approach.

In summary, ISO 27001 sets the requirements for an ISMS, ISO 27002 provides guidelines for implementing controls within an ISMS, and ISO 27005 outlines the process for conducting information security risk assessments in line with the requirements of ISO 27001. These standards work together to help organizations establish and maintain effective information security management and risk assessment processes.

By following these best practices and ISO standards, software developers can help to ensure that their code is secure and resistant to attacks, impacting resiliency negatively as a result.

Using Software Bill of Materials (SBOM) best practices

A **Software Bill of Materials** (**SBOM**) is a critical component of software security. A good SBOM should include information about the software components, their versions, and their dependencies. This information can be used to identify security vulnerabilities and risks and to track the provenance of software.

Following the underlying best practices for creating SBOMs as follows can reduce target surface areas:

- Use a standardized format for SBOMs. This will make it easier to share and use SBOMs.
- Include as much information as possible in the SBOM. This includes information about the software components, their versions, their dependencies, and any known security vulnerabilities.
- Keep the SBOM up to date. This is important to ensure that the SBOM reflects the latest information about the software components.
- Use SBOMs to identify security vulnerabilities and risks. This information can be used to prioritize security remediation efforts.
- Track the provenance of software. This information can be used to identify and mitigate software supply chain risks.

So far, we have learned some key concepts in designing resilient architectures, particularly focusing on the concepts of redundancy, fault tolerance, isolation, automation, and monitoring. We have highlighted the importance of building redundant systems to mitigate the impact of failures and discussed the shared responsibility model between CSPs and customers to ensure resilience. We have also emphasized the need to consider factors such as MTTR, RTO, and RPO when designing resilient applications. We explored this concept through the Gimli Glider incident by correlating them with aircraft design principles to illustrate the practical implementation of resilience strategies.

In the next section, we are going to look into different AWS services that you as a builder can leverage to build resilient systems. We will also learn about the importance of practicing resilience as a continuous journey rather than a one-time activity.

Empowering yourself with AWS services

AWS provides a diverse range of services and features to facilitate the development of resilient applications. These services cover several key foundational areas, such as network, compute, database, monitoring, and messaging. Builders can leverage these services to create a strong, resilient foundational architecture that is highly available and acts as a base to deploy applications.

The accompanying diagram illustrates a selection of AWS services that can be utilized for this purpose. It is important to note that the design decisions pertaining to application architecture ultimately reside with the customers, and as a user, you retain complete autonomy to select the solution that optimally addresses your specific requirements in terms of performance, security, and cost.

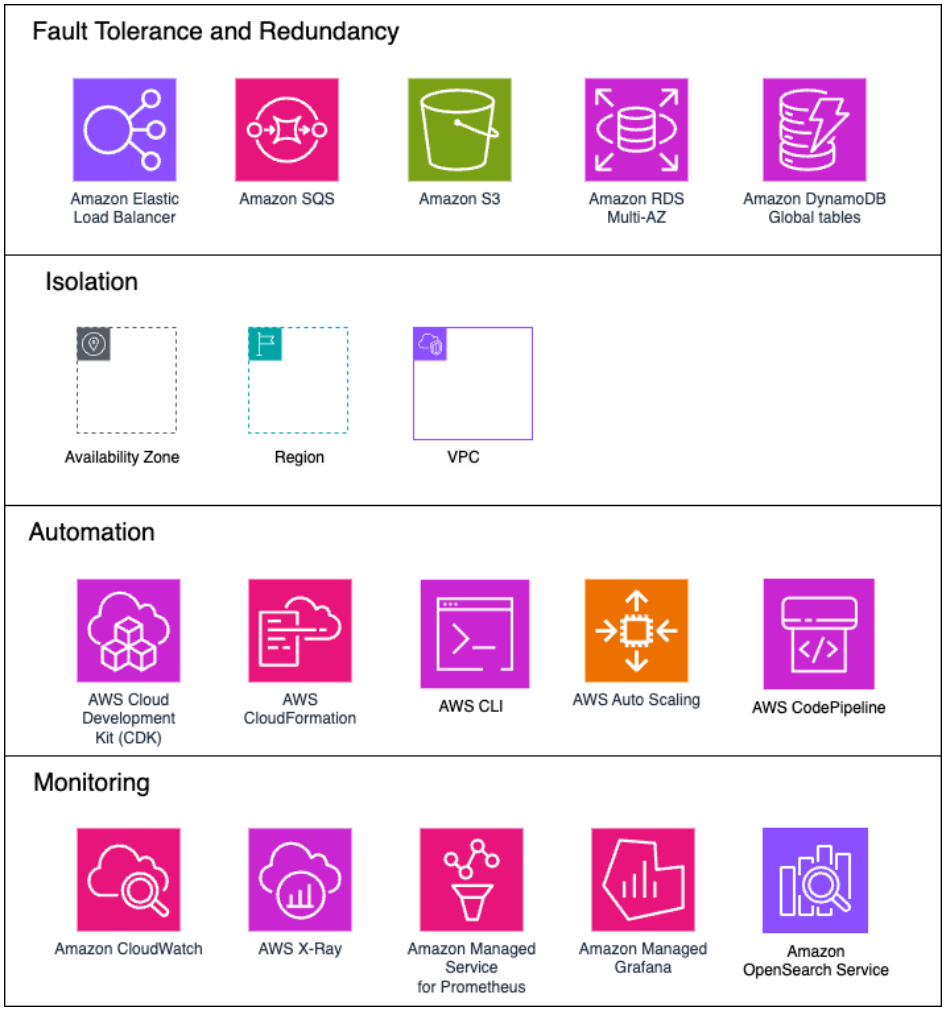

Figure 1.5 – Different AWS services for builders to make use of to design resilient architectures

Not all applications can be optimally hosted on the same type of infrastructure or architecture. The implementation decisions are predicated upon input gathered from both business and technical teams. Business teams' insights will inform you about the scale of application usage, the target user base, geographical Region support requirements, response time expectations, availability requirements, and more. This information will assist you in determining the SLAs for the application in terms of latency, availability, error rate, and other relevant metrics.

You can take these inputs to then determine how you want to design the application architecture. For example, if the application does not need to respond to user requests in a synchronous fashion, then designing the application based on batching services such as AWS Batch would be an ideal solution that makes use of Amazon SQS to retrieve messages offline and process appropriate jobs.

Upon utilizing fully managed services such as AWS Lambda, DynamoDB, Amazon Aurora/**Relational Database Service** (**RDS**), Amazon **Simple Queue Service** (**SQS**), Amazon **Simple Storage Service** (**S3**), Amazon CloudWatch, Amazon Managed Service for Prometheus, and numerous other services, customers automatically acquire multi-AZ availability. These services are engineered to inherently support cross-AZ availability.

For instance, when establishing an Aurora or RDS database, you retain the ability to denote whether the instances are deployed across various AZs or within a single AZ, which is strongly advised against for production use. Aurora employs a dispersed storage system across multiple AZs for data distribution, storage, and replication.

Furthermore, it delivers failover recovery within seconds and automated scaling of storage and computing resources independently, without any downtime. As a user, you can delegate the substantial responsibility of designing and preserving the availability of the database to Aurora to handle.

When utilizing Amazon EC2 for computational purposes, customers are granted substantial control over their environment's capabilities. However, this level of control necessitates responsible behavior from customers to ensure optimal resource utilization.

Amazon EC2 Auto Scaling, in conjunction with AWS Auto Scaling and Amazon CloudWatch services, offers a convenient method for setting up auto scaling. EC2 launch templates enable the automatic launching of EC2 instances of your choosing, thereby facilitating the scaling of EC2 workloads. This also encompasses scaling cluster nodes on both Amazon EKS and Amazon ECS clusters.

Beyond EC2, AWS Auto Scaling extends its support to scaling DynamoDB tables, as well as Amazon ECS and Amazon Aurora services.

As we know, monitoring is a key aspect of building resilient architectures. Amazon CloudWatch is a fully managed observability platform that can be used for collecting, storing, and querying metrics, logs, and traces from any environment, including non-AWS environments such as on-premises or other cloud providers. Without managing any servers or capacity planning, you can simply use CloudWatch to monitor infrastructure and applications at planet scale.

You can employ CloudWatch to visualize performance and metric details through dashboards; set up alarms; correlate, query, and analyze logs and metrics; and trace data to troubleshoot problems.

Now, let's understand resilience as something that needs to be continuous.

The continuous resilience journey

The resilience lifecycle framework provided by AWS is a structured and iterative process designed to foster resilience at every stage of an application's lifecycle. It encompasses a five-stage approach that addresses the entire spectrum, from planning and design to operation and continuous improvement. By adhering to this framework, organizations can construct and maintain highly resilient applications capable of withstanding disruptions and delivering value consistently, even in challenging circumstances.

Nevertheless, achieving resilience is not a one-time endeavor; rather, it is an ongoing process. Resilience constitutes a journey that necessitates continuous monitoring, refinement, and improvement. Tracking crucial metric parameters, such as SLAs, RTOs, and RPOs enables organizations to adopt a scientific approach to resilience goals. Furthermore, implementing processes such as **Operational Readiness Reviews** (**ORRs**) assists in identifying potential issues during the operational phase and ensures that the system is equipped to handle real-world scenarios. By embracing resilience as a continuous journey, organizations can guarantee that their applications remain resilient and reliable and consistently deliver value. Let's take a look at the details of these in the following section.

Resilience Lifecycle Framework

The **resilience lifecycle framework** is a five-stage approach developed by AWS to help organizations continuously improve the resilience of their applications. It provides a structured, iterative process for building resilience at every stage of an application's lifecycle, from planning and design to operation and ongoing improvement.

Here's a breakdown of the five stages:

1. **Set objectives**: This stage involves defining the desired level of resilience for your application, considering factors such as acceptable downtime, RTOs, and RPOs. You also identify potential threats and vulnerabilities to understand what your resilience strategies need to address.
2. **Design and implement**: In this stage, you design your application with resilience in mind. This includes implementing techniques such as redundancy, fault tolerance, disaster recovery, and automation. You also choose and configure AWS services and features that support your resilience goals.
3. **Evaluate and test**: Before deploying your application to production, it's crucial to assess its resilience capabilities. This can involve running simulations, chaos testing, and penetration testing to identify weaknesses and make improvements.

4. **Operate**: Once deployed, you continuously monitor your application's performance and health. You also need to have incident response plans in place to address any disruptions or outages that may occur.
5. **Respond and learn**: After an incident or during ongoing operations, analyze what happened and identify opportunities for improvement. This could involve updating your resilience strategies, implementing new technologies, or refining your operational procedures.

The resilience concepts of each stage are applied at different levels, ranging from individual components to entire systems. The framework helps anticipate disruptions and their impact on your application, allowing you to identify the mitigations needed to build a resilient, reliable application. It is recommended to update your resilience model with every iteration of your application's lifecycle.

Resilience, as defined by AWS, is *"the ability of an application to resist or recover from disruptions, including those related to infrastructure, dependent services, misconfigurations, and transient network issues"* (`https://docs.aws.amazon.com/prescriptive-guidance/latest/resilience-lifecycle-framework/introduction.html`).

By following the resilience lifecycle framework, organizations can build and maintain highly resilient applications that can withstand disruptions and continue to deliver value even in the face of challenges.

Using the resilience lifecycle framework, you are able to also achieve some critical additional benefits such as the following:

- **Reduced risk of downtime and data loss**: Proactive planning and testing help mitigate the impact of incidents and ensure business continuity
- **Improved operational efficiency**: Automated processes and efficient incident response plans streamline operations and save time
- **Enhanced customer experience**: By minimizing disruptions and ensuring service availability, organizations can deliver a better experience for their customers
- **Increased confidence and trust**: Demonstrating a commitment to resilience can build trust with stakeholders and potential customers

That covers the framework that can be used to establish resilience. In the following section, we will look into why making this a continuous activity will help achieve greater results.

Resilience is a continuous journey

Achieving resilience is not a one-time activity; rather, it's an ongoing process. Resilience is a journey that you continue to monitor, tune, improve, and build upon. Tracking important metric parameters, such as SLA, RTO, and RPO, allows you to take a scientific approach to resilience goals.

These metrics are well established in the industry and there are several tools and processes to track them.

To get ahead of the problems that might occur during the operational phase, Amazon follows a process called ORR. The ORR process is essentially a checklist of questions that you would prepare, keeping the target goals in mind. The aim is for these questions to help you solve the business problems defined. Having said that, to get started with implementing an ORR in your company, the first step is to identify the business challenges you want to solve as a result of implementing the ORR. It could be something such as wanting to reduce outages or increase scalability and operational efficiency.

It is recommended to cover the following three key areas in your ORR:

- Architectural recommendations
- Operational processes
- Event management

Covering the previous areas as part of the ORR helps ensure your system has the following characteristics:

- **Built to handle real-world usage**: Can it scale under pressure and recover from failures?
- **Operated efficiently**: Do you have clear procedures for deploying changes and responding to incidents?
- **Monitored and maintained**: Are you proactively tracking its health and identifying potential issues?

By going through the ORR, teams gain valuable insights and actionable steps to improve their system's operational readiness.

The AWS Well-Architected Framework covers the ORR process in detail and is a great source to learn more about this in detail (`https://docs.aws.amazon.com/wellarchitected/latest/operational-readiness-reviews/wa-operational-readiness-reviews.html`).

Summary

In this chapter, we discussed the significance of resilience in building reliable and robust applications. We introduced the resilience lifecycle framework, developed by AWS to help organizations continuously improve the resilience of their applications.

We also highlighted the benefits of following the resilience lifecycle framework, including reduced risk of downtime and data loss, improved operational efficiency, enhanced customer experience, and increased confidence and trust.

To achieve resilience, organizations should view it as an ongoing process, continuously monitoring, tuning, and improving their resilience strategies. Tracking important metric parameters such as SLA, RTO, and RPO enables a scientific approach to resilience goals.

We explored the ORR process, a checklist of questions designed to help organizations identify and solve business problems related to resilience.

Finally, the chapter emphasized the importance of incorporating security measures into the resilience strategy, considering the evolving threat landscape and the need to protect applications from cyberattacks and other security breaches.

By following the resilience lifecycle framework and adopting a proactive approach to resilience, organizations can build and maintain highly resilient applications that can withstand disruptions and continue to deliver value even in challenging circumstances.

In the subsequent chapter, we will thoroughly examine how various techniques and AWS services enable the implementation of a resilient architecture. Emphasis will be placed on utilizing auto scaling as a fundamental method for achieving reliability objectives. Additionally, we will explore strategies for cost optimization through the use of AWS Spot instances, offering redundancy and scalability for your applications.

2

Implementing Resilient Compute and Auto Scaling

This chapter will provide a comprehensive guide to implementing resilient compute and auto scaling solutions on AWS. We will delve into the significance of designing systems that can withstand failures by incorporating redundancy and fault tolerance into compute resources. We will also explore various factors that can disrupt system stability, including resource issues, service disruptions, application/code issues, security threats, and environmental factors. After, we will learn about key principles and strategies for addressing these factors, such as multi-**Availability Zone** (**AZ**) deployments, redundant environments, and stateless architectures, providing practical examples and architectural illustrations.

The next stage will be to introduce **AWS Auto Scaling** as a solution for dynamic resource management and explain its key components and benefits. We will further discuss cost-saving strategies using Spot Instances and Reserved Instances, offering insights into their advantages and effective management. This chapter emphasizes the importance of monitoring and maintaining a healthy infrastructure, highlighting AWS observability services. Finally, we will touch on extending resilience to containers and serverless architectures while discussing relevant AWS services.

By understanding the concepts and strategies presented in this chapter, you will be equipped to design and implement resilient compute solutions on AWS, ensuring high availability, scalability, and cost-efficiency for your applications.

In this chapter, we will cover the following topics:

- Redundancy and fault tolerance in compute
- Embracing auto scaling for dynamic resource management
- Optimizing cost-efficiency with Spot and Reserved Instances
- Monitoring and maintaining a healthy infrastructure
- Extending resilience to containers and serverless

Redundancy and fault tolerance in compute

Before engaging in an in-depth exploration of resilient architecture design implementation, it is imperative to gain an understanding of the various types of interruptions that have the potential to destabilize a system operating on AWS.

The factors that disrupt the stability of a system may originate from both internal (those that you, as a user, have control over) and external (factors that are controlled by AWS that you have little or no control over) sources. Furthermore, they can be categorized as either controllable or uncontrollable factors. For instance, a sudden surge in usage resulting from the unexpected popularity of a piece of software is an occurrence that can't be fully anticipated. However, it is also true that proactive planning for such spikes can be accomplished by implementing appropriate measures during the design phase of the application architecture.

Let's consider some of the important factors that can impact system stability. These factors can arise either internally or externally:

- **Resource issues**:
  - **Capacity overload**: Sudden spikes in traffic or resource-intensive tasks can overwhelm allocated resources such as CPU, memory, or network bandwidth. This can lead to throttling, slowdowns, and even crashes.
  - **Insufficient resources**: Running your system on inadequate resources for its workload, such as inadequate CPU or memory, can lead to chronic instability. Stretched resources struggle to handle even routine operations, making the system vulnerable to failure.
  - **Resource configuration errors**: Misconfigured resources such as security groups, IAM roles, or VPC settings can unintentionally restrict access or functionality, leading to unexpected behavior and instability.
- **Service disruptions**:
  - **Underlying service outages**: While rare, underlying AWS services can experience outages or throttling, impacting applications that rely on them. This can be mitigated through redundancy and multi-AZ deployments.
  - **API throttling**: Exceeding API quotas or making excessive API calls can trigger throttling, limiting your application's functionality and causing performance degradation.
  - **Service updates and changes**: New features or changes introduced by AWS services can sometimes lead to unforeseen compatibility issues or bugs, impacting your applications' stability.
- **Application and code issues**:
  - **Software bugs**: Bugs in your application code can lead to crashes, memory leaks, or unexpected behavior. Rigorous testing and code reviews are crucial to prevent such issues.

  - **Configuration errors**: Misconfigurations within your application's settings or environment variables can lead to malfunctions and instability.
  - **External dependencies**: Applications relying on external APIs or services can inherit their instability if those dependencies experience outages or errors.

- **Security threats**:
  - **Denial-of-service (DoS) attacks**: Malicious actors can launch DoS attacks to flood your system with traffic, overwhelming resources and causing outages. Security best practices and **distributed denial-of-service (DDoS)** mitigation strategies are essential.
  - **Security vulnerabilities**: Unpatched vulnerabilities in your application or underlying systems can be exploited by attackers to gain unauthorized access or disrupt operations. Regular security audits and patching are crucial.

- **Environmental factors**:
  - **Network connectivity issues**: Network outages or latency issues can disrupt communication between your system components, leading to slowdowns and errors.
  - **Power outages or hardware failures**: While uncommon in AWS, hardware failures or power outages in data centers can temporarily disrupt service.

Previously, we discussed a variety of factors that can have an impact on the stability of your environment. Operators must be attentive to these aspects to enhance resilience. In the following section, we will discuss some of the principles that you can follow to improve system resilience.

Key principles for addressing factors influencing system stability

Increasing system stability is a core part of having a resilient environment. While attaining a truly stable environment that never fails is extremely challenging and cost-prohibitive, it is important to be intentional about making architectural decisions to improve stability. The following are some important factors to consider when you make architectural design decisions:

- **Multi-AZ deployments**
- **Redundant environments**
- **Stateless architectures**

We introduced the multi-AZ deployment architecture in the previous chapter and took a look at how AWS allows us to build de-coupled infrastructure through AZs. An AZ is a distinct physical location within an AWS Region. AZs are designed to be independent of one another, with their own power, cooling, and networking. Each Region has multiple AZs, which are typically located within proximity to each other. This allows applications to be deployed across multiple AZs to achieve high availability.

If one AZ experiences an outage, applications can continue to run in the other AZs. This isolation helps protect applications and data from failures in a single AZ.

AZs are a fundamental building block of AWS's resilient infrastructure. By designing applications and systems to take advantage of AZs, you can improve the reliability, availability, and scalability of your applications. The following architecture shows how a really simple multi-AZ architecture can be designed using Amazon EC2 and Amazon Aurora:

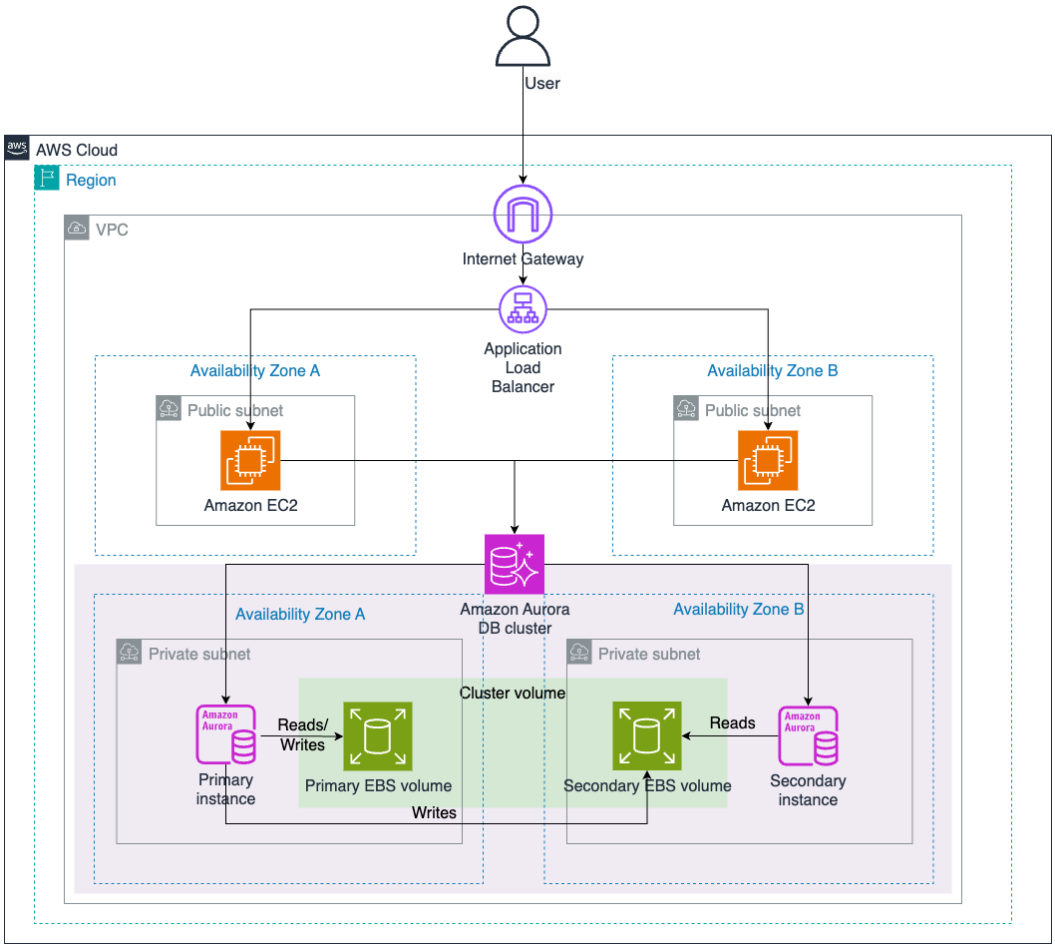

Figure 2.1 – Simple multi-AZ architecture

Let's assume that there is an e-commerce website hosted on this infrastructure where the EC2 instances have web servers running on them and the database is hosted on Amazon Aurora. Incoming requests from users are handled through an **Application Load Balancer** (**ALB**) to provide load balancing across

EC2 instances behind them. ALBs can be configured to route traffic to the EC2 instances behind them using different techniques, such as the following:

- **Round-robin**: Requests are routed to targets in a round-robin fashion. Essentially, the requests are distributed equally across target instances.
- **Least outstanding requests**: When a new request comes in, the load balancer that uses this algorithm will forward it to the target with the fewest outstanding requests.
- **Weighted random**: Weighted random routing is a load balancing algorithm that takes into account the capacity of each instance in a target group when distributing traffic. This algorithm assigns a weight to each instance, which determines the likelihood of that instance receiving a request. The higher the weight, the more likely the instance is to receive a request. Weighted random routing is useful when you have instances of different sizes or capacities in a target group. For example, you might have some instances that are more powerful than others or some instances that have more memory or storage. Weighted random routing ensures that traffic is distributed to instances in a way that is proportional to their capacity.

This setup ensures user requests to the website are served even if one of the servers were to fail. This is a simple yet classic example of setting up high availability on AWS.

The design of high availability for a web server demands meticulous attention to detail, whereas the high availability design of Amazon Aurora is inherently integrated into the service. Amazon Aurora provides multi-AZ availability as a service by automatically provisioning a primary instance and a secondary instance in separate AZs. Amazon Aurora also assumes responsibility for synchronizing data across the different Amazon Aurora instances.

In this configuration, the primary Amazon Aurora instance consistently responds to read and write requests. If a failure renders the primary instance inoperable, Amazon Aurora will automatically designate the secondary instance as the primary instance to handle read and write requests from the web server.

Now, let's delve deeper into this matter. What transpires with users who were receiving services from one of the web servers that just experienced an outage? It is possible that the users' sessions were invalidated, and their workflow was disrupted, potentially leading to adverse business implications. This problem is of critical importance as it fundamentally undermines the commitment to delivering high availability.

To address this challenge, it is necessary to separate state data from the web server, effectively implementing a stateless architecture. Such an approach enables failover to occur without compromising the functionality of the web server.

The following diagram shows how that can be done easily on AWS:

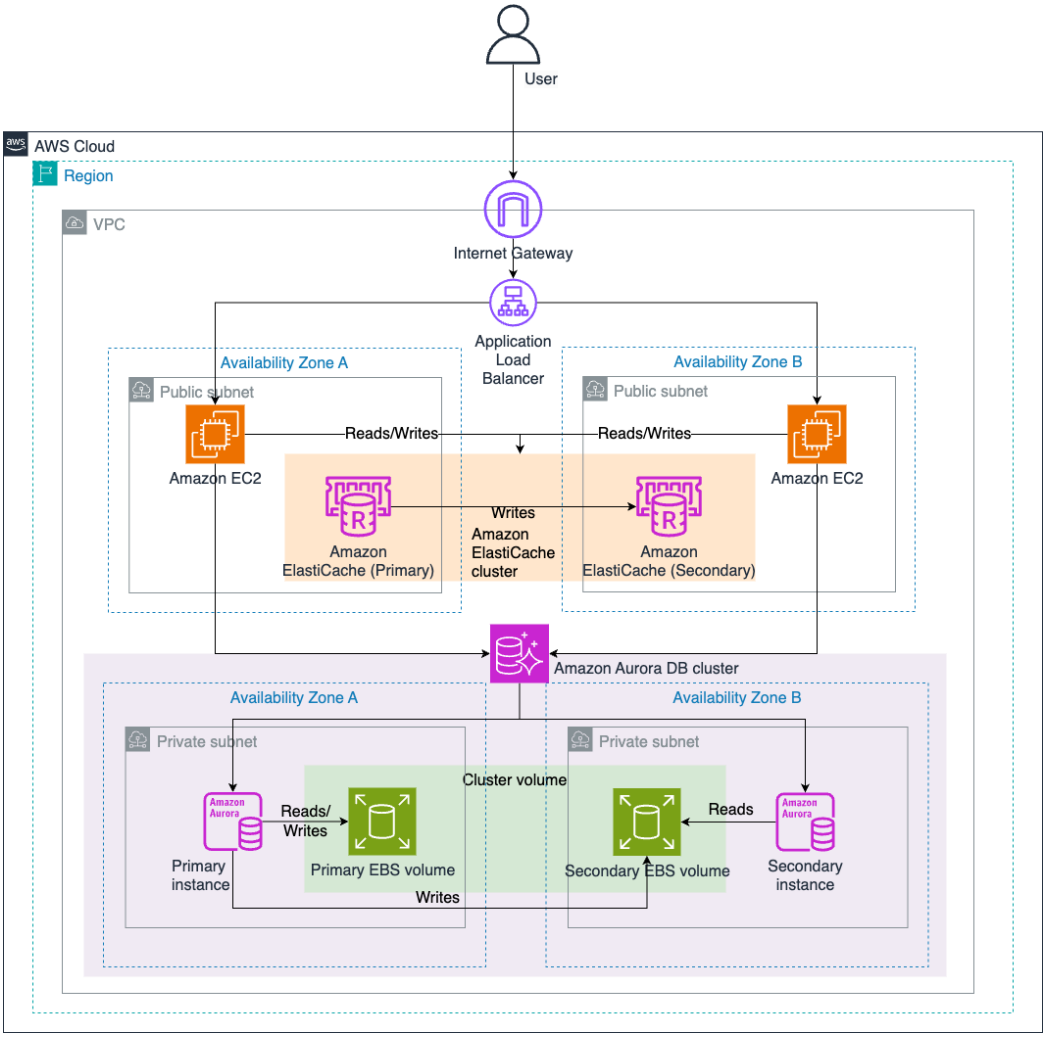

Figure 2.2 – An enhanced multi-AZ architecture with stateless web server design

In the preceding architecture, you can see Amazon ElastiCache for Redis being used to store user sessions. The web servers do not store any session state in memory within themselves; rather, they offload that responsibility to the Amazon ElastiCache for the Redis cluster.

Amazon ElastiCache for Redis is a managed, in-memory data store service that is compatible with Redis. It provides fast, scalable, and secure data access for applications that require low latency and high throughput. It is designed to make it easy to deploy and operate Redis, and it offers a range of features that make it a great choice for applications that need to store data in memory.

Amazon ElastiCache for Redis offers a serverless option that automatically provisions and scales a Redis cluster based on demand, eliminating the need for manual cluster management. You could also choose to make use of Amazon ElastiCache for Memcached or Amazon MemoryDB for Redis for the same purposes.

In this section, we learned about different strategies we can deploy to improve system stability by adopting various AWS services, such as AWS Elastic Load Balancer, Amazon ElastiCache, Amazon Aurora multi-AZ, and Amazon EC2. However, note that this is not a definitive set of services to use to attain higher stability in your environment. AWS offers several building blocks for you to design a stable architecture for your specific needs. In the following section, we will learn about auto scaling and how you can make use of that to allocate resources dynamically just in time as needed.

Embracing auto scaling for dynamic resource management

In the dynamic world of cloud-based applications, efficient resource management is crucial for both optimal performance and cost control. With AWS Auto Scaling, AWS offers a flexible way to adjust compute resources based on demand to deliver a seamless user experience while keeping your budget in check.

What is Auto Scaling in AWS?

Imagine a world where your application effortlessly scales up during peak traffic hours, preventing frustrating slowdowns, and gracefully scales down during quiet periods, saving you precious resources. That's the magic of AWS Auto Scaling. It eliminates the manual guesswork of resource allocation by automatically provisioning and deprovisioning compute resources (think EC2 instances, Lambda functions, and more) based on predefined metrics such as CPU utilization, memory usage, or network traffic.

This translates to several key benefits:

- No more scrambling to predict resource needs with Auto Scaling reacting in real time based on actual demand.
- When user traffic spikes, Auto Scaling automatically scales up resources, preventing application slowdowns or crashes.
- By scaling down during low-traffic periods, you avoid paying for unused resources. Think of it as going into power-saving mode when you're away from home.

Some pitfalls and assumptions in the e-commerce architecture

Regarding the e-commerce architecture we visited in the previous section, a couple of assumptions were made:

- The resources (CPU and memory) that are allocated for the EC2 instances of the web server are assumed to be sufficient enough to handle peak traffic.
- There will be a stable and appropriate utilization of resources. This means there will be steady traffic to the website in such a way that the allocated resources will be used meaningfully.

These two assumptions serve as the primary determinants in selecting capacity for the resources that are employed within the architectural framework. However, what implications arise if these assumptions prove to be less than optimal? The potential consequence is insufficient resources being allocated, which could adversely impact the application's resilience.

Conversely, over-allocating resources may result in waste and incur unnecessary expenditures beyond the actual requirement, thereby undermining the core objective of utilizing a cloud service provider.

This is exactly where AWS Auto Scaling comes into play. It allows us to be dynamic in terms of resource allocation and only use appropriate resources when necessary, thereby providing optimal resource utilization and spending.

The key components of Auto Scaling in AWS

Auto scaling in AWS is a partnership between two key components:

- **Auto Scaling groups** (**ASGs**): This is the group of compute resources that Auto Scaling manages. In AWS, this could be a group of EC2 instances, a fleet of Lambda functions, or even a cluster of Amazon Aurora replicas. All of these work together to handle the workload.
- **Scaling policies**: These are the key components behind the operation, defining triggers and actions for scaling. Triggers specify metrics (for example, CPU utilization exceeding 80%) and thresholds that initiate scaling, while actions determine how many resources to add or remove from the ASG (for example, add two new instances or terminate idle Lambdas).

Now, let's take a look at how we can modify the e-commerce website architecture while including AWS Auto Scaling:

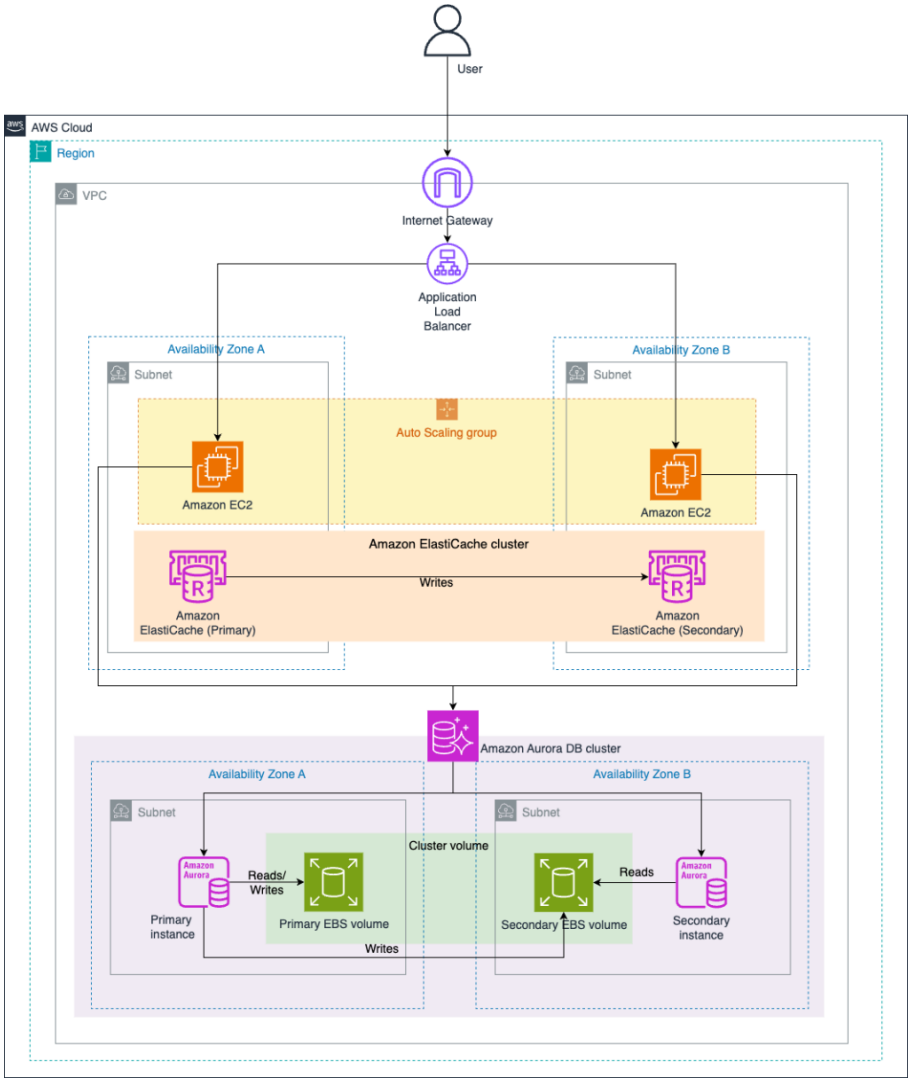

Figure 2.3 – An enhanced multi-AZ architecture with stateless web server design and auto scaling EC2 instances

With this architecture in place, the website can scale out on demand to accommodate additional user traffic while also scaling in when additional resources aren't needed. This allows us to leverage on-demand cloud resources and ensure a smooth and responsive user experience that adapts to application needs.

Some use cases and benefits of Auto Scaling in AWS

In this section, we will provide an overview of the challenging use cases that can be solved through auto scaling, along with its benefits. We will also discuss some of the considerations for the optimal implementation of auto scaling.

Auto Scaling in AWS tackles diverse challenges, adapting to various application needs, such as the following:

- **Web applications**: Auto Scaling based on active users ensures a smooth and responsive experience, even during peak hours.
- **Batch processing jobs**: Auto Scaling automatically provisions resources for specific workloads, optimizing job completion time.
- **Serverless functions**: Auto Scaling quickly adjusts concurrency based on demand, ensuring your functions execute seamlessly.
- **Databases**: Auto Scaling prevents database throttling by automatically scaling replicas based on read/write load.

Some key benefits of using AWS Auto Scaling are as follows:

- **Improved performance**: No more bottlenecks, just smooth-sailing applications that delight your users
- **Cost optimization**: Pay only for the resources you use, maximizing your cloud budget
- **Increased agility**: Adapt to changing demands in real time, ensuring your application stays ahead of the curve
- **Reduced operational overhead**: Let Auto Scaling handle the heavy lifting, freeing up your time for other strategic initiatives

While Auto Scaling helps us solve the challenges discussed earlier, you should pay attention to some important factors when you set it up. The following list briefly describes factors that help ensure that your setup is optimal, mainly in terms of efficiency and cost:

- **Metric selection**: Choose metrics that accurately reflect your application's needs. Don't rely on a single metric; consider a combination (for example, CPU, memory, and network) for a holistic view.
- **Cooldown periods**: Prevent excessive scaling in and out with short-lived spikes. Implement cooldown periods to avoid unnecessary resource churn. A cooldown period allows you to determine a waiting period before you take action on resource scaling.

- **Scaling granularity**: Define the minimum and maximum number of resources per ASG. This ensures you have enough resources for peak moments but don't overprovision during low periods.
- **Cost monitoring**: Keep track of scaling costs and optimize policies accordingly. Remember, even superheroes need to keep an eye on their budget!
- **Advanced features**: Explore advanced features such as predictive scaling, scheduled scaling, and target tracking policies for even more fine-grained control.

In this section, we went through a few use cases for using Auto Scaling in AWS and how you can make use of it to solve some important challenges. We also redesigned the e-commerce website architecture so that it makes use of Auto Scaling. Finally, we learned about some important steps and considerations to follow when making detailed design decisions regarding auto scaling that help us create an optimal architecture. In the next section, we will focus on how you can optimize and reduce costs while having a scalable infrastructure through auto scaling.

Optimizing cost-efficiency with Spot and Reserved Instances

AWS offers various pricing models for its compute resources, including **Spot Instances** and **Reserved Instances**, which provide significant cost savings compared to On-Demand Instances. Spot Instances allow users to bid on unused EC2 capacity, offering up to a 90% discount. However, AWS can reclaim Spot Instances with a 2-minute notice if the spot price exceeds the bid or capacity is needed. Reserved Instances, on the other hand, provide a significant discount compared to On-Demand Instances in exchange for a commitment to a specific instance configuration for a 1 or 3-year term. Reserved Instances are best suited for steady-state workloads, while Spot Instances are ideal for flexible, fault-tolerant workloads.

In this section, we will dive deeper into Spot Instances and how you can leverage these to your advantage in optimizing cost.

Using Spot Instances

Spot Instances are an innovative offering from AWS that enables organizations to leverage spare compute capacity at significantly reduced costs. Spot Instances are ideal for workloads that can tolerate interruptions and are suitable for various use cases, such as batch processing, data analysis, and web applications.

Consider the following graph, which shows a correlation between resource usage and cloud spend:

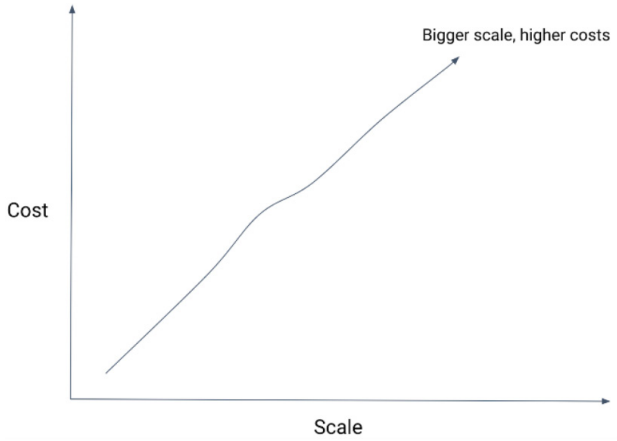

Figure 2.4 – A graph showing the linear progression of cost based on the scale of resources used

Cloud services operate under the principle of charging users based on actual usage of resources. This point of view is very true on AWS also, but certain techniques can be applied to reduce costs.

However, AWS **Spot Instances** can change this notion to a large extent for certain use cases. Let's dive deep into how Spot Instances work so that we understand the cost benefits of using them in our environments.

Spot Instances are unused Amazon EC2 instances that are available at steep discounts compared to **On-Demand Instances**. However, they come with a caveat – they can be reclaimed by AWS at any time based on demand fluctuations. Despite this potential disruption, Spot Instances provide a cost-effective way to scale resources elastically without compromising performance.

The following are some of the key advantages of Spot Instances:

- **Cost savings**: Spot Instances offer significant cost savings, typically ranging from 60% to 90% compared to On-Demand Instances (`https://aws.amazon.com/ec2/spot/`). Organizations can optimize their cloud expenses by utilizing Spot Instances for bursty or non-critical workloads.
- **Scalability**: Spot Instances enable rapid scaling of compute resources to meet fluctuating demand. Organizations can quickly provision and terminate instances based on their requirements, allowing for a flexible and adaptive infrastructure.
- **Spot Fleet**: AWS Spot Fleet simplifies the management of Spot Instances by automatically launching and managing instances across multiple AZs based on users' pre-defined configuration. This feature enhances reliability and ensures that workloads are distributed efficiently.

To leverage Spot Instances effectively, robust strategies must be implemented. A well-informed strategy aids developers and architects in designing applications that utilize Spot Instance infrastructure effectively and appropriately. Let's look at some of the strategies that you can consider for this:

- **Interruption handling**: To mitigate the impact of instance interruptions, organizations should implement strategies such as graceful shutdown scripts and automatic instance replacement mechanisms.
- **Hybrid approach**: Combining Spot Instances with On-Demand Instances can provide a balanced approach, ensuring both cost savings and reliability.
- **Bid strategies**: Understanding the Spot Instance market dynamics and employing optimal bidding strategies can help organizations secure instances at competitive prices.

When utilizing Spot Instances, adherence to standard best practices remains crucial. The following list encompasses recommended practices that end users should contemplate when formulating architectures that incorporate Spot Instances:

- Using multiple Spot capacity pools across different AZs to maximize available Spot capacity
- Leveraging AWS-managed automation with EC2 auto scaling, EC2 Fleet, or Spot Fleet to dynamically find the lowest cost capacity across Spot and On-Demand Instances
- Designing applications to be fault-tolerant and able to resume from interruptions gracefully
- Monitoring Spot price changes and instance interruptions closely

When utilizing Spot Instances, it is important to note that infrastructure costs do not scale linearly with usage. The pricing structure of Spot Instances enables significant cost reductions, particularly when deployed at scale, as shown in the following figure:

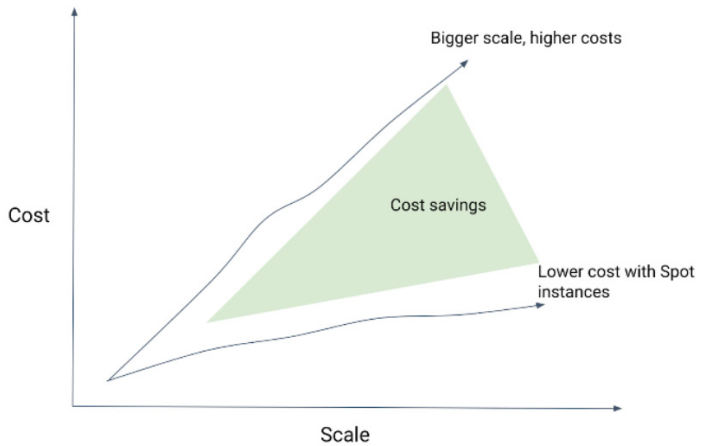

Figure 2.5 – A graph showing the cost-saving impact due to the usage of Spot Instances

In this section, we examined strategies and best practices for using Spot Instances. Due to the possibility of AWS reclaiming capacity at any time, using Spot Instances can present challenges. If not properly designed, this can have a substantial negative impact on application stability. Therefore, in addition to adhering to the best practices outlined in this section, careful consideration should be given to workload types when using Spot Instances.

The ephemeral nature of resource availability can make Spot Instances unsuitable for certain mission-critical, long-term workloads. In the following section, we will take a look at how you can cut costs for such workloads by using Reserved Instances.

Using AWS Reserved Instances

In addition to Spot Instances, AWS users can leverage AWS **Reserved Instances**, which offer significant cost savings compared to On-Demand Instances. By committing to a specific instance type, size, and term, organizations can save up to 75% on their cloud computing costs (`https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-reserved-instances.html`). Reserved Instances are particularly beneficial for predictable, steady workloads such as production databases, enterprise applications, and mission-critical systems.

There are several use cases where Reserved Instances can be leveraged effectively:

- **Database instances**: Reserved Instances can be applied to Amazon RDS instances, providing cost-effective and reliable database hosting.
- **Compute-intensive workloads**: For applications requiring consistently high compute power, such as data processing or scientific simulations, Reserved Instances offer significant savings.
- **Long-term applications**: When running applications for an extended period, Reserved Instances lock in lower rates, ensuring predictable and stable pricing.
- **Seasonal or predictable workloads**: Businesses with predictable seasonal spikes or regular usage patterns can save costs by utilizing Reserved Instances during peak times.

To maximize the benefits of Reserved Instances, organizations should carefully assess their resource usage patterns, forecast future needs, and select the appropriate instance type and term. Additionally, monitoring Reserved Instance utilization and adjusting the portfolio as needed can optimize cost savings.

By putting all of these new ideas together, the e-commerce website architecture can make use of the following techniques to reduce operational costs while also offering higher resiliency:

- Assuming that we require at least two EC2 servers behind the load balancer, at a bare minimum, we can use AWS Reserved Instances for those web servers, which can reduce costs by 60% (`https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-reserved-instances.html`).
- AWS ASGs support multiple instance types in a single launch template. This allows you to specify the percentages of On-Demand Instances and Spot Instances you wish to launch. Since our web server is stateless, this strategy will further reduce operational costs.

- AWS Reserved Instances can be used for Amazon Aurora nodes given that we would need at least a single Primary and Secondary pair for resiliency purposes. Using Spot Instances for database workloads is not the most prudent choice and can prove to reduce the resilience of the architecture. Hence, using On-Demand Instances is a more suitable option for such workloads.

In this section, we examined Spot Instances and Reserved Instances, exploring their potential for various workload types to curtail compute expenses. Both Spot Instances and Reserved Instances are innovative offerings from AWS, enabling users to construct robust architectures at scale while mitigating significant cost implications. In the following section, we will investigate diverse monitoring options and their role in upholding a resilient infrastructure.

Monitoring and maintaining a healthy infrastructure

Irrespective of whether an individual is managing a large-scale operation within a public cloud or a more limited server deployment within an on-premises environment, it is imperative to comprehend the performance of resources. Without diligently tracking key metrics, analyzing logs, and collecting application traces, obtaining an accurate understanding of resource performance and health is impossible. The information that's gathered through these processes involves optimizing resource allocation, implementing auto scaling mechanisms, making informed software architecture design decisions, and enhancing user experience. Ultimately, making informed decisions is crucial to the long-term success of operating technical infrastructure and environments.

By prioritizing continuous health checks, implementing auto-recovery configurations, and embracing proactive maintenance, we can empower our workloads with the resilience and reliability they deserve.

AWS observability services

AWS offers a robust portfolio of observability options for users to make use of. The following figure shows different native and open source managed services AWS offers for users to leverage:

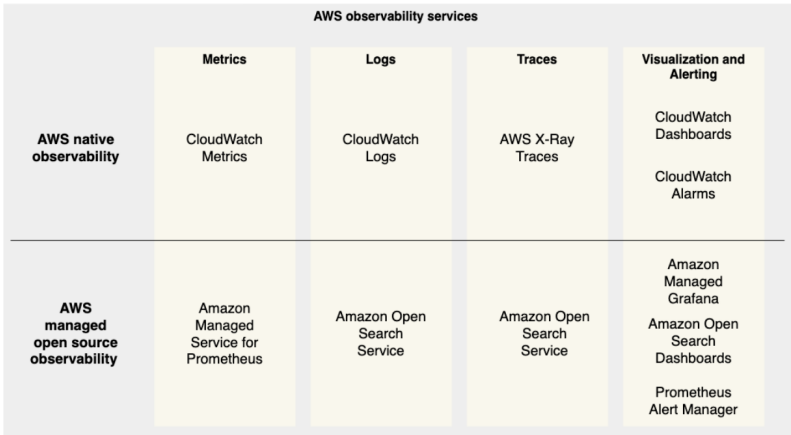

Figure 2.6 – AWS observability services

AWS offers several purpose-built solutions to monitor specific workload types and to cover certain use cases. The following figure depicts different features within CloudWatch that solve a variety of use cases:

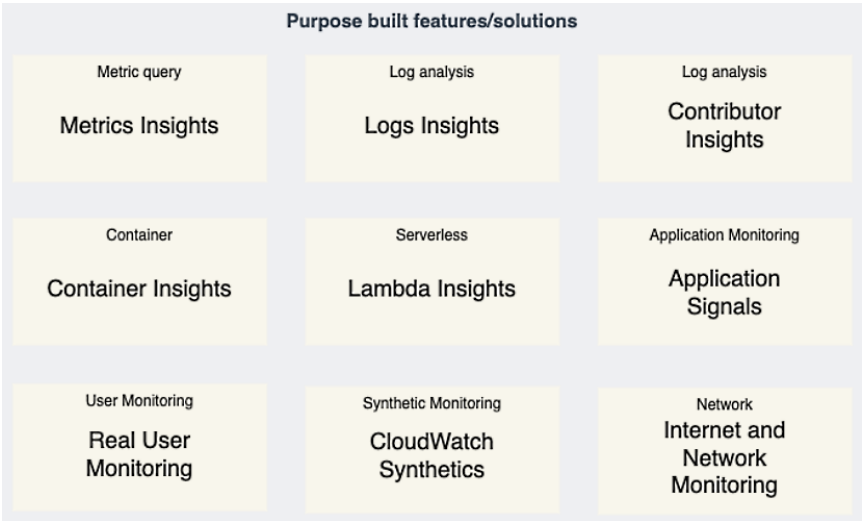

Figure 2.7 – Purpose-built features on Amazon CloudWatch

Amazon CloudWatch is a fully managed observability platform that offers a rich set of features for collecting, storing, and querying signals such as **metrics**, **logs**, and **traces** from any environment, whether it's AWS, on-premises, or third-party cloud service providers. CloudWatch Agent can be deployed in any of the aforementioned environments to collect signals.

CloudWatch has built-in out-of-the-box features that are purpose-built for specific use cases, such as **EC2 monitoring**, **container monitoring**, and **Lambda monitoring**.

Monitoring applications and infrastructure proactively

While setting up metric alarms, it is essential to leverage CloudWatch Anomaly Detection for Metrics. This is a powerful tool that leverages machine learning algorithms to identify deviations from normal patterns in your metrics. By continuously analyzing historical data, it can detect anomalies and potential issues, allowing you to take proactive measures before they impact your systems or applications. This advanced monitoring capability is designed to enhance the reliability and performance of your cloud infrastructure. Setting up alerts on Anomaly Detection allows you to be dynamic in terms of alerting and, as a result, increases the signal-to-noise ratio by reducing false alarms.

CloudWatch also allows users to continuously monitor by setting up synthetic monitoring through CloudWatch Synthetics. This allows you to write scripts to automate the process of testing your APIs, create websites for various use cases, and be alerted when their performance degrades. This strategy allows you to be aware of issues and address them even before your users find out.

Tracking the end user experience of your web application's usage can give you deep input on various factors of your application performance in the real world. CloudWatch Real User Monitoring offers invaluable insights into your web application's performance by actively monitoring and measuring page load times, resource utilization, and overall user experience. By leveraging this proactive approach, you can detect and resolve issues before they impact your users, ensuring a seamless digital experience and maintaining high customer satisfaction.

Debugging and troubleshooting issues using metrics, logs, and traces

The most important reason to collect signal data is to query it effectively when in need. Querying logs for analysis is a task that all developers and operators perform regularly to troubleshoot issues. Amazon CloudWatch Logs Insights empowers users to derive meaningful insights from their log data. This powerful tool leverages advanced analytics to identify patterns, trends, and anomalies in log data, enabling proactive problem identification and faster troubleshooting. By harnessing the capabilities of Logs Insights, organizations can make informed decisions and ensure optimal performance and reliability of their applications and systems.

During troubleshooting sessions in production environments, having the ability to trail log events while being able to easily identify events with specific keywords empowers root cause analysis and resolves problems quickly. Amazon CloudWatch Live Tail is a powerful tool that enables real-time monitoring and analysis of log data. It provides a continuous stream of log events, allowing DevOps engineers and administrators to proactively identify and troubleshoot issues, track application performance, and gain deeper insights into system behavior. By harnessing the capabilities of Live Tail, organizations can enhance their observability and ensure the smooth operation of their applications and infrastructure.

Similar to querying logs with Logs Insights, CloudWatch offers a SQL-like query language with Amazon CloudWatch Metrics Insights for querying metric data. Metrics Insights is a powerful tool that enables users to gain deep insights from their metric data. With its querying mechanism, users can easily analyze and identify patterns, trends, and anomalies in their metrics, allowing for proactive problem identification and faster troubleshooting. By leveraging the capabilities of Metrics Insights, organizations can optimize performance, detect issues before they impact users, and ensure the reliability of their applications and systems.

Correlating logs, metrics, and traces for easier root cause analysis

The ability to quickly get back to normal operations directly contributes to having higher reliability in your environment. This essentially means that you need to have a system in place that allows you to quickly identify issues by analyzing various signals that are collected such as logs, metrics, and traces, by deeply correlating the information contained within. This gives you a 360-degree view of an incident and helps you narrow down a specific piece of infrastructure, network, application, or code that causes issues. CloudWatch Application Signals empowers you to proactively monitor, troubleshoot, and optimize your applications effectively. By leveraging advanced analytics and real-time insights into logs, metrics, and traces, you gain unparalleled visibility into your systems.

In this section, we learned about the significance of proactive monitoring, effective debugging, and troubleshooting techniques to ensure the reliability and optimal performance of applications and infrastructure. By leveraging advanced tools such as CloudWatch Anomaly Detection for Metrics, Synthetics, and Real User Monitoring, organizations can identify deviations from normal patterns, potential issues, and performance degradation before they impact users, allowing for timely mitigation measures.

We also emphasized the importance of effective debugging and troubleshooting strategies. Tools such as CloudWatch Logs Insights, Live Tail, and Metrics Insights empower users to query and analyze log data, metrics, and traces, uncovering patterns, trends, and anomalies for faster problem identification and resolution. Additionally, CloudWatch Application Signals provides a comprehensive view by correlating logs, metrics, and traces, enabling deeper insights and facilitating efficient root cause analysis during incidents. The key takeaway is that proactive monitoring, effective debugging and troubleshooting, and correlating different data sources are essential for maintaining high reliability, performance, and observability in modern applications and infrastructure.

AWS-managed open source observability services

In addition to using AWS native observability services, users also can make use of managed open source services such as Amazon OpenSearch Service, Amazon Managed Service for Prometheus, and Amazon Managed Grafana to monitor their environments. These services are built on top of popular open source software that are widely used in the industry. Users have the option of deploying these open source software themselves by managing and operating the infrastructure.

However, many users find it very challenging to operate a reliable, efficient, and secure infrastructure that allows them to use this software at scale. With AWS-managed open source observability services, users can focus all their effort and attention on actually using the software in their environment rather than focusing on building infrastructure and deploying, patching, securing, scaling, and managing the environment themselves, which typically results in a higher **total cost of ownership** (**TCO**) in most cases.

Large-scale metrics collection and management in microservice environments

Amazon Managed Service for Prometheus is a fully managed Prometheus-compatible service that allows users to ingest, query, and alert on Prometheus metrics collected from any environment, including AWS, on-premises, and other cloud providers. Users can simply deploy a collector such as the Open Telemetry collector or Prometheus agent to collect metrics from their environments and send their metrics to an Amazon Managed Service for the Prometheus workspace they want to send to. Behind the scenes, AWS operates a large-scale open-source-based Cortex environment deployed across multiple AZs to provide reliability and high availability.

Log collection and analytics in a centralized environment

One of the popular open source log analytics solutions is OpenSearch. It is open source software created by Amazon as a fork from ElasticSearch when ElasticSearch changed the license of ElasticSearch software from the open source **Apache License** to a **Server Side Public License** (**SSPL**). AWS offers a managed offering called Amazon OpenSearch Service that users can use to create OpenSearch clusters that AWS will manage for them, including replacing faulty nodes that replace the clusters automatically when their performance degrades. Amazon OpenSearch Service allows users to run powerful queries to analyze large volumes of data with ease. The service also comes with advanced built-in analytical abilities for Log Analytics, Security Analytics, Machine Learning, and more, which makes it extremely useful for solving a variety of use cases.

Providing a centralized, single-pane view across your environments

It is common to find users deploying multitudes of observability platforms to monitor their environments to suit specific needs. The convenience of using the *right tool for the right job* can come at the cost of data silos and lack of visibility across systems. Users would be better served by deploying a platform that can consolidate data across systems and provide a single visualization pane with dashboards connected to various data sources. Grafana is a powerful open source tool that connects several dozens of data sources to query data from. AWS offers a fully managed Grafana service called Amazon Managed Grafana that users can use to consolidate their data across different sources. Amazon Managed Grafana makes it easy to set up, operate, and scale Grafana. It provides a unified view of metrics, logs, and traces from multiple sources, allowing you to troubleshoot issues quickly and effectively. With Amazon Managed Grafana, you can easily create dashboards and visualizations to monitor your applications and infrastructure, and you can set up alerts to notify you of any potential problems.

This section provided insightful information on the various managed open source observability services offered by AWS, all of which enable users to monitor diverse environments by collecting signals such as metrics, logs, and traces. These fully managed open source services present customers with an advantageous combination of a scalable and secure managed observability platform, while simultaneously offering the benefits associated with open source solutions. In the subsequent section, we will explore additional compute options, including containers and serverless services, that can be leveraged to construct resilient environments.

Extending resilience to containers and serverless

When it comes to building resilient applications on AWS, extending your resilience strategy beyond instances to containers and serverless services is crucial. Containers offer a lightweight and portable way to package and deploy applications. By utilizing containers, you can isolate application dependencies and ensure consistent and predictable behavior across different environments. Additionally, containers provide resource isolation, allowing you to run multiple applications on a single host while maintaining high availability and security.

Serverless computing, on the other hand, allows you to build and run applications without managing infrastructure. With serverless, you only pay for the resources your application consumes, eliminating the need to provision and maintain servers. This can significantly reduce the operational overheads and complexity associated with traditional infrastructure management. Furthermore, serverless services are typically designed to be highly scalable and fault-tolerant, making them an ideal choice for building resilient applications.

AWS offers a range of container and serverless services to support your resilience needs. **Amazon Elastic Container Service** (**Amazon ECS**) and **Amazon Elastic Kubernetes Service** (**Amazon EKS**) provide managed container orchestration services that make it easy to deploy, manage, and scale containerized applications. Amazon Lambda is a serverless compute service that allows you to run code without provisioning or managing servers. On the other hand, AWS Fargate is a serverless compute engine for containers that allows you to run containers without managing the underlying infrastructure.

By leveraging these services, you can build highly resilient applications that are scalable, fault-tolerant, and cost-effective.

Container environments are ephemeral by nature. You never treat a container as a pet but as cattle, which means you just replace the container resource when it isn't performing to your expectations. For example, if a Kubernetes Pod is running out of memory, you never go into the Pod to increase the allocated memory. Rather, you create a new Pod configuration and redeploy the resource, which results in the existing Pod being completely replaced by a new Pod with the new configuration with higher memory. This essentially means that it is dangerous to keep the state within the Pod memory as it can result in applications losing track of their state often.

Depending on the container orchestration technology, you always deploy more than one copy of the application at any given point to ensure redundant resources are always available to serve your customer requests. It is typical to see customers deploy multiple **Replicas** of their applications in a Deployment configuration in Amazon EKS. In this case, Kubernetes deploys copies of Pods and also ensures each Pod is deployed in different underlying nodes to ensure high availability and resiliency. Please refer to the Kubernetes documentation (`https://kubernetes.io/docs/concepts/workloads/`) to learn about concepts such as Pods, Deployments, and more.

All the concepts and logic behind building resilient architectures on virtual machines or other environments apply to container environments as well. This includes emphasizing stateless architectures, maintaining redundancy, automating deployment orchestration, closely monitoring performance and health metrics through monitoring platforms, reducing the blast radius through infrastructure isolation, and more.

Summary

In this chapter, we delved into implementing resilient compute and auto scaling solutions on AWS. We emphasized the significance of designing systems to withstand failures by incorporating redundancy and fault tolerance into compute resources. We also explored various factors that can disrupt system stability, including resource issues, service disruptions, application and code issues, security threats, and environmental factors. Then, we discussed key principles and strategies for addressing these factors, such as multi-AZ deployments, redundant environments, and stateless architectures, providing practical examples and architectural illustrations. After that, we introduced AWS Auto Scaling as a solution for dynamic resource management, explaining its key components and benefits. Furthermore, we discussed cost-saving strategies using Spot Instances and Reserved Instances, offering insights into their advantages and effective management. The importance of monitoring and maintaining a healthy infrastructure was highlighted, and we explored AWS observability services. Finally, we touched on extending resilience to containers and serverless architectures, discussing relevant AWS services and the applicability of resilience principles in these environments.

You've now learned how to design highly resilient, large-scale internet systems by leveraging various AWS services through redundancy while optimizing cost. While this chapter primarily focused on compute, in the next chapter, you will learn how to make use of various techniques so that you can design and maintain highly resilient data through security and best practices.

3

Securing and Backing Up Critical Data

Within every organization, it is imperative that each and every team member prioritize security as their paramount role requirement. Regardless of the user-friendliness of the software, the aesthetic appeal of the user interface, or the efficacy of the product in addressing specific customer needs, users will cease utilizing the software upon discovering that the system lacks adequate security measures. Safeguarding infrastructure and applications constitutes an ongoing endeavor. Even the simplest application catering to a limited number of users must be secured in every possible aspect.

In this chapter, we will learn about the importance of access control to ensure data security, layering backup strategies to increase reliability, making use of multi-region models to improve data availability, automating recovery mechanisms, and best practices to follow in **disaster recovery drills** (**DR drills**). With the learnings from this chapter, you will be able to design systems that support high data resiliency and availability by using various AWS services.

The chapter will have the following headings:

- Data security as resilience foundation
- Layering backup strategies for reliable resilience
- Embracing multi-region and geo-replication
- Continuous monitoring and recovery orchestration
- Disaster recovery planning and drills

Let's get started!

Data security as resilience foundation

In addition to securing the network from external threats, writing secure application code that thwarts **injection attacks** (where attackers exploit vulnerabilities in an application to send malicious code into a system) or cross-origin attacks (where a malicious script that doesn't belong to the source domain attacks a website), which typically protects threats originating from outside organizations while providing data security in transit, it is also important to secure data at rest. Several studies have shown that bad actors can also originate from within organizations and have exploited poorly secured data storage environments (`https://arcticwolf.com/resources/glossary/threat-actor/`).

Essentially, securing data in transit and data at rest are both critical to ensuring that data integrity is maintained.

Taking precautions and preventive measures is critical to providing data security in large organizations. With the increasing reliance on technology and the vast amounts of sensitive data being processed and stored, organizations face a constant threat of cyberattacks and data breaches. Implementing robust security practices is essential to protect against unauthorized access, data theft, and other security incidents that can have severe consequences.

Implementing access control mechanisms such as multi-factor authentication, role-based access control, privileged identity management practices, and strong password policies is crucial to ensuring that only authorized individuals have access to sensitive data. Additionally, encrypting data at rest and in transit, using secure communication channels, and regularly backing up data are vital measures to safeguard information.

Furthermore, organizations should establish and enforce a comprehensive security policy that outlines the security measures employees must adhere to. Conducting regular security audits, vulnerability assessments, and penetration testing can identify weaknesses and vulnerabilities in the security infrastructure, allowing organizations to address them promptly.

Educating employees about security risks and best practices is also critical. Regular security awareness training programs can help employees understand their role in protecting data and identify potential threats.

By taking proactive measures and adhering to best practices, organizations can create a solid foundation for data security. This not only protects sensitive information but also ensures operational continuity, maintains customer trust, and mitigates the risks associated with data breaches.

AWS offers a variety of database options for their customers. Users can pick and choose the right database for their use case based on their needs.

Whether the requirement demands a relational database that prioritizes consistency over availability, a document database that scales horizontally to support flexible storage options by prioritizing availability, or perhaps the application needs an in-memory database that offers response times in microseconds, a graph database that offers superior scalability and security, or a purpose-built time series database to easily analyze historical data using SQL queries – in all such instances and more, you are in good hands with AWS.

Let's now look at some of the typical AWS services used for data storage:

- Amazon **Relational Database Service** (**RDS**)
- Amazon **DynamoDB**
- Amazon **DocumentDB**
- Amazon **Simple Storage Service** (**S3**)
- Amazon **Redshift**
- Amazon **Neptune**
- Amazon **Timestream**
- Amazon **ElastiCache**
- Amazon **MemoryDB** for **Redis**

Controlling access to data

One of the fundamental best practices for securing data is by controlling access to the data.

Depending on the database type, techniques used to secure data can vary slightly. Securing access to databases is crucial to protecting sensitive information and maintaining data integrity. One key measure is to implement access control mechanisms such as **user authentication** and **authorization**. This involves creating individual user accounts with strong passwords and assigning specific permissions to each user based on their roles and responsibilities. Additionally, it's important to regularly review and update user privileges to ensure that they are aligned with current business requirements. Furthermore, implementing features such as two-factor authentication and role-based access control can provide additional layers of security. By controlling access to Amazon RDS databases, organizations can minimize the risk of unauthorized access and protect their data from potential threats.

You can use secure **virtual private cloud** (**VPC**) design practices to protect your database from threats. One of the fundamental network design principles to follow when deploying databases such as Amazon RDS is to deploy them in a private subnet within an AWS VPC. In addition to this, there are several other important design principles to consider:

- **Multi-AZ deployment**: Place your RDS instances across multiple **Availability Zones** (**AZs**) for high availability and fault tolerance. This ensures that if one AZ fails, your database can automatically fail over to the standby instance in another AZ.
- **Subnet groups**: Create a dedicated subnet group for your RDS instances. This group should include private subnets from at least two AZs to support multi-AZ deployments and provide redundancy.
- **Security groups**: Implement strict security group rules to control inbound and outbound traffic to your RDS instances. Only allow necessary connections from application servers or other authorized sources.

- **Network ACLs**: Use **network access control lists** (**NACLs**) as an additional layer of security at the subnet level. Configure NACLs to allow only essential traffic to and from your database subnets.
- **VPC peering or Transit Gateway**: If you need to access the database from multiple VPCs, consider using VPC peering or AWS Transit Gateway to establish secure connections between VPCs.

These principles help ensure secure and resilient database deployment within your VPC.

For S3 buckets, controlling access is vital for ensuring the security of sensitive data. One approach is to enforce user authentication and authorization mechanisms. This involves creating individual user accounts with strong passwords and defining specific permissions for each user based on their roles and responsibilities. By implementing access control lists, organizations can control which users and groups have access to specific objects within the bucket. Additionally, leveraging features such as server-side encryption and bucket policies can enhance data protection and prevent unauthorized access to sensitive information. You can learn about the different access management options available for Amazon S3 here: `https://docs.aws.amazon.com/AmazonS3/latest/userguide/access-management.html#access-management-tools`.

Overall, the techniques deployed involve the following:

- Deploy strict VPC network design principles that prevent unwanted access
- Identify users that need access to data and classify them based on their needs
- Create strong passwords and rotate the passwords frequently
- Enable two-factor authentication for users that require AWS console access
- Use strict **identity and access management** (**IAM**) permissions to restrict unwanted users from accessing data
- Enable data access tracking through auditing services such as AWS CloudTrail and monitor anomalous access by setting up alerts

In this section, we learned about the importance of securing critical data through access control. This is the foremost step in protecting any infrastructure, but it becomes even more significant when it comes to data protection as it involves very serious consequences when compromised. Having your teams adequately trained in not only employing proper access control but also continuously auditing the implementation is paramount to maintaining the desired security.

Encryption, intrusion detection, and prevention

Encryption ensures that even if data is accessed without authorization, it will be unintelligible to the adversary. Encryption methods scramble data, rendering it unreadable without the correct decryption key. AWS provides robust encryption solutions, such as AWS **Key Management Service** (**KMS**), facilitating secure key management. Encryption at rest (for data stored on volumes or in databases) and in transit (for data moving across networks) are both essential components of a resilient data security strategy. Encrypted data, even if breached, remains secure, minimizing the impact of the incident.

Even with the best access controls and encryption, the possibility of intrusions remains. **Intrusion detection systems** (**IDSs**) and **intrusion prevention systems** (**IPSs**) are vital for safeguarding against unauthorized activities in the AWS environment. Tools such as Amazon **GuardDuty** offer intelligent threat detection using machine learning, user behavior analysis, and other techniques. IDSs and IPSs alert administrators of suspicious activities, provide insights into security incidents, and allow for rapid incident response, crucial for ensuring operational continuity in the case of an attack.

Resilience advantages of a secure data strategy

The benefits of investing in a strong data security posture within AWS extend far beyond data protection itself. The following are some of the advantages of a secure data strategy:

- **Protection against data loss**: The primary purpose of data security is the prevention of data loss, whether due to cyberattacks, accidental deletion, or hardware failures. By implementing access controls, encryption, and intrusion detection, organizations dramatically reduce the chances of data becoming unavailable or unusable. Data backups, a key aspect that we'll expand on later (in the *Layering backup strategies for reliable resilience* section), become the crucial last resort if disaster strikes.
- **Maintaining operational continuity**: Data underpins business operations in modern organizations. Disruption of data can bring key processes to a standstill. Data security practices minimize the potential for such disruptions. Even in cases where a breach does occur, the damage is often contained thanks to access controls and encryption, allowing business operations to continue with minimal or controlled impact.
- **Rapid recovery**: Data breaches or disasters necessitate swift recovery to mitigate damage. Effective incident response plans are a key pillar of resilience. Data security, particularly when involving encryption, aids in rapid recovery by ensuring that compromised data cannot be readily exploited, thus buying the organization valuable time to investigate and remediate. Moreover, with stringent access control and intrusion detection, organizations gain better visibility into the nature of a breach, allowing for faster and more precise corrective actions.
- **Compliance and trust**: In numerous industries, data security is not merely a best practice but a legal necessity. Regulatory frameworks such as the **Health Insurance Portability and Accountability Act** (**HIPAA**) of 1996, **General Data Protection Regulation** (**GDPR**), and **Payment Card Industry Data Security Standard** (**PCI-DSS**) have strict data protection requirements. Robust data security within AWS demonstrates compliance efforts and instills trust among customers, partners, and regulators. This trust is a key component of resilience, ensuring continuity of operations and partnerships in even the most challenging environments.

Throughout this section, we have explored the implementation of security best practices as a crucial mechanism for data protection and integrity maintenance. These measures serve as effective safeguards against data breaches and unauthorized system infiltration. However, it is imperative to recognize that other significant threats exist, such as data loss resulting from hardware and software failures, which can lead to catastrophic consequences beyond the scope of traditional access control measures.

In the subsequent section, we will delve deeply into a comprehensive set of best practices designed to mitigate the impact of data loss on your business operations. This exploration will provide valuable insights into proactive strategies and robust solutions that can enhance your organization's resilience against unforeseen data loss incidents.

Layering backup strategies for reliable resilience

The integrity and accessibility of data are absolutely paramount for business continuity in the cloud. Disruptions, ranging from inadvertent deletions and hardware failures to targeted cyberattacks, can render vital data unavailable, leading to operational roadblocks and even financial losses. To ensure resilience in AWS, it's essential to meticulously plan and implement a multi-pronged backup and recovery strategy. Such a strategy ensures that in the face of any incident, data can be restored promptly, allowing operations to resume with minimal disruption.

Here are the common backup strategies generally followed in the industry:

- **Full backups**: This most comprehensive backup type creates a complete replica of your entire dataset at a specific point in time. Full backups offer a surefire way to recover all data in case of major disasters or extensive data corruption. However, they can be time-consuming and storage-intensive, especially for large datasets.
- **Incremental backups**: Incremental backups are designed to improve efficiency and reduce storage costs. After an initial full backup, incremental backups only capture changes made to the data since the last backup of any type (full or incremental). These changes are added to the backup chain, significantly reducing the volume of data that needs to be backed up in each cycle.
- **Differential backups**: Differential backups offer a middle ground between full and incremental backups. After the initial full backup, differential backups store all changes made since that initial full backup. Hence, differential backups tend to grow in size with each cycle, but still usually remain smaller than subsequent full backups.
- **Continuous data protection** (**CDP**): CDP is a near-real-time backup approach. Instead of scheduled backups, CDP systems constantly capture changes to data as they occur. This provides the most granular recovery points, minimizing the amount of data potentially lost in an incident. However, CDP solutions can be more complex and can impose greater performance overhead on production systems.

In summary, backup strategies encompass various approaches to protect data efficiently. Incremental backups capture only changes since the last backup, reducing storage needs and backup time. Differential backups store all changes since the last full backup, offering a balance between storage efficiency and ease of restoration. CDP provides near-real-time backups by constantly capturing data changes, offering the most granular recovery points but potentially increasing system complexity and overhead. Each strategy presents unique trade-offs between storage efficiency, recovery speed, and system impact, allowing organizations to choose the most suitable approach for their specific needs and resources.

Implementing layered backups in AWS

AWS provides a suite of tools and services to facilitate the implementation of your layered backup strategy. Layered backups in AWS leverage multiple services to create a comprehensive data protection strategy. This approach combines various backup methods and storage options to ensure data resilience and flexible recovery options. By utilizing services such as S3, EBS snapshots, Storage Gateway, and AWS Backup, organizations can implement a multi-tiered backup system that addresses different recovery time objectives and retention needs. This strategy allows for quick restoration of recent data, long-term archiving, and protection against various failure scenarios. Layered backups in AWS enable businesses to balance performance, cost, and security while maintaining robust data protection across their cloud infrastructure. AWS offers a variety of services that can be used for backing up your data. In the following list, we will look at those services to understand how they can be made use of to help with different use cases:

- **Amazon S3**: S3 is a highly scalable, durable, and cost-effective object storage service, well-suited for storing backups. It offers versioning and lifecycle policies to manage backup retention and automate the transition of older backups to lower-cost storage tiers such as S3 Glacier.
- **Amazon EBS snapshots**: **Elastic Block Store** (**EBS**) snapshots provide point-in-time backups of EBS volumes attached to EC2 instances. Snapshots are incremental in nature and are an excellent choice for protecting frequently changing data.
- **AWS Backup**: A centralized service, AWS Backup manages and automates backups across various AWS storage services including Amazon **Elastic File System** (**EFS**), Amazon Relational Database Service (RDS), DynamoDB, and more. It simplifies the creation of backup plans and lets you centrally monitor your backups.
- **Storage Gateway**: AWS Storage Gateway offers hybrid cloud storage solutions. This can be employed to create local backups of data in your on-premises environment, providing an additional layer of redundancy.

AWS offers a range of services for comprehensive backup solutions. Amazon EBS snapshots provide point-in-time backups for EC2 instance volumes, ideal for frequently changing data. AWS Backup centralizes and automates backup management across various AWS services, simplifying the process and enabling centralized monitoring. AWS Storage Gateway facilitates hybrid cloud storage solutions, allowing for local backups of on-premises data and adding an extra layer of redundancy. These services collectively enable businesses to create robust, scalable, and efficient backup strategies tailored to their specific needs and infrastructure requirements.

Designing your AWS backup strategy

As businesses increasingly rely on cloud infrastructure, developing a robust backup strategy for your AWS environment has become crucial. A well-designed backup plan can safeguard your data, ensure business continuity, and mitigate the impact of unexpected events, such as system failures, human errors, or cyber threats.

When designing your AWS backup strategy, consider the following key elements:

- **Data criticality and retention requirements**: Identify the most critical data and applications in your AWS environment and determine the appropriate backup frequency and retention periods. This will help you prioritize your backup efforts and ensure that your most valuable data is protected.
- **Backup storage and durability**: Leverage AWS services such as Amazon S3, Amazon Glacier, or AWS Backup to store your backup data. These services offer varying levels of durability, cost-effectiveness, and accessibility to meet your specific needs.
- **Backup automation and scheduling**: Automate your backup processes to ensure consistent and reliable data protection. Leverage AWS services such as AWS Backup or Amazon Data Lifecycle Manager to schedule and manage your backups.
- **Backup monitoring and testing**: Regularly monitor the health and integrity of your backups, and periodically test your ability to restore data from the backups. This will help you identify and address any issues before they become critical.

By following these best practices, you can design a comprehensive AWS backup strategy that safeguards your data, ensures business continuity, and aligns with your organization's specific requirements.

Backup validation and disaster recovery testing

A backup strategy is only as good as your ability to execute a recovery. Backup validation and **disaster recovery** (**DR**) testing are critical processes for ensuring data integrity and system resilience. Regular tests verify the reliability of backups and the effectiveness of recovery procedures. These practices help identify potential issues, streamline restoration processes, and provide confidence in an organization's ability to recover from various disaster scenarios, minimizing downtime and data loss risks. Next, we will look at a couple of important activities that you need to follow to improve backup efficiency:

- **Regular backup validation**: It is crucial to not simply presume that your backup systems are operational. Validate their functionality through regular testing. Use AWS Backup to set up automatic test restores or create manual snapshots and perform trial recoveries on a regular cadence.
- **DR drills**: Comprehensive DR drills based on set **recovery time objectives** (**RTOs**) and **recovery point objectives** (**RPOs**) serve to test your overall resilience posture. Simulate disruptions, practice restoring data and systems from backups in isolated environments, and identify gaps in your DR plans.

Additional considerations for managing backup data include the following:

- **Security**: Data backups must be protected as rigorously as the operational data itself. Encrypt backups both at rest and in transit, and carefully control access using IAM policies.

- **Regulatory compliance**: Ensure your backup strategy addresses any industry-specific regulations your organization must comply with (e.g., HIPAA for healthcare data).
- **Automation**: Automate the creation and management of backups using AWS Backup or custom scripting solutions. Automation minimizes manual errors and streamlines operations. **Runbooks** are essential for automating DR in AWS environments. These predefined procedures streamline recovery processes, reducing errors and response times. By codifying complex workflows, runbooks enable consistent, repeatable actions across AWS services. Integrated with **AWS Systems Manager Automation**, they allow the programmatic execution of recovery tasks, accelerating service restoration and minimizing downtime. Runbooks also facilitate regular testing, ensuring DR plans remain effective as infrastructure evolves.

In this section, we learned about the importance of layering your backup strategy through different AWS services and about some of the critical actions/best practices you need to follow to improve the effectiveness of the backup strategies. In the following section, we will learn about how you can deploy a multi-region approach to further improve resilience in data availability.

Embracing multi-region and geo-replication

In an era marked by increasing interconnectedness, the potential for disruptions remains a lingering concern. Localized failures, regional outages, and even widespread natural disasters can threaten the availability and integrity of data stored and processed within a single geographic location. To combat this and safeguard mission-critical applications, modern cloud architectures must adopt strategies that distribute data across multiple regions for superior resilience.

Let's take a look at the architectural patterns and AWS services that enable the implementation of resilient multi-region and geo-replication data solutions. We'll dissect various replication techniques, their suitability in different scenarios, and the complexities of ensuring data consistency in geographically distributed environments.

The case for geographic redundancy

Before delving into the technical aspects, let's solidify the case for multi-region and geo-replication strategies:

- **Regional outages**: While AWS Regions are designed with high availability in mind, occasional service failures or disruptions affecting the entire Region cannot be entirely dismissed. Distributing your data across multiple regions offers protection against such events.
- **Natural disasters**: Large-scale natural disasters such as earthquakes, floods, or hurricanes can cripple infrastructure in a specific geographic area. Replicating data outside that affected region safeguards your operations.
- **Compliance**: Regulations in certain industries mandate the storage and processing of data across multiple locations for redundancy. Geo-replication, when done correctly, can help satisfy such compliance needs.

- **Performance and latency**: Placing data closer to end users in geographically distributed locations can improve application responsiveness and deliver a better user experience, especially for global audiences.

Multi-region data distribution in AWS addresses several critical concerns for businesses. It provides resilience against regional outages and natural disasters by ensuring data availability even if one region is compromised. This approach also helps meet compliance requirements in industries that mandate data redundancy across multiple locations. Additionally, geo-replication can enhance performance and reduce latency for global users by placing data closer to end users. By leveraging multiple AWS regions, organizations can create a robust, compliant, and high-performing infrastructure that safeguards operations and improves user experience across diverse geographical locations.

Understanding replication techniques

AWS offers a versatile toolbox for replicating data across regions. The techniques to be employed will depend on the type of data and your recovery objectives:

- **Storage-level replication**: Services such as Amazon S3 offer **cross-region replication** (**CRR**), enabling the automatic, asynchronous replication of objects across buckets in different AWS regions. This approach is ideal for replicating datasets stored in object storage.
- **Database replication**: For relational databases, AWS RDS provides multi-AZ deployments with synchronous replication for high availability within a region and the option for asynchronous read replicas in other regions. Services such as Amazon Aurora leverage a shared storage architecture enabling near real-time replication across regions.
- **Asynchronous replication (general)**: In scenarios where real-time, synchronous replication is infeasible or unnecessary, asynchronous replication techniques are used. These might involve custom solutions using AWS Lambda, Amazon SQS/SNS, or data streaming services such as Amazon Kinesis for continuous replication of data changes in a loosely coupled manner.

As you can see, AWS offers various replication strategies for multi-region data distribution. AWS RDS supports multi-AZ deployments and asynchronous read replicas. Amazon Aurora uses shared storage for near-real-time cross-region replication. Asynchronous replication can be implemented using AWS Lambda, Amazon SQS/SNS, or Amazon Kinesis, allowing tailored multi-region strategies.

Active-active versus active-passive architectures

When utilizing multiple regions, you must decide how traffic is routed and how data is kept consistent. High availability is crucial for modern systems to ensure continuous operation and minimize downtime. Two common approaches to achieving this are active-active and active-passive architectures. These designs differ in how they distribute workloads and handle failures, offering varying trade-offs between resource utilization, complexity, and failover speed. Understanding these architectures helps

organizations choose the most suitable strategy for their specific needs, balancing factors such as performance, cost, and reliability.

- **Active-passive**: In this configuration, one region serves as the primary, handling all traffic. The second region acts as a standby, replicating data for failover scenarios. This is typically a simpler model with minimal consistency-related challenges.
- **Active-active**: In an active-active setup, both (or multiple) regions actively serve traffic. Distributing load with services such as Amazon Route 53 is common. However, this introduces complexities with regard to managing concurrent updates and conflict resolution between the regions.

Dealing with data consistency

Data consistency becomes a critical challenge in multi-region architectures. Data consistency in backup strategies is a critical aspect of ensuring the reliability and usability of backed-up information. It refers to the state where all data in a backup accurately reflects the source system at a specific point in time. Achieving data consistency is vital for maintaining the integrity of backups, especially for complex systems with interdependent components or frequently changing data. Proper consistency measures help prevent data corruption, ensure successful restorations, and support effective DR processes.

Various techniques and considerations come into play when designing backup strategies that prioritize data consistency across different types of systems and applications. Let's take a look at some of them:

- **Eventual consistency**: With asynchronous replication, a degree of lag between writes in the primary region and their visibility in the replica region is inherent. Applications need to be designed with tolerance for eventual consistency.
- **Strong consistency**: Where strong consistency is mandatory, synchronous replication techniques are required. However, this introduces trade-offs between performance (latency) and consistency guarantees.
- **Conflict resolution**: In active-active scenarios, concurrent writes to the same data in different regions can happen. Strategies such as *last-writer-wins* or custom reconciliation logic may be needed.

Multi-region cloud architectures face data consistency challenges, including managing eventual consistency, balancing strong consistency with performance, and handling conflicts in active-active setups. Successful implementation requires designing applications to accommodate these models, incorporating conflict resolution, and balancing consistency, performance, and application requirements for robust systems.

AWS services for multi-region and geo-replication

AWS offers a range of services for multi-region and geo-replication strategies, enabling businesses to enhance their global presence, improve DR capabilities, and reduce latency for users worldwide.

These services allow organizations to replicate data and applications across multiple AWS regions, ensuring high availability and data durability. By leveraging AWS's global infrastructure, companies can implement robust backup and failover mechanisms, comply with data sovereignty requirements, and optimize performance for geographically dispersed users. Multi-region and geo-replication solutions in AWS provide the foundation for building resilient, scalable, and globally distributed architectures. Let's dissect the key AWS services that are instrumental in implementing multi-region data architectures:

- **Amazon S3 CRR**: CRR provides object-level replication across S3 buckets in different regions. It can be configured with granular policies, including the ability to replicate only objects with specific tags or prefixes. S3 offers versioning and lifecycle management features for managing backup retention and controlling storage costs.
- **Amazon RDS read replicas**: For managed relational databases, read replicas can be deployed in different regions asynchronously. They improve read query performance and provide a DR option. RDS multi-AZ deployments within a region offer synchronous high availability.
- **Amazon Aurora global database**: Aurora's global database capability offers near-real-time replication across regions with dedicated read replicas for scaling read-intensive workloads. These replicas also offer regional failover capabilities.
- **Amazon Route 53**: This highly available DNS service can be used to distribute traffic intelligently across regions, for both failover (active-passive) and load balancing (active-active) scenarios. Route 53 supports geo-proximity routing (crucial to adopt data residency requirements), latency-based routing, and weighted routing policies for flexible traffic management.
- **AWS Lambda, SQS/SNS, and Kinesis**: For custom replication scenarios, serverless functions (Lambda), messaging queues (SQS), notification services (SNS), and real-time data streaming (Kinesis) can be combined to build tailored data replication pipelines, often used to push real-time changes from one region to another.

Multi-region architectures in AWS leverage a diverse set of services to achieve robust data replication and distribution. Key learnings include the flexibility of S3 CRR for object storage, the scalability of RDS and Aurora for database replication across regions, and the critical role of Route 53 in managing traffic distribution. Additionally, AWS's serverless and messaging services provide customizable solutions for specific replication needs. These tools collectively enable businesses to create resilient, globally distributed systems that balance data consistency, performance, and DR capabilities. The key to success lies in understanding each service's strengths and combining them effectively to meet specific business requirements for global data management and availability.

Design considerations and challenges

Building a genuinely resilient multi-region architecture requires careful planning and consideration of several key factors. Multi-region architectures present both opportunities and complexities in designing resilient, high-performance systems. Key considerations include data consistency across regions, latency management, compliance with varied regional regulations, and cost optimization.

Challenges often arise in synchronizing data, managing traffic routing, ensuring uniform security measures, and handling the increased complexity of deployments and monitoring. Organizations must carefully balance the benefits of improved availability and DR capabilities against the added operational overhead and potential cost increases.

Successfully navigating these design considerations and challenges is crucial for implementing effective multi-region architectures that enhance global reach and system reliability. Several factors need to be considered to ensure your strategy is sustainable in the long term. Let's take a look at some of them:

- **Cost**: Geographic distribution of data and services involves additional costs. Storage in multiple regions, cross-region data transfer, and the potential for more complex architectures need to be factored into budgetary considerations.
- **Complexity**: Multi-region architectures inherently introduce greater operational complexity. Monitoring, replication management, and failover procedures require thoughtful design and automation to ensure smooth operation.
- **Application design**: Applications must be either architected with data consistency requirements in mind or refactored to work in an environment characterized by eventual consistency if asynchronous replication is chosen.
- **Network latency**: Synchronizing data over large geographic distances can introduce noticeable latency. Applications with stringent real-time requirements might need special design accommodations or might limit the degree to which synchronous replication is used.
- **Testing and validation**: Thorough testing of failover procedures and data restoration in your multi-region setup is vital. Ensure that you have robust DR plans and regular drills to test resilience.

Implementing multi-region AWS architectures involves challenges such as managing increased costs, handling complexity, adapting to data consistency requirements, addressing network latency, and ensuring thorough failover testing. Success requires careful planning, robust design, and regular testing to create resilient, globally distributed systems balancing performance, cost, and reliability.

A simple, resilient, global web application architecture

In this section, we will design a global web application that is designed based on a multi-region architecture providing higher resiliency. For simplicity's sake, we will assume that the website is a monolith that can be hosted on a single EC2 instance. The architecture will consist of the following components:

- **Load balancing**: Route 53 with latency-based routing allocates traffic among regions based on user proximity for optimal performance.
- **Active-active (or active-passive as needed)**: Application servers in multiple regions, potentially using **Auto Scaling groups** (**ASGs**) for elasticity.

- **Data layer**: You can choose your data layer from one of the following:
  - **Aurora global database** for strong consistency and near real-time cross-region replication
  - **RDS with cross-region read replicas** for a cost-optimized solution, accepting eventual consistency
- **Object storage**: S3 buckets in each region using CRR to maintain copies of static assets or user-uploaded files.

The following architecture diagram shows how these components are put together.

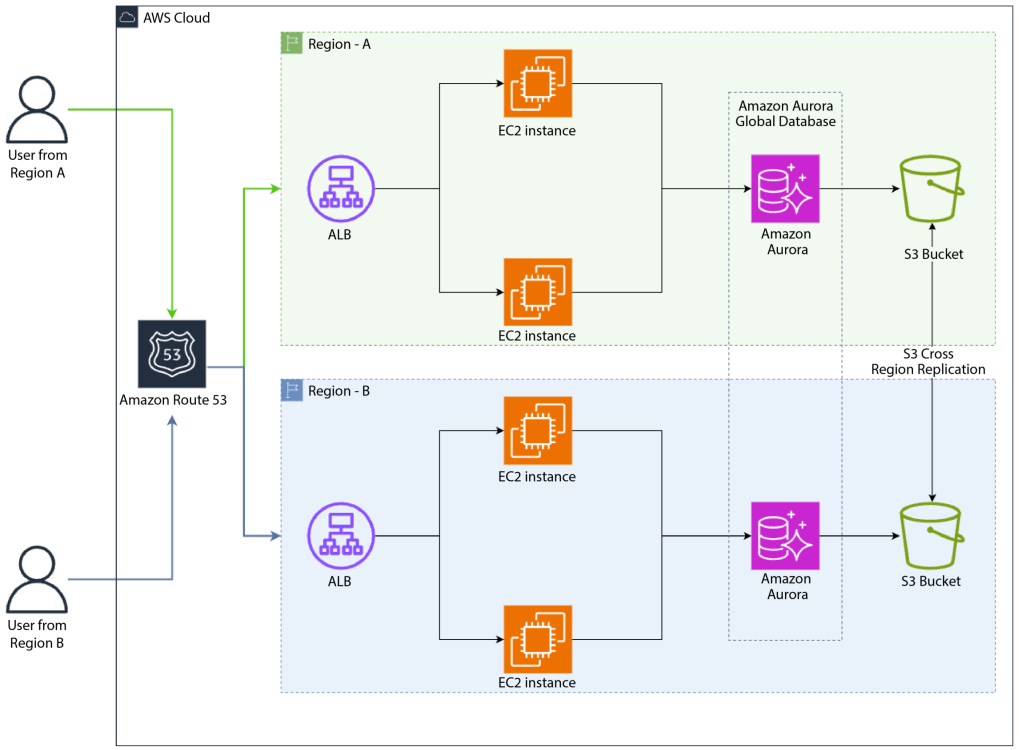

Figure 3.1 – Sample resilient, multi-region application architecture

As you can see, compute, database, storage, and load balancing are all distributed across AWS Regions for high availability and resilience. The chances of more than one AWS Region getting impacted are extremely unlikely given the historical performance of AWS infrastructure, which gives us greater assurance on the resilience of this architecture.

Continuous monitoring and recovery orchestration

The agility and flexibility of the cloud bring many rewards, yet they also present unique challenges for ensuring resilience. Reactive measures in response to data loss events or operational disruptions are often insufficient for today's always-on, mission-critical systems. Building a truly resilient AWS environment necessitates a shift toward a proactive, self-healing posture underpinned by continuous monitoring and intelligent recovery automation.

In this section, we'll investigate how a well-orchestrated automation strategy empowers you to detect anomalies early, mitigate data loss with self-healing processes, and maintain operational continuity in the face of disruptions. We will analyze essential best practices and explore how to leverage AWS services for effective implementation.

Three key areas form the backbone of an automated approach to resilience:

- **Continuous monitoring**: The crux of proactive defense is in setting up rigorous monitoring of key system metrics, logs, and infrastructure health. Establishing meaningful thresholds and baselines creates an invaluable reference point for identifying deviations that might signal impending issues.
- **Anomaly detection**: Monitoring data, armed with the right analysis techniques, allows for spotting anomalies indicative of potential failures. Employing machine learning models along with carefully designed rules can offer predictive insights, catching problems often before they fully manifest.
- **Automated recovery orchestration**: Self-healing systems take automatic corrective actions to address issues or failures. Predefined workflows and runbooks, executed through automation scripts or event-driven triggers, dramatically cut down the time required for recovery and minimize human intervention.

To summarize, proactive IT defense employs continuous monitoring, anomaly detection, and automated recovery. Monitoring establishes baselines, while anomaly detection predicts issues using this data. Automated recovery implements self-healing through predefined workflows. This strategy quickly identifies and addresses problems, minimizing downtime and manual intervention in IT environments.

Best practices for automation in resilience

Best practices for automation in resilience focus on leveraging technology to enhance system reliability, recovery speed, and overall operational efficiency. These practices aim to minimize human error, reduce response times, and ensure consistent execution of critical processes:

- **Holistic monitoring**: Instrument your applications, infrastructure, and services for all-encompassing visibility spanning network traffic, resource utilization (CPU, memory, I/O), API error rates, log analysis, and user activity.

- **Prioritize critical assets**: Tier your monitoring and recovery strategies based on the impact of various systems or datasets on your business operations. Focus resources on ensuring rapid response to anomalies in the most critical areas.
- **Baselines and thresholds**: Understanding *normal* system behavior is paramount in pinpointing anomalies. Establish clear baselines for relevant metrics and define meaningful thresholds that trigger alerts or initiate recovery actions.
- **Test and iterate**: Resilience automation is an iterative journey. Conduct regular testing of your monitoring configurations, automation scripts, and recovery workflows. Learn from these tests and make continuous improvements to increase your systems' self-healing efficacy.
- **Leverage managed services**: Opting for managed services where possible can reduce overheads. Such services typically have built-in monitoring and automated responses, taking some of the responsibility out of your hands.

AWS provides a powerful collection of tools for implementing continuous monitoring and recovery orchestration:

- **Amazon CloudWatch**: At the heart of AWS monitoring, CloudWatch collects metrics, logs, and events, providing the foundation for creating custom dashboards and setting trigger-based alarms. CloudWatch Logs Insights supports complex log analysis.
- **AWS CloudTrail**: AWS CloudTrail logs and monitors account activity across your AWS infrastructure. It records API calls, tracks user actions, and provides event history for auditing, compliance, and security analysis purposes.
- **Amazon GuardDuty**: This intelligent threat detection service utilizes machine learning and anomaly detection techniques to flag suspicious behavior on your AWS accounts, networks, and workloads.
- **AWS Lambda**: Lambda functions offer a versatile way to execute custom logic. Employ them to initiate recovery actions, adjust resource allocations, or trigger notifications in response to alarms or anomalies.
- **AWS Systems Manager**: Manage fleets of EC2 instances or on-premises servers with Systems Manager. Automation documents allow for the execution of well-defined recovery procedures or troubleshooting routines across multiple systems in response to events.
- **AWS Config**: Track resource configuration changes and receive notifications for non-compliance against established rules. Config can trigger corrective actions for remediation.
- **AWS Auto Scaling**: Automate the scaling in or out of EC2 instances based on metrics such as CPU load or request queues. This is an example of a self-healing mechanism for handling load variations, ensuring application responsiveness.

Effective resilience strategies in cloud environments involve holistic monitoring, prioritizing critical assets, setting baselines, testing resilience automation, and using managed services. These practices enhance anomaly detection, response efficiency, and self-healing capabilities, ensuring the high availability and performance of cloud services.

Automating data loss prevention and recovery

Automating data loss prevention and recovery focuses on implementing proactive systems to safeguard critical information and streamline recovery processes. This approach combines preventive measures with automated recovery solutions to minimize data loss risks and reduce downtime:

- **Automated backups with lifecycle management**: Use AWS Backup lifecycle management policies in conjunction with Amazon S3 lifecycle policies to manage storage tiers and retention durations. This optimizes costs, balancing the need to have historical recovery points with efficient storage management.
- **Anomaly detection on backup jobs**: Monitor CloudWatch or your orchestration tool's logs for backup failures or unexpectedly high changes in data volume. These deviations from the norm could signal issues that warrant attention, potentially related to data corruption or even ransomware activity.
- **Recovery validation**: Merely taking backups is not enough. Automated test restores into isolated environments. These tests should verify the integrity of data and measure the time taken, allowing you to refine procedures for actual recovery scenarios.

Effective cloud backup and recovery strategies involve automated backups with lifecycle management, anomaly detection, and regular recovery validation tests. This approach enhances data protection, improves disaster readiness, and ensures reliable restoration of critical systems and information during emergencies.

Scenario-based automation examples

Let's outline examples of how automation can respond to specific incidents related to data:

- **Sudden spike in database CPU utilization**:
  - **Monitoring**: CloudWatch detects the CPU spike, exceeding a defined threshold.
  - **Initial response**: A Lambda function triggers and scales up database resources (e.g., upgrades RDS instance type) to address immediate capacity issues.
  - **Incident analysis**: GuardDuty and additional logs from CloudWatch are analyzed to determine the cause of the spike. If necessary, a human-in-the-loop workflow kicks off for further troubleshooting.

- **EBS volume degradation**:
  - **Monitoring**: CloudWatch monitors EBS volume performance metrics (e.g., I/O operations, latency), so you can configure alarms to automate remediation actions to address the issue.
  - **Automated remediation**: A combination of Systems Manager runbooks and Lambda functions execute a predefined recovery plan. This might involve snapshotting the volume, creating and attaching a fresh EBS volume from the snapshot, and potentially failing over applications.
- **S3 object key changes**:
  - **Monitoring**: CloudWatch Events or S3 Event Notifications trigger actions based on modification of critical objects with specific prefixes.
  - **Anomaly detection**: Lambda functions analyze the nature of the change(s). If unexpected patterns suggest potential tampering or data corruption, automated actions can be taken to revert changes, alert administrators, and isolate affected objects to prevent further damage.

As discussed here, unique situations require specific solutions to be deployed. This includes using a variety of techniques and services to monitor and perform remediation actions. This is not an all-encompassing list and hence, as a user, you should innovate and devise new solutions that fit into solving your own architectures and scenarios.

Further considerations to improve recovery mechanisms

Following industry best practices to improve recovery mechanisms will help optimize operations. This includes identifying weak areas, attaining clarity in recovery workflow actions, and being able to automate infrastructure deployments. The following list provides some examples of such actions:

- **Chaos engineering**: Intentionally introduce *failure* scenarios into a controlled environment to rigorously test automation effectiveness. Netflix's Chaos Monkey is the inspiration for this practice; consider controlled variations within your AWS environment to increase confidence.
- **Prioritizing recovery workflows**: Define the sequence and priority of systems to be recovered during a significant data loss incident. This dictates the order of automated orchestration steps – for example, prioritizing recovering the identity management system before addressing a backend storage or database system.
- **Versioning and rollbacks**: Use version control for **infrastructure as code** (**IaC**) templates, Lambda functions, and other automation components. This facilitates safe rollbacks if any changes introduce unintended side effects.

Now that we have learned about continuous monitoring and recovery orchestration in detail, let's dive into the details of planning for DR and drills.

Disaster recovery planning and drills

Hope for the best, but plan for the worst.

This adage is remarkably apt when approaching resilience within complex cloud environments. While we meticulously build fault-tolerant architectures, implement robust backup strategies, and leverage a multitude of tools for redundancy and automation, the hard truth is this: disasters can still strike. The differentiating factor for organizations that endure such disruptions with minimal impact lies in their preparedness through DR planning and the relentless practice of DR drills.

This section delves deep into the methodology of designing effective DR plans tailored to your AWS workloads and, crucially, dives into the various types of drills and simulations that validate your resilience preparedness.

But before we go any further, let us look at why DR plans and drills matter:

- **Beyond theoretical protection**: Backup systems, multi-region architectures, and standby environments are only as good as they perform under pressure – during an actual disaster. Theoretical resilience is often far removed from the reality of restoring systems under stressful conditions.
- **Validating assumptions**: DR plans are rife with assumptions about recovery time, resource availability, and operational procedures. Drills expose the gaps between these assumptions and reality, allowing you to fix weaknesses before a crisis occurs.
- **Minimizing downtime**: The purpose of DR is to reduce operational downtime, and it is determined by the time taken to get critical systems back online. Effective drills help teams optimize their processes and refine automation, dramatically improving speed during a real disaster.
- **Muscle memory**: DR protocols should not be neglected documents gathering dust. Regular exercises verify that stakeholders and teams comprehend their responsibilities, actions, and channels of communication when encountering genuine disruptions.

Improving cloud recovery mechanisms involves proactive testing, prioritization, and version control. Use chaos engineering for controlled failure scenarios, prioritize recovery workflows, and employ versioning for IaC. These practices enhance resilience, boost recovery confidence, and enable efficient DR strategies.

Crafting your AWS disaster recovery plan

A comprehensive DR plan tailored to your specific resilience goals in AWS is the cornerstone of your preparedness. Here's a breakdown of key elements:

- **Identifying critical systems**: Not all systems have the same priority in a disaster. Prioritize mission-critical applications and datasets, classifying them into tiers based on their impact on business operations.

- **Defining RTOs and RPOs per tier**: Establish target RTOs and RPOs for each tier of criticality. These drive your choices of backup strategies, technologies, and failover procedures.
- **Outlining AWS recovery procedures**: Detail step-by-step procedures for infrastructure restoration, data recovery (database snapshots, S3 object versions), DNS failover (Route 53), and application redeployment in a recovery region.
- **Communication and escalation**: Clear communication is paramount during a crisis. Define contact information, escalation chains, and pre-scripted stakeholder notifications to ensure a coordinated response.
- **Testing and documentation**: Most importantly, your DR plan should include a schedule for regular testing and a mechanism for documenting results and updating the plan based on drill insights.

Effective DR in AWS requires prioritizing critical systems, setting RTOs and RPOs, detailing AWS recovery procedures, and establishing communication protocols. Regular testing and updates based on drills are essential to ensure swift, efficient responses, minimizing downtime and data loss while maintaining business continuity.

Types of disaster recovery drills

Select the types of drills that best align with your criticality tiers, recovery goals, and organizational maturity. Consider a gradual progression of complexity:

- **Checklist review/tabletop exercise**: Teams walk through the DR plan, visualizing the steps and validating communication channels without actually executing technical actions. This offers a lightweight way to ensure everyone understands their role.
- **Simulated component failure**: Isolate a non-critical component or a test environment and simulate its failure. Practice recovery procedures, observe system behavior, and validate backup integrity for that specific component.
- **Pilot light DR**: In this drill, a minimal version of your core applications and data is kept running in a secondary AWS region. Test the process of quickly scaling up this *pilot light* to full production capacity for rapid failover.
- **Warm standby DR**: Maintain a near-complete scaled-down replica of your production environment, regularly updated from backups. During this drill, practice scaling up the warm standby and verifying data consistency.
- **Full failover DR drill**: The most intensive exercise. Simulate a complete production outage and fail over to a secondary region. This comprehensively tests all recovery actions and allows for validating actual RTOs against your goals.

DR testing in AWS includes exercises such as checklist reviews, simulated failures, pilot light drills, warm standby exercises, and full failover drills. These methods validate DR plans at different complexity levels, ensuring preparedness and allowing teams to identify weaknesses and refine procedures for business continuity.

AWS tools to power your DR drills

AWS offers an array of features and services that assist in the design and execution of your DR drills:

- **AWS Elastic Disaster Recovery Service** (**AWS DRS**): AWS DRS is a comprehensive service designed to minimize downtime and data loss during unforeseen events. It continuously replicates servers, databases, and applications from on-premises or cloud environments to AWS. DRS enables rapid failover with minimal RPO and RTO, allowing businesses to quickly resume operations in the event of a disaster. The service offers cost-effective storage solutions and minimal compute requirements during non-disaster periods. With features such as automated recovery plans, non-disruptive testing, and point-in-time recovery, AWS DRS provides organizations with a robust, scalable, and efficient DR solution tailored to their specific needs.
- **Resilience Hub**: Gain a centralized view of your applications' resilience posture. Resilience Hub can conduct automated assessments and generate recommendations aligned with AWS best practices.
- **Fault Injection Simulator**: This newly released service allows you to intentionally introduce controlled disruptions into your AWS environment. Use it to simulate everything from EC2 instance failures to degraded database performance or API throttling.
- **Isolated testing environments**: Create separate VPCs or even dedicated testing accounts to stage DR drills. Use security groups and **network access control lists** (**NACLs**) to strictly isolate the drill environment from your true production infrastructure, reducing risks during testing.
- **IaC**: Tools such as AWS CloudFormation help you define both production and DR environments as templates. Automate the deployment and teardown of your DR environments for testing, ensuring consistency and speed during drills.
- **Route 53 failover**: For active-passive configurations, Route 53's health checks and failover routing assist in simulating the cutover of traffic to your DR region during a drill.

AWS enhances DR testing with tools such as Resilience Hub for centralized assessment, Fault Injection Simulator for disruption testing, and isolated testing environments. Using IaC and Route 53's failover capabilities, these tools help test DR strategies, identify weaknesses, and improve resilience.

Execution best practices for your drills

DR drills are one of the most important and very challenging activities to employ in organizations. When appropriate attention is not given, this could be a very time-consuming activity, which might eventually result in a lack of enthusiasm amongst team members. If not appropriately planned, a failed DR drill can even cause outages resulting in business impact. The following list covers some of the important best practice actions to be followed in this journey:

- **Start small**: Begin with simple checklist reviews and progress to more complex drills as your DR strategies and team confidence mature.
- **Pre- and post-mortem analysis**: Before a drill, set clear objectives and metrics. Afterward, conduct a rigorous analysis of successes, failures, and bottlenecks. Iterate and improve your DR plan based on these insights.
- **Treat drills like the real thing**: Enforce a sense of urgency during drills. This will help identify weaknesses that might not surface in casual walk-throughs and better prepare teams for stressful real-world scenarios.
- **Involve stakeholders**: Cross-functional participation is key. Include not only technical staff but also management and customer-facing teams, ensuring everyone's understanding of DR implications.
- **Don't fear failure**: Drills are bound to reveal shortcomings. Embrace them as valuable learning opportunities and proactively improve your resilience posture.

Start with simple checklist reviews and progress to complex drills, conduct pre- and post-mortem analyses, treat drills like real emergencies, involve cross-functional stakeholders, and embrace failures as learning opportunities. This systematic approach enhances DR capabilities and fosters continuous improvement in resilience.

Specific scenarios for DR drills

Let's illustrate how targeted drills can test different aspects of your AWS environment:

- **Regional failure simulation**: Purposefully disable services in an entire AWS region. Monitor automated multi-region failover, observe data replication behavior, and test recovery procedures for region-specific dependencies.
- **Ransomware attack drill**: Treat a test dataset as compromised and practice a *clean room* recovery strategy. This might involve restoring the latest viable backup in a new environment, quarantining affected accounts, and validating data integrity before going back online.
- **Dependency mapping drill**: Identify potential cascading failures and bottlenecks. Simulate the failure of seemingly minor components to analyze the ripple effects on your most critical systems.

Finally, let's look at some additional considerations to improve DR mechanisms:

- **Drill frequency**: The frequency of drills depends on the rate of change in your systems, the criticality of your applications, and regulatory requirements. Highly critical systems might warrant monthly or even weekly drills, while less critical ones could involve quarterly or yearly exercises.
- **Automation for speed**: As your DR strategies evolve, integrate automated recovery actions with tools such as AWS Lambda and Systems Manager runbooks. Drills test and refine this automation, driving down your RTO over time.
- **Cost control during drills**: Employ on-demand resources, lower-tier instance types, scheduled start/stop, and temporary adjustments to Auto Scaling configurations. This helps minimize expenses while ensuring drill environments are sufficiently representative of production.

Summary

In this chapter, we learned about various aspects of securing and backing up critical data. We learned about a variety of AWS services that can be used to create a layered backup strategy, how we can adopt a multi-region approach to increase data availability, various automation techniques that can be used for backup, and best practices for DR. This information will be useful for developers and architects to design infrastructure and applications that adopt security best practices for data access and also design systems that can take advantage of data-resilient backend systems.

In the following chapter, we will learn about how you can design systems that handle failure gracefully in such a manner that the failure is not directly exposed to the end user, resulting in business impact and customer satisfaction challenges.

4

Orchestrating Graceful Degradation

Graceful degradation, a fundamental design principle, aims to ensure the continued functionality of complex systems, even in the face of partial failure or significant disruptions. By incorporating this approach, systems can maintain a certain level of operability, albeit potentially with reduced capacity or performance, rather than experiencing complete collapse.

The primary objective of graceful degradation is to prevent catastrophic system failures, which can lead to substantial downtime, data loss, and compromised security. In a well-designed system, individual components or subsystems are designed to degrade gradually and predictably as they encounter failures or performance issues. This allows the system as a whole to continue operating, albeit with potentially diminished capabilities, until repairs or replacements can be made. In summary, graceful degradation is a technique that allows you to achieve resilience in your systems.

In this chapter, we will take a look at the following topics:

- Understanding graceful degradation through an example
- Identifying the fault – diagnosing partial failures and minimizing impact
- Isolating the wound – containment strategies to prevent cascading outages
- Streamlining recovery with preconfigured actions
- Leveraging ML and GenAI to enhance issue detection and response

Understanding graceful degradation through an example

Here is an example of a system that handles a system degradation graciously.

This architecture diagram (*Figure 4.1*) illustrates how a resilient system can handle a user requesting a file, with the ability to return the file from a cache if the underlying system fails or the file doesn't exist anymore:

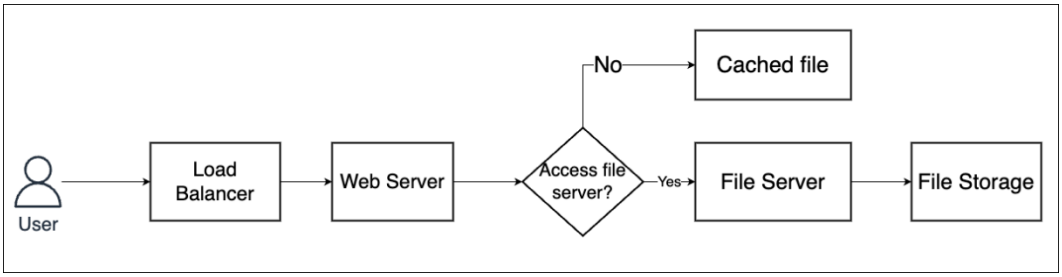

Figure 4.1 – Logical architecture diagram showing a decision being made to return a cached file when the target file server is unavailable

The following diagram (*Figure 4.2*) shows a simplified example sequence of actions that take place within the system when a user requests a file during a file server failure:

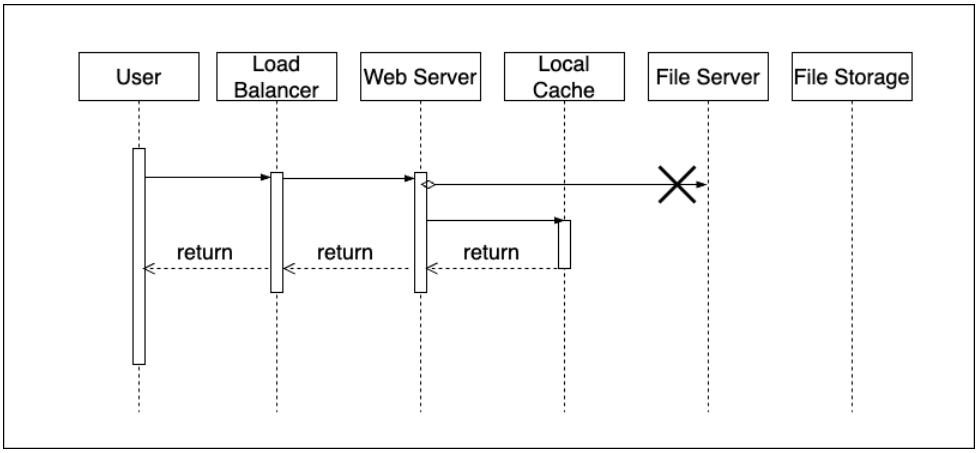

Figure 4.2 – Sequence diagram showing an example sequence of actions that could take place when a user requests a file during a file server failure

Let's walk through the details of what is happening here:

1. The user sends a request to the system to retrieve a file.
2. The load balancer sends the request to the web server to handle it.

3. The web server receives the user's request, checks and calls the file server to fetch the file, and finds that there is an error in completing the request.
4. The web server then fetches an already cached older file and returns it to the requester.

In this architecture, the requesting user is not faced with an ugly error message when the file server fails to process the request. Instead, the user receives the cached file potentially informing the user about the situation with a polite message on the user interface.

Graceful degradation is implemented in various domains:

- **Corporate systems**: In enterprise environments, graceful degradation is essential for maintaining **business continuity** (**BC**). Critical systems, such as **customer relationship management** (**CRM**) platforms, e-commerce websites, and financial transaction systems, are designed to degrade gracefully in the event of hardware failures, software glitches, or network outages. This ensures that while certain features or functionalities may become unavailable or slower, users can still access essential services and perform core tasks.
- **Security systems**: Graceful degradation is crucial for maintaining the integrity and effectiveness of security systems. **Intrusion detection systems** (**IDSs**), surveillance cameras, and **access control systems** (**ACSs**) are designed to continue functioning, even if specific components fail. This ensures that security breaches can still be detected and mitigated, albeit potentially with reduced coverage or responsiveness.
- **Web applications**: Graceful degradation is commonly employed in web development. Websites and web applications are designed to handle various scenarios, such as slow network connections, missing assets, or browser incompatibilities. By gracefully degrading, websites can ensure that users still have a positive browsing experience, even if certain elements or features are unavailable or slow to load.

To summarize, graceful degradation is a critical design principle that enables systems to maintain core functionality and operational integrity even during failures, resource constraints, or adverse conditions. By prioritizing essential services and adapting to limitations, this approach ensures that critical operations can continue uninterrupted, albeit with reduced capabilities. It plays a crucial role in maintaining BC, security, and consistent user experiences across various domains, including corporate environments, security infrastructure, and web development. Ultimately, graceful degradation significantly enhances system resilience, minimizes disruptions, and serves as a vital strategy in modern system design and **disaster recovery** (**DR**) planning.

Implementing graceful degradation requires careful design and engineering considerations:

- **Modularity**: Systems should be designed with modular components that can be independently tested and replaced. This allows for easier identification and isolation of failed components, minimizing the impact on the overall system.

- **Redundancy**: Redundancy is an important strategy to ensure the reliability of a system. In this approach, multiple instances of a component or service are deployed so that if one fails, another can immediately take over its tasks. For instance, in a distributed system, multiple instances of a particular service can be run simultaneously. If one instance fails, the load balancer can automatically redirect incoming requests to another instance, ensuring that the system continues to function seamlessly without any disruption.
- **Fault tolerance (FT)**: Systems should incorporate FT mechanisms, such as redundancy and failover, to handle component failures or performance degradation. This ensures that critical functions can continue to operate, even if specific components fail.
- **Monitoring and logging**: Robust monitoring and logging mechanisms are essential for detecting and diagnosing issues early on. By identifying potential problems promptly, proactive measures can be taken to mitigate their impact and prevent complete system failure.

Graceful degradation is a valuable design principle that enables systems to handle failures and disruptions with resilience. By implementing this approach, organizations can minimize downtime, maintain BC, and enhance the overall reliability and robustness of their systems.

In the upcoming sections, we will take a look at five major strategies that you need to implement in order to build a system that supports the principles of graceful degradation. The first of these is the ability to identify faults.

If only we were aware of the cause of the failure, we would be empowered to take appropriate corrective actions. Establishing comprehensive monitoring and observability measures within your environment is a critical contributing factor. In the ensuing section, we will investigate this topic in greater depth and acquire knowledge of some of the optimal practices to be followed for the identification of faults.

Identifying the fault – diagnosing partial failures and minimizing impact

In this section, we will focus on how you can use different techniques and **Amazon Web Services** (**AWS**) services to identify failures in the system in order to identify issues and reduce their impact on overall system stability.

AWS offers a variety of tools to collect signals from your environments, whether they are hosted on AWS services such as Amazon **Elastic Compute Cloud** (**EC2**), Amazon **Elastic Kubernetes Service** (**EKS**), Amazon **Elastic Container Service** (**ECS**), AWS Lambda, and so on or hosted on a non-AWS environment such as on-premises servers or on other **cloud service providers** (**CSPs**).

Mastering pinpoint accuracy in identifying the root causes of partial failures within your AWS environment can be a game-changer in minimizing troubleshooting time and expediting recovery. Here's how you can achieve this.

Log analysis through Amazon CloudWatch

Enable detailed logging throughout your AWS resources to capture valuable insights into partial failures. Capture both application and infrastructure logs to be able to get a complete understanding of the environment. Infrastructure logs such as Amazon **Virtual Private Cloud** (**VPC**) flow logs can be vital in understanding your infrastructure performance. If you are operating a container environment such as Kubernetes, you might want to consider collecting logs and events from the control plane in order to understand bottlenecks and other resource/capacity constraints that might affect your environment.

It is a good idea to design the monitoring architecture in such a way that you get the level of visibility required to operate the environment. CloudWatch allows you to designate a central monitoring account to which you can collect all signals such as logs, metrics, and traces free of cost in most cases. Refer to the *Cross-account observability* section on the CloudWatch pricing page for specifics and updated information on pricing (`https://aws.amazon.com/cloudwatch/pricing/`). CloudWatch Logs pattern analysis, CloudWatch Logs Anomaly Detection, and CloudWatch Logs Contributor Insights are features that you should strongly consider making use of to pinpoint issues easily and quickly. These features are built to help users identify issues by increasing the signal-to-noise ratio.

The following logical architecture diagram (*Figure 4.3*) shows how you can set up multiple layers of monitoring accounts to create centralized monitoring on CloudWatch in a distributed fashion. As you can see in the following diagram, you can set up your monitoring accounts to collect signals from different workloads and design the architecture in a way that works for you:

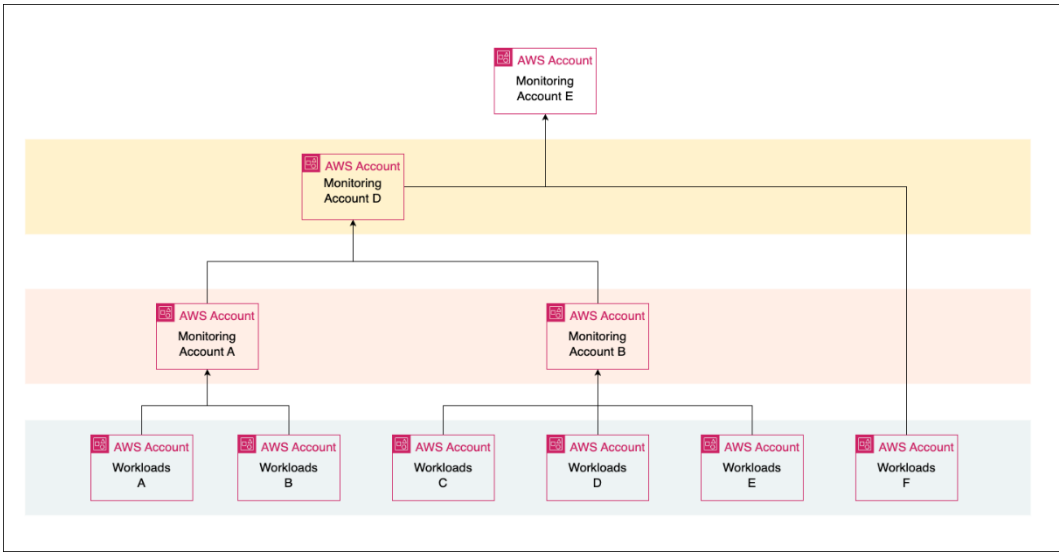

Figure 4.3 – An advanced CloudWatch central monitoring architecture

With flexible central monitoring options available, CloudWatch can be used to set up central monitoring of workloads for your specific needs. In the architecture in *Figure 4.3*, workloads A and B are being monitored from monitoring account A. Users who only have access to monitoring account A will be able to see data from both workload accounts A and B. Similarly, users with access to only monitoring account B will be able to access monitoring data from workload accounts C, D, and E. As you can see, the pattern continues in this fashion where users who have access to monitoring account E are those who can see all monitoring data from all accounts across your AWS environments. The AWS documentation provides details about setting up central monitoring on the docs page here: `https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch-Unified-Cross-Account.html`.

Performance monitoring

Implement performance monitoring tools such as Amazon CloudWatch to track key metrics such as CPU utilization, memory usage, and network latency. Determine the exact type of metrics to collect, how frequently you should collect those metrics, and what metadata should be included in the metrics. When using CloudWatch for metrics, you need to plan the dimensions in which you want to collect metrics, which is the equivalent of labels in Prometheus (`https://prometheus.io/docs/introduction/overview/`)-compatible environments such as Amazon Managed Service for Prometheus (`https://docs.aws.amazon.com/prometheus/latest/userguide/what-is-Amazon-Managed-Service-Prometheus.html`).

In addition to collecting such key metrics, you should also consider collecting application-specific metrics such as runtime metrics (for example, **Java Virtual Machine** (**JVM**) metrics) and custom business metrics that are exposed through application instrumentation. These metrics give you granular information about the application's performance and state.

Container orchestration environments are already very popular among users who want to host their applications on cloud platforms. Amazon EKS is widely used by customers who simply want to use Kubernetes without the overhead of managing the underlying infrastructure or the control plane. When using EKS, collecting metrics at the Pod and container level will give deeper insights into your infrastructure performance.

CloudWatch Enhanced Container Insights is a powerful tool that provides comprehensive monitoring and observability for containerized applications running on Amazon ECS, Amazon EKS, and AWS Fargate. It offers deep insights into the performance and health of containers, enabling DevOps teams to quickly identify and troubleshoot issues.

Enhanced Container Insights leverages the capabilities of CloudWatch and integrates seamlessly with the AWS ecosystem. It automatically collects metrics, logs, and traces from containers, providing a unified view of the application's behavior. The rich dashboards and visualizations make it easy to analyze and correlate data, helping teams identify performance bottlenecks, resource constraints, and potential security vulnerabilities.

One of the key benefits of Enhanced Container Insights is its ability to provide real-time visibility into container metrics. It collects metrics such as CPU utilization, memory usage, and network traffic, allowing teams to monitor the resource consumption and performance of individual containers. This information is crucial for optimizing resource allocation and ensuring that applications have the resources they need to perform optimally.

Possessing such detailed metrics about the environment allows you to configure alarms to alert you when metrics deviate from expected ranges, indicating potential issues easily and resulting in being able to take remediation measures quickly.

Root cause analysis through traces

Use AWS X-Ray to trace requests and identify the exact components or services involved in a partial failure. Analyze X-Ray traces to understand the sequence of events leading up to the failure. In a microservice environment, an outage or a performance challenge could have originated from any service that is part of the equation. If you want to take immediate action, it is important to understand how the fabric of the services is connected.

The following diagram (*Figure 4.4*) is a partial service map that shows how different services interact with each other. You can also see that a few nodes have a red indication that shows potential problems in those services. This visual indication, along with alerts set up on metrics such as latency, error rate, response rate, and so on generated from traces, helps easily pinpoint issues so that you can take quick action:

Figure 4.4 – Partial service map of a microservice architecture showing service interaction and potential issues on certain services

In this section, we explored the critical role of signal collection, including logs, metrics, and traces, in understanding application performance and health. We examined how Amazon CloudWatch provides flexibility in designing monitoring account structures within AWS. While effective log analysis remains crucial for issue identification and troubleshooting, AWS also offers diverse strategies for designing proactive monitoring systems. These systems aim to prevent issues from occurring, a topic we will delve into in the following section. This proactive approach to monitoring and management represents a significant shift from reactive troubleshooting to preemptive problem-solving in cloud environments.

Predicting issues before they occur

The collection of signals from the environment is of paramount importance, and this fact is not subject to debate. However, it must be acknowledged that the data we are observing represents historical information. When an alarm is triggered, the system is already in trouble and you are in troubleshooting mode, indicating that an issue affecting the end user has already occurred. Therefore, it would be advantageous to identify and address potential issues before they manifest. One approach is to leverage collected signal data, such as logs, traces, and metrics, and subject them to an ML model

to predict the future behavior of the system. This is an exhaustive exercise and can also get expensive. This approach is done in mission-critical systems where customers have the ability to do so. We will learn about how you can use **CloudWatch Anomaly Detection** for advanced troubleshooting use cases later in this chapter.

Additionally, it is important to identify potential architectural design issues, such as unknown bottlenecks, **single points of failure** (**SPOFs**), and critical components that lack redundancy. AWS provides a service called **Fault Injection Service** (**FIS**) (`https://aws.amazon.com/fis/`) that allows for the artificial simulation of issues, enabling the identification of problematic areas in the system.

AWS FIS enables engineers to simulate faults and failures in their systems to test their resilience and identify potential issues. By injecting controlled faults, FIS helps engineers gain insights into how their systems respond to various failure scenarios, allowing them to proactively identify and address weaknesses.

FIS provides a wide range of fault injection capabilities, including the ability to simulate common infrastructure failures such as network latency, packet loss, and instance reboots. It also supports simulating application-specific faults, such as database connection failures or API timeouts.

The benefits of using FIS are numerous. First, it helps engineers identify potential problems early in the development process before they can impact production systems. Second, FIS can help engineers optimize their systems for reliability by identifying and eliminating SPOFs. Third, FIS can be used to test **DR plans** (**DRPs**) and ensure that systems can recover quickly from failures.

In terms of reliability, FIS can be a valuable tool for improving the reliability of systems. By simulating faults and failures, engineers can gain insights into how their systems will behave under stress and take steps to mitigate the impact of failures.

Using information collected from this exercise will help you identify weak points and take action by rearchitecting the system to improve reliability.

In this section, we have learned various methodologies used for the identification of faults within systems, with the objective of enhancing resilience. We strongly advise that you conduct a thorough examination of your environment, assess the various monitoring options available for your systems, and subsequently adopt the most suitable option. The subsequent section will provide an in-depth analysis of the rationale behind designing systems in a manner that minimizes the impact of failures on downstream systems, thereby mitigating the risk of catastrophic failure. Additionally, we will explore various industry-standard best practices that can serve as a guide during this process.

Isolating the wound – containment strategies to prevent cascading outages

In a complex AWS environment, a single failure can have a cascading effect, leading to widespread outages and data loss. To mitigate this risk, it is essential to implement containment strategies that prevent failures from spreading beyond their source.

This chapter will discuss various containment strategies that can be employed to improve the resiliency of AWS environments. We will cover topics such as automated troubleshooting, **incident management** (**IM**), and architectural design patterns for containment.

Automated troubleshooting

One of the most effective ways to contain failures is to automate the troubleshooting process. This can be achieved using tools such as AWS Systems Manager OpsCenter, which can analyze alerts and diagnose common issues.

By configuring automated actions to resolve simple problems without manual intervention, troubleshooting time can be significantly reduced, minimizing the impact of failures.

Incident Management

Another important aspect of containment is Incident Management. By establishing a structured Incident Management process, organizations can ensure that partial failures are handled effectively. This includes assigning clear roles and responsibilities to team members, using tools such as AWS Incident Manager to centralize incident tracking and communication, and conducting regular incident reviews to identify areas for improvement.

The following diagram shows the process that can be used to address a situation where the CloudWatch Alarm fires and triggers a series of actions to take an automated remediation action:

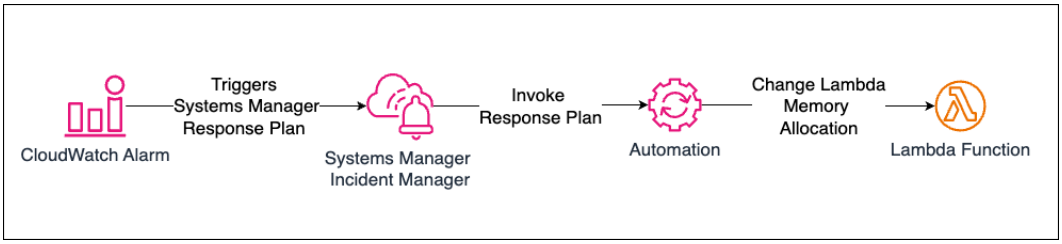

Figure 4.5 – CloudWatch alarm triggering Systems Manager response plan to fix Lambda memory issue

Automation is a crucial aspect of modern IT systems, as explored in this section. Organizations should aim to automate all possible processes to maximize efficiency and reliability. Well-designed automation significantly reduces downtime and enhances system reliability, freeing engineers to focus on high-value, mission-critical tasks. However, it's essential to approach automation design with caution. Careful consideration must be given to automated responses to prevent unintended consequences or potential catastrophic issues. Striking the right balance between comprehensive automation and safeguards against unexpected outcomes is key to leveraging automation's full potential while maintaining system integrity and safety.

Architectural design patterns for containment

In addition to automated troubleshooting and IM, several architectural design patterns can be used to contain failures:

- **Bulkhead pattern**: In this context, the term *bulkhead* refers to the compartmentalized partitions frequently constructed within the hull of a vessel. Bulkheads in ships serve several critical functions, with the primary purpose of preserving buoyancy during hull breaches. These compartmentalized barriers help contain flooding, limiting water ingress to specific sections of the vessel. This design allows the ship to remain afloat even if one or more compartments are compromised, significantly enhancing overall maritime safety and survivability. If water infiltrates a specific section of the vessel, the leak will presumably remain confined to the bulkhead where the breach occurred, preventing the flooding of the entire vessel. Certain modern bulkhead designs are even adept at containing fires and minimizing potential damage induced by electromagnetic pulses.

  In the context of software systems, when multiple subsystems operate as components of a single, overarching process, a fault in one component can readily trigger cascading failures. Drawing inspiration from the bulkhead concept employed in ship hulls, this resilience pattern enables developers to design systems comprising multiple independent subsystems and services functioning within distinct machines or containers. This approach constrains the impact of a failure on neighboring processes, allows teams to analyze such failures in isolation, and is arguably a distinguishing advantage of architectural styles such as microservices, which adamantly advocate for loose coupling between software components.

- **Backpressure pattern**: Backpressure is a defensive approach that allows systems and services to automatically refuse excess workload beyond their current capacity. When a service experiences slowdowns due to factors such as database query delays or network congestion, it can reject new requests to maintain its performance. This rejection also signals the source of the request about the issue, potentially preventing it from hanging or making repeated failed attempts. The strategy enhances system resilience by allowing components to protect themselves from overload and communicate their status effectively.

  Effective backpressure systems should propagate across multiple nodes in a service chain. When one component can't handle its current load, it should be able to push this information back through the entire upstream process. As the overloaded system recovers, it can gradually accept the previously rejected requests. This cascading approach helps regulate throughput organically, preventing any single component from being overwhelmed by unchecked requests.

- **Circuit breaker pattern**: Although backpressure effectively manages load, it often can't solve performance issues alone. In complex software environments, organizations need systems that can detect emerging performance problems and completely disconnect from data sources to recover. This is where the circuit breaker pattern becomes crucial.

> Inspired by electrical circuit breakers that cut power during surges, this pattern sets a specific stress threshold for workloads. When reached, the receiver switches to an *open* state, rejecting new requests and pausing the message queue. Once stress levels and throughput return to normal, the circuit *closes*, resuming regular operations.
>
> Martin Fowler's article on this pattern is a great read to dive deep into. You can find it here: `https://martinfowler.com/bliki/CircuitBreaker.html`.

By equipping these isolated entities with the ability to open and close connections, this resilience pattern can prevent temporary outages from escalating into cascading failures that spread uncontrollably across vast segments of the software stack. As such, it constitutes a particularly significant pattern for individuals designing large, distributed systems to familiarize themselves with.

By implementing containment strategies, organizations can significantly improve the resiliency of their AWS environments. These strategies help to prevent failures from spreading beyond their source, minimizing the impact on users and businesses. In addition to these, you are encouraged to read about other best practices such as implementing timeouts, backing off from excessive retries, and how introducing randomness in backoff strategies can help improve containing issues. Read more about this in the article here: `https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/`.

In the following section, we will learn about why having capabilities to automate responses to incidents is important for improving system stability and resilience. We will look at different AWS services and industry-standard best practices that can be used in the process.

Streamlining recovery with preconfigured actions

In the realm of large-scale operational environments, a fundamental best practice is the systematic documentation and recording of issues and their corresponding resolutions in a timely manner. During the process of issue identification, it is imperative to discern recurring patterns and explore feasible avenues for automating remedial measures. The ultimate objective is to attain a state wherein manual intervention becomes obsolete upon the re-emergence of similar issues in the future.

While acknowledging that not all issues lend themselves to automated solutions, it is essential to strive for automation whenever it proves both practical and advantageous. However, prior to fully embracing this strategy, careful consideration must be given to potential adverse effects that could arise from automation malfunctions.

As businesses increasingly rely on cloud-based applications and services, the need to ensure continuous availability and minimize downtime has never been more critical. AWS provides a comprehensive suite of tools and services to help organizations build resilient applications and systems.

AWS Systems Manager is a cloud-based management service designed to streamline the management and operations of infrastructure on AWS and those that are hosted on non-AWS environments. It offers a centralized view of all resources, enabling users to automate tasks, collect and analyze performance

data, and configure updates effectively. Systems Manager encompasses several components, including Operations Management, Application Management, Change Management, and Node Management. These components collectively provide a comprehensive management solution for AWS resources.

In this set of components, Operations Management contains a specific feature called **Incident Manager** that is relevant to us in this context.

AWS Systems Manager Incident Manager is a cloud-based service that helps you manage and respond to incidents in your AWS environment. It provides a centralized view of your AWS resources and enables you to quickly identify and resolve issues. Incident Manager also integrates with other AWS services, such as Amazon CloudWatch, Amazon **Simple Notification Service** (**SNS**), and Amazon Chime, Slack, and several other third-party IM solutions to provide a comprehensive IM solution.

One of the key features of Incident Manager is the ability to create and manage response plans. A response plan is a set of pre-defined actions that can be automatically executed in response to a specific incident. Response plans can be used to address a wide range of issues, including the following:

- **Infrastructure failures**: Response plans can be used to automatically fail over to redundant resources in the event of a hardware or software failure
- **Security breaches**: Response plans can be used to automatically isolate infected systems and contain the spread of malware
- **Performance issues**: Response plans can be used to automatically scale up resources to handle increased traffic or load

Response plans can be created in the AWS console or using the AWS CLI. When creating a response plan, you will need to specify the following information:

- **Name**: The name of the response plan
- **Description**: A description of the response plan
- **Triggers**: The events or conditions that will trigger the response plan
- **Actions**: The actions that will be performed in response to the trigger
- **Notifications**: The people or groups who will be notified when the response plan is triggered

Once a response plan has been created, it can be tested to ensure that it works as expected. Response plans can be tested in the AWS console or using the AWS CLI.

In addition to response plans, Incident Manager also provides a number of other features to help you manage and respond to incidents. These features include the following:

- **Incident timeline**: The incident timeline provides a visual representation of the events that have occurred during an incident

- **Incident chat**: The incident chat functionality allows you to communicate with other team members about an incident
- **Incident metrics**: Incident metrics provide you with insights into the performance of your IM process

Incident Manager is a powerful tool that can help you to improve your IM process. By using Incident Manager, you can reduce the time it takes to resolve incidents, improve the quality of your **incident response** (**IR**), and reduce the impact of incidents on your business.

Here are some best practices for using AWS Systems Manager Incident Manager:

- **Use Incident Manager for critical incidents only**: Incident Manager is a powerful tool, but it should only be used for critical incidents. Too many incidents can make it difficult to manage and troubleshoot your system.
- **Create well-defined response plans**: Your response plans should be well defined and easy to understand. They should include clear instructions on what to do in the event of an incident.
- **Test your response plans regularly**: Your response plans should be tested regularly to ensure that they work as expected.
- **Use incident metrics to improve your IM process**: Incident Manager provides a number of incident metrics that can help you improve your IM process. These metrics can be used to identify trends and areas for improvement.

There are several benefits to using preconfigured actions:

- **Reduced downtime**: Reduce downtime by automating the recovery process and eliminating the need for manual intervention
- **Improved security**: Improve security by automatically isolating infected systems and containing the spread of malware
- **Increased efficiency**: Improve efficiency by automating common tasks and reducing the need for manual labor
- **Enhanced compliance**: Help organizations meet compliance requirements by automating the response to security incidents and other events

There are a few best practices to keep in mind when using preconfigured actions:

- **Use automations sparingly**: You should only use automated response plans for critical events or conditions. Forcing automation without making necessary validations can make it difficult to manage and troubleshoot your system.
- **Test your automations regularly**: You should test your automated response plans regularly to ensure that they work as expected with changing system needs and designs.

- **Use a consistent naming convention**: Use a consistent naming convention to make automations easier to identify and manage.
- **Document your response plans thoroughly**: Follow consistent guidelines and templates to document your response plans so that others can understand how they work.

This section explored various aspects and considerations involved in streamlining recovery processes through preconfigured actions. We examined how to design and implement these automated responses effectively. Additionally, we delved into the diverse features offered by AWS Systems Manager, which provides powerful tools for easily creating and managing these preconfigured actions. Understanding these capabilities enables organizations to develop more robust, efficient, and responsive recovery strategies in their AWS environments.

Leveraging ML and GenAI to enhance issue detection and response

GenAI is a rapidly developing field of AI that has the potential to revolutionize many industries and aspects of our lives. In this section, we will learn about how you can leverage GenAI to improve the performance and efficiency of your teams in diagnosing issues and automating responses quickly. Before we do that, let's expand on what GenAI and its applications in general are. By utilizing ML and **natural language processing** (**NLP**), GenAI models can create new content, such as text, images, music, and even code, which is indistinguishable from human-generated content.

GenAI for IR

GenAI models are not only known to generate text but also have a strong ability to summarize and extract information from a very large text input. You can also build chatbots that can answer questions on a given text source through **Retrieval Augmented Generation** (**RAG**). RAG is a technique that enhances the performance of **Large Language Models** (**LLMs**) by incorporating external knowledge sources. This method allows the model to access and utilize information beyond its initial training data, enabling it to generate more accurate and up-to-date responses.

RAG models address limitations in existing text generation models by combining retrieval-based and generation-based approaches. They leverage pre-trained language models for text generation and utilize external knowledge sources to enhance the accuracy and quality of the generated text. RAG models can produce more informative, diverse, and coherent text compared to models that rely solely on generation, making them particularly useful for tasks such as question answering, summarization, and dialogue generation.

AWS offers several ML and GenAI services that you can make use of. Amazon Q is a GenAI-powered assistant built on top of Amazon Bedrock, a fully managed service that provides access to high-performing **foundation models** (**FMs**) from leading AI companies. Amazon Bedrock serves as the underlying platform, offering a choice of FMs that can be customized and integrated into applications to build GenAI capabilities.

Amazon Q leverages these FMs available through Bedrock to power its conversational AI features, enabling it to assist with tasks such as answering questions about AWS, generating and improving code, and providing recommendations.

One of the use cases to employ Amazon Q in your operations workflow is to feed past incident documents to Amazon Q in order to be able to easily search for information about past incidents and take necessary action. When a DevOps engineer is responding to an alarm, they can be informed not only about the issue that is going on but also a potential fix based on past information.

The logical workflow after implementing a solution on these lines might look similar to the following diagram:

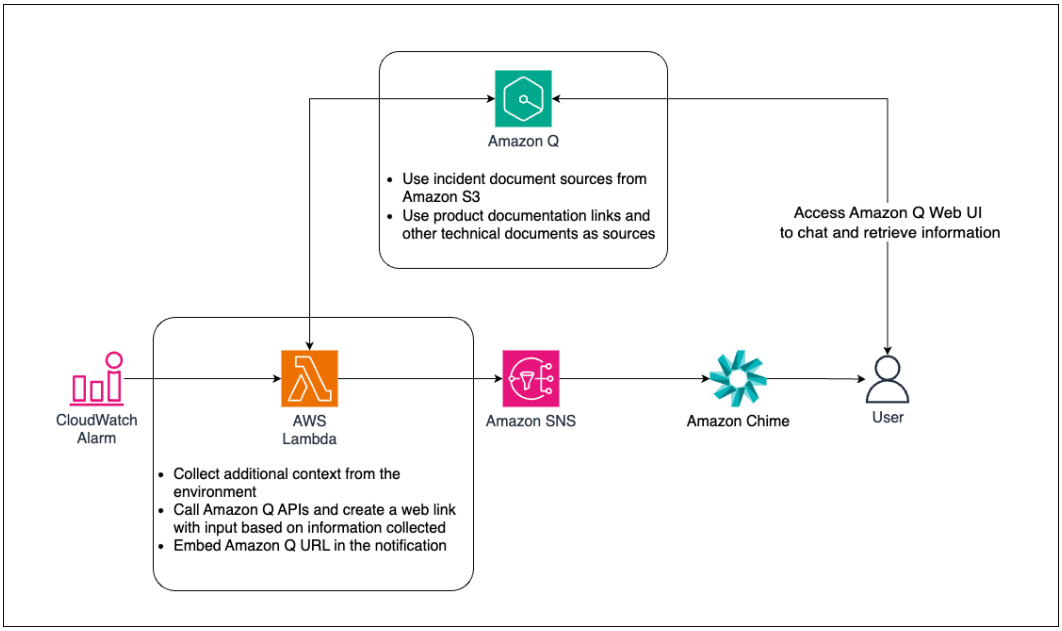

Figure 4.6 – A potential architecture that makes use of Amazon Q for RAG-based chat on IR

In the architecture in *Figure 4.5*, firstly, Amazon Q is set up to make use of past incident documents that were collected over a period of time. When a CloudWatch alarm goes off, a Lambda function can be used to collect additional context/information about the alarm, use Amazon Q APIs, and construct a query URL with input from the information obtained from the alarm. That URL can be embedded into the notification, which can then be sent to Amazon SNS, which invokes collaboration software such as Amazon Chime.

Upon receipt of a notification from the chatbot by the end user, the message will contain a hyperlink that can be utilized to initiate a dialogue with Amazon Q. Subsequently, the user will have the ability to pose inquiries based on the problem encountered and potentially obtain a solution that can be implemented based on the responses provided by Amazon Q.

This workflow significantly diminishes the time required for issue resolution, as knowledge derived from previous issues can be utilized to promptly resolve issues. Additionally, it can serve educational purposes for new team members by providing access to the chatbot, enabling them to familiarize themselves with the system through inquiries to Amazon Q. GenAI services are acknowledged for occasional fabrication of information and providing misguided responses; however, this is circumvented in this instance due to the architecture being based on RAG, with the context narrowly focused on the specific use case. At the time of writing this chapter, Amazon Q supports connecting to several case management systems such as ServiceNow, Confluence, Salesforce, and so on. You can take a look at the updated list of connectors here: `https://docs.aws.amazon.com/amazonq/latest/qbusiness-ug/connectors-list.html`.

ML for issue identification

Anomaly detection is a great way to reduce false alarms and thereby improve the signal-to-noise ratio. Metric anomaly detection is a technique used to identify unusual patterns or deviations in time series data. It is widely employed in various domains, including IT operations, financial monitoring, and healthcare. Amazon CloudWatch, a monitoring and observability service from AWS, leverages ML algorithms to perform anomaly detection on metrics collected from applications, infrastructure, and services.

ML algorithms, such as **Seasonal Autoregressive Integrated Moving Average** (**SARIMA**) and **Exponential Smoothing**, are used to establish a baseline for the normal behavior of a metric. This baseline is created by analyzing historical data and identifying patterns and trends. When new data points are received, they are compared against the baseline to detect significant deviations.

CloudWatch Anomaly Detection automatically identifies anomalous data points and provides insights into potential causes of anomalies. It employs statistical methods and ML models to analyze metric data and identify patterns that deviate from the norm. The service also considers factors such as seasonality, trend, and noise to distinguish between genuine anomalies and normal fluctuations.

This enables quick troubleshooting, reduces downtime, and ensures the smooth operation of applications and infrastructure. CloudWatch Anomaly Detection can be integrated with other AWS services, such as Amazon SNS and Amazon Chime, to trigger notifications and alerts when anomalies are detected, ensuring timely response and resolution.

Look at the following sample screenshot that shows an anomaly detection band from CloudWatch metrics:

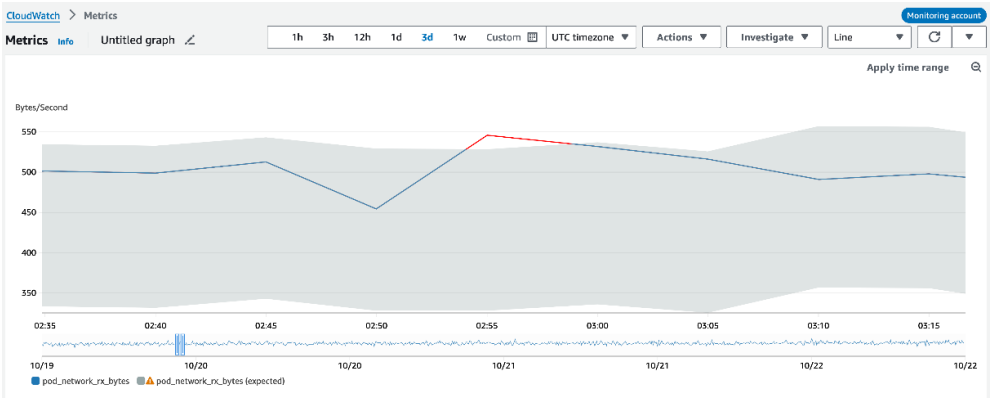

Figure 4.7 – CloudWatch metrics anomaly detection

Figure 4.7 shows CloudWatch Anomaly Detection enabled on a metric called `pod_network_tx_bytes`. Once enabled, you can see that CloudWatch has used ML algorithms behind the scenes to understand and analyze the metric data over a period of time and create a gray color band indicating the range of values the metric values could go to that can be considered to be normal.

The red line toward the top right-hand side of the screenshot indicates the metric values that fell outside the expected range, indicating outlier values. Now, when creating a CloudWatch alarm, you could simply configure it to get triggered only if the metric values fall outside the anomaly detection range instead of a static value. This means you are only triggering alarms when there is a genuine issue going on in your environment, hence improving the signal-to-noise ratio.

The following screenshot shows the CloudWatch alarm setup screen where you can set trigger values based on the anomaly detection band:

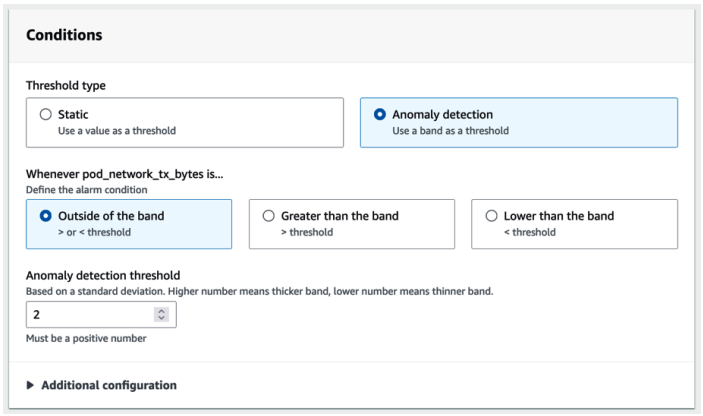

Figure 4.8 – CloudWatch alarm setup screen showing alarm values being set on the anomaly detection band

As has been previously discussed, the integration of anomaly detection and GenAl in issue detection and troubleshooting workflows allows for a significant enhancement in the efficiency of teams managing IT environments, thereby facilitating seamless operations.

In this section, we discussed how you can leverage ML and GenAI to accurately identify issues by increasing the signal-to-noise ratio and also use GenAI to accelerate the troubleshooting process. This helps increase performance, reduce cost, and overall increase the availability and resilience of your applications.

Summary

As we wrap up this chapter, let's take a look at the different aspects we learned. First, we dove deep into the concept of identifying partial failures and how you can use different AWS services such as Amazon CloudWatch to effectively identify those. In the same section, we also discussed the AWS FIS service, which helps you to conduct simulated failure drills in order for your teams to identify potential weak points in your architecture. In the second section, we learned about deploying various architecture patterns to follow in order to reduce dependencies in such a way that failures can be confined to a smaller portion of the application/infrastructure. Subsequently, we explored the benefits of using preconfigured actions in order to automatically take corrective actions during outages. In the final section, we learned about using ML and GenAI to accelerate and optimize issue identification and troubleshooting.

In the following chapter, we will explore more deeply the AWS shared responsibility model and how this model varies depending on the services used. You will learn about how you can design your architecture by leveraging the shared responsibility model to help achieve resilient architectures.

Further reading

The following articles are good sources to learn about the concept of graceful degradation:

- What is Graceful degradation?: `https://www.techtarget.com/searchnetworking/definition/graceful-degradation`.
- Exploring Graceful degradation: `https://www.ituonline.com/tech-definitions/what-is-graceful-degradation/`.

5

Exploring the AWS Shared Responsibility Model

Resilience, the ability of a system to withstand and recover from disruptions, is a fundamental pillar of any successful cloud infrastructure. In the **Amazon Web Services** (**AWS**) ecosystem, achieving this resilience is not a solitary endeavor but a collaborative partnership. The **AWS shared responsibility model** encapsulates this partnership, delineating the distinct yet interconnected roles of both AWS and the end user (customer) in fortifying infrastructure against a spectrum of potential threats.

In this chapter, you will get an introduction to what shared responsibilities between AWS and customers look like and the different roles these parties play in designing and operating a resilient infrastructure.

We'll cover the following topics:

- The essence of collaboration
- The synergy of shared resilience
- Adapting shared responsibilities for specific services
- Shared responsibility and cost
- The importance of continuous testing for critical infrastructure resilience in AWS environments

The essence of collaboration

At its core, the **shared responsibility model** is a framework that distributes the onus of security and operational reliability between AWS and its customers. This is not a division of labor meant to dilute responsibility, but rather a strategic alignment of expertise and control.

AWS, as the cloud provider, shoulders the responsibility for the resiliency *of* the cloud, ensuring the underlying infrastructure's reliability, availability, and security, ranging from physical data centers to foundational services. AWS is responsible for the resiliency of its underlying infrastructure, including hardware, software, networking, and facilities. They strive to ensure service availability meets or exceeds their **service-level agreements** (**SLAs**). The **AWS Global Cloud Infrastructure** is designed to enable customers to build highly resilient workloads, with each **AWS Region** consisting of multiple, isolated

Availability Zones (**AZs**) interconnected with high-bandwidth, low-latency networking. This design helps protect against various faults and disruptions.

The end user, meanwhile, assumes the responsibility for the resiliency *in* the cloud. This encompasses designing workloads by implementing best practices that leverage the underlying infrastructure on which the applications are hosted. The specific responsibilities vary based on the services used. The customer's responsibility for resiliency in the AWS cloud depends on the services they choose. For services such as Amazon **Elastic Cloud Compute (EC2)**, customers are responsible for configuring and managing all aspects of resiliency, such as deploying instances across AZs, implementing self-healing techniques, and using resilient workload architecture best practices. For managed services such as Amazon **Simple Storage Service** (**S3**), Amazon **Relational Database Service** (**RDS**), and AWS **DynamoDB**, customers are responsible for managing data resiliency, including backups, versioning, and replication. Deploying workloads across multiple AZs and Regions is a key aspect of high availability and **disaster recovery** (**DR**) strategies.

The synergy of shared resilience

The collaborative nature of the model is what makes it a catalyst for resilience. AWS's focus on the foundational layers allows them to invest heavily in redundancy, failover mechanisms, and threat detection at a scale that most individual organizations could not replicate. This provides a robust foundation upon which users can build.

The customer's role is equally crucial. By managing their data, applications, and configurations, they can tailor their environment to their specific security and resilience needs. This includes implementing encryption, network firewalls, logging, and monitoring. This granular control, coupled with AWS's foundational security, creates a multi-layered defense mechanism that is far more resilient than either party could achieve alone.

Figure 5.1 shows a simplified version of what the responsibilities between AWS and customers look like. A more detailed diagram can be found in the official AWS documentation here: `https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/shared-responsibility-model-for-resiliency.html`.

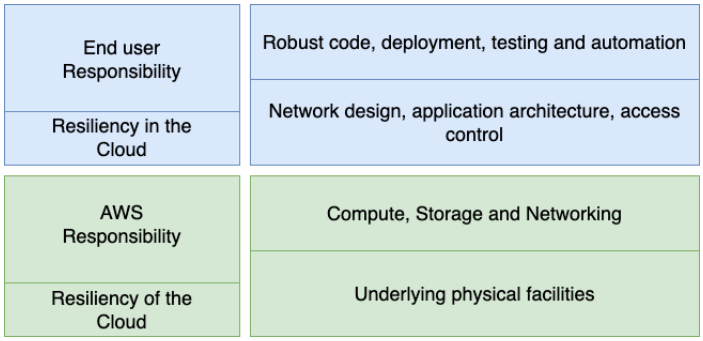

Figure 5.1 – Shared responsibility model for resilience on AWS

Understanding the shared responsibility model is the first step toward building a truly resilient architecture on AWS. In the subsequent sections of this chapter, we will delve deeper into the specifics of this model, exploring the nuances of AWS's and the customer's responsibilities across various services. We will also examine some best practices and tools that can help you navigate this shared landscape to create cloud infrastructure that not only survives but thrives in the face of adversity.

Adapting shared responsibilities for specific services

The AWS shared responsibility model provides a framework for building resilient cloud infrastructure, but its application varies depending on the specific services you employ. Understanding how this model adapts to different services is crucial for crafting comprehensive resilience strategies. In this section, we will learn about differences and similarities in AWS and customer responsibilities with respect to some of the important AWS services.

When it comes to customer responsibility in managing resilience in AWS services, the higher the abstraction, the lower the workload for customers in managing the resilience of the infrastructure.

In *Figure 5.2*, we attempt to showcase the variance in the level of responsibility customers have depending on the type of AWS service being used.

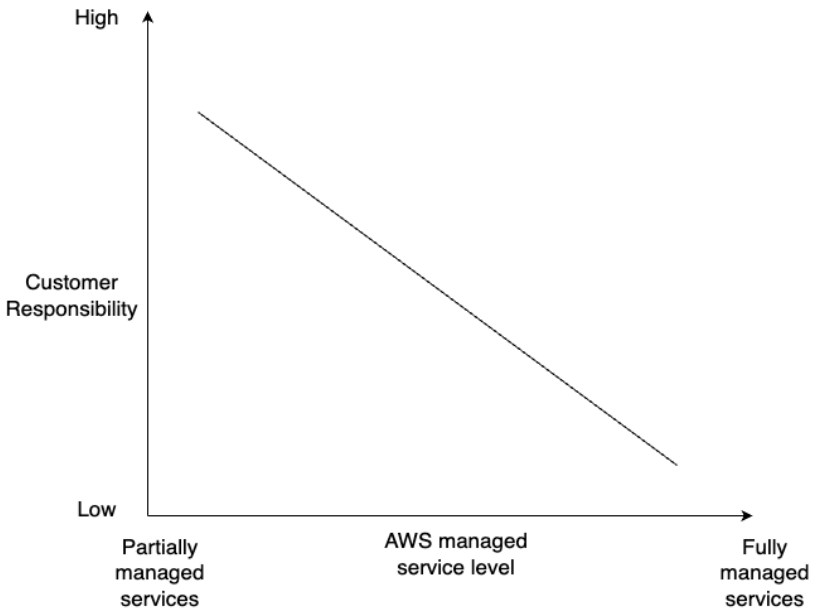

Figure 5.2 – Varying customer responsibility based on the level of the AWS-managed service being used

In a nutshell, customers should always aim to use services that are fully managed by AWS if they want to take less ownership of the resilience of their environment.

For example, when using Amazon DynamoDB for storage, the responsibility between these parties looks like the following:

- **AWS responsibility**: AWS manages the underlying infrastructure, including hardware, replication, and backups. They also provide automated scaling and failover mechanisms.
- **Customer responsibility**: Customers are responsible for designing their tables and indexes for optimal performance and resilience. They must also implement appropriate data access controls and monitor their tables for throughput and latency.

As explained previously, AWS manages almost all the infrastructure portion of the environment for DynamoDB, which allows the customer to simply use the service with minimal setup, such as configuring security and table design. This allows customers to maximize their time in focusing on other meaningful areas, such as writing better code, automating deployment, testing, and so on, rather than spending time designing infrastructure for auto scaling or availability.

Let's look at another AWS service, such as Amazon **Elastic Kubernetes Service** (**EKS**). The responsibilities between the parties look like the following:

- **AWS responsibility**: AWS manages the control plane components, including the Kubernetes API server, etcd database, and scheduler. They also handle patching, scaling, and maintenance of the control plane.
- **Customer responsibility**: Customers are responsible for the data plane components such as configuring and managing their worker nodes, deploying and managing applications, and securing their containers. They must also apply security patches to their worker nodes and implement appropriate network security controls.

The complete ownership and operation of a Kubernetes environment is a significant undertaking. A Kubernetes environment consists of two primary components: the *control plane* and the *data plane*. The data plane encompasses virtual machines that serve as hosts for **Kubernetes Pods**, which in turn host applications. However, the Kubernetes control plane is a mission-critical part of the infrastructure that needs close to 100% availability to ensure applications in the data plane do not go down. Let's take a look at the control plane operations in detail to understand the various functions it supports.

Kubernetes control plane and its operations

The Kubernetes control plane orchestrates the cluster's overall operations, maintaining the desired state of applications. At its core is the API server, which serves as the cluster's gateway, processing RESTful requests and updating the cluster's state in etcd, a consistent and highly available key-value store.

The controller manager oversees various controllers that regulate the cluster's state, ensuring that the actual state matches the desired state by managing tasks such as node lifecycle and replication. The scheduler assigns workloads to nodes based on resource availability and constraints, optimizing resource utilization and performance.

Together, these components maintain the cluster's health, manage workloads, and facilitate communication between the control plane and the data plane. Robust security measures, including authentication, authorization, and admission control, are implemented to protect the cluster's integrity. The control plane is often replicated for high availability, ensuring resilience and reliability in managing the cluster's operations.

As seen earlier, a Kubernetes cluster consists of various components with its control plane being the center of the infrastructure whose health and performance can directly impact the entire cluster's health.

By utilizing Amazon EKS, customers can entrust the management of the control plane's health, reliability, security, and availability to AWS. EKS provides a fully managed control plane environment that is prepared to integrate with worker nodes.

As a user, you will still be responsible for configuring the number of worker nodes, selecting the instance types to be used for these nodes, grouping them together under *node groups* to simplify management, as well as overseeing the configuration and deployment of pods, load balancing, networking between pods, security of pods and nodes, scaling of pods based on requirements, and storage management.

As evident in this scenario, you possess the significant authority to manage and control the architecture and behavior of the Kubernetes environment.

Now, let's take a look at another use case where the customer gets even more control and hence responsibility and power to build their infrastructure. This is the use case where you want to host your own database servers on Amazon EC2 instances:

- **AWS responsibilities**:
  - Provisioning and managing the underlying infrastructure, including hardware, networking, and storage
  - Providing a secure and compliant platform
  - Offering a variety of tools and services to help customers manage their databases
- **Customer responsibilities**:
  - Designing and implementing the database architecture
  - Configuring and managing the database software
  - Monitoring and maintaining the database
  - Backing up and restoring the database
  - Securing the database
  - Patching the host and the database engine
  - Ensuring high availability and reliability

As seen in the preceding list, in this use case, customers have great control over every single aspect of the lifecycle of the database servers used in the environment. It is apparent that these are serious actions that need to be carefully undertaken to operate a reliable database environment.

Now that we have learned about how AWS and customer responsibilities vary across different AWS services, let's overlay the services on *Figure 5.1* to get a clearer picture of where these aforementioned and other similar services fall on the chart:

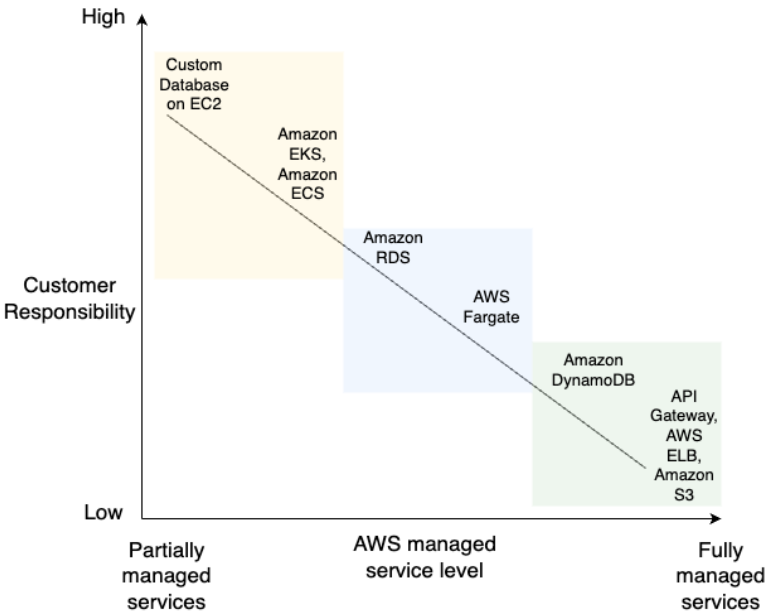

Figure 5.3 – Customer responsibility scale for various AWS services

As illustrated in *Figure 5.3*, the greater the control AWS exercises in managing the underlying infrastructure and service components, the lower the customer's responsibility. Using fully managed services, such as Amazon **S3**, **API Gateway**, **AWS Elastic Load Balancer**, Amazon **Simple Queueing Service** (**SQS**), Amazon **Simple Notification Service** (**SNS**), **AWS Route 53**, and so on, lowers customer responsibility to the minimum whereas managing workloads on EC2, EKS, and alike increase customer responsibility significantly higher.

To summarize, in this section, we learned about how customer responsibility varies across different AWS services depending on how much AWS takes responsibility for operating the service infrastructure and components. It is always encouraged to leverage AWS's rich experience and expertise, resources, and technical abilities to manage as much infrastructure as possible. This greatly reduces customer responsibility and increases resiliency without much effort.

In the following section, we will learn about how shared responsibility impacts the overall operation cost of the infrastructure and how as a user you need to equip yourself to fully inform yourself of the total cost of ownership before making key decisions.

Shared responsibility and cost

Cost is a critical factor in making architectural decisions, and getting it correct is important to ensure long-term success in operations. It is easy to get carried away with superficial information and make uninformed decisions that can eventually prove to be expensive in the long run.

Many users do not see the benefits fully managed services bring to the table before making decisions. While fully managed services may initially appear more expensive than self-managed options, organizations often find that the **total cost of ownership** (**TCO**) is lower in several situations. For example, simply going by the sticker price for a service such as **Amazon Aurora** can lead to one thinking that it is less expensive to operate a database by themselves. To understand the full picture of the cost, you have to consider several factors.

The following section covers a list of areas where fully managed services offer significant advantages over self-managed options:

- **Reduce operational costs**: When using a fully managed service, the responsibility of management and operation of the service infrastructure, including activities such as upgrading hardware and underlying software, such as firmware, virtualization stack, operating system, and so on, lies with AWS. AWS manages the day-to-day management and operation of the service, freeing up internal resources to focus on core business activities. Users can also enjoy the economy of scale that allows them to leverage the functionality of the service at a lower cost than what they can achieve on their own.
- **Improve efficiency**: AWS has highly skilled employees who have deep experience in managing and operating a very large-scale infrastructure, which leads to higher efficiency and reduced downtime. This enables users to benefit from a reliable, high-quality infrastructure on which they can develop applications that meet their customers' expectations. Higher availability and efficient infrastructure directly contribute to developer and operator satisfaction, which positively impacts productivity. This can result in cost savings in terms of lost productivity and revenue.
- **Enhance security**: AWS maintains a large-scale, multi-tenant environment that emphasizes security as a fundamental principle. This encompasses the physical security of data center facilities, adherence to secure software development practices, and the implementation of high-quality threat defense mechanisms designed and deployed by industry experts. When utilizing a fully managed service from AWS, one can reasonably assume the safety and security of the software and underlying infrastructure relative to self-managed workloads. In the context of self-managed workloads, users bear the responsibility of ensuring both a secure infrastructure perimeter and the security of access and software. This can be a demanding endeavor, requiring highly skilled resources to remain vigilant in the face of evolving threats. Entrusting AWS with this responsibility can assist organizations in mitigating the risk of data breaches and other security incidents. This proactive approach can result in cost savings related to potential fines, legal expenses, and reputational harm.

- **Ensure compliance**: Fully managed offerings on AWS go through plenty of scrutiny to get compliance certifications. It involves regular inspections of infrastructure, security practices, and certification requirements to ensure the stack is fully compliant with the regulatory requirements. While it is relatively easy to operate a regulatory-compliant environment using AWS infrastructure, such as Amazon EC2, it still requires careful operational practices to be implemented by the user. **Managed service providers** (**MSPs**) can help organizations comply with industry regulations and standards, which can be complex and time-consuming to manage on their own. This can lead to cost savings in terms of fines and penalties.
- **Accelerate time to market**: Businesses around the world are constantly looking at adding meaningful value to their customers and one of the key aspects to doing that is to be able to deliver new products and update the existing portfolio of products frequently. To support your business to do just that, the technical teams need to be ready to spin up appropriate infrastructure in a fast manner. The easiest and sometimes most cost-efficient way to do that is by using fully managed services on AWS against self-managed software stacks.

For example, imagine you expect a new feature launch to become extremely popular that could overload your systems. In order to meet the new user growth, you decide to use **Apache Kafka** as a messaging system. You could set up **Amazon Managed Service for Kafka** (**MSK**) simply in a few minutes and get going with building your application, versus setting up the entire complex infrastructure yourself. Using MSK not only gives you a quick head start but also an environment that requires far less maintenance overhead compared to building and managing one on your own. Quickly getting the products to market through AWS-managed services not only saves operational costs but also helps businesses earn market share and increase profits.

In addition to these cost savings, fully managed services can also provide organizations with greater peace of mind knowing that their service is being managed and operated by experts. This can lead to improved productivity and morale among employees, which can also contribute to cost savings.

While fully managed services may have a higher upfront cost than self-managed options, they often provide organizations with a lower TCO and a range of additional benefits. By carefully considering the factors discussed earlier, organizations can make an informed decision about whether a fully managed service is right for them.

The importance of continuous testing for critical infrastructure resilience in AWS environments

Continuous testing is a key component of building resilient critical infrastructure. By continuously testing critical infrastructure systems, organizations can identify and fix vulnerabilities before they can be exploited. This helps to reduce the risk of outages and service disruptions that could have a significant impact on the resilience of the infrastructure.

Most, if not all, outages that have taken place so far are due to a specific scenario not having been tested and mitigation actions planned for. One such incident that had a major economic impact on the

organization was the Equifax data breach. In the year 2017, Equifax, an American credit monitoring agency suffered a massive, large-scale data breach, exposing sensitive information of 147 million customers. As a result, Equifax faced multiple fines and settlements related to the 2017 data breach, totaling up to $700 million, in addition to losing customer trust and goodwill.

Two key factors were identified as the source of this breach:

- Failure to patch a known vulnerability in the **Apache Struts framework**
- Inadequate security testing and monitoring

As you can see, if only Equifax had appropriate continuous testing mechanisms in place to identify code and dependency vulnerabilities, they would have been able to identify the fault in advance and potentially mitigate the risk. Learn more about the Equifax breach in the article here: `https://www.csoonline.com/article/567833/equifax-data-breach-faq-what-happened-who-was-affected-what-was-the-impact.html`.

Another interesting incident that we can learn from is where an insecure S3 bucket exposed millions of customer data publicly. In November 2020, it was reported that Prestige Software, a supplier of channel management software services for the online travel industry, configured their S3 bucket incorrectly, which resulted in exposing private information, such as credit card details, reservation details, and other **personally identifiable information** (**PII**), such as name, email, address, phone number, and national IDs of customers of Booking.com, Expedia, and Hotels.com, totaling about 24 GB of data.

If only Prestige Software had a regular scanning process that checked their S3 bucket configuration on a frequent basis, this exposure could have been averted, saving the company from going through a massive security nightmare, bad press, and losing customer goodwill and trust. Learn more about the incident in this news report here: `https://www.computerweekly.com/news/252491842/Leaky-AWS-S3-bucket-once-again-at-centre-of-data-breach`.

There are several such incidents that we can quote here, but the message is loud and clear. It is extremely critical for us to be vigilant when it comes to securing the infrastructure and ensuring it continues to stay secure on an ongoing basis through continuous testing and validation.

AWS offers a wide range of fully managed services that can help organizations build resilient critical infrastructure. However, even when using fully managed AWS services, continuous testing is still important. This is because simply using AWS services in the architecture doesn't automatically guarantee a fully resilient, foolproof infrastructure. In addition, organizations may need to integrate AWS services with their own on-premises systems, which can introduce additional complexities.

There are several benefits to continuous testing of critical infrastructure in AWS environments:

- Continuous testing can help organizations identify and fix vulnerabilities before they can be exploited. This helps to reduce the risk of outages and service disruptions that could have a significant impact on public safety and the economy.

- Continuous testing can help organizations comply with regulations that require them to test their critical infrastructure systems regularly.
- Continuous testing can help organizations improve the overall security and availability of their critical infrastructure systems by identifying and fixing vulnerable points that could pose performance bottlenecks and security challenges.

Let's explore specific strategies for key AWS services: Amazon S3, Amazon RDS, and Amazon EC2:

- **Amazon S3 resilience testing**: Amazon S3 offers durable object storage, but continuous testing is essential to verify its resilience. Key strategies include the following:
  - **Versioning and replication**: Enable versioning and configure cross-region replication to test data recovery scenarios. Regularly simulate regional failures and verify that objects can be accessed from the replicated bucket.
  - **Access control**: Continuously test S3 bucket policies and IAM roles to ensure proper access controls. Use AWS Config rules to detect and alert on publicly accessible buckets.
  - **Performance testing**: Utilize tools such as Apache JMeter to simulate high-concurrency scenarios and verify S3's ability to handle peak loads without compromising availability.
- **Amazon RDS resilience testing**: For Amazon RDS, focus on these continuous testing strategies:
  - **Multi-AZ deployments**: Test failover scenarios by forcing a failover using the AWS CLI or RDS API. Monitor the time taken for the standby instance to become the new primary and verify application connectivity.
  - **Backup and restore**: Regularly test the restoration process from automated backups and manual snapshots. Verify data integrity and measure the time required for full restoration.
  - **Read replica performance**: For read-heavy workloads, test the latency and throughput of read replicas under various load conditions. Ensure that the application can effectively distribute read operations across replicas.
- **Amazon EC2 resilience testing**: EC2 instances form the backbone of many AWS applications. Implement these testing strategies:
  - **Auto Scaling**: Continuously test **Auto Scaling groups** (**ASGs**) configured to leverage multiple AZs by simulating traffic spikes and instance failures. Verify that new instances are launched and integrated into the application stack seamlessly.
  - **Instance recovery**: Use AWS CloudWatch alarms to trigger automatic recovery of impaired EC2 instances. Test this mechanism regularly to ensure rapid recovery without data loss.
  - **Chaos engineering**: Implement controlled chaos experiments using **AWS Fault Injection Simulator** (**AWS FIS**). Randomly terminate EC2 instances to verify that your application can handle unexpected instance failures.

- **Comprehensive resilience testing**: To ensure overall infrastructure resilience, focus on the following strategies:
    - **DR drills**: Conduct regular DR drills that simulate complete region failures. Test the ability to fail over to a secondary region and measure the **recovery time objective** (**RTO**) and the **recovery point objective** (**RPO**).
    - **Load testing**: Use services such as AWS CloudFormation to deploy test environments that mirror production. Conduct load tests to verify that your infrastructure can handle peak traffic without degradation.
    - **Continuous monitoring**: Implement comprehensive monitoring using Amazon CloudWatch and AWS X-Ray. Set up alarms for key metrics and regularly review and update monitoring thresholds. By implementing these continuous testing strategies for S3, RDS, EC2, and overall infrastructure resilience, organizations can significantly enhance their ability to maintain critical services in the face of unexpected events and ensure high availability in AWS environments. Create CloudWatch dashboards that give you clear visibility into performance indicators that matter to you. Well-defined **Service Level Objectives** (**SLO**) should be the driving factor behind identifying these performance indicators.

AWS offers a variety of security and monitoring services for users to make use of to automate the continuous testing process and also keep close track of the infrastructure's performance and health. In the following section, we will look at those tools and their capabilities.

Tools and techniques to perform continuous testing of AWS environments

To perform continuous testing for critical infrastructure resilience in AWS environments, several tools and services can be utilized:

- **AWS Resilience Hub**: AWS Resilience Hub is a central service for defining, validating, and tracking application resilience. It offers the following capabilities:
    - It assesses applications to uncover potential resilience enhancements.
    - It validates RTOs and RPOs.
    - It provides actionable recommendations to improve resilience.
    - It generates code snippets for creating recovery procedures as AWS Systems Manager documents.
    - It integrates with AWS FIS for chaos engineering tests.
- **AWS FIS**: AWS FIS is a fully managed service for running chaos engineering experiments that does the following:
    - It simulates real-world failures such as network errors or database connection issues.

  - It offers pre-built scenarios and a wide selection of disruptive actions.
  - It includes controls and guardrails for safely running experiments in production.
  - It can be integrated with AWS Resilience Hub for automated resilience testing.

- **AWS CodePipeline**: AWS CodePipeline can be used to automate resilience assessments as part of the **Continuous Integration/Continuous Deployment** (**CI/CD**) process. It offers the following capabilities:
  - It triggers AWS Step Functions workflows to run resilience assessments.
  - It stops deployments if resilience issues are detected.
  - It ensures changes don't compromise application resilience.
- **AWS Step Functions**: AWS Step Functions can orchestrate resilience assessment workflows. It offers the following capabilities:
  - It updates and publishes the latest application version in Resilience Hub.
  - It initiates resilience assessments.
  - It checks assessment status and compliance.
  - It triggers alerts if policy breaches are detected.
- **Additional testing tools**: There are some additional testing tools, such as the following:
  - **Load testing tools**: Services such as Apache JMeter can simulate high-concurrency scenarios.
  - **Chaos engineering tools**: Tools such as Chaos Monkey can be used to randomly terminate EC2 instances.
  - **AWS CLI and APIs**: For scripting custom resilience tests, such as forcing RDS failovers.
- **Continuous monitoring**: AWS's native monitoring tools come in handy to keep track of the performance of infrastructure and applications. Monitoring specific performance and health indicators does serve as an important tool to quickly be in the know of any abnormalities that could be a sign of an ongoing security challenge:
  - **Amazon CloudWatch**: Set up alarms for key metrics and regularly review thresholds
  - **AWS X-Ray**: For distributed tracing to identify performance bottlenecks and failures

By leveraging these tools and services, organizations can implement a comprehensive continuous testing strategy for critical infrastructure resilience in AWS environments. This approach helps ensure that systems can withstand failures, recover quickly, and maintain high availability across all AWS services, including S3, RDS, and EC2.

Overall, the following actions are necessary to pay attention to performing the continuous testing of critical infrastructure in AWS environments:

- Use various testing tools and techniques to ensure all aspects of your critical infrastructure systems are tested.
- Test your systems regularly, and more frequently during times of heightened risk.
- Automate your testing process as much as possible to reduce the cost and time required to test your systems.
- Use a risk-based approach to testing, focusing on the most critical systems and components.
- Work with AWS to develop a testing plan that is tailored to your specific needs.

By following these tips, your organization can ensure that its critical infrastructure systems are resilient to disruptions.

Adapting your security practices alongside AWS's ever-evolving landscape

As AWS continues to innovate and expand its offerings, security practices must evolve to keep pace with these changes. AWS regularly introduces new services and updates existing ones, which can significantly impact how organizations manage their security posture.

One of the primary challenges is the introduction of new services. Each new service, such as the recently launched AWS Parallel Computing Service, brings unique security considerations. Organizations must stay informed about these services and understand their security implications. This involves regularly reviewing AWS announcements and documentation to identify new features and potential vulnerabilities.

Making changes to existing services to adopt new services necessitates an adaptive security strategy. AWS frequently updates its services to enhance functionality and security. For example, the addition of conditional writes to Amazon S3 requires organizations to update their security policies and access controls to leverage these new capabilities effectively. Continuous monitoring and updating of security configurations are essential to ensure they align with the latest service updates.

Moreover, the behavior and operation of services can evolve, impacting security practices. For instance, the shift toward more integrated and automated security tools, such as Amazon CodeGuru Security, which can identify code security issues through static analysis and help reduce runtime vulnerabilities, highlights the need for organizations to adopt DevSecOps practices. Integrating security into the development lifecycle ensures that security measures are applied consistently and automatically across all stages of service deployment and operation.

To adapt effectively, organizations should implement a few key strategies:

- **Continuous learning and training**: The cornerstone of adapting to AWS's evolving landscape is a commitment to continuous learning. Security teams must stay informed about new AWS

services, features, and best practices. Regular attendance at AWS events, monitoring of official blogs, and participation in the AWS community are crucial for staying ahead of the curve.

- **Embracing infrastructure as code (IaC)**: As AWS services become more programmable, adopting IaC practices becomes essential. IaC allows organizations to define and manage their cloud resources using code, enabling version control, automated testing, and consistent security configurations across environments.
- **Leveraging AWS native security services**: AWS frequently introduces new security services and enhances existing ones. Organizations should regularly evaluate and integrate these native tools into their security strategies. Services such as AWS Security Hub, GuardDuty, and Macie offer powerful capabilities that can strengthen an organization's security posture.
- **Implementing least privilege access**: As AWS services become more granular, the principle of least privilege becomes increasingly important. Regularly review and refine IAM policies to ensure users and services have only the permissions necessary for their tasks.
- **Automating security processes**: With the growing complexity of AWS environments, manual security processes become unsustainable. Investing in automation for tasks such as security assessments, compliance checks, and incident response is crucial for maintaining security at scale.
- **Adopting a DevSecOps approach**: Integrating security into the development pipeline is essential in a rapidly changing cloud environment. By adopting DevSecOps practices, organizations can ensure that security considerations are addressed throughout the development lifecycle, from design to deployment and beyond.
- **Proactive monitoring and auditing**: Implement continuous monitoring and regular audits to detect and respond to security incidents promptly.
- **Adopt a zero-trust model**: Ensure that every access request is authenticated, authorized, and encrypted, regardless of its origin.

By staying informed and proactive, organizations can effectively adapt their security practices to align with AWS's dynamic environment, ensuring robust protection for their cloud resources and maintaining competitive advantage.

Sharing lessons learned and engaging with the community

AWS users have several avenues to contribute to a collective knowledge base, engage with fellow users, and continuously learn. Here are some potential options:

- **AWS re:Post**: A community-driven Q&A platform where users can ask questions, share knowledge, and receive expert technical guidance on AWS services. It features contributions from AWS experts, a reputation system, and integration with the AWS Knowledge Center, fostering continuous learning and collaboration within the AWS community.

- **AWS user groups**: Local community-led groups that meet regularly provide a platform to discuss AWS services, share experiences, and network with peers. These gatherings often feature presentations from AWS experts or local users, fostering a collaborative environment for learning and professional growth.
- **AWS events**: AWS offers numerous opportunities for learning and networking through various events. You can attend conferences, such as AWS re:Invent, AWS Summit, and AWS DevDay, where you can participate in workshops, hands-on labs, and networking sessions. Additionally, many of these events provide virtual participation options for remote attendees.
- **AWS online communities**: Join AWS-focused groups on platforms such as LinkedIn and Reddit. By participating in discussions, sharing insights, and staying updated on AWS news, members can enhance their knowledge and connect with other AWS enthusiasts.
- **AWS blogs**: To share your unique experiences, best practices, and learnings about adopting AWS services, you can collaborate with your AWS contacts to co-author blog posts that are published on the official AWS blog channels. AWS strongly encourages customers to participate in such activities to share knowledge and learn from each other.
- **Open source contributions**: Contributing to AWS-related open source projects on GitHub allows you to share your own AWS tools, scripts, or templates with the community. By doing so, you can collaborate with other developers, enhance your skills, and help build a collective knowledge base for AWS users.

Summary

In this chapter, we learned about how the shared responsibility model impacts the resilience of applications and infrastructure. Through some examples, we learned about the various reasons why choosing a fully managed AWS service could contribute to increased resilience and reduced cost compared to self-managed workloads on AWS. As emphasized in this chapter, it is important for users to make conscious efforts to calculate the TCO before making decisions. We also learned about the importance of continuous testing and the tools you can use to perform these tests in an automated fashion, which helps improve the resilience of the infrastructure significantly. The **AWS Well-Architected Framework** contains specific guidelines under the Resiliency pillar for users to follow, which we will talk about in the next chapter.

Part 2: Building Resilient Cloud Architectures on AWS

This part covers the core principles and practices for designing and implementing resilient architectures on the AWS cloud. It provides practical guidance on implementing redundancy, loose coupling, error handling, observability, and regional resilience strategies.

This part has the following chapters:

- *Chapter 6*, *Learning AWS Well-Architected Principles for Resiliency*
- *Chapter 7*, *Architecting Fault-Tolerant Applications*
- *Chapter 8*, *Resiliency Considerations for Serverless Applications*
- *Chapter 9*, *Using Containers to Improve Resiliency*
- *Chapter 10*, *Resilient Architectures Across Regions*

6

Learning AWS Well-Architected Principles for Resiliency

Building resilient applications on the cloud is a critical requirement for modern businesses. Resilience ensures that your applications can withstand failures, recover quickly, and continue to operate seamlessly, minimizing downtime and its associated costs. The **AWS Well-Architected Framework** (or Well-Architected for short) provides a comprehensive set of best practices and guidance to help you design and build resilient architectures on AWS. By understanding and implementing the resilience-focused recommendations from Well-Architected, you will be better equipped to design and operate robust, resilient applications that meet your business requirements for availability, durability, and fault tolerance. This knowledge will help you minimize downtime, reduce the impact of failures, and ensure that your applications can recover quickly, ultimately improving customer satisfaction and protecting your business from the consequences of disruptions.

This chapter will cover the following topics:

- Gaining Operational Excellence for improved resilience
- Building reliable architectures
- Scaling applications to meet demand
- Architecting cost-effective resilience
- Implementing security for improved resilience

Technical requirements

This chapter provides a few practical examples to demonstrate a few concepts. To use them, here are some requirements to have installed in your environment:

- A major requirement is an AWS account, preferably a non-production account, with enough permissions to create/delete, **Virtual Private Cloud** (**VPC**), and associated resources (subnets,

route tables, internet gateway) and EC2 instances. It's generally recommended to follow the principle of least privilege and grant only the necessary permissions required for the specific tasks. You can create an IAM policy with the required permissions and attach it to an IAM user, group, or role as needed.

- A Bash terminal is also a must. If you don't have a Linux host, you can use AWS CloudShell.
- The **AWS Command-Line Interface** (**AWS CLI**). AWS CloudShell has the latest version of the AWS CLI pre-installed (`https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html`).
- Terraform (`https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli`) is also required.
- Git (`https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`) is also required.

Gaining Operational Excellence for improved resilience

The Operational Excellence pillar streamlines cloud workload operations through standardized processes, automation, and monitoring. It aims to simplify administration tasks, enhance system availability, improve customer satisfaction with consistent performance, and accelerate innovation through efficient iteration. Key aspects include change management, automated event response, root cause analysis, and continuous learning from operational events.

So how does that relate to reliability? If we look into the key design principles of the operational excellence pillar we can find performing operations with code, making frequent small, implementing reversible changes, refining operation procedures frequently, anticipating failures when they happen, learning from failures, leveraging AWS managed services, and implementing observability for actionable insights.

Performing operations as code

In the cloud, most operations can be performed through **Application Programming Interfaces** (**APIs**). AWS provides low-level service APIs that allow programmatic access to provision and manage resources. AWS also offers the AWS CLI, which is built on these APIs to enable infrastructure automation through the command line. With the AWS CLI, you can list and view the current state of resources such as EC2 instances, VPC networks, and Lambda functions. The CLI also allows you to create, modify, and delete resources to orchestrate your infrastructure. For example, you can launch new EC2 instances, create a VPC, or deploy a Lambda function by running CLI commands that call the underlying AWS service APIs. By leveraging the AWS CLI and service APIs, you can automate and script infrastructure management without needing to manually click through the AWS console. This makes it easier to treat infrastructure as code and replicate AWS environments through automated scripts rather than manual configuration. AWS CLI is most suited for shell scripting.

Here's a sample application we will deploy on AWS using only the AWS CLI. You can see its architectural components in this diagram.

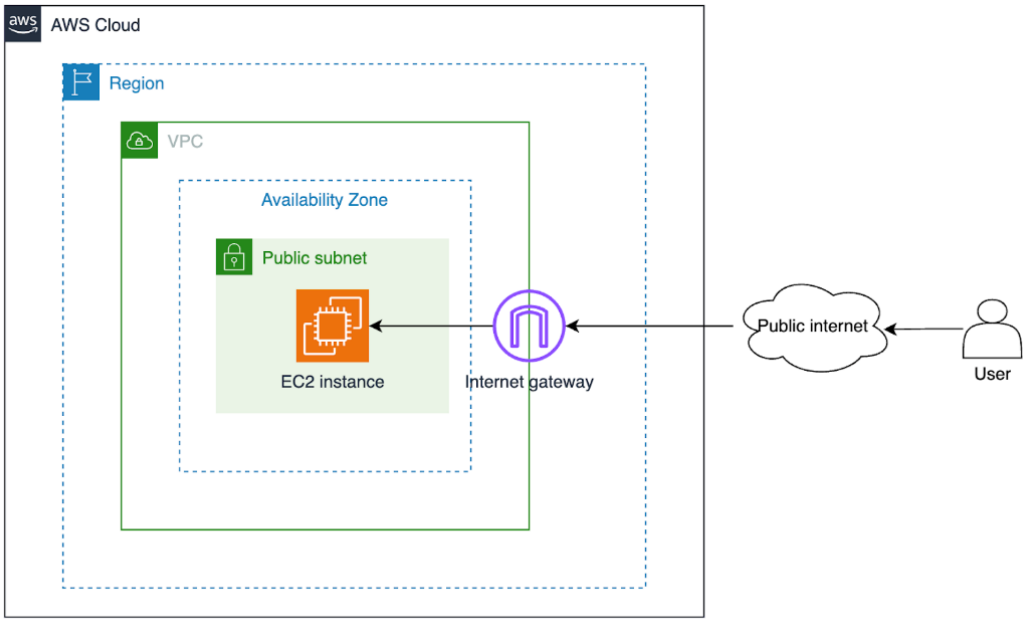

Figure 6.1 – Simple application diagram

To create this architecture in your AWS account, we have provided a sample Bash script that uses AWS APIs to deploy a web application all the way from the network building blocks. Execute these commands to see it in action:

```
$ git clone https://github.com/PacktPublishing/Building-Resilient-Workload-on-AWS
$ cd Building-Resilient-Workload-on-AWS/Chapter6/1-aws-cli
$ ./cli.sh
```

Important note

Running those scripts will incur charges on your account. Clean up as fast as possible. To clean up, run the `cleanup.sh` script in the same directory.

A few minutes after running the script, you will see a URL displayed on your terminal that will show an Apache web page if you open it on your browser, as displayed here:

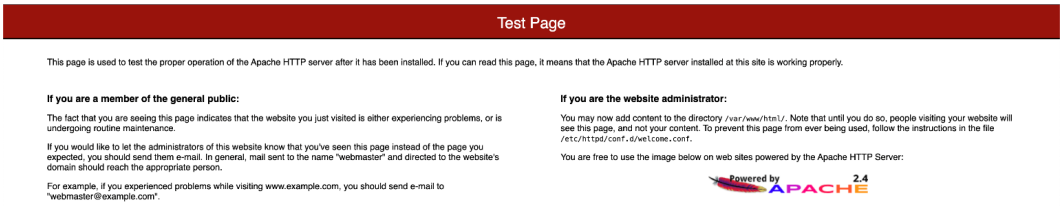

Figure 6.2 – A simple application test page

You can see with this example how a script can automate error-prone tasks. Although the AWS CLI is great for shell scripting, it can become complex to handle failure scenarios and edge cases. For example, if you hit a VPC limit in a region, CLI scripts may need to restart after the limit is raised based on custom logic. Additionally, scripts need awareness of the current state to support idempotency. This is where **Infrastructure as Code** (**IaC**)frameworks such as AWS CloudFormation, the **AWS Cloud Development Kit** (**AWS CDK**), Terraform, and Pulumi shine compared to CLI scripting. Some of the open source tools provide reusable constructs for defining AWS resources using the same underlying APIs as the CLI.

However, they handle failure cases, maintain state, manage concurrent runs, and avoid deadlocks without complex script logic. For instance, Terraform can determine what AWS resources need to be created versus modified to reach your desired end state. IaC frameworks also support modularity and abstraction for easy reusability.

AWS CloudFormation provides IaC capabilities (JSON and YAML) as a fully managed service. This can offer advantages over open source options in terms of state management, access controls, and not having to maintain tooling. For example, CloudFormation centrally manages the state and provides rollback, avoiding issues such as accidentally deleting Terraform state files. You only pay for the deployed AWS resources, not for CloudFormation itself. CloudFormation also integrates tightly with IAM and other AWS services.

There are also ways to gain CloudFormation benefits while still using programming languages such as JavaScript, Python, and so on. The **AWS Cloud Development Kit** (**AWS CDK**) allows you to define infrastructure in code using TypeScript, JavaScript, Python, and other languages without needing to use YAML/JSON directly. Overall, CloudFormation provides enterprise-grade IaC capabilities as a service while still allowing integration with DevOps-centric tools.

The choice of IaC tooling can spark lengthy debate, as each organization has its own requirements and policies. However, there are some key considerations:

- **Simplicity is an ally**: Simpler IaC approaches tend to be more resilient

- **Evaluate engineering skills as well**: Using an unfamiliar language such as CDK Go when, for example, only 10% of the engineers in your company know Go adds significant complexity
- **Idempotence and consistency**: For example, compared to a bash script, Terraform lets you run `terraform apply` multiple times, from any host, by any engineer, and reach the same end state

Overall, consider how well an IaC tool aligns with your team's skills, your environment's complexity, and your needs for consistency and reproducibility. Lean toward simpler solutions with built-in state management rather than complex scripting. Also, leverage IaC frameworks suited to your tech stack that will make infrastructure management easy, reliable, and scalable as your organization grows.

Here's the Terraform version of the Bash script we used before:

```
$ cd Building-Resilient-Workload-on-AWS/Chapter6/2-terraform
$ terraform init && terraform apply
```

To clean up, simply run the command that follows. Notice the difference with cleanup.sh, where the terraform config alone is enough to create, modify, and delete resources:

```
$ terraform destroy
```

With infrastructure defined and provisioned as code, the next enabler of operational resilience is making frequent, small, reversible changes to your workloads. The following section covers strategies and practices for incremental, safe deployments on AWS.

Making frequent, small, reversible changes

Now that we have IaC, we can make changes to infrastructure in the same way we make changes to application code – through small, reversible, and frequent updates. With IaC, infrastructure changes should be small and incremental, allowing for changes to be easily rolled back if needed. **Version control systems** such as Git enable tracking changes to IaC definition files in the same way as application code. Pull requests and code reviews allow collaboration and safety checks on infrastructure changes. **Continuous integration** and **continuous deployment** automate testing and the rollout of infrastructure changes side-by-side with application updates. By using IaC and applying proven software engineering patterns, we gain agility, safety, and collaboration in our infrastructure changes. The same principles of small, reversible updates applied to application code now also apply to your server fleet, networks, and cloud environments thanks to IaC. Infrastructure can now move at the speed of software development.

This entire workflow can be visualized in the following example, where you can see a continuous process from a user submitting changes to the infrastructure via code. The changes are applied by the CI/CD pipeline to the live infrastructure.

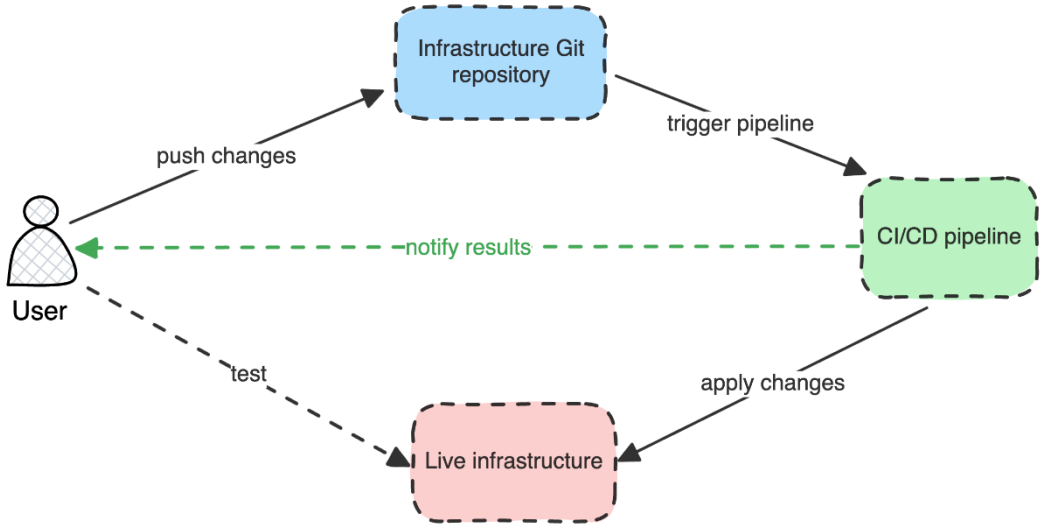

Figure 6.3 – Infrastructure automation

To effectively make small, frequent infrastructure changes, your overall architecture should be loosely coupled, with small modular components that can be updated independently. This limits the blast radius of any failed deployments, allowing for easy rollback. There are a few patterns to follow: incremental rollouts deploy changes to a few instances first before updating all resources. This lets you test in production with reduced risk. Similarly, **canary deployments** allow you to roll out applications progressively by splitting traffic between an already-deployed version and a new version. A subset of your user base would get the new version so you can gain quick feedback to decide how to move forward. If the canary succeeds, roll forward; otherwise, roll back. **Blue-green deployments** provision separated environments, then switch incoming user traffic from old blue to new green once validated. All these methods make changes in small steps, allowing you to continually deliver infrastructure safely.

An example of blue-green deployment looks like the figure that follows, where you can provision an entirely new subset of the infrastructure, along with the new version application. This allows for testing without compromising real user experience. Once the changes are valid, you can simply flip a DNS switch and route all users to the new version.

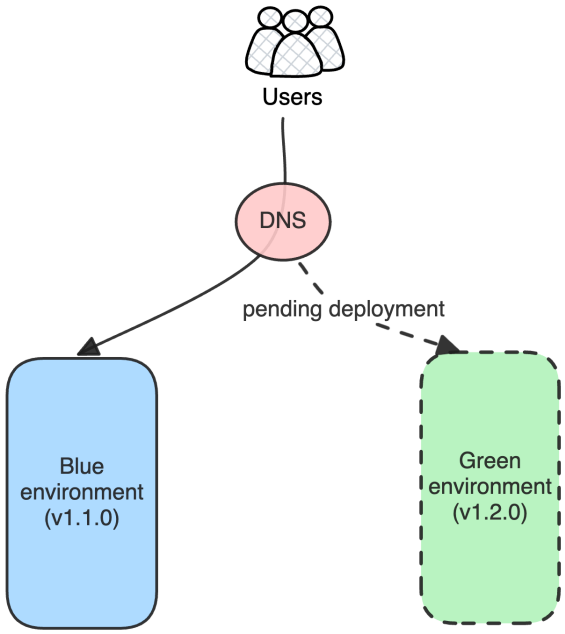

Figure 6.4 – Blue-green deployment

Combined with automation, small incremental updates to loosely coupled architectures let you evolve infrastructure fearlessly like code, enabling velocity while ensuring resilience.

Beyond architecture, the organizational culture needs to embrace agile methodologies and rapid release cycles. Tooling alone does not enable velocity – the people and processes are critical. There must be shared goals across teams to align priorities and resources. Everyone should understand how their work contributes to business outcomes. With the right automated IaC tooling, modular architectures, and a culture of ownership, collaboration, and iteration, organizations can deliver changes quickly and safely. Infrastructure and application code can move in lockstep. Failed changes become learning opportunities rather than crises. By applying complementary principles across people, processes, and technology, companies can unlock the full benefits of IaC, allowing infrastructure to be an enabler of innovation rather than a bottleneck.

Refining operations procedures frequently

An important aspect of operational resilience is defining procedures so that team members know how to respond when failures occur. As operational excellence dictates, we should outline playbooks and runbooks so responders can follow clear steps for detection, investigation, and recovery. Although we aim to automate as much as possible, some human intervention will be necessary during incidents. Having defined procedures gives on-call staff guidance on which dashboards and observability tools – CloudWatch, X-Ray, Managed Prometheus, and Managed Grafana – to consult, which scripts to run,

and which steps to take even when woken up at 3 A.M. Well-documented playbooks help minimize additional damage and speed up resolution when lack of sleep may hinder focus and clear thinking.

Let's say our application relies on customers being able to self-register and pay for access to features. The application is decoupled, and registrations are processed from a queue and written to a database. At 2:45 A.M., a CloudWatch alarm wakes you up, indicating increased latency in response times, which damages our **Service-Level Agreement** (**SLA**) and directly impacts revenue. In this critical moment, having a detailed runbook is invaluable. The runbook directs you to confirm the issue on specific dashboards, then use X-Ray traces to isolate the root cause as high latency from the database tier. The runbook could provide the exact SQL script to run to clean the stuck DB operation. By following the pre-defined runbook, you can quickly mitigate the issue and limit damage to the business, even when lacking sleep and focus. Well-designed response procedures are crucial for maintaining resilience when critical systems fail.

A best practice under the operational excellence pillar is to frequently review and evolve operational procedures and runbooks. Service owners should gather regularly to update these living documents based on recent experiences. Take an opportunity to celebrate operation wins, cross-feed a culture of improvement across teams, review and update dashboards, review postmortems of operational events, and take action. In particular, after major incidents such as the database latency issue, it is important to conduct a blameless postmortem. The goal of a blameless postmortem is to thoroughly understand what happened, why it happened, and how to prevent it in the future without assigning blame to individuals. As Google's *Site Reliability Engineering* book details, blameless postmortems focus on systemic issues and process gaps, not mistakes made by people. By adopting a blameless culture, teams can openly discuss errors, learn from failures, and improve procedures without fear of punishment. The insights from each postmortem should feed back into enhancing detection, investigation, and recovery processes. This continuous improvement of runbooks and learning from outages is crucial for advancing operational resilience over time.

After conducting a blameless postmortem on our database latency issue, we identified several bottlenecks to improve. We decided to implement queuing as soon as a user registers to decouple the initial subscription from payment processing. Additionally, we added a queue after payment is complete to finalize account setup in a separate step. These changes mitigate the original root cause. However, they also make portions of our runbooks and dashboards outdated.

From a simple application like in *Figure 6.1*, you can see how the architecture evolves and introduces new components after operational events.

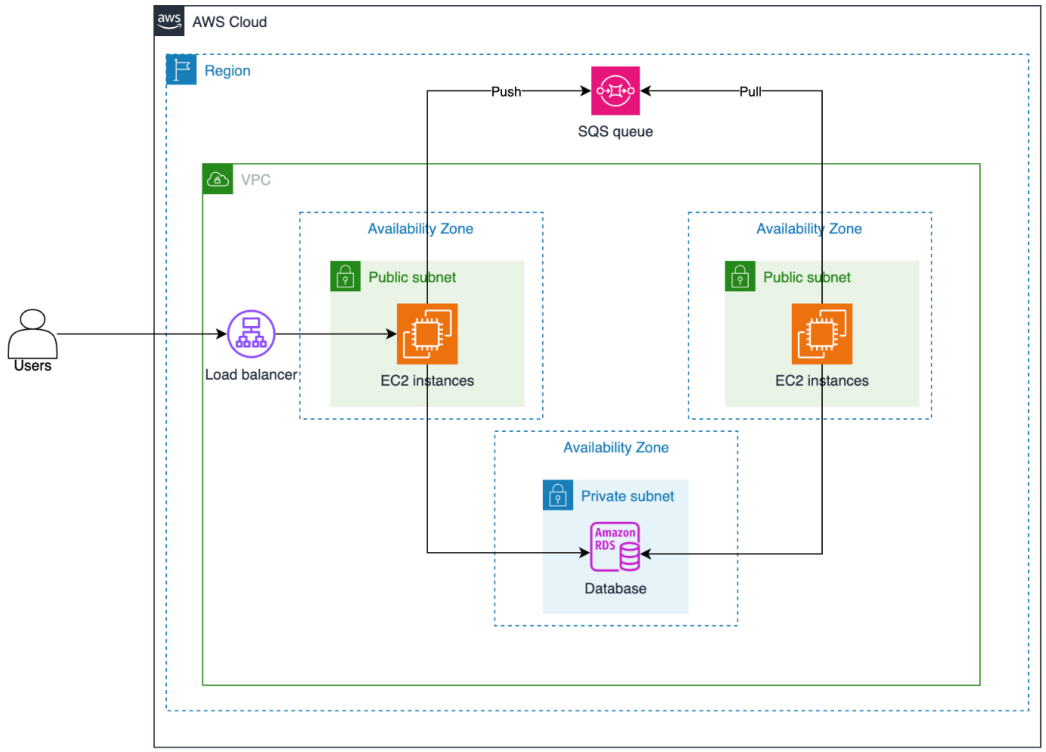

Figure 6.5 – New architecture with queuing

It is critical that we communicate documentation changes to all stakeholders involved in incident response. The runbooks and observability tools should be updated to reflect the new queues and system architecture. Outdated documentation could lead to confusion during future outages. Updating our procedures is part of the continuous improvement loop. As we refine our systems for greater resilience, we must also evolve our operational playbooks accordingly. Keeping these living documents in sync is key to maximizing their value during incidents.

Anticipating failure

A key tenet in resilience engineering is that *everything fails all the time*. You have probably seen that quote multiple times in this book already. Rather than vainly trying to prevent all failures, we must architect systems under the assumption that components will inevitably fail. Though we cannot predict every possible scenario, we can leverage experience, also called *production scars*, and monitoring metrics to hypothesize likely failures. For our subscription queue evolution, we can anticipate the

queue growing if too many messages build up, or buggy releases poison the queue with malformed messages that prevent processing. To proactively validate these failure theories, we could intentionally inject faults during testing to observe system behavior. If our hypotheses are correct, we may uncover vulnerabilities that should be addressed before reaching production. Of course, we cannot foresee every failure mode, but this mindset of proposing and validating failure scenarios builds intuition and resilience. The failures will happen regardless, so our systems and procedures must evolve to better withstand the inevitable unknowns.

The practice of proactively testing failure modes has evolved into a discipline known as **chaos engineering**. The goal of chaos engineering is to improve system resilience by intentionally injecting faults into production environments in a controlled manner. Services such as AWS **Fault Injection Service** (**FIS**) allow engineers to carefully craft experiments that induce failures such as high CPU, network delays, instance shutdowns, and more. Teams can start with lower-risk mock environments, and then gradually test production systems once procedures are refined. Each experiment should have a defined hypothesis and expected result. If the system behaves differently than anticipated, the learnings fuel further resilience improvements. Chaos engineering provides a powerful way to validate the ability to withstand inevitable real-world failures. When thoughtfully applied, these controlled experiments build confidence and uncover weaknesses before they cause customer-impacting events. Chaos engineering is explored further in *Chapter 13*.

Operational excellence can be reinforced through fun *game days* (`https://aws.amazon.com/buildon/gamedays/`) that simulate failures. Game days bring together teams to roleplay responses to simulated production incidents. They help refresh knowledge as architecture evolves, people leave companies, and knowledge is lost. By replicating production environments through IaC, realistic scenarios can be staged for teams to detect, investigate, and mitigate. Game days not only test operational readiness but also uncover gaps in runbooks, monitoring, and coordination between teams. They provide a safe space to learn and improve skills in incident response. When run regularly, game days keep staff sharp, validate new hires, and ensure documentation stays up to date. Gamifying failure scenarios through competitions and leaderboards taps into human competitiveness to drive engagement. Game days are a powerful tool for maintaining and advancing operational excellence over time as conditions continuously change.

Using managed services

A key way to enhance operational resilience is by using fully managed AWS services whenever possible. By relying on services such as Amazon RDS instead of self-managed databases on EC2, we gain the benefits of AWS's world-class operational practices without having to build them ourselves. Managed services reduce our maintenance overhead so we can focus on core business logic. AWS handles replication, failover, patching, backups, and other complex operations that are easy to get wrong when self-managed. AWS has far more operational expertise than any single company could hire. In our example, services such as Amazon SQS remove the need to develop queueing systems from scratch. Of course, managed services won't be a fit for every unique workload. However, in general, leveraging AWS's operational excellence lifts much of the burden from our shoulders. AWS' track

record of availability and performance is far superior to what we could achieve on our own in most cases. Offloading undifferentiated heavy lifting to AWS increases our resilience posture.

It's important to note that managed AWS services do not completely absolve us of operational responsibilities. The **shared responsibility model** dictates that we are still accountable for factors such as data security, identity management, and architectural best practices when leveraging services. For example, even if we use Amazon RDS, we must still define operational procedures around database backups, failover testing, and access controls. Managed services reduce but do not eliminate many operational concerns. We still need to monitor service health, thoroughly test changes, and run game days to validate our responses to potential AWS-related incidents. Our architecture decisions directly impact the resilience of systems built on AWS. So, while they provide a strong foundation, we must continue to invest in operational excellence through regular reviews, training, and validations to minimize business disruption. The shared responsibility model balances the advantages of managed services with user accountability.

Implementing observability for actionable insights

A critical pillar of operational excellence is the strong **observability** of system health and behavior. We cannot wait for customers to inform us of issues – robust monitoring provides the earliest possible detection of anomalies. Beyond just alerting, observability gives insight into the root cause by collecting metrics, traces, and logs. Metrics reveal overall trends and seasonal traffic patterns, allowing **predictive scaling**. **Distributed tracing** maps the flow of transactions across services to pinpoint latency sources. **Structured logging** captures detailed diagnostics on application execution and failures. Together, these data streams provide comprehensive observability so teams can quickly investigate issues. The famous quote rings true: *you cannot improve what you cannot measure*. Observability supplies the raw data to uncover optimization opportunities before they become outages. It shifts operations from a reactive to a proactive posture. With sufficient visibility, we can get ahead of many failures rather than waiting for reactive monitoring to alert us. The right telemetry is essential for rapid detection and diagnosis, as well as continuous improvement of operational excellence. Observability will be covered in greater detail in *Chapter 12*.

To maximize the value of observability, we must identify **Key Performance Indicators** (**KPIs**) and business objectives that define *what good looks like* for a system. With complex microservices architectures, there are endless signals we can monitor. We risk information overload without clear guidance on the metrics that matter most. Tying observability to KPIs and **Service Level Objectives** (**SLOs**) gives North Star metrics to orient around when investigating issues. For example, an e-commerce site should focus on monitoring checkout success rates, abandoned carts, and revenue trends. Aligning observability with business outcomes focuses attention on the signals that directly indicate customer experience and business impact. Observability without purpose leads to aimless monitoring. Defining objectives and KPIs gives direction to detect deviations quickly and understand which issues require urgent response.

To enable robust observability for our subscription service example, we will leverage several AWS services:

- Structured JSON logging from each microservice will be ingested into CloudWatch Logs for storage and analysis.
- X-Ray distributed tracing will capture full transaction details, including latency between steps.
- From the structured logs, we can derive custom metrics on sign-up conversions using regular expressions. These key metrics can trigger alerts to on-call staff without creating alert fatigue.

CloudWatch Synthetics canaries will mimic customer workflows to proactively detect issues before customers are impacted. CloudWatch Real User Monitoring will provide additional visibility into the true end-user experience. Together, these observability data streams will give us comprehensive insight into transaction lifecycles and business KPIs. We will know immediately if any step of the signup process slows down, or if errors increase. This observable system aligns with our core goals of minimizing latency, as well as maximizing sign-ups and retention. With these signals calibrated to business outcomes, we can maintain excellent customer experience. Observability will be explored further in *Chapter 12*.

Fostering an organizational culture for operational excellence

Before launching any new service, it is critical that we perform structured **Operational Readiness Reviews** (**ORRs**) and **Production Readiness Reviews** (**PRRs**). These proactive processes bring together stakeholders to validate that all dimensions of operational excellence have been addressed. ORRs and PRRs generate valuable documentation on system architecture, functionality, and procedures. They provide a framework for making risk trade-offs and gathering expert feedback to catch issues early. By taking a methodical approach, organizations can avoid major outages that result from rushing changes into production. Readiness reviews promote accountability across teams to uphold high operational excellence standards. They ensure all aspects are in place – monitoring, playbooks, capacity planning, rollback plans, and so on – before the software touches customers. Conducting rigorous readiness reviews epitomizes the proactive, preventative mindset required for operational excellence. They complement ongoing reviews by vetting readiness before deploying changes.

When applying operational excellence principles, we must tailor efforts appropriately based on business objectives. Not all applications require the same level of resilience investment. We should default to simplicity unless specific business needs dictate more sophisticated solutions. Beyond technical practices, organizational culture and structure impact operational success. A culture focused on incremental improvement through small, frequent changes is ideal. Best practices around incident response, observability, and change management should be shared across teams. SLOs help align priorities by framing operational excellence as a business enabler rather than just a cost center. Leadership buy-in and cross-team collaboration are key. Operations teams cannot drive resilience alone. Expertise must be distributed through training and knowledge sharing. With the right organizational support, operational excellence transitions from an aspirational goal to a practical reality.

Operational excellence is a critical enabler of resilience in the cloud, laying the foundation for robust, self-healing systems through best practices such as performing operations as code, making frequent, small, and reversible changes, refining procedures, anticipating failures, leveraging managed services, and implementing comprehensive observability. However, achieving true operational excellence requires fostering the right organizational culture, tailoring efforts based on business objectives, and aligning with SLOs. By combining technical best practices with the proper organizational support and mindset, companies can unlock operational excellence's full potential, enabling highly available architectures that can withstand failures and recover quickly. While operational excellence establishes the groundwork, the **Reliability pillar** of the AWS Well-Architected Framework provides specific guidance on designing and architecting fault-tolerant, highly available systems, which will be explored in the next section.

Building reliable architectures

Reliability is a critical pillar within the Well-Architected Framework for building robust workloads on AWS. It refers to a system's ability to perform its intended function correctly and consistently whenever called upon. To put it simply, a reliable system is one you can count on to work as intended when needed. Reliability encompasses the entire lifecycle of a workload, from initial design through deployment and operations, until decommissioning. A reliable system should operate without failure throughout its lifespan. This guidance will provide in-depth, best-practice recommendations for architects and engineers on how to build workloads that adhere to the Reliability pillar's principles and practices. Following AWS's reliability tenets allows organizations to create resilient workloads that meet customer expectations, prevent business disruptions, and avoid the costs of system outages. Reliability is foundational to operational excellence and business continuity. The Reliability pillar is centered on several core design principles. These include automatically recovering from failures, rigorously testing recovery procedures, horizontally scaling to increase aggregate availability, accurately predicting capacity needs, and managing changes through automation. We will cover disaster recovery and testing in *Chapter 14*. While operational excellence focuses on processes and procedures, reliability aims to build inherently resilient architectures. Reliable systems are designed to be highly available and withstand inevitable failures through redundancy, decoupling, and fault isolation. They are cloud-native architectures that leverage the elasticity and automation of AWS. By baking reliability into the foundation of system design, organizations can minimize outages and ensure workloads meet business needs. The practices under the Reliability pillar work in concert with operational excellence to deliver robust and resilient cloud applications.

Automatically recovering from failure

While monitoring business KPIs indicates overall system health, underlying components can still fail without impacting the whole system. As the aviation analogy illustrates, minor failures left unchecked can cascade into major incidents. Therefore, in addition to holistic monitoring, reliable systems should automatically detect and recover from component failures. Self-healing mechanisms that restore failed parts, retry transient errors, and redirect traffic away from unhealthy subsystems

maintain resilience without manual intervention. **Automated recovery** limits the blast radius when localized issues occur. Quickly restoring functionality and preventing propagation through isolation and redundancy is crucial. Reliable architectures cannot wait for human operators to notice and fix problems. They must be designed to automatically handle commonly anticipated failure scenarios; just as modern planes automatically compensate for equipment issues while notifying ground staff. Automatic recovery keeps small disruptions from becoming big outages.

When using managed AWS services such as SQS, we gain access to vendor metrics on queue performance. We should identify metrics tied to anticipated failure modes, such as queue size growing from backpressure. SQS provides queue length metrics we can baseline and alert on. We can also embed schema version metadata in messages, letting consumers automatically discard malformed messages by using the **dead letter queue** functionality in SQS. Monitoring the ratio of normal versus dead letter queue length detects bad releases poisoning queues. By leveraging vendor metrics and building automated remediation, we maintain resilience. If queues grow beyond thresholds, which indicates business health, we can scale consumers, pause producers, or drain queues manually. The key is detecting and recovering from common failure modes automatically by using events generated by Event Bridge and passing them to trigger orchestration tools such as AWS Step Functions. Reliable systems leverage vendor-provided signals and self-healing capabilities to stay available without manual intervention.

To scale reliably, systems should be decoupled as much as possible. Smaller, loosely coupled components can scale independently as needed. In our queue example, we can horizontally scale consumers to match throughput needs without scaling the front end. RDS Aurora's read replicas let us elastically scale database reads separately from writes. By monitoring queue depth and database utilization, we can trigger auto scaling rules to add consumers and read replicas during traffic spikes. Loose coupling and auto scaling enable scaling out at the bottleneck tier without affecting other layers. Well-architected distributed systems allow incremental horizontal scaling in an automated and granular way. This maintains reliability during load spikes and reduces the chance of overprovisioning resources.

When selecting AWS services, opt for managed and serverless options whenever possible. Services such as AWS Lambda and Amazon ECS make scaling faster and more granular than manually provisioning EC2 instances. However, for EC2, pre-baked **Amazon Machine Images** (**AMIs**) and immutable infrastructure remove drift and speed deployment. In general, by splitting business logic into smaller components, we can right-size each to the appropriate service, using Lambda for transient, event-driven functions and ECS or other container platforms for longer-running processing. While every workload is unique, defaulting to simplicity reduces maintenance overhead. Leveraging AWS' automation and managed services improves reliability by reducing our responsibility for capacity planning, OS patching, scaling, and code deployments. The more we can offload to AWS services, the less we have to get right ourselves. Choosing the right mix of services and architectures is essential for scaling reliably.

Capacity and quotas management

Unlike on-premises infrastructure, the cloud eliminates the guesswork for capacity planning by allowing on-demand consumption. AWS provides detailed usage metrics to track resource utilization. Responsibility for scaling capacity varies by service – Lambda and SQS automatically scale to meet demand, while EC2 requires configuring **Auto Scaling groups** (**ASGs**). By monitoring usage metrics and setting thresholds tied to business needs, we can trigger scaling to maintain performance without over- or under-provisioning. The key is architecting to leverage the elasticity of the cloud, whether automatic or configurable. Resilient systems ensure sufficient capacity to handle usage spikes and failover events.

Capacity is finite. Over-provisioning can be costly and misconfiguration errors can happen, especially when humans are involved. AWS enforces service quotas (formerly limits) to prevent over-provisioning risks such as unintended costs or resource exhaustion. Quotas limit resources per account and region, such as max EC2 instances or VPCs. Some quotas require manual increase requests. It's critical to monitor usage against quotas and proactively raise limits. Hitting a quota ceiling can directly cause failures, such as launching new EC2 capacity during a spike. CloudWatch quota metrics help track usage against limits. Ideal solutions automatically request quota increases via API when thresholds are crossed. Quotas protect against mistakes but can also cause outages if usage trends are not watched closely. Resilient architectures ensure sufficient quota headroom through careful tracking and automation. Quotas should be seen as helpful guardrails, not barriers, with overhead minimized through CloudWatch monitoring and automated increases.

Just as AWS applies quotas to protect itself and its customers, we as system owners should follow similar practices at the application level. Through testing and monitoring, we can discover the limits of each application component. For example, through load testing, we can find the maximum simultaneous connections a container or EC2 instance can handle before overload. In a database, we can determine the connection limit before rejections cascade into service disruption. In microservices, communication between services via messaging or API calls should have appropriate timeouts applied. When opening a connection, define a reasonable time to wait for a response before marking it as failed, avoiding long hangs that propagate failures. When a resource call fails, employ exponential backoff retries. Without thoughtful limits, client timeouts, and retries, poor implementations can do more harm than good. Well-chosen application quotas create helpful guardrails, allowing graceful degradation when failures occur. Just as broken escalators become stairs, our systems should provide minimum functionality during failures, though this isn't always possible. Like AWS quotas, application limits should be monitored and tuned based on real-world data. While building reliable architectures focuses on resilience and fault tolerance, the ability to scale applications seamlessly is another critical aspect of maintaining high availability on AWS. The next section explores strategies and AWS services for scaling workloads to handle fluctuations in user traffic and demand without impacting performance or availability.

Scaling applications to meet demand

We covered most of the performance efficiency aspects throughout the other pillars. However, as the Well-Architected Framework defines it, performance efficiency is about using computing resources efficiently to meet system requirements while minimizing costs. This pillar emphasizes eliminating anything that doesn't directly add business value. We can summarize its learnings in the following points:

- **Diversifying instance types**: Using a diverse mix of instance families and types in your ASGs is an important resiliency strategy. Relying on a single instance family or size could limit your ability to scale if that particular instance type is unavailable. By utilizing multiple instance types tailored for different workload needs – such as compute-, memory-, or storage-optimized – you build redundancy into your architecture. If a particular type of instance that your application favors becomes constrained, auto scaling can launch an alternative instance family that still meets the core resource demands. As you adopt more managed compute services such as AWS Fargate, the undifferentiated heavy lifting of capacity provisioning and infrastructure resilience is handled by AWS, allowing you to focus on your applications.
- **Using predictive scaling**: Going further, you can not only monitor and scale reactively but also be proactive and have scheduled scaling if you know your workload patterns. For example, when all your users connect at once, it can be hard to meet the demand instantly. If you use EC2 instances and Auto Scaling, Predictive Scaling (`https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-predictive-scaling.html`) is a feature that can help you with forecasting machine learning models to anticipate the capacity.
- **Keeping track of the newest AWS releases**: AWS launches new capabilities at a rapid pace to solve customer challenges and meet emerging needs. What is difficult or complex to build today may be simplified by an AWS announcement tomorrow. Regularly reviewing AWS news feeds, blog posts, social media channels, and the console can uncover new features that streamline your architecture. For example, a managed capability may now exist to replace a solution you previously self-managed. Additionally, engaging with AWS representatives can provide valuable visibility into upcoming releases that may soon make your life easier. You can also leverage AWS-published reference architectures that demonstrate proven practices for building resilient, high-availability systems on AWS across various domains. Architecting with a future-ready mindset will enable you to rapidly adopt the latest resiliency advances from AWS.

While building reliable architectures is crucial for ensuring system availability and resilience, it is equally important to design systems that can scale seamlessly to meet fluctuating demand. The ability to scale resources up or down in response to changes in traffic or workload is a key advantage of cloud computing, and it is essential for maintaining performance and avoiding disruptions. In the next section, we will explore the principles and best practices for scaling applications on AWS.

Architecting cost-effective resilience

While resiliency maximizes system availability, it can also introduce additional infrastructure complexity that impacts costs. The AWS Well-Architected Cost Optimization pillar demonstrates how to build resilient systems while optimizing your AWS spend. By architecting solutions that take advantage of AWS's global infrastructure and managed services, you can reduce costs through operational excellence while automating resilience capabilities. Using serverless and containers enables inherent resiliency and auto scaling capabilities without overprovisioning. Continually evaluating utilization metrics and right-sizing based on data empowers you to optimize costs for resilient workloads. However, excessive focus on cost optimization can lead to architectural choices that reduce resilience, fault tolerance, and availability. Careful analysis of the tradeoffs between resiliency patterns and cost reduction strategies is required to balance both objectives. With governance, automation, and data-driven insights you can deploy architectures that achieve cost-efficient resilience.

When architecting resilient solutions on AWS, cost optimization considerations should remain ever-present to guide decisions. Achieving maximal resilience capabilities may carry a high price tag, so target resilience levels should directly map to business requirements and risk tolerances. For example, replicating critical production workloads across multiple regions enhances fault tolerance and availability, but at increased infrastructure expenses. Compare the return on investment of high resilience against the business objectives to right-size architectures appropriately. Approaching new projects with a cloud cost optimization mindset baked in from inception can reveal more efficient ways to build resilience capabilities. With AWS' variable pricing, you only pay for the resilience capabilities that align with the mission outcomes demanded by the organization. For every architecture decision towards greater resilience, there is a cost associated that should be put in the balance, without losing sight of the business' return on investment.

Implementing security for improved resilience

The AWS Well-Architected Security Pillar outlines critical best practices for building secure cloud architectures that are equally vital for building resilient systems. While foundational security protects against external threats, by holistically incorporating these security design principles, you also enhance the reliability, fault tolerance, and availability of your systems. To reduce risk from within, apply least privilege access, encrypt data both in transit and at rest, enable traceability through logs, and implement rigorous identity and access controls. Additionally, limit the blast radius to further curtail potential threats. Detecting events and responding rapidly becomes easier when you have baseline-secured architectures in place. Just like with resilience, achieving effective security necessitates culture change, cross-team collaboration, and centralized governance across your organization. With modern automated security tools and AWS' shared responsibility model, cloud-native security fosters operational excellence. AWS CISO recommends **Zero Trust architectures** (see `https://aws.amazon.com/blogs/security/zero-trust-architectures-an-aws-perspective/`, where they expand on the preceding concepts).

As we discussed in previous chapters, AWS also applies the shared responsibility model for security, so you only manage the security of your data and applications. Depending on the service, the responsibility will change.

The following sections dive deeper into key security areas that directly impact resilience – identity and access management, governance, protection, and incident response. Implementing robust controls in each of these domains hardens your systems against threats while enabling faster recovery when issues occur.

Identity and access management

Strict **identity and access management** is critical for resilience by ensuring only authorized entities can access AWS environments. Uniquely defining identities for every user, application, and resource enables properly scoped access. Leverage AWS identity services such as IAM for users and roles or **IAM Identity Center** (formerly **SSO**) for identity federation and Amazon Cognito if you need to grant applications for consumers or customers access to your AWS resource. Protect root credentials obsessively, only using them when absolutely necessary, and enable **Multi-Factor Authentication** (**MFA**) for all users, especially administrators. Ideally, we should be able to have emergency procedures for security issues, the same way we have runbooks. Organizing users into IAM groups with least privilege policies or applying **Attribute-Based Access Control** (**ABAC**) simplifies appropriate access.

Leveraging IAM roles instead of keys enables least-privilege access for AWS services. EC2 instances should have individual roles granting precisely scoped permissions rather than generically elevated rights. Similarly, AWS services such as Lambda, ECS, EKS, and more are configured with IAM roles dictating their access permissions. While IAM best practices focus on security, they also help limit the blast radius if credentials are leaked. If an attacker gains access to one system, they should not be able to traverse horizontally across an account or vertically to other environments. Quickly rotating compromised credentials or deprovisioning user access reduces dwell time. Apply identity management fundamentals before disasters strike to have controls in place to manage incidents. The Well-Architected Identity pillar covers these in more detail, but resilience similarly relies on identity hygiene.

For robust security-based resilience, utilize AWS secrets management services such as Secrets Manager and Systems Manager Parameter Store rather than hard-coding credentials in code or configs, as centrally storing secrets lets you rotate credentials without necessarily redeploying applications. Auditing permission usage history via CloudTrail provides visibility into access trends and can detect surges in access denials indicating a policy misconfiguration or breach. By implementing formal secrets management and monitoring for credential misuse, you give yourself resilience by separating code from credentials, enabling rotation, and detecting identity issues early before they cascade into system failures. Keeping access credentialing dynamic and auditable reduces your downtime risks from expired, leaked, or inadequate permissions as architectures evolve over time.

Avoid taking shortcuts with broad permissions that could expose resources later. For example, in our subscription application, EC2 instances should have a role allowing only necessary SQS and CloudWatch actions. If you're adding S3 file storage, add the bucket ARN in the policy to provide

access to the desired S3 bucket. Regularly review policies as architectures evolve to trim unnecessary access. Automated testing and IaC help catch permission issues early while facilitating policy updates. However, don't remove permissions prematurely before older instances and deployments conclude accessing those resources. Meticulous management of service roles provides the minimum required access for maximum resilience through change.

Services such as **IAM Access Analyzer** will help you achieve the least privilege by continuously looking at the permissions you created and providing you with insights to refine them. AWS Trusted Advisor will also help you optimize many Well-Architected pillars, including Performance, Security, Resilience, and Operations depending on your AWS Support subscription level. Read more about Trusted Advisor in the AWS Documentation (`https://aws.amazon.com/premiumsupport/technology/trusted-advisor/`).

You can enhance resilience by thoughtfully structuring AWS accounts to isolate environments, blast radius, and access controls. Create separate AWS accounts for production, testing, and development environments instead of combining all resources within a single account. However, we need to ensure cross-account access follows identity best practices using roles and federation instead of long-lived keys and also centralize identity providers connected to all accounts for consistency. This facilitates a culture of experimentation and innovation without risking production stability and helps avoid issues such as development quotas inadvertently restricting production scale.

Sometimes, those accounts need to communicate. For example, microservices isolated into different production accounts may need to communicate to provide a feature. We still need to establish patterns and tools for multi-account management as your environments scale up. Set account quotas, tags for cost allocation, and consolidated billing to facilitate governance.

Enable advanced account security features such as **AWS Organizations** and **Service Control Policies** (**SCPs**) to apply permission guardrails, GuardDuty, and Security Hub at inception. By compartmentalizing accounts, you reduce risks, costs from issues spreading, and complexity for your operators. Account planning is foundational for us to segment access, security domains, and failure isolation. The AWS multi-account security and governance guide provides detailed best practices for you to build resilience through account management. AWS Control Tower is a fully managed service that can help automate the setup of multiple accounts by implementing all aforementioned best practices. Here's an example of how Control Tower and AWS Organizations can enhance security:

1. Set up AWS Organizations and create OUs for different environments (e.g., development, staging, and production).
2. Enable AWS Control Tower, which automatically sets up the landing zone and applies baseline security guardrails across your organization.
3. Define additional custom guardrails using SCPs and apply them to the appropriate OUs or accounts within AWS Organizations.

4. When you need to create a new account for a new project, use Account Factory to provision the account within the desired OU. Account Factory is a solution provided by AWS Control Tower that automates the process of creating and provisioning new AWS accounts.
5. The new account inherits the baseline security guardrails from Control Tower and the custom guardrails defined for the OU it belongs to.

Protection

We have seen ransomware attacks halt companies' operations when intruders take control of systems and data. AWS provides various network, compute, and data security mechanisms you can architect to restrict lateral traversal and blast radius. Segment your VPCs using subnets, **Network Access Control Lists** (**NACLs**), and security groups to allow only necessary traffic between tiers based on zero-trust principles. Inspect flows at edge networks, denying by default what isn't explicitly permitted. Protect compute layers by securing instance access, applying OS-hardening, and enabling host-based firewalls. Encrypt data at rest and in transit while controlling access. By implementing least privilege and inspection at infrastructure layers, you proactively minimize your attack surface.

Services such as CloudFront, Global Accelerator, API Gateway, and Application Load Balancer with AWS Shield provide **Distributed Denial-of-Service** (**DDoS**) protection capabilities while allowing you to globally distribute applications closer to users. CloudFront is a really powerful **Content Delivery Network** (**CDN**) service that helps you distribute your static and dynamic content quickly and reliably with high speed and points of presence around the world. You can even benefit from free SSL certificates managed by AWS when using CloudFront or importing your own. CloudFront is an example of undifferentiated heavy lifting as it is very easy to use and allows personalization with computing capabilities directly at the edge.

Earlier in the chapter, *Figure 6.5* showed the architecture of a subscription application example, decoupling incoming tasks with SQS queues. By applying security best practices, we can see the following in the next figure:

- A CloudFront distribution, protecting the application from potential DDoS attacks, is visible. CloudFront can cache common requests and reduce the load on the application. By forwarding specific HTTP headers from CloudFront to the load balancer and configuring the load balancer to reject any traffic that doesn't match these headers, we can establish mutual trust between CloudFront and the load balancer to make sure no traffic reaches out to the load balancer directly by passing CloudFront. You can also limit access to the load balancer using AWS-managed prefix lists.
- A security group for each layer authorizing only legitimate traffic is included. An example would be when the instances behind the load balancer would only allow traffic from the load balancer and the database would only allow traffic from the EC2 instances.

- An IAM role with specific permissions to read and write from the SQS queues is also included.

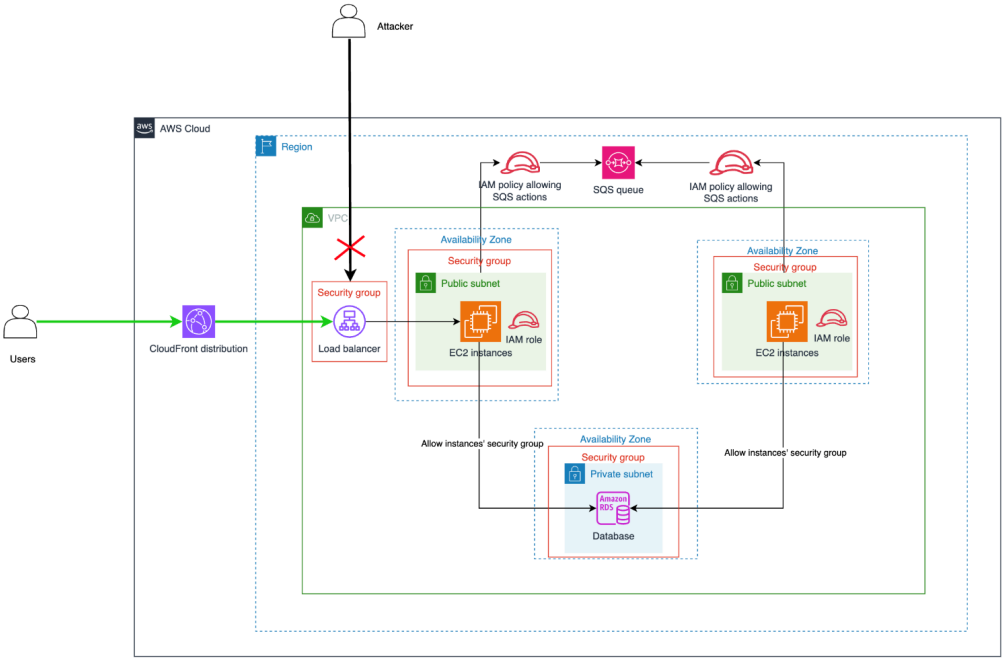

Figure 6.6 – Subscription management example architecture with security controls, including CloudFront, security groups, and IAM roles

If you manage customer data, you should have a data classification model by sensitivity, then apply encryption, tokenization, and access controls appropriately to protect information. **AWS Key Management Service** (**AWS KMS**) is a core AWS service that lets you easily create, manage, and control the cryptographic keys used for data encryption across AWS services and your applications. Many services leverage KMS by default with service-managed keys. For highly critical data, go further by creating and managing your own **Customer-Managed Keys** (**CMKs**) within KMS for greater control over encryption infrastructures. Rotating these CMKs periodically ensures keys are not accidentally exposed for long periods. With robust data classification, encryption, and access management powered by AWS IAM, you can architect resilient data protections aligned to your risk tolerance.

Resilient software development lifecycles test for vulnerabilities and dependencies before release. Incorporate scanning tools such as CodeGuru to leverage machine learning for identifying security risks. Establish automated pipelines for building, testing, and deploying releases to minimize human error. Use AWS Systems Manager Patch Manager to seamlessly patch fleets of EC2 instances. Leverage Amazon Inspector and Systems Manager to run security checks, then aggregate findings in Security Hub for central visibility. While AWS manages the security of the cloud, you remain responsible

for securing workloads and data within it. Prioritizing secure engineering and operations processes enhances resilience against emerging threats.

Incident response

Resilient architectures must plan for inevitable security events or system failures by having robust incident response capabilities. The **NIST cybersecurity framework** (shown in the following figure) outlines key incident handling phases – detect anomalies, analyze impact, contain spread, eradicate root cause, and recover operations. AWS Well-Architected reinforces establishing mechanisms to quickly detect, analyze, prioritize, isolate, remediate, and validate incidents through automation and instrumentation. Implementing mature incident response not only enables organizations to manage events but also helps meet compliance obligations for customer data protection. You need to keep in mind the goals of the incident that you design with all stakeholders, with the ultimate result of ensuring business continuity. In theory, you should run incident response simulations and use tools with automation to increase your speed for detection, investigation, and recovery. For security testing, you could apply a red and blue team approach, where both teams are security individuals playing a role in improving a company's security. A red team will play the role of an attacker when a blue team defends. In practice, it can be difficult to implement depending on the company's size and maturity. Gradually augmenting security can increase complexity – rely on AWS managed services to reduce your management overhead.

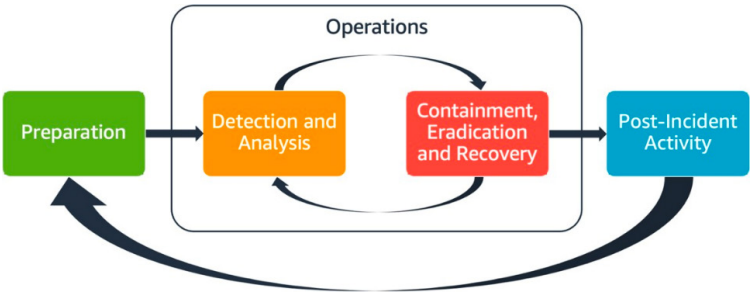

Figure 6.7 – NIST cybersecurity framework

Robust security measures are essential for building resilient systems on AWS. By following the AWS Well-Architected Security Pillar's best practices, such as implementing least privilege access, encrypting data, enabling traceability, and enforcing rigorous identity and access controls, you can limit the blast radius of potential incidents and minimize cascading failures. Key considerations include strict identity and access management, thoughtful governance and account structuring, network and data protection mechanisms, secure software development lifecycles, and robust incident response capabilities. Incorporating these security principles not only safeguards against external threats but also enhances overall system reliability, fault tolerance, and availability, ultimately contributing to improved resilience and business continuity.

Summary

In this chapter, we explored critical pillars of the AWS Well-Architected Framework such as operational excellence, reliability, security, efficiency, and cost optimization, specifically through the lens of architecting resilient cloud solutions. While each of these pillars plays a crucial role in enhancing resilience, they are interconnected and must be addressed in tandem, as we learned in this chapter. Well-Architected provides continuously updated tactical guidance on using AWS services purposefully to eliminate undifferentiated tasks unrelated to core business goals. We discussed leveraging reference architectures, infrastructure automation, robust monitoring, secure design principles, and cost-effective operations to build reliable and resilient architectures.

The key takeaways included using AWS managed services to reduce heavy lifting, automating deployments and procedures, conducting reviews to incrementally improve, scaling to demand, understanding system health proactively, and prioritizing security by design to protect months of innovation work. While Well-Architected covers many facets, adhering to its recommendations equips teams with best practices for creating resilient architectures able to rapidly detect issues, minimize impact, and reliably recover when inevitable yet unpredictable failures occur. Keep in mind that improving cloud resilience is an iterative journey that should be prioritized based on business requirements and risk tolerances. Next, we will see how to design fault-tolerant applications. We will explore best practices for building resilient applications on AWS that can withstand component failures.

7
Architecting Fault-Tolerant Applications

Fault tolerance is a system's ability to continue operating correctly in case of one or more failures of its components. Building resilient and fault-tolerant applications is a critical aspect of ensuring high availability and meeting business requirements in today's cloud-native world. System failures are inevitable and can occur due to various reasons, such as hardware failures, software bugs, network issues, or even human errors. Architecting applications to withstand these failures and continue operating with minimal disruption is essential for providing a seamless experience to customers and maintaining business continuity.

While fault tolerance and high availability are related concepts, they differ in their focus and approach. High availability aims to minimize downtime and maintain continuous operation, even during failures or maintenance, often through redundancy, failover mechanisms, and load balancing techniques. On the other hand, fault tolerance focuses on the ability to continue operating correctly despite failures, without necessarily recovering or switching components. It often involves techniques such as **error detection**, **error correction**, and **fault isolation** to prevent failures from propagating and affecting the overall system. Although fault tolerance and high availability can overlap, they have distinct implications.

In this chapter, we will explore architectural patterns and best practices for building fault-tolerant and highly available applications on AWS. We will learn about **redundancy**, **loose coupling**, **graceful degradation**, and fault isolation, which are fundamental to creating resilient systems.

This chapter contains the following topics:

- Leveraging AWS global infrastructure for redundancy
- Load balancing workloads across redundant systems
- Handling data redundancy
- Implementing loose coupling for isolating faults

Leveraging AWS global infrastructure for redundancy

Redundancy is a fundamental principle in designing fault-tolerant and highly available systems. It involves the intentional duplication or multiplication of critical components, such as hardware, software, or data, to ensure that if one component fails, there are redundant components available to take over and maintain system operation. Redundancy is essential for mitigating the impact of failures, which are inevitable in complex systems due to various factors, including hardware malfunctions, software bugs, network issues, or human errors. By incorporating redundancy into system architecture, organizations can achieve higher levels of reliability, minimize downtime, and ensure continuous operation even in the face of component failures. Redundancy is a proactive approach to fault tolerance, enabling systems to gracefully handle failures without compromising functionality or causing complete system outages. In the next sections, let's dive into how we can implement redundancy for systems.

Hardware or infrastructure redundancy

In the AWS cloud, our discussions usually center around services rather than hardware. Given that AWS oversees the hardware, as explained in the previous chapter on the **shared responsibility model**, the approach to redundancy is somewhat different. Even though redundancy ultimately means having multiple copies of infrastructure components, there are some fundamental constructs of AWS we can leverage.

Let's start with **AWS Regions**, also referred to as Regions. AWS Regions are physical geographic areas consisting of multiple, isolated locations called **Availability Zones** (**AZs**). Typically, when companies build on AWS, they will deploy their applications inside one region as a starting point and expand to multiple Regions if required by the business. Redundancy for Regions or multi-region deployment will be developed later. That said, before tackling multi-regions, there is a lot we can do inside a single Region, which is generally a wide geographic area.

Each AWS Region consists of at least three isolated, and physically separate, AZs within a geographic area with single-digit latency synchronous replication. AZs are designed for high availability and can be leveraged for fault isolation. To put that into perspective, a single AZ typically refers to at least three clustered data centers (closer together than two AZs) to provide redundancy and fault tolerance. This layered redundancy inside a Region provides a very high tolerance to faults and disasters. Read the AWS documentation to know more about the AWS global infrastructure key components (`https://docs.aws.amazon.com/whitepapers/latest/aws-fault-isolation-boundaries/aws-infrastructure.html`). AWS encourages applications to take advantage of this construct by deploying parts of the systems across multiple AZs, as much as possible. By leveraging AZs, we ensure redundancy, fault tolerance, and high availability for our applications. Even if one AZ experiences an outage or failure, the application can continue to operate from the other AZs. One of the exceptions to this construct is **High-Performance Computing** (**HPC**) workloads, where there's a strong constraint on latency and compute clusters must happen on the same physical host, meaning that multiple AZs cannot be leveraged. As you might have seen so far in this book, in architecture, there are no silver bullets, mostly tradeoffs.

After looking at redundancy at the physical infrastructure level, let's see in the next section how AWS core infrastructure helps achieve redundancy.

Leveraging AWS core infrastructure for redundancy

Many systems on AWS start with an AWS **Virtual Private Cloud** (**VPC**), which is a fundamental service that closely resembles a traditional network that you'd operate in your typical data center. A VPC is an isolated environment at the network level where you can provision and deploy AWS resources.

It includes subnets, which provide network segregation within the VPC, routing tables, network gateways, and other components that enable you to control your virtual networking environment.

By spanning a VPC across multiple AZs within a region, you can achieve redundancy and resilience for your applications and services. This allows you to run parts of your application resources, such as EC2 instances or containers, physically separated over great distances. In the unlikely event of an AZ disruption, the duplicated virtual hardware in other AZs can take over, ensuring continuity of operations. Additionally, regional services such as Amazon SQS or Amazon Lambda are built on top of multiple AZs on your behalf, providing built-in redundancy and fault tolerance.

In the previous chapter, we covered that one of the easiest ways to achieve better resilience is to leverage managed services, where the service applies operational excellence best practices, including security, high availability, fault tolerance, and disaster recovery, across multiple AZs by default. However, Zonal services offer single AZ options for specific use cases or cost optimization, such as the Amazon **S3 One Zone-Infrequent Access** (**S3 One Zone-IA**) storage class, which stores data in a single AZ to reduce costs for infrequently accessed data that doesn't require multi-AZ redundancy. Another example is Amazon **Relational Database Service** (**RDS**) with the Single-AZ deployment option, which provides a cost-effective solution for non-critical applications that can tolerate potential data loss in case of an AZ failure. Always check the AWS documentation for services you intend to use, so you understand their operating model.

In the next section, we'll work our way up the stack from the global infrastructure to applying redundancy with load balancing.

Load balancing workloads across redundant systems

To effectively leverage the redundancy provided by AZs, you need to employ load-balancing techniques, especially if you receive traffic from outside your system. Load balancing allows you to distribute incoming traffic across multiple targets, such as EC2 instances or containers, running in different AZs. At the time of writing this book, AWS **Elastic Load Balancing**, a fully managed service that, as the name implies, allows you to use load balancing capabilities at a very high scale, proposes three types of load balancers according to the usage.

Application Load Balancers are generally meant for HTTP and HTTPS traffic (layer 7, OSI model) and route traffic based on the content of the request, such as HTTP headers, paths, hosts, and more.

Network Load Balancers (**NLB**), however, are used for routing TCP (layer 4), UDP, and TLS traffic. NLBs trade features for extreme performance. The third one is a **Gateway Load Balancer** (layer 3) to deploy, scale, and manage virtual appliances such as firewalls, intrusion detection and prevention systems, and deep packet inspection systems.

For fault tolerance, they allow you to enable multi-AZ deployments and ensure that each target group has at least one target in each enabled AZ. Additionally, load balancers have a health check mechanism to monitor the status of registered targets. This way, when one of the targets becomes unavailable, the load balancer can automatically stop routing traffic to it. *Figure 7.1* shows a common architecture with load balancing across multiple AZs with traffic not routed to failed targets.

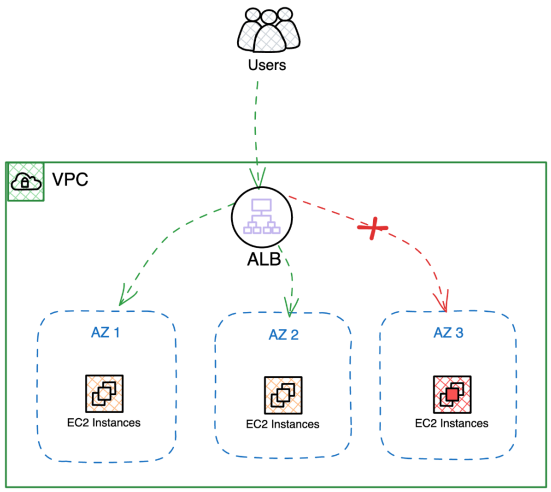

Figure 7.1 – Load balancing traffic towards healthy targets

Health checks are quite straightforward and should be fast so failures can quickly be mitigated. In general, health checks should be kept simple and consume very few resources (CPU cycles or memory). For example, if you run an HTTP server that serves an API, your health check would be a specific path that responds with `HTTP 200`, indicating that the server is alive:

```
$ curl http://localhost/health
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 12
(...Omitted for brevity)
```

It is tempting to go further with health checks and set up more complex scenarios such as checking external dependencies (databases, caches, third-party APIs, etc.) or simulating an entire user interaction. This may require more effort to implement and maintain and could lead to heavy resource consumption. The key here is to start simple and gradually introduce more elaborate checks as needed, if needed.

That said, regardless of their complexity, health checks can result in non-consistent errors, where a low percentage of requests fail. This is commonly known as **gray failures**. These gray failures are harder to detect and may require more sophisticated monitoring systems to identify and isolate the affected targets or AZs. By monitoring error rates, latency, and other performance metrics across AZs, we can identify gray failures. The following figure shows an example dashboard monitoring load balancer metrics across multiple AZs:

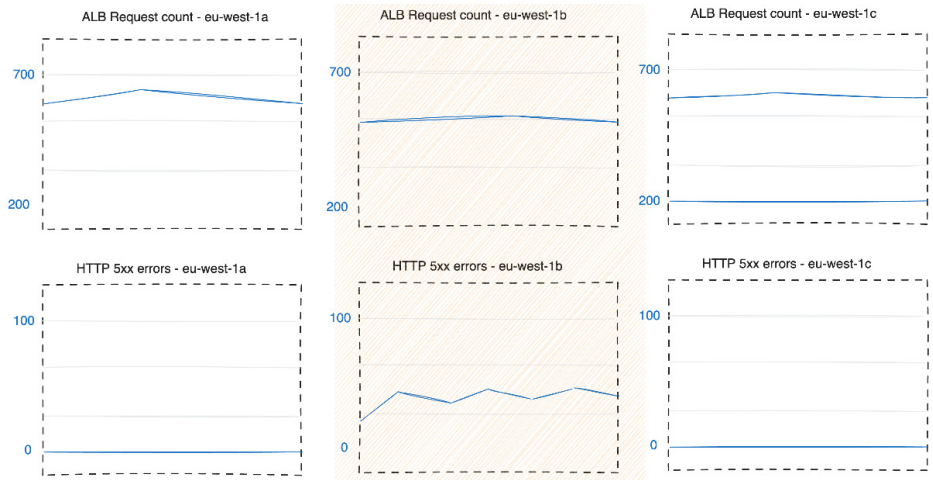

Figure 7.2 – Load balancer metrics on a dashboard

In the preceding figure, we see that those gray failures are isolated to an AZ as the request count is slightly lower in `eu-west-1b` (AZ b) than the other AZs, and we can see few `HTTP 5xx` errors in this AZ compared to other AZs. We can leverage AWS to consider them as hard failures (where everything fails consistently). For example, Amazon Route 53 Application Recovery Controller's **zonal shift** helps pause traffic to an entire AZ that we consider impaired, such as `eu-west-1b` in our example. It can be achieved through the load balancer quite easily. This allows you to continue operating from other healthy AZs while you dive into what is causing the issue. This can be done in the console or API, or with an AWS command line as follows:

```
$ aws arc-zonal-shift start-zonal-shift \
--resource-identifier arn:aws:elasticloadbalacing:...
--away-from euw1-az2
--comment "possible issue isolated to AZ2"
--expires-in 12h
```

Important note

Before turning off an AZ for your application, you need to make sure that you have the capacity to continue operating with one less zone. Otherwise, this can degrade your customer's experience even more.

AWS allows you to go further and automate this with the **zonal autoshift** feature to recover even more quickly from operational events. You hold the button to make the shift, but you must be prepared, and preparation comes with practice. Zonal autoshift will require you to schedule practice runs. In the beginning, you can start during non-business hours. With time and confidence, you'll be able to plan it during normal operating hours so when an event comes, you are prepared.

AWS will automatically shift application resource traffic away from an AZ on your behalf to help reduce the time required for recovery during disruptive events. To ensure that the zonal autoshift feature is safe for your application, you must configure and perform practice runs before enabling zonal autoshift for a resource. This practice run is a mandatory requirement to verify that you have adequate capacity across all AZs within the AWS Region, allowing your application to continue operating normally when traffic for a resource is shifted away from one AZ.

Redundancy can be extended beyond AZs to encompass entire application architectures. You can create multiple copies of your entire setup, including the VPC, applications, and load balancers, effectively treating each copy as a self-contained unit. This approach is commonly used in **blue-green deployments**, as discussed in *Chapter 6*, where you maintain two identical production environments (blue and green) and switch between them during deployments or failover events. The **Domain Name System** (**DNS**) plays a crucial role in enabling this level of redundancy. Amazon Route 53, AWS' highly available and scalable DNS service, allows you to manage the mapping between your domain names and the IP addresses or load balancers associated with each copy of your architecture. By updating the DNS records, you can seamlessly redirect incoming traffic from one environment to another. You can see how this comes into play in *Figure 7.3*.

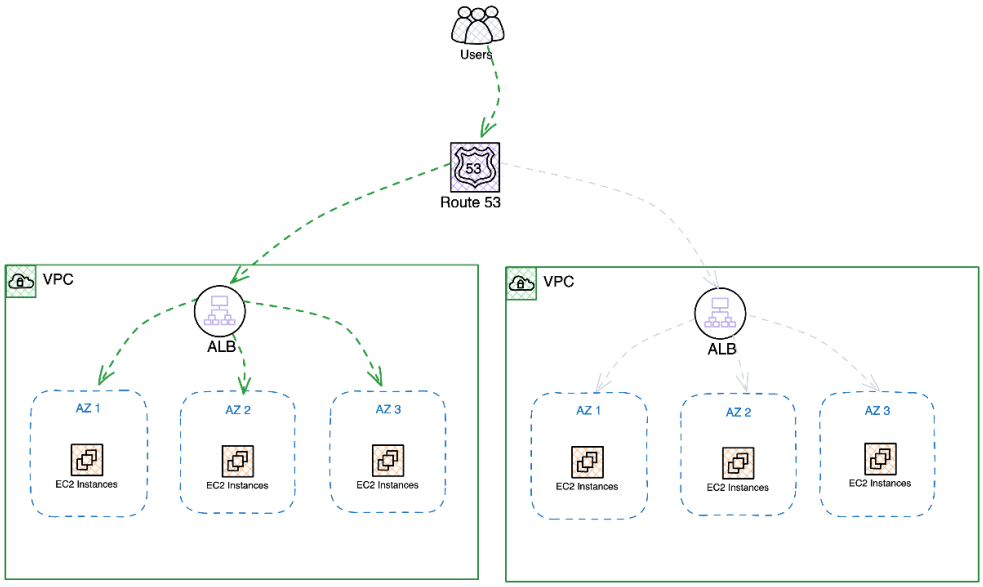

Figure 7.3 – Extending redundancy to the entire application in the same Region

While maintaining multiple copies of your entire architecture can be costly, Amazon Route 53 provides features to mitigate these costs. Depending on your requirements for automation and failover speed, you can proactively scale down or turn off the inactive environment (e.g., the *green* environment) and pre-scale the active environment before an anticipated event or failover. This approach allows you to optimize resource utilization and costs while maintaining the ability to rapidly switch between environments when needed. Additionally, Route 53 supports various routing policies and health checks, enabling you to automate the failover process based on predefined conditions or monitored health metrics. For example, you can configure Route 53 to automatically route traffic to the healthy environment if it detects failures or degraded performance in the active environment, minimizing downtime and ensuring seamless failover.

So far, we have covered leveraging AZs to achieve redundancy for AWS services such as VPCs and DNS, as well as using load balancing to distribute traffic across multiple targets in different AZs. Load balancers play a crucial role in fault tolerance by monitoring target health and automatically routing traffic away from failed instances. However, to fully leverage this redundant infrastructure, applications themselves must be designed with redundancy in mind. In the next section, we will explore patterns and considerations for building fault-tolerant and resilient software that can take advantage of the underlying redundant architecture provided by AWS services and load-balancing capabilities.

State management – stateless versus stateful approaches

Load balancing works best with **stateless applications**, where each request can be handled independently by any of the available targets. Stateless applications don't maintain client-specific data on the server side, making it easier to distribute requests across multiple targets without worrying about session affinity. On the other hand, **stateful applications** maintain client-specific data on the server side, requiring requests from the same client to be routed to the same target for consistency. While load balancers can be configured to support session affinity for stateful applications, they introduce complexity and can limit the ability to distribute load evenly across targets.

Let's put this concept into an example. Imagine a popular online video game where players compete for high scores, and their rankings are displayed on a leaderboard. In a stateful approach, each game server instance maintains the leaderboard data in its memory, and players' requests are routed to the same server instance using session affinity or sticky sessions. Each server instance can only handle a limited number of concurrent players, and if a server fails, all the leaderboard data stored in its memory is lost, causing disruption and inconsistencies for players.

If one of the game server instances crashes or becomes unresponsive, all players connected to that instance will lose their leaderboard data, and their rankings will be unavailable until the server is restored, or their sessions are redirected to another instance.

In the following figure, we can see an example of sticky sessions with two groups of users always being redirected to the same instances by the load balancers.

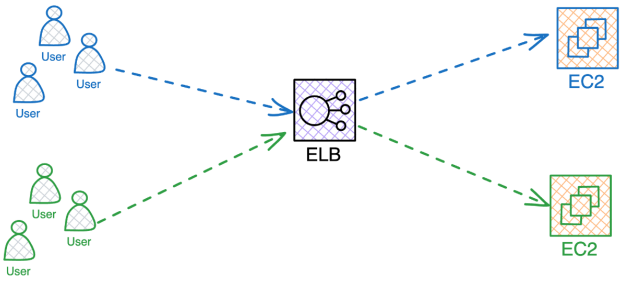

Figure 7.4 – Stateful application routing (sticky sessions)

At this point, it doesn't matter whether we have prepared the infrastructure for redundancy because the application cannot take advantage of it.

In a stateless approach, the game servers don't maintain any leaderboard data locally. Instead, they interact with a managed service such as Amazon DynamoDB to store and retrieve leaderboard data with (probably) a session ID. DynamoDB is a fully managed, scalable, and highly available NoSQL database service that can handle massive amounts of data and traffic. With this approach, each player's request can be handled by any available game server instance, as there is no need for session affinity. The game servers simply read and write leaderboard data to and from DynamoDB, which replicates the data across multiple AZs for redundancy and fault tolerance. If a game server instance fails, players connected to that instance can seamlessly reconnect to another available instance without losing their leaderboard data or rankings. The new instance can retrieve the leaderboard data from DynamoDB and continue serving the players without any disruption.

In comparison with *Figure 7.4*, the next figure shows how using an external session store lets us make use of all instances for all users, per transaction.

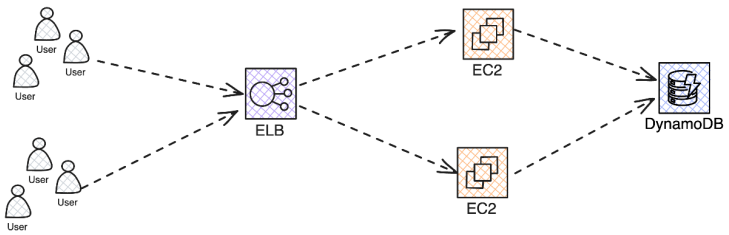

Figure 7.5 – State management deferred to DynamoDB

By adopting a stateless architecture with DynamoDB in this example, the video game can scale more easily to accommodate a growing player base. Game server instances can be added or removed dynamically without worrying about data loss or session affinity. Additionally, the leaderboard data is

highly available and durable, ensuring that players' rankings are always accessible, even in the event of server failures or AZ outages. AWS offers multiple database and cache solutions for managing states, depending on the use case. DynamoDB is often a straightforward choice with fewer infrastructure components to manage. However, for more complex state management requirements, solutions such as Amazon ElastiCache or MemoryDB (both for in-memory caching) may be more suitable. To dive deeper into state management, check out the *Performance at Scale with Amazon ElastiCache* AWS whitepaper (`https://docs.aws.amazon.com/whitepapers/latest/scale-performance-elasticache/scale-performance-elasticache.html`). We will delve deeper into fault tolerance considerations for data storage and management in the next section.

Handling data redundancy

In addition to redundancy at the compute layer (e.g., load balancing across multiple EC2 instances), it's crucial to address redundancy at the data layer to ensure complete fault tolerance for your applications. Most applications rely on some form of data storage, such as object stores or databases (NoSQL, SQL, etc.), to persist user data or system data required for proper functioning. Running your database on a single EC2 instance introduces a single point of failure, which can compromise the availability and durability of your application, as you can see in *Figure 7.6*.

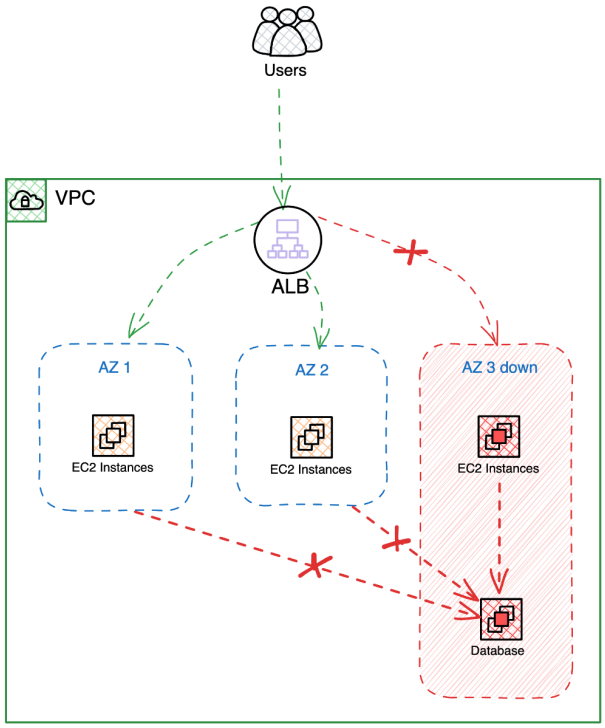

Figure 7.6 – A single point of failure with a database

While you can take regular snapshots of the instance to prevent data loss in case of an AZ disruption, this approach alone does not provide fault tolerance. In the event of a failure, you would need to restore the snapshot, which can lead to prolonged downtime and potential data loss or inconsistencies, negatively impacting your users and the overall system reliability.

As data can be handled with different forms of storage, let's see in the next sections how to apply redundancy for files and databases. Then, we'll discuss backup to maintain fault tolerance.

Applying redundancy for file storage

Many applications and workloads still rely on traditional file storage, such as content management systems, media processing pipelines, or legacy applications that require file-based operations. Additionally, scenarios involving data migrations or integrations with on-premises systems often necessitate the use of file storage solutions.

Amazon **Simple Storage Service** (**S3**) is a cost-effective, popular, scalable, and highly resilient service to handle object storage (files of any nature). It provides various redundancy options, including cross-AZ replication and cross-Region replication, ensuring that your data is durably stored and accessible even in the event of an AZ or Regional disruption. With S3's built-in redundancy features, you can easily configure your data to be replicated across multiple AZs within a Region (for high availability) or across different AWS Regions (for disaster recovery and compliance purposes).

However, S3 may not always be the best fit for use cases that demand low-latency access to data stored on disk. Object storage typically interfaces through APIs or SDKs, which can introduce latency and complexity for workloads that require direct file system access.

To address this challenge, AWS provides file storage solutions such as **Amazon Elastic File System** (**Amazon EFS**) or **Amazon FSx**, which bridge the gap between the need for low-latency, disk-based access and the durability and redundancy requirements of modern applications. Amazon EFS is a scalable file system that can be mounted concurrently by multiple compute resources across AZs, ensuring high availability and data durability.

To go further, if data needs to be stored on S3 for archival or compliance purposes, services such as AWS DataSync can be leveraged to copy data from various sources (such as EFS or on-premises systems) to the highly durable and cost-effective S3 storage. Transitioning data between storage layers as needed is one great strength and flexibility of AWS, and the cloud in general. This allows you to choose the right storage solution based on your application's requirements, performance needs, and cost considerations, while still ensuring data redundancy and fault tolerance.

Leveraging managed database services

If your use case requires it, you can run your own database service across multiple AZs and replicate the data. However, this approach comes with challenges, such as managing the replication process, ensuring data consistency, handling failover scenarios, and maintaining security patches for the OS

and database engine. Implementing and maintaining a highly available and fault-tolerant database deployment can be complex and resource-intensive. Yes, you might have guessed it; managed services can help you achieve fault tolerance with less hassle. AWS offers more than 15 services for data storage, including managed database services and object storage solutions.

While we won't dive into all AWS' managed database services here, let's address how you can leverage redundancy for fault tolerance with some of the most popular offerings. One of the most straightforward and cost-effective ways to run MySQL or PostgreSQL on AWS is through Amazon RDS. By default, RDS doesn't come with AZ replication enabled, but you can configure Multi-AZ deployments. When you enable a Multi-AZ deployment, RDS will automatically set up another copy of your database instance synchronously. This instance, called a replica, is always in a different location than the primary instance, for high availability. In the event of an AZ disruption, RDS will automatically fail over to the standby replica, minimizing downtime and data loss. Additionally, RDS allows you to create many read replicas across AZs, which can help offload read traffic from the primary instance and improve read performance, especially if the read replica is in the same AZ as the application. Many SQL libraries allow you to configure separate read and write endpoints, so read queries (`SELECT`, `SHOW`, `EXPLAIN`, and so on) are sent to the read replica endpoint, while write operations (`INSERT`, `UPDATE`, `DELETE`, and so on) are sent to the primary write endpoint. For example, the default Java connector for MySQL, **MySQLConnector/J** (`https://dev.mysql.com/doc/connector-j/en/connector-j-source-replica-replication-connection.html`), **ActiveRecords** with Ruby on Rails (`https://guides.rubyonrails.org/v6.0.2.1/active_record_multiple_databases.html`), or **dbresolver** in Go (`https://github.com/bxcodec/dbresolver`) support various high availability and load balancing configurations, including failover and read/write splitting. You can also build a higher-level abstraction class for your database connection pool manager to have a custom behavior.

Another popular managed database service is Amazon Aurora, a MySQL and PostgreSQL-compatible relational database built for the cloud. Unlike standard RDS, Aurora is designed from the ground up for high availability and fault tolerance. Aurora uses a distributed, multi-master architecture that replicates data across multiple AZs within a Region. This means that there is no single point of failure, as Aurora automatically provisions and maintains multiple replicas of your data across different AZs. In the event of an AZ disruption, Aurora automatically fails over to one of the remaining replicas, ensuring continuous availability and data durability. Additionally, Aurora allows you to create multiple read replicas across AZs, enabling read scaling and further improving fault tolerance by distributing read traffic across multiple replicas.

For NoSQL workloads, Amazon DynamoDB, a fully managed NoSQL database service, abstracts away the complexities of data redundancy and fault tolerance. DynamoDB automatically replicates data across multiple AZs within a Region, ensuring that your data remains available even in the event of an entire AZ failure. DynamoDB also offers global tables, which allow you to replicate your data across AWS Regions, providing an additional layer of redundancy and disaster recovery capabilities.

Furthermore, DynamoDB supports **Point-in-Time Recovery** (**PITR**), which automatically creates incremental backups of your DynamoDB table's data and metadata. With PITR, you can restore your

table to any point in time within the last 35 days, protecting you from accidental writes or deletes and providing an additional safeguard against data loss. To make the most of DynamoDB's resilience, it's crucial to follow best practices for partitioning and distributing your data across partitions (known as **shards**). Improper partitioning strategies can lead to hot shards, where a disproportionate amount of traffic is directed to a small number of shards, resulting in throttling and performance degradation, as seen in *Figure 7.7*.

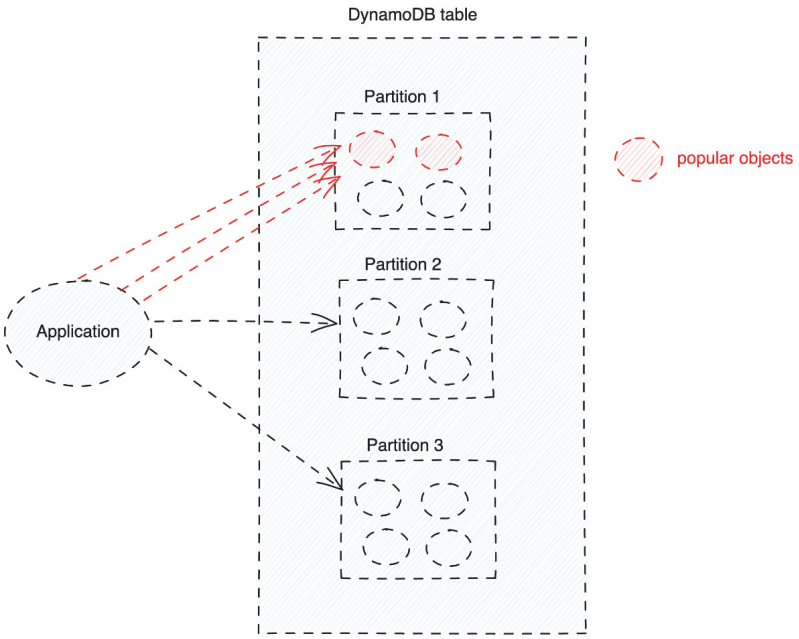

Figure 7.7 – DynamoDB hot shards

By carefully designing your partition keys and distributing your data evenly across shards, you can avoid these performance bottlenecks and ensure that your application can fully leverage DynamoDB's fault tolerance capabilities. Here are a few ways you can avoid hot shards:

- **Use composite partition keys**: Instead of using a single attribute as the partition key, consider using a combination of attributes to distribute the data more evenly across partitions. For example, if you have a table for storing user data, instead of using the `userId` as the partition key, you could use a composite key such as `userType#userId`. This way, the data for different user types (e.g., admin, regular, and premium) will be distributed across different partitions, reducing the likelihood of hot partitions.
- **Split overloaded partitions**: In some cases, you may need to store data that naturally tends to cluster around certain partition key values. In such scenarios, you can employ a technique called partition overloading, where you intentionally introduce additional attributes into the partition

key to distribute the data more evenly. For example, if you have a table for storing product data, and you expect a few popular products to have significantly more data than others, you could use a composite partition key such as `productCategory#productId#randomSuffix`. This way, even the data for popular products will be distributed across multiple partitions.

- **Introduce randomness or hashing**: Another technique is to introduce randomness or hashing into your partition key design. For instance, if you have a table for storing sensor data, instead of using the `sensorId` key as the partition key, you could use a hashed value of the `sensorId` key combined with a random suffix, such as `hash(sensorId)#randomSuffix`. This approach helps distribute the data more evenly across partitions, even if the sensor IDs are not uniformly distributed.

It's important to note that the optimal partition key design depends on your application's specific data access patterns and requirements. You can use CloudWatch Contributor Insights to gain visibility into the access patterns of your DynamoDB tables, helping you identify potential hot keys or skewed access patterns that may impact performance and resilience. Designing your DynamoDB tables correctly, with a deep understanding of your application's query patterns, is essential to achieving both high performance and fault tolerance.

AWS offers a diverse range of managed database services on top of what we discussed to support various data models, including relational (RDS and Aurora), key-value (DynamoDB), document (DynamoDB and **Amazon DocumentDB**), in-memory (**Amazon ElastiCache** and **Amazon MemoryDB**), graph (**Amazon Neptune**), time series (**Amazon Timestream** and **Amazon Managed Service for Prometheus**), wide column (DynamoDB, **Amazon Keyspaces for Apache Cassandra**), and data warehouse (**Amazon Redshift** and **Amazon Athena**). These services are built with scale and reliability in mind, leveraging AWS's global infrastructure to provide fault tolerance and high availability out of the box.

Backing up data regularly

As established in *Chapter 3*, backing up data is an essential part of being fault tolerant. Many of the managed data services such as Amazon RDS, Aurora, and DynamoDB provide built-in backup and restoration capabilities. Enabling automated backups and configuring appropriate retention periods can be done with a simple API call or through the AWS Management Console. These services also provide PITR capabilities, allowing you to restore your data to a specific point in time. Amazon S3 can help you replicate objects in another Region, providing an additional layer of protection against data loss or corruption.

For self-managed data stores, such as those running on Amazon EC2 instances or using Amazon **Elastic Block Store** (**EBS**) volumes, you can implement a backup strategy using AWS Backup. AWS Backup is a fully managed service that automates and centrally manages backups across various AWS services and on-premises resources. With AWS Backup, you can define backup plans that specify the backup frequency, retention periods, and storage locations (e.g., Amazon S3 and Amazon Glacier)

for your data. Additionally, AWS Backup supports cross-account and cross-Region backups, enabling you to store backups in separate accounts or Regions for added security and compliance.

Regardless of the specific services or tools you use, it's essential to regularly test your backup and recovery processes to ensure they work as expected and meet your recovery objectives.

In the next section, we'll explore the concept of **loose coupling** and its role in building resilient architectures on AWS. We will explore how **microservices** help with loose coupling and best practices to ensure communication between them.

Implementing loose coupling for isolating faults

While redundancy plays a vital role in achieving fault tolerance, it is not a panacea on its own. Even with redundant components in place, tightly coupled systems can propagate failures across the entire architecture, leading to widespread outages and compromising the overall resilience of the system. This is where the principle of loose coupling becomes crucial.

Loose coupling is a design approach that emphasizes decoupling components and services within a system, reducing their interdependencies, and minimizing the impact of failures. By embracing loose coupling, you can create more modular and independent systems, where failures in one component do not cascade through the entire architecture. This isolation of faults not only enhances fault tolerance but also simplifies the maintenance, scalability, and future extensibility of your applications. In a tightly coupled system, components are heavily dependent on each other, often communicating synchronously and sharing state or resources. Any failure or disruption in one component can quickly ripple through the system, causing cascading failures and potentially bringing down the entire application. Conversely, a loosely coupled system consists of independent components that interact through well-defined interfaces and asynchronous communication patterns, minimizing direct dependencies and shared state.

Imagine a complex e-commerce application with tightly coupled components for product catalogs, shopping carts, order processing, and payment gateways. If the order processing component fails, it could potentially block the entire checkout flow, preventing customers from completing their purchases and impacting the entire application, regardless of how redundant it is. However, in a loosely coupled architecture, these components would be decoupled, communicating asynchronously through message queues or event streams. If the order processing component experiences an issue, the other components can continue functioning, allowing customers to browse products, add items to their carts, and even initiate the checkout process. The orders would be queued or buffered until the order processing component recovers, minimizing the impact on the overall user experience and system availability.

In the next sections, we'll review software architecture designs for improving fault tolerance and strategies for decoupling components. AWS maintains prescriptive guidance on architecture patterns for AWS (`https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/introduction.html`). As this guidance is constantly maintained and kept up to date, we will cover highly popular architecture patterns such as microservices, **Event-Driven Architectures** (**EDAs**), and asynchronous communication patterns.

Using microservices for decoupling services

Microservices architectures have emerged as a response to the challenges faced by monolithic applications, which can become increasingly complex, difficult to maintain, and resistant to change as they grow. Microservices aim to solve these problems by breaking down a monolithic application into a collection of small, independently deployable services, each focused on a specific business capability or domain. This approach promotes modularity, scalability, and agility, as individual services can be developed, deployed, and scaled independently, without affecting the entire application.

A key principle of microservices architectures is domain-driven design, which emphasizes the importance of aligning software design with the underlying business domains. Each microservice should be bounded by a specific business domain and have complete ownership over its data store. This approach promotes loose coupling and encapsulation, as changes within a domain are isolated and less likely to impact other domains. By owning their data stores, microservices can maintain data consistency and integrity within their respective domains, reducing the risk of data corruption or inconsistencies. This ownership model also enhances resilience, as issues within a domain can be contained and designed to affect the least number of other domains.

While microservices architectures offer numerous benefits, their deployment and operation can be complex. One popular deployment model for microservices is **containerization**, which involves packaging each service and its dependencies into a lightweight, portable container. Platforms such as Amazon **Elastic Kubernetes Service** (**EKS**) and **Elastic Container Service** (**ECS**) provide managed container orchestration and scaling capabilities, simplifying the deployment and operation of microservices. However, it's important to note that containers are merely a deployment model and do not inherently equate to a microservices architecture. Microservices are an architectural style that emphasizes loose coupling, domain-driven design, and independent deployments, while containers are a technology that can facilitate the deployment and operation of microservices.

Let's take the example of a simple e-commerce website. We can divide the website into smaller functions that are easier to operate, in isolation. Let's consider the following domains:

- **Product catalog service**: Responsible for managing product information, such as descriptions, images, pricing, and inventory levels; owns the product catalog data store
- **Shopping cart service**: Handles the management of user shopping carts, including adding, removing, and updating items; owns the shopping cart data store
- **Order management service**: Responsible for processing and fulfilling customer orders; owns the order data store, which includes order details, shipping information, and payment details
- **Payment service**: Handles payment processing and integration with payment gateways; owns the payment data store, which stores transaction details and payment histories
- **Shipping service**: Manages shipping carriers, rates, and tracking information; owns the shipping data store, which includes carrier information and shipment details

- **Customer service**: Handles customer profiles, authentication, and account management; owns the customer data store, which includes customer information and preferences
- **Recommendation service**: Provides personalized product recommendations based on customer behavior and purchase history; owns the recommendation data store, which stores customer interactions and purchase data
- **Search service**: Responsible for indexing and searching product catalogs based on customer queries; owns the search index data store

By dividing the e-commerce application into these microservices, each service is bounded by a specific business domain and owns its respective data store. This approach promotes loose coupling, as changes within a domain (e.g., updating the product catalog) can be isolated and less likely to impact other domains (e.g., order management or shipping).

Additionally, each microservice can be developed, deployed, and scaled independently, in different programming languages, allowing for greater agility and flexibility in responding to changing business requirements or traffic patterns. For example, if the product catalog service experiences high traffic during a sale, it can be scaled independently without affecting other services. The search service can leverage a full-text, search-based engine such as Amazon OpenSearch, while the product catalog service stores its data on Amazon DynamoDB. The recommendation service could train a machine learning model with Amazon Personalize based on database extracts in S3. See in the following figure how domains and services can separate themselves by boundaries and data stores.

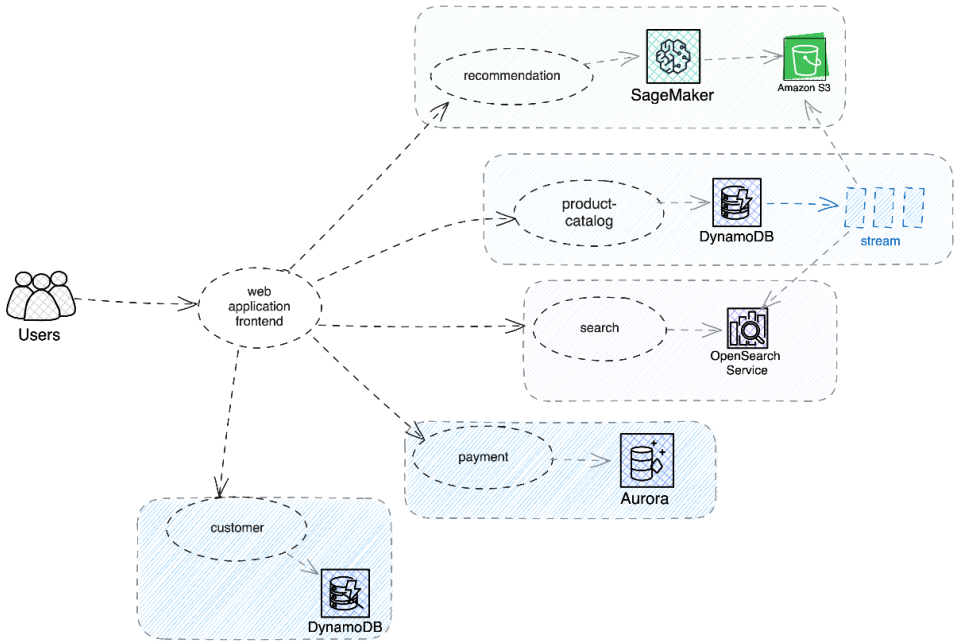

Figure 7.8 – Microservices with different data stores

Microservices can be overwhelming to operate as complexity can add up fast. It's important for organizations to provide guidance around them. For example, it must be established which columnar databases should be used for the data store, which set of programming languages should be used for microservices, and so on.

Service-to-service communication

With services decoupled into domains and microservices, one question arises: *how do they communicate with one another?* One popular architectural pattern is to communicate by API. Each service provides a well-defined set of operations with a clear contract, and ideally, no service should directly access the data store of another domain or service. This is important to allow greater flexibility for each service.

There are several protocols that can be used for **service-to-service communication** over APIs, such as the following:

- **Remote Procedure Call** (**RPC**) protocols such as **Google Remote Procedure Call** (**gRPC**) provide a high-performance, efficient, and language-agnostic way for services to communicate. gRPC uses **protocol buffers** for data serialization and supports features such as streaming, flow control, and load balancing.
- **Representational State Transfer** (**REST**) is a widely adopted architectural style that uses HTTP as the underlying protocol. Services expose resources through URLs, and clients interact with these resources using HTTP methods such as `GET`, `POST`, `PUT`, and `DELETE`.
- Communicate asynchronously by publishing and consuming messages from messaging queues such as Amazon **Simple Queue Service** (**SQS**) or Apache Kafka. This decouples the services and provides a buffer for handling spikes in traffic or temporary service outages.

Regardless of the protocol chosen for service-to-service communication, various issues can arise, such as network packet loss, service unavailability, or timeouts. To be fault-tolerant, services should implement several patterns and practices.

Let's see next how to protect service communication with **limits** and **timeouts**.

Limits and timeouts

Limits and timeouts are crucial practices for ensuring the resilience and fault tolerance of microservices architectures. Before deploying to production, it's essential to perform load testing. This not only provides confidence in how the system performs under load but also gives operators valuable information about the limits of the systems. Understanding these limits allows services to communicate them to clients, enabling them to architect their applications accordingly. Setting up limits allows a service to gracefully reject queries if they are likely to compromise the service's health, preventing cascading failures and resource exhaustion.

When a service communicates with its dependencies, it's crucial to set appropriate timeouts. Nothing can be left to chance, as a thread that hangs and blocks while waiting for a response can cause a cascading series of events, generating more threads and potentially taking down the entire service. By implementing timeouts, services can limit the amount of time they wait for a response from a dependency, preventing resource starvation and ensuring that requests are processed within a reasonable timeframe.

Timeouts should be carefully configured based on the expected response times of dependencies and the overall performance requirements of the system. Too short of a timeout can lead to premature failures and unnecessary retries, while too long of a timeout can result in resource exhaustion and unacceptable latencies. It's essential to strike a balance and continuously monitor and adjust timeouts based on real-world performance data.

Retries and backoff

In addition to timeouts, services should implement **retry** mechanisms with **exponential backoff** to handle transient failures. This allows services to automatically retry failed requests after a brief delay, increasing the chances of success without overwhelming the failing dependency with excessive retries. Exponential backoff helps to distribute retries over time, reducing the risk of cascading failures and providing a more graceful degradation of service.

Isolate services into pools, so if one pool fails, the others can continue operating, like a ship's bulkheads. Provide fallback responses or default values when a service is unavailable, allowing the system to continue functioning with degraded functionality. Distribute requests across multiple instances of a service to improve availability and prevent overloading any single instance. Implement robust monitoring and alerting systems to quickly detect and respond to service failures or performance issues.

The following is an implementation of exponential backoff in Python. It is also available in the `Chapter07` folder of the GitHub repository of the book as `Chapter7/1-exponential-backoff`.

```python
import time
import random

MAX_RETRIES = 5  # Maximum number of retries
INITIAL_DELAY = 1  # Initial delay in seconds
MAX_DELAY = 60  # Maximum delay in seconds

def exponential_backoff(retries, delay):
    """
    Exponential backoff function to calculate the delay for the next retry.

    """

    delay = min(delay * 2, MAX_DELAY)
    jitter = random.uniform(0, delay / 2)
    return delay + jitter

def make_request(retry_count=0):
    """
    Function to make a request and handle retries with exponential backoff.

    """

    try:
        # Make the request here
        # If successful, return the response
        print("Request successful!")
        return "Success"
    except Exception as e:
        # Request failed
        if retry_count < MAX_RETRIES:
            delay = exponential_backoff(retry_count, INITIAL_DELAY)
            print(f"Request failed. Retrying in {delay} seconds...")
            time.sleep(delay)
            return make_request(retry_count + 1)
        else:
            print("Maximum retries exceeded. Request failed.")
            raise e

# Example usage
response = make_request()
print(response)
```

Figure 7.9 – Exponential backoff Python template

Many libraries, including the AWS SDK, propose native techniques to implement exponential backoff when calling AWS services to allow retry without overwhelming your service in case of an issue or getting throttled by AWS (`https://aws.github.io/aws-sdk-go-v2/docs/configuring-sdk/retries-timeouts/`).

Circuit breaker

Consider the example of the order management service in our e-commerce application. This service relies on the payment service to process customer payments during the order fulfillment process. If the payment service becomes overloaded or experiences an outage, the order management service could become overwhelmed with failed requests, leading to resource exhaustion and potentially impacting

other services. To mitigate this risk, the order management service can implement a **circuit breaker** for the payment service. The circuit breaker acts as a proxy between the order management service and the payment service, monitoring the success rate of requests to the payment service.

Initially, the circuit breaker is in the *Closed* state, allowing requests to flow through to the payment service. However, if the failure rate exceeds a predefined threshold (for example, if 50% of requests fail within a specific time window), the circuit breaker transitions to the *Open* state. In this state, the circuit breaker immediately rejects all incoming requests to the payment service, preventing further requests from being sent and potentially overwhelming the failing service.

Instead of failing outright, the order management service can provide a fallback response or alternative behavior. For example, it could temporarily store the order details and payment information, allowing customers to complete the order process and retry the payment later when the payment service has recovered. After a predefined period (e.g., 60 seconds), the circuit breaker transitions to a *Half-Open* state, where it allows a limited number of requests to pass through to the payment service. If these requests are successful, the circuit breaker transitions back to the *Closed* state, allowing normal operation to resume. If the requests fail, the circuit breaker returns to the *Open* state, and the cycle repeats.

We can see these three states displayed in the following figure:

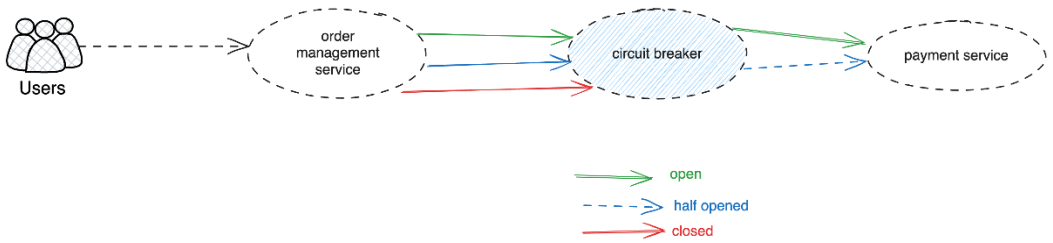

Figure 7.10 – Circuit breaker example

By implementing the circuit breaker pattern, the order management service can isolate failures in the payment service, preventing cascading failures and providing a graceful fallback mechanism. This improves the overall resilience and fault tolerance of the e-commerce application, ensuring that customers can complete their orders even in the face of partial service outages or failures.

Circuit breakers are a powerful way to stop propagating failures across microservices. To implement circuit breakers in practice, read the AWS prescriptive guidance on cloud design patterns, architectures, and implementations (`https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/circuit-breaker.html`). Let's see next another technique where communication between services happens through events.

Event-driven architecture (EDA)

In addition to communicating via APIs, microservices can also communicate through an EDA. EDA is an architectural pattern that promotes the production, detection, consumption, and reaction to events. Instead of direct method calls or message passing, components in an event-driven system communicate by emitting and consuming events, which represent significant changes in state or occurrences within the system.

Consider the example of the payment service in our e-commerce application. When a customer pays for an article, the payment service could emit an `OrderPaid` event, which would be picked up by other interested services, such as the `product-catalog` and `shipping` services. The `product-catalog` service could consume the `OrderPaid` event and update the product inventory levels accordingly, while the `shipping` service could initiate the shipping process based on the order details, as shown in the following figure:

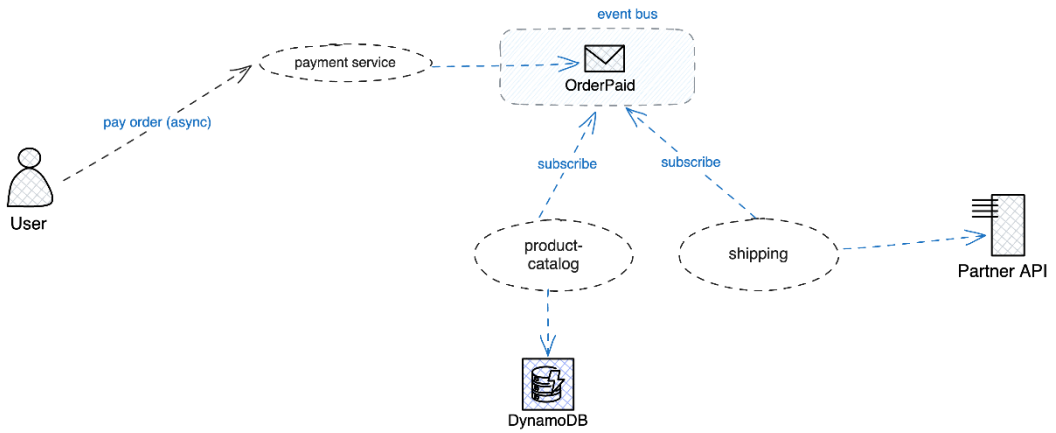

Figure 7.11 – Microservices with EDA

In an EDA, there are typically three main components: producers, routers (or event buses), and consumers. Producers are the services that emit events, while consumers are the services that listen for and react to those events. The router, or event bus, acts as a central hub that routes events from producers to the appropriate consumers.

AWS provides several services that can help implement an EDA, such as Amazon EventBridge, Amazon SQS, and Amazon Kinesis. EventBridge is a serverless event bus that can ingest events from various sources (e.g., AWS services, custom applications, or third-party providers) and route them to target consumers based on rules and patterns. SQS and Kinesis can also be used as event buses, with SQS providing a simple message queuing service and Kinesis offering real-time data streaming capabilities.

EDA enables developers to achieve loosely coupled inter-service communication, promoting scaling and greater flexibility. Asynchronous communication empowers services to operate independently and allows easy retries and replays.

> **Important note**
>
> However, this architecture pattern can be overwhelming for small teams and easy use cases. Event delivery can also become a bottleneck, but as we have stated multiple times in this book, you need to favor managed services to do the heavy lifting. To operate EDA correctly, you will need strong observability to find root causes when an issue happens.

In many cases, it can be beneficial to combine both API-based communication and EDA within a microservices architecture. APIs can be used for synchronous request-response interactions, while EDA can handle asynchronous, event-driven communication. For example, in our e-commerce application, the payment service could expose APIs for placing orders synchronously, while also emitting `OrderPaid` events asynchronously for other services to consume and react to.

By combining these two communication patterns, microservices can leverage the benefits of both approaches, enabling efficient request-response interactions while also supporting real-time event-driven workflows and decoupled, scalable architectures.

Now let's explore another popular guidance for designing and operating microservices: the Twelve-Factor App.

The Twelve-Factor App methodology

The Twelve-Factor App (`https://12factor.net/`) is a set of principles and best practices for building modern, scalable, and maintainable software-as-a-service applications. Developed by the team at Heroku, this methodology serves as a blueprint for designing applications that can run reliably on modern cloud platforms. The twelve factors cover various aspects of application design, such as code base management, dependency management, configuration, backing services, and build, release, and run processes, as well as concurrency, disposability, dev/prod parity, logs, and administrative processes.

By adhering to these principles, developers can create applications that are highly portable, scalable, and easy to maintain and deploy. The Twelve-Factor App methodology emphasizes practices such as separating config from code, treating backing services as attached resources, maximizing robustness with fast startup and graceful shutdown, keeping development and production as similar as possible, and enabling administrative tasks as one-off processes. It would be remiss to discuss software architecture best practices without acknowledging the Twelve-Factor App as a comprehensive and widely adopted guide for building well-designed, cloud-native applications.

Summary

In this chapter, we defined fault tolerance as a system's ability to continue operating correctly despite component failures, differentiating it from high availability. We emphasized the importance of building resilient, fault-tolerant applications in today's cloud environment. In the first section, we covered implementing redundancy by leveraging AWS infrastructure such as AZs and load balancing. We discussed the implications of stateless versus stateful application designs for fault tolerance. We saw how to use data redundancy strategies such as cross-AZ replication and backups, along with managed database services such as RDS, Aurora, and DynamoDB that provide built-in redundancy.

Furthermore, we learned that redundancy is not free, and that it often comes with increased complexity and costs. While it is essential for fault tolerance, it should be implemented judiciously, prioritizing the most critical components and aligning with business requirements and customer expectations. Not all systems or components may require the same level of redundancy, and organizations should carefully evaluate the trade-offs between cost, complexity, and the desired level of resilience.

We also focused on the principle of loose coupling to prevent cascading failures from impacting the entire system. We explored the microservices architecture pattern and domain-driven design as ways to decouple components. We touched on inter-service communication patterns such as APIs and EDAs. We also discussed resilience patterns such as circuit breakers, timeouts, retries, and fallbacks to handle failures gracefully.

It's important to make the right software architecture choice to make the most of the AWS Cloud offerings. Every application is unique and has unique challenges but there are a few essential design principles to keep in mind, if possible, at the design phase, before any implementation, especially if resilience is a key property of the system being built. The key takeaway from this chapter is embracing faults in systems and preparing for them. In the next chapter, we will dive into specific considerations for resiliency with serverless applications.

8

Resiliency Considerations for Serverless Applications

In the previous chapter, we explored architectural patterns and best practices for building fault-tolerant and highly available applications on AWS. We discussed the importance of embracing failures and designing systems that can withstand and recover from component failures. While these principles apply to all types of applications, serverless computing introduces unique considerations and challenges when it comes to building resilient systems.

Serverless computing, with its event-driven, pay-per-use model, offers a compelling value proposition for building scalable and cost-effective applications. Serverless requires a different mindset and approach to achieving fault tolerance and resilience. In this chapter, we will delve into the considerations and strategies for building resilient serverless applications on AWS. We will explore techniques for fault tolerance, idempotent and asynchronous function design, and leveraging managed services for resiliency. By understanding these concepts and best practices, you can design and implement serverless applications that are highly available, fault-tolerant, and able to withstand failures gracefully.

Building resilient serverless applications is not just about implementing individual patterns or techniques; it's about embracing a holistic approach that considers the unique characteristics of serverless computing. We will explore various factors that contribute to the overall resilience of your serverless applications through the following topics:

- Defining serverless applications
- Building resilience into serverless
- Monitoring and observability
- Testing serverless applications

Defining serverless applications

First things first, what is considered a **serverless application**? In the context of AWS, serverless is a modern approach to building and running applications that leverage AWS services designed to abstract away the underlying infrastructure management tasks. These applications are event-driven, scalable, and follow a pay-per-use pricing model, allowing developers to focus on writing code without worrying about provisioning, scaling, or maintaining servers or clusters.

The core components of serverless applications on AWS typically include AWS Lambda, Amazon API Gateway, AWS Step Functions, and Amazon EventBridge. AWS Lambda is a serverless compute service that allows you to run your code without provisioning or managing servers. **API Gateway** acts as the entry point for serverless applications, routing incoming API requests to the appropriate Lambda functions or other backend services. **Step Functions** is a serverless function orchestrator that enables you to coordinate multiple AWS services into serverless workflows. EventBridge is a serverless event bus that routes events from various AWS services and your own applications, enabling event-driven architectures and decoupled communication between components.

In addition to these core services, serverless applications often leverage other managed services such as Amazon Simple Storage Service (Amazon S3) for object storage, Amazon DynamoDB for NoSQL databases, Amazon Aurora Serverless for relational databases, AWS Simple Notification Service (Amazon SNS) for pub/sub messaging, and Amazon Simple Queue Service (Amazon SQS) for message queuing. These managed services provide built-in scalability, availability, and fault tolerance without the need to manage the underlying infrastructure.

The key characteristics of serverless applications on AWS include no server management, event-driven execution, automatic scaling, pay-per-use pricing, leveraging managed services, and a stateless and ephemeral design for serverless functions. By embracing serverless services on AWS, developers can build and deploy applications quickly without worrying about infrastructure provisioning or scaling concerns. This allows them to focus on writing code and delivering business value, while AWS handles the underlying infrastructure management tasks.

We cover observability in *Chapter 12*, and it's a key characteristic of any resilient system. However, it's even more important for serverless applications. Effective monitoring and observability are crucial for detecting and responding to failures, especially when you don't manage the underlying infrastructure yourself. You need to put enough visibility into your application code and use AWS-provided observability signals (metrics, logs, traces) for these serverless components. In this chapter, we will cover key signals you should be looking at when building and running serverless applications.

Now that we have covered the fundamentals, let's see next how to build resilience into serverless.

Building resilience into serverless

Serverless architectures are inherently more resilient and fault-tolerant than traditional monolithic applications. The serverless model eliminates the need for provisioning and managing servers, automatically scaling resources based on demand, and providing built-in redundancy and failover mechanisms. However, serverless applications still need to be designed and implemented with fault tolerance in mind to ensure reliable and consistent operation.

Building resilient serverless applications involves adopting specific design patterns and best practices to handle failures, errors, and unexpected scenarios gracefully. In this section, we will explore the key considerations for enhancing the resiliency of serverless applications, focusing on idempotent and asynchronous function design. Idempotent functions are designed to produce the same result regardless of how many times they are executed. We will cover error handling, retries, and managing throttling and quotas, which are key factors in managed services in general.

Idempotent and asynchronous function design

AWS Lambda is an event-driven, serverless AWS service and one of the main components of serverless architectures. With Lambda, the minimal unit of deployment is called a function, and it should be succinct, unitary, and stateless. We will often use the word *Function* to denote Lambda Functions. When using AWS Lambda, being idempotent means that your function should produce the same result, regardless of whether it's invoked once or multiple times with the same input data. Techniques such as deduplication, locks, and unique identifiers can help ensure idempotency and prevent unintended side effects.

Let's consider a scenario where your Lambda function processes orders in an e-commerce application from a queue. Each order has a unique order ID, and the function needs to update the order status in a database (e.g., Amazon DynamoDB) when the order is processed.

When the Lambda function receives an order event, it first checks if the order ID already exists in the database. If the order ID exists, it means the order has already been processed, and the function can skip further processing to avoid duplicates. If the order ID doesn't exist, the function proceeds with processing the order.

Before processing an order, the Lambda function can acquire a lock associated with the order ID. If the lock is already held by another instance of the function (indicating concurrent processing of the same order), the function can release the lock and skip processing to prevent conflicts. If the lock is acquired successfully, the function can proceed with processing the order and release the lock after completing the operation.

When updating the order status in the database, the Lambda function can use a unique identifier (e.g., a combination of the order ID and a timestamp) to ensure idempotent updates. If the unique identifier already exists in the database, it means the order status has already been updated, and the function can skip the update operation. If the unique identifier doesn't exist, the function can proceed

with updating the order status using the unique identifier. The following Python Lambda Function implements this behavior:

```
import boto3
import time
from botocore.exceptions import ClientError


dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Orders')

def lambda_handler(event, context):
    order_id = event['orderId']

    # Check if order already exists (deduplication)
    try:
        response = table.get_item(Key={'orderId': order_id})
    except ClientError:
        return {'statusCode': 500, 'body': 'Error retrieving order'}

    if 'Item' in response:
        return {'statusCode': 200, 'body': 'Order already processed'}

    # Acquire lock
    lock_key = f'lock_{order_id}'
    try:
        table.put_item(Item={'key': lock_key}, ConditionExpression='attribute_not_exists(key)')
    except ClientError as e:
        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            return {'statusCode': 200, 'body': 'Order already being processed'}
        raise e

    try:
        # Process order
        unique_id = f'{order_id}_{int(time.time())}'
        table.put_item(Item={'orderId': order_id, 'status': 'processed', 'uniqueId': unique_id})
    finally:
        # Release lock
        table.delete_item(Key={'key': lock_key})

    return {'statusCode': 200, 'body': 'Order processed successfully'}
```

Figure 8.1 – Idempotency with Lambda

You can find the complete implementation for this example on GitHub in listing `Chapter8/1-idempotency-lambda` (`https://github.com/PacktPublishing/Building-Resilient-Architectures-on-AWS/tree/main/Chapter8/1-idempotency-lambda`)

Serverless functions, such as Lambda, are designed to be stateless and ephemeral. This means that each function invocation is independent and isolated, with no persistent state or data stored within the function instance. The function's execution environment is created on demand, processes the event or request, and is then terminated. This stateless and ephemeral nature is a fundamental characteristic of serverless architectures, enabling automatic scaling, high availability, and cost-efficiency.

The stateless nature of serverless functions has significant implications for resilience. Since each function invocation is independent and isolated, failures or errors in one invocation do not affect other invocations or the overall application state. This inherent isolation helps prevent cascading failures and improves the overall resilience of the system. Additionally, the ephemeral nature of function instances means that any transient issues or corrupted states within a function instance are automatically resolved by terminating and recreating the instance for the next invocation. We'll see in the next subsection how to handle retries and error handling.

Retries and error handling in AWS Lambda

AWS Lambda, the core compute service for serverless applications, provides built-in retry mechanisms for failed invocations. While AWS is responsible for the Lambda service, you are responsible for the code running inside a Lambda function, retries, and error management. You need to make sure the function has the appropriate permissions when making calls to other AWS APIs, configure the Lambda timeouts so the execution is not stopped prematurely, and keep the function runtimes up to date with the latest security patches and language versions. AWS usually informs you about new runtimes and runtimes deprecation (e.g., Lambda Python stopped supporting Python 3.7 in December 2023 and, at the time of writing this book, is announcing Lambda Python 3.8's deprecation in October 2024).

When it comes to retries and error handling, Lambda's behavior varies depending on the invocation type, as listed here:

- **Synchronous invocations** (request/response): When you invoke a Lambda function synchronously (e.g., via the AWS Lambda console, AWS CLI, or AWS SDKs), Lambda does not automatically retry the function if it encounters an error or times out. In these cases, the invoker (your application or service) is responsible for implementing retry logic and error handling. This could involve manually re-invoking the function, sending the failed event to a queue for debugging, or ignoring the error based on your application's requirements. We discussed retry strategies with exponential backoff that are still applicable here in the previous chapter.
- **Asynchronous invocations** (event): For asynchronous invocations, such as those triggered by an Amazon SQS queue or an Amazon SNS topic, Lambda automatically retries the function twice if it encounters an error or times out. If the function still fails after the retries, Lambda sends the event to a **dead-letter queue** (**DLQ**), if one has been configured.
- **Stream-based invocations** (e.g., Amazon Kinesis or Amazon DynamoDB Streams): For stream-based invocations, Lambda automatically retries the function up to twice if it encounters an error or times out. If the function still fails after the retries, Lambda skips the batch and continues processing the next batch of records from the stream.

> **Important note**
>
> The retrying twice behavior might change as AWS publishes new features to AWS Lambda. Refer to the AWS documentation (`https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html`) for the latest information on retries.

It's important to note that Lambda's automatic retries are intended to handle transient failures, such as network issues or temporary service outages. If your function's code encounters a non-transient error, such as a programming error or a misconfiguration, Lambda's retries will not resolve the issue, and you'll need to update and redeploy your function code. When designing Lambda functions, you must use idempotent logic to handle duplicate invocations due to retries. To better handle retries, especially for asynchronous transactions, one best practice is to use DLQs, which we'll see in the next section.

DLQs for failed asynchronous invocations

A DLQ is typically an Amazon **Simple Queue Service** (**SQS**) queue or an Amazon **Simple Notification Service** (**SNS**) topic. When a Lambda function fails during an asynchronous invocation, AWS Lambda sends the event payload to the configured DLQ. You can then implement a separate Lambda function or process to consume messages from the DLQ, handle the failures, and potentially retry the original invocation. This approach allows you to decouple the failure-handling logic from the main application flow, providing greater flexibility and resilience.

It's important to note that DLQs are not limited to SNS or SQS; any queue dedicated to replaying failed messages can be considered a DLQ. However, when using DLQs, you should be careful to have a limit or a mechanism in place to prevent indefinite retries, which could lead to resource exhaustion or other unintended consequences. If the issue requires a code fix, for example, processing the DLQ indefinitely can have an impact on the rest of the system and your quotas (service limits).

Essentially, DLQs allow you to set aside failures for automatic retries or manual intervention, but they should be handled with care and appropriate safeguards. SQS provides a built-in feature called *redrive policy* that simplifies the configuration of DLQs. Using AWS CloudFormation, you can set up a DLQ for an SQS queue like this:

```
Resources:
  MainQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: my-main-queue
      RedrivePolicy:
        deadLetterTargetArn: !GetAtt DeadLetterQueue.Arn
        maxReceiveCount: 3

  DeadLetterQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: my-dead-letter-queue
```

You can find this CloudFormation template in the `Chapter8/2-sqs-redrive-policy` listing in the book's GitHub repository (`https://github.com/PacktPublishing/Building-Resilient-Architectures-on-AWS`).

In this example, `RedrivePolicy` property specifies that messages that fail to be processed after three attempts (`maxReceiveCount: 3`) will be moved to the `my-dead-letter-queue` SQS queue.

Important note

Discarding failures is outside of the application (Lambda) code context, and this separation of concerns is one of the strengths of serverless architectures. By offloading failure handling to dedicated components such as DLQs, you can simplify your application code and improve overall system resilience.

Embracing the inevitability of errors and implementing graceful error handling is a fundamental aspect of building resilient serverless applications. This concept is not limited to AWS Lambda but applies to any serverless compute service, including AWS Step Functions. By expecting and proactively planning for errors, developers can better control failure modes and ensure their applications respond gracefully to unexpected situations. In the following section, we will explore another potential failure mode: throttling due to service quotas.

Handling throttling and service quotas (service limits)

AWS imposes quotas and limits on the usage of its services to prevent resource exhaustion and ensure fair allocation across multiple tenants. If these quotas are not respected, AWS will interrupt or limit the usage of the service, potentially leading to service disruptions or failures.

While serverless services such as AWS Lambda and Amazon API Gateway provide automatic scaling capabilities, they still have service quotas, previously called limits. These mechanisms are in place to prevent abuse and ensure fair usage across all customers. In *Chapter 6*, we discussed how to set up quotas and boundaries for your own services.

Handling quotas with compute – Lambda example

One of the best ways to prevent throttling and service quotas is to optimize your resource utilization. AWS Lambda, for example, has quotas (`https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html`) on the number of concurrent executions (the number of inflight requests a function is handling), function memory allocation, and deployment package size. Some of those quotas are increasable, such as concurrent execution, and need to be monitored. We discussed in *Chapter 6* how unmonitored AWS quotas utilization can hurt the overall availability of your systems. When using any AWS (serverless) service, look at its quotas and plan accordingly. By setting up an alarm on quotas, we can keep an eye on how close we are to thresholds and take preventive action.

Acting on quota alarms

It's also important to note that *taking action* doesn't necessarily mean increasing quotas. Sometimes, another part of the system can blow up our quotas unintentionally. Setting up strong monitoring and observability is key to getting visibility of your system operations and eventually getting to the point of answering why we are close to the quota limits.

AWS Lambda proposes a **provisioned concurrency** option (`https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html`) to help remove **cold starts** for functions sensitive to latency, prevent being throttled for concurrency, and make sure that there is always a pool for concurrency available for critical functions. You can also leverage the **reserved concurrency** feature (`https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html`), which can make sure a function is guaranteed its limit of concurrent invocations.

Lambda cold starts

A cold start in the context of serverless computing, such as AWS Lambda, refers to the initial invocation of a function instance after a period of inactivity or when a new instance needs to be launched. When a Lambda function is invoked for the first time or after a prolonged period of inactivity, AWS needs to allocate compute resources, provision the runtime environment, download the function code and its dependencies, and initialize the function before it can start executing. This process can take some time, typically ranging from a few milliseconds to several seconds, depending on the function's code size, dependencies, and the runtime environment. For Java-based functions, AWS introduced **SnapStart**, a purpose-built solution to reduce cold starts. You can read more about it in this AWS blog (`https://aws.amazon.com/blogs/compute/reducing-java-cold-starts-on-aws-lambda-functions-with-snapstart/`).

Handling quotas with queues – SQS example

All quotas are documented by AWS, but SQS is one of the most commonly used services in serverless architectures, so it is right that we cover it in this chapter. SQS has various quotas and limits in place to ensure fair usage across all customers. SQS queues come in two flavors:

- **FIFO queues**, where message order and unicity are preserved, and throughput is sacrificed (20k inflight messages)
- **Standard queues**, where messages can be delivered more than once, in no order; this is designed for high throughput (120k infight messages)

While SQS quotas are kept up to date in the AWS documentation (`https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-quotas.html`), let's see what actions to take when facing throttling or quotas related failures with SQS (or any messaging service at this point):

- **Exponential backoff and jitter**: When your application encounters a throttling exception, you can implement an exponential backoff strategy with jitter to retry the request after a gradually increasing delay. This helps to prevent the service from being overwhelmed with retries and allows time for the throttling to subside.

- **Batching and buffering**: Instead of sending individual messages to SQS, you can batch multiple messages into a single request, up to the maximum batch size. This reduces the number of API calls and helps to stay within the send message rate quota. Additionally, you can buffer messages in memory or a local queue before sending them to SQS, allowing you to control the rate at which messages are sent.
- **Parallel processing**: If your application needs to process many messages concurrently, you can leverage multiple Lambda functions to distribute the workload and stay within the receive message rate quota.

This Python snippet demonstrates how to implement exponential backoff and jitter while trying to send a message to an SQS queue:

```
def send_message_with_backoff(message_body):
    delay = 1  # Initial delay in seconds
    max_delay = 60  # Maximum delay in seconds

    while True:
        try:
            response = sqs.send_message(
                QueueUrl=queue_url,
                MessageBody=message_body
            )
            return response
        except sqs.exceptions.TooManyEntriesInBatchException as e:
            # Handle throttling exception
            if delay > max_delay:
                raise e
            # Introduce jitter to avoid synchronous retries
            sleep_time = delay + random.random()
            time.sleep(sleep_time)
            delay *= 2  # Exponential backoff
        except Exception as e:
            # Handle other exceptions
            raise e
```

Figure 8.2 – Exponential backoff with SQS

You can find the complete implementation for this example in listing `Chapter8/3-sqs-exponential-backoff` in the book's GitHub repository.

By implementing these strategies, you can ensure that your serverless applications can gracefully handle throttling and service limits imposed by AWS services such as Amazon SQS.

Handling quotas with databases – DynamoDB example

DynamoDB is a highly performant and scalable NoSQL database service designed for serverless architectures. DynamoDB can achieve millions of requests per second while maintaining low latency

and high availability. This exceptional performance is made possible by DynamoDB's distributed architecture, which leverages techniques such as consistent hashing, log-structured merge trees, and quorum-based replication. Despite its impressive performance capabilities, DynamoDB too has service quotas in place to ensure fair usage and prevent abuse. Understanding and managing these quotas is essential for building resilient and scalable serverless applications backed by DynamoDB. DynamoDB has several quotas that can impact the resilience and performance of your applications. One of the most relevant quotas to consider is the throughput quota. Throughput quotas come in DynamoDB in two flavors:

- **Provisioned throughput**: DynamoDB enforces quotas on the maximum provisioned read and write capacity units for tables and global secondary indexes.
- **On-demand throughput**: There are quotas on the maximum read and write request units per second for on-demand mode.

Exceeding these throughput quotas can result in throttling, which can impact the availability and responsiveness of your application. To handle throughput quotas, you can implement strategies such as the following:

- **Implementing a strong design table**: Even though DynamoDB offers schema flexibility, starting with a good table design will save time in the future. DynamoDB supports two types of primary keys, a **hash key** and a **hash and range key**. A hash key consists of a single attribute that uniquely identifies an item. A hash and range key consists of two attributes that, together, uniquely identify an item. To make a good choice, you need to think about how the data will be stored and queried. Advanced techniques such as **single table design** (`https://aws.amazon.com/blogs/compute/creating-a-single-table-design-with-amazon-dynamodb/`) will help with optimal data access, reduce throughput, and maximize performance by avoiding hot shards.
- **Distributing queries over multiple tables**: Sometimes, it's easier to spread data over multiple tables for simplicity. If tables have very different access patterns, they can be provisioned differently (on-demand or provisioned).
- **Implementing exponential backoff and retries for throttled requests**: Here, we can also leverage SQS, for example, for retries to prevent long durations in Lambda executions.

On the same table, DynamoDB allows local and global indexes to be set, which provides different data lookup access patterns. There's a limit on how many indexes can be created per table, so you need to plan in the design phase.

In this section, we covered how building fault tolerance in serverless applications is crucial for ensuring reliable and consistent operation. While serverless architectures are inherently more resilient than traditional monolithic applications, they still need to be designed, implemented, and operated with fault tolerance in mind. In the next section, we'll see how observability is key to building more resilient serverless applications.

Monitoring and observability for serverless applications

Remember that *"Everything fails all the time"* (Werner Wogels, CTO, Amazon). To operate serverless applications, it's nearly impossible to bypass **monitoring** and **observability**. We have seen how AWS services protect themselves with quotas. There's a strong need to monitor them, along with the application's telemetry signals.

AWS provides observability and monitoring tools to help you monitor and troubleshoot your serverless applications. Amazon CloudWatch and AWS X-Ray (traces) are two essential services for monitoring serverless applications. These services are serverless and require no additional operational burden for customers. Imagine if you had to monitor the monitoring system itself!

CloudWatch collects and processes raw data into readable, near-real-time metrics and logs. For AWS Lambda, some important metrics to monitor are as follows:

- **Invocations**: The number of times your Lambda function is invoked
- **Throttles**: The number of invocation requests that were throttled due to hitting the concurrent execution limit
- **IteratorAge**: For event source mappings (e.g., Kinesis or DynamoDB Streams), this metric indicates how far behind the iterator is from the stream head
- **ConcurrentExecutions**: The number of function instances that are currently executing
- **Errors**: The number of invocations that failed due to errors in your function code or configuration

CloudWatch also provides pre-built dashboards and recommended alerts for these metrics, making it easier to monitor and set up alarms for potential issues. CloudWatch Lambda Insights is a purpose-built, monitoring capability for Lambda that provides a clear view across all functions and gives insights into what information should be looked at when monitoring Lambda functions.

By default, Lambda logs are collected into Amazon CloudWatch Logs and allow users to dive deeper into the root causes of events.

Like most services, DynamoDB also publishes usage metrics into CloudWatch, but it additionally offers Contributor Insights to analyze and identify the root cause of throttling and performance issues. Contributor Insights provides visibility into the operations, users, or tenants that are contributing to the load on your DynamoDB tables, enabling you to take targeted action to improve performance and resilience.

AWS X-Ray, on the other hand, provides end-to-end tracing for serverless applications, allowing you to analyze and debug distributed applications. X-Ray integrates with AWS Lambda, API Gateway, and other AWS services, providing visibility into request paths, latencies, and potential bottlenecks. Services such as SNS and SQS have active tracing enabled by default, without requiring any instrumentation on your part. *Chapter 12* will provide a lot more information on observability but it's a critical part of building resilient serverless applications.

Observability outside of AWS

It's important to note that you are not restricted from using AWS tools for monitoring and observability of your serverless applications. AWS proposes more and more native features that give back visibility and productivity to customers. But with features such as OpenTelemetry, you can easily instrument once and choose any vendor for your observability stack. The most important thing is that you have observability for your applications.

We just learned about observability for serverless applications, but observability as a whole is a very critical component for any kind of application. Let's see in the next section how testing can help build more resilient serverless applications.

Testing serverless applications

Implementing robust testing and deployment strategies is crucial for ensuring the reliability and resilience of your serverless applications. **Continuous Integration** and **Continuous Deployment** (**CI/CD**) pipelines can automate the build, testing, and deployment processes, reducing the risk of human error and ensuring consistent deployments. In *Chapter 6*, in the *Making frequent, small, reversible changes* section, we defined different strategies for CI and CD.

When it comes to serverless, testing may be a little bit harder. AWS has published a comprehensive *Prescriptive Guidance* document on *Testing Serverless Applications* (`https://docs.aws.amazon.com/prescriptive-guidance/latest/serverless-application-testing/introduction.html`) that covers various techniques, challenges, best practices, and resources related to this topic.

According to the AWS Prescriptive Guidance, testing serverless applications can be challenging due to their distributed nature, event-driven architecture, and the involvement of multiple AWS services. However, it is essential to adopt a robust testing strategy to ensure the reliability and fault tolerance of your serverless applications. Three main non-exclusive techniques come to mind when implementing tests for serverless apps.

Mock testing

Mock testing is an essential technique for testing serverless applications, especially when dealing with external dependencies or services. In a serverless environment, your functions often interact with other AWS services, third-party APIs, or databases. Mock testing allows you to isolate and test your function's logic by simulating or *mocking* these external dependencies. Mocking is particularly useful for unit testing, where you want to test individual functions or components in isolation. By mocking external dependencies, you can ensure that your tests are focused on the logic of your function, rather than being affected by the behavior of external services or resources.

The guidance provides examples of using popular mocking libraries such as `aws-lambda-mock-context` and `aws-lambda-mock-libs` for mocking the Lambda execution context and AWS SDK clients, respectively. These libraries allow you to create mock objects that simulate the behavior of AWS services, making it easier to test your Lambda functions without relying on actual service calls. Depending on the language, the AWS SDK provides built-in mock interfaces (`https://aws.github.io/aws-sdk-go-v2/docs/unit-testing/#mocking-client-operations`) for SDK operations (API calls), which helps with unit testing.

Additionally, the guidance recommends using mocking techniques for integration testing, where you need to test the interactions between multiple components or services. By mocking certain dependencies, you can isolate and test specific integration points without being affected by the behavior of other components or services.

Overall, mock testing is a crucial technique for ensuring the reliability and fault tolerance of serverless applications. By simulating external dependencies and edge cases, you can thoroughly test your application's logic and behavior, leading to more resilient and robust serverless systems.

Emulation testing

Emulation testing is a technique that's used to test serverless applications by emulating or replicating the actual execution environment of AWS services. Unlike mock testing, which simulates external dependencies, emulation testing aims to mimic the real-world behavior and conditions of the serverless environment.

Using emulation testing to validate the behavior of your serverless applications in an environment that closely resembles the production environment can help identify issues that may not be apparent during unit or integration testing, such as performance bottlenecks, resource limitations, or compatibility issues.

One way to perform emulation testing is by using local emulators or testing tools that replicate the behavior of AWS services. For example, the **AWS Serverless Application Model** (**AWS SAM**) provides a local testing environment called `sam local` that emulates AWS Lambda, API Gateway, DynamoDB, and other services. This allows developers to test their serverless applications locally before deploying them to the cloud. Emulation testing can be particularly useful for end-to-end testing, where you want to validate the entire application flow and ensure that all components and services are working together as expected. It can also be valuable for performance testing, load testing, and stress testing because you can simulate real-world conditions and workloads to identify potential bottlenecks or scalability issues.

Testing on AWS

While local emulation and mocking can be valuable for certain testing scenarios, it can be more reassuring to test serverless applications directly in the cloud environment to ensure accurate and realistic testing conditions.

Testing in the cloud involves deploying your serverless application or components to an AWS account or environment that closely resembles your production setup with no real customers. This approach allows you to test your application's behavior and interactions with real AWS services, rather than relying on emulated or simulated environments. With a service such as AWS Organizations, which is an account management service that enables you to consolidate multiple AWS accounts into an organization that you create and centrally manage, you can easily dedicate test accounts to serverless applications that simulate reality.

By testing in the cloud, you can validate your application's behavior in the actual production environment, including factors such as network latency, service quotas, and real-world conditions. This approach allows you to carry out end-to-end testing of your entire serverless application, including external dependencies and third-party integrations, ensuring that the complete application flow works as expected. You can simulate load to carry out performance and load testing under realistic conditions, allowing you to identify potential bottlenecks or scalability issues.

Summary

In this chapter, we explored the unique considerations for building resilient serverless applications on AWS. We started by defining serverless applications and their core components, such as AWS Lambda, Amazon API Gateway, AWS Step Functions, and Amazon EventBridge. We learned about design patterns such as error handling, managing quotas, observability, and testing for serverless applications. Serverless applications come with a lot of simplification for the operations and allow builders to focus on their business logic. Following the serverless applications considerations we've discussed in this chapter will ensure smoother operations and improve their resilience.

In the next chapter, we will explore resilience considerations for container-based applications, which introduce unique challenges and opportunities for building fault-tolerant systems on AWS.

9

Using Containers to Improve Resiliency

Containers have emerged as a game-changing technology in the world of cloud computing, offering a lightweight and efficient way to package and deploy applications. At their core, containers are isolated environments that encapsulate an application and its dependencies, allowing it to run consistently across different computing environments. Unlike traditional **virtual machines** (**VMs**), which virtualize the entire hardware stack, containers virtualize at the operating system level, sharing the host's **kernel** and reducing overhead. Containers are particularly well-suited for microservices architectures, which we introduced in previous chapters. By breaking down **monolithic applications** into smaller, independent services, containers enable each service to be developed, deployed, and scaled independently. This modular approach not only improves resiliency by isolating failures to individual services but also facilitates faster development cycles and easier maintenance.

In this chapter, we will explore various aspects of using containers to improve resiliency on AWS, as follows:

- Immutable infrastructure with containers
- Scaling and load-balancing containerized applications
- Inter-service communication with containers
- Security considerations for container resilience

Technical requirements

This chapter provides a few practical examples to demonstrate a few concepts. To use them, here are some requirements to have installed on your environment:

- A non-production AWS account
- A Bash terminal; if you don't have a Linux host, you can use AWS CloudShell

- The **AWS command line interface** (**AWS CLI**); AWS CloudShell has the latest version of the AWS CLI pre-installed (`https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html`)
- Terraform (`https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli`)
- Git (`https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`)
- Go programming language (`https://go.dev/doc/install`)
- Docker engine (`https://docs.docker.com/engine/install/`)
- Docker compose (`https://docs.docker.com/compose/install/linux/`)
- Packer (`https://developer.hashicorp.com/packer/tutorials/docker-get-started/get-started-install-cli`)

> **Important note**
>
> We will use a Go application that is placed in the `Chapter9/1-example-app` folder (`https://github.com/PacktPublishing/Building-Resilient-Architectures-on-AWS/tree/main/Chapter9/1-example-app`) in the GitHub repository of this book as an example throughout this chapter. All code references for the chapter will be in the `Chapter9` folder.

Immutable infrastructure with containers

Immutable infrastructure is a paradigm shift in the way we manage and deploy applications. Instead of modifying existing systems, immutable infrastructure advocates for creating new, immutable instances whenever changes are required. This approach brings several benefits in terms of resiliency, consistency, and manageability. The core idea behind immutable infrastructure is to treat VMs and cloud infrastructure components as disposable entities.

> **Important note**
>
> On AWS, VMs are referred to as **Elastic Compute Cloud service** (**EC2**) **instances** or just *instances* for short. Instances are also referred to as **servers**. That said, a server is not always virtual, and AWS does propose **bare metal** (non-VMs) with various CPU architectures. Check the EC2 catalog for more information (`https://aws.amazon.com/ec2/instance-types/`).

To really grasp what immutable infrastructure means, let's build up from traditional VMs and then show the advantages of using containers. We'll propose hands-on examples to better illustrate the concepts. After building the example application using VMs, we'll build and deploy with containers on AWS.

Concept of immutable infrastructure

Immutable infrastructure boils down to how we manage changes. For example, rather than making changes to a running instance, which can lead to configuration drift and potential inconsistencies, the immutable infrastructure concept dictates that a new instance should be created with the desired configuration instead. This new instance is then tested, validated, and deployed, while the old instance is discarded or terminated. One of the primary benefits of immutable infrastructure is increased reliability and consistency. Since each instance is created from a known, immutable source, for EC2, an immutable source is **Amazon Machine Image** (**AMI**). The risk of configuration drift is minimized, ensuring that all instances are identical and behave consistently. This consistency also simplifies troubleshooting and reduces the potential for human error during manual configuration changes. We can visualize this workflow with *Figure 9.1* where we start an EC2 **Auto Scaling group** (**ASG**) (group of instances) with `AMI-001` that has the current version of an application. Next, a developer publishes a new version of the application, which triggers a **CI/CD pipeline** that results in a newer AMI version that includes the new `AMI-002` code. Finally, the pipeline updates the launch configuration of the ASG, and we end up with completely new instances with all dependencies packaged once in the AMI and launched multiple times.

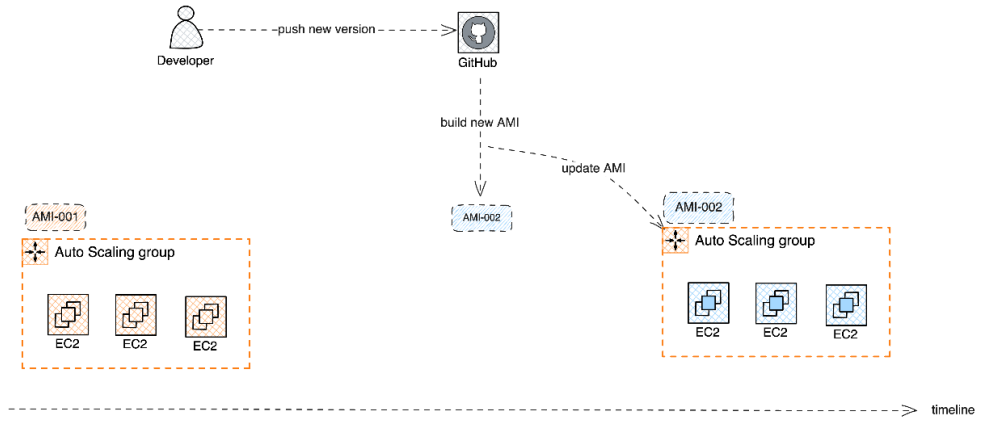

Figure 9.1 – Immutable instances deployment timeline

Immutable infrastructure also promotes better separation of concerns. The process of building and configuring instances is decoupled from the runtime environment, allowing for more robust and automated deployment pipelines. This separation enables faster and more frequent deployments and easier rollbacks in case of issues, improving overall resiliency and reducing downtime.

In the example in `Chapter9/3-container-example` in the GitHub repository, run the following script to build and publish a new AMI to AWS. Replace `<AWS_REGION>` with your current AWS region:

```
$ cd Chapter9/3-container-example
$ ./build-publish-ami.sh <AWS_REGION>
```

After a few minutes, the image should be published, and running the following command should list the new image:

```
$ aws ec2 describe-images --owners self
```

Next, let's build the same application with containers and see their advantages.

Building and managing container images

Containers are at the heart of immutable infrastructure, and this section itself could be an entire book. We'll try to summarize the main concepts. When we talk about containers, we refer to three key components: **container images**, **container runtimes**, and **container orchestration platforms**.

Container images

A container image is a lightweight, stand-alone, executable package that includes everything needed to run an application: code, runtime, system tools, libraries, and settings. Container images are the starting point of the container lifecycle and serve as the immutable building blocks for deploying applications. If you think about the *Figure 9.1* example, the result of the CI/CD pipelines here will be a container image that would be easier and faster to build. We'll see that in action!

Container images are packages, typically stored and distributed through container registries, which act as central repositories for managing and sharing container images. Popular container registries include **Docker Hub** (the official Docker registry), Amazon **Elastic Container Registry** (**ECR**), **Quay.io** (offered by Red Hat), and **JFrog Artifactory**. We'll cover Docker in the next section with container runtimes.

> **Important note**
>
> You might have seen a lot of use of **Docker image** when talking about container images because *Docker, inc.* is a software company that created the **Docker** technology, which is a set of tools (including Docker Hub and many others) to manage and automate the lifecycle of containers. They did it in a developer-friendly manner, focusing on developer productivity, which created worldwide, massive adoption, but Docker and containers are two different things. Containers existed way before Docker!

Let's transform our example application deployment by building a container image using Docker. Create a file named `Dockerfile` in the root folder of the Go application. The directory should look like *Figure 9.2*.

Figure 9.2 – Dockerfile in application directory

In the Dockerfile in *Figure 9.3*, we start from a Docker base image (Golang version 1.21) that already has Go installed, which allows us to build the application. You can notice another similar instruction on *Line 8* with a Linux Alpine image. This process is called a **multi-stage build**. This allows us to use any larger image with development requirements (*Lines 3-6*), and since Go produces a compiled binary that doesn't need the Go runtime, we can take a lightweight Linux image to finalize the image, copying the result of the `builder` image (*Line 11*). This results in a small image that can be downloaded faster, for faster **scaling**! Finally, we instruct Docker that we will need port `8000` (*Line 12*) and we run the compiled application (*Line 13*) by default.

> **Important note**
> To dive deeper into the Dockerfile specification, check out the Dockerfile reference online (`https://docs.docker.com/reference/dockerfile/`).

Dockerfile

```
1  FROM golang:1.21 as builder
2  ENV GOPROXY=https://goproxy.io,direct
3  WORKDIR /go/src/app
4  COPY . .
5  RUN go get .
6  RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
7
8  FROM alpine:latest
9  WORKDIR /app
10 RUN apk --no-cache add ca-certificates
11 COPY --from=builder /go/src/app/app .
12 EXPOSE 8000
13 CMD ["./app"]
```

Figure 9.3 – Dockerfile content

Now, let's build the image locally with the following commands. Make sure to be in the directory where the Dockerfile is located. The `-t` flag allows us to specify a name tag and version for the image being built. Note that the latest tag is not mandatory and will be applied regardless of whether you specify it or not:

```
$ cd Chapter9/3-container-example
$ docker build . -t example-chapter-9:latest
```

For production, it's recommended that you follow semantic versioning (`https://semver.org/`), for example, `v0.1.0` representing `major.minor.patch`, CI/CD build number, or Git tag. You can also combine multiple tags with each image version. The following command will apply a second tag to our freshly built image:

```
$ docker tag example-chapter-9:latest v0.1.0
```

In this current form, the image is just a static asset that is only available in your local environment and not usable by anyone. With the following command, we can create a private ECR repository, available to your entire AWS account. You will typically do it once per image, and then push multiple versions to the image:

```
$ export ECR_REPOSITORY_URI=$(aws ecr create-repository --repository
example-chapter-9 --query repository.repositoryUri --output text)
```

Authenticate your local Docker client to the ECR registry. Make sure to replace `AWS_REGION` with your current AWS Region or set the variable with `export AWS_REGION=<your_region>`. The command line goes like this:

```
$ aws ecr get-login-password --region $AWS_REGION | docker login
--username AWS --password-stdin $ECR_REPOSITORY_URI
```

Next, tag your local image with the ECR repository URI:

```
docker tag example-chapter-9:latest "${ECR_REPOSITORY_URI}:latest"
```

Push the tagged image to the ECR repository. As an exercise, also tag the `v0.1.0` version and push it to ECR:

```
docker push "${ECR_REPOSITORY_URI}:latest"
```

Now, your Go application is packaged as an immutable container image and stored in Amazon ECR, ready for deployment to various container orchestration platforms, such as ECS or EKS, which we'll see in one of the upcoming subsections.

If we compare it with the AMI build example, you'll immediately notice the speed advantage that containers bring. As containers have a smaller footprint, we package fewer dependencies, which impacts their sizes. In the next section, we'll see ways to run these container images.

Container runtimes

A container runtime is the software responsible for running and managing containers on a host operating system. It provides the necessary environment and resources for containers to execute and isolate their processes, networking, and storage. Examples of popular container runtimes include Docker (`https://www.docker.com/` by Docker Inc.), **Finch** (`https://runfinch.com/`; it's open source and originally by AWS), **containerd** (`https://containerd.io/` by CNCF), and **CRI-O** (`https://cri-o.io/`; it's runtime optimized for the **Kubernetes** platform, which we'll cover in the next section).

To standardize all container runtimes with a need to address open, vendor-neutral container standards, the **Open Container Initiative** (**OCI**) (`https://opencontainers.org/`) was launched by Docker Inc. The OCI is a lightweight, open governance structure for creating open industry standards around container formats and runtimes. The OCI currently hosts two specifications:

- **The Image Specification** (**OCI image spec**): This specification defines how to describe and distribute a container image, including the content, metadata, and layer distribution formats.
- **The Runtime Specification** (**OCI Runtime spec**): This specification defines how to run a filesystem bundle that is unpacked on disk to create a container. It outlines the configuration, execution environment, and lifecycle of a container.

By adhering to the OCI standards, container runtimes such as containerd, CRI-O, and even Docker's own runtime (which is OCI-compliant) can ensure the interoperability and portability of containers across different platforms and environments.

Using the Docker runtime, we will continue with our hands-on example and run the container we built previously with the following commands:

```
$ docker run --rm -it -p 8080:8080 example-chapter-9
$ curl localhost:8080/
```

You should have a response from the application and the same app can be hosted anywhere, provided the environment runs the Docker engine. Note that you could also run the container with the ECR name, as it's just a tag on the actual image that is already available in your local environment.

> **Important note**
>
> Images are built for specific CPU architectures. For example, if you build a docker image on macOS, it will not run on Linux by default. However, Docker provides you the capability to do a cross-platform build, for example, by building from macOS to Linux:
>
> `$ docker buildx build -t example-chapter-9 . --platform=linux/amd64`

When you have an image and a container running, how do you operate your containers in a production environment? Let's see how container orchestration platforms help achieve production-grade, resilient systems.

Container orchestration platforms

While container runtimes manage individual containers, container orchestration platforms are responsible for automating and managing the deployment, scaling, and networking of containerized applications across multiple hosts. These platforms provide features such as **load balancing**, service discovery, self-healing, and automated rollouts and rollbacks. Let's have an overview of the most popular ones in the industry:

- **Docker Compose**: This is a tool for defining and running multi-container Docker applications. It allows you to define and manage the services that make up your application in a declarative YAML file, specifying the configuration, dependencies, and runtime environment for each service. Docker Compose simplifies the process of running and managing containerized applications locally or on a single host. It's usually used for development environments with multiple containers (e.g., a web app container and its database in two different containers launched at the same time).
- **Docker Swarm**: This is a native clustering solution for Docker, enabling you to create and manage a swarm of Docker nodes as a single virtual system. Swarm provides features such as service discovery, load balancing, and scaling, allowing you to deploy and manage containerized applications across multiple hosts. While Swarm is a relatively simple orchestration tool, it can be useful for smaller-scale deployments or as a stepping-stone toward more advanced orchestration platforms.
- **Kubernetes**: This is an open source container orchestration platform that has become the industry standard for deploying, scaling, and managing containerized applications. Kubernetes provides a robust set of features, including automatic bin-packing, self-healing, horizontal scaling, service discovery, load balancing, and rolling updates and rollbacks. It supports a wide range of workloads, from stateless to stateful applications, and can be deployed on-premises or in the cloud.

You might have heard the famous phrase *it works on my machine*. This usually happens when an issue is not reproducible in another environment. Containers partially solve this problem as it's easy to replicate the deployment conditions. In the next section, we'll see how to deploy container applications on AWS.

Deploying containers on AWS

It is always possible to deploy containers on an EC2 instance with a Docker engine installed, or maybe with docker-compose, to treat containers as standard applications on EC2. However, this would not take advantage of the scalability and flexibility that containers have. AWS offers several container

orchestration platforms tailored for different use cases and workload requirements. At the time of writing this book, here are the most popular services to run containers on AWS:

- **AWS App Runner**: App Runner is a fully managed service that provides an abstracted experience for deploying and running containerized web applications and APIs. I would say this is the easiest way to run containers on AWS. It automatically provisions and scales the required infrastructure, including load balancing, scaling, and security, making it a simple and cost-effective option for running containerized applications without managing the underlying infrastructure. Developers only need to focus on writing resilient application code and leave all the burden to AWS. With App Runner, you can deploy directly from a Git source or provide a container image. This service is very useful and straightforward for most simple use cases.
- **Amazon Elastic Container Service** (**Amazon ECS**): ECS is a fully managed container orchestration service that simplifies running and scaling containerized applications on AWS. It supports Docker containers and allows you to launch and stop container-based applications with simple API calls. ECS integrates with many other AWS services, including **Elastic Load Balancing** (**ELB**), AWS **Identity and Access Management** (**IAM**), and Amazon CloudWatch for monitoring and logging. ECS has its own constructs and is a bit more elaborate than App Runner. We'll explore a bit more details in the next section.
- **Amazon Elastic Kubernetes Service** (**Amazon EKS**): EKS is a managed Kubernetes service that simplifies the deployment, management, and scaling of Kubernetes clusters on AWS. It provides a certified Kubernetes conformant platform, allowing you to leverage the full power and flexibility of Kubernetes while offloading the operational overhead of managing a Kubernetes cluster to AWS. While you can run your own Kubernetes platform on EC2, EKS manages the Kubernetes control plane components (such as the API server, etcd, and controller manager) across multiple Availability Zones for high availability. EKS will handle the process of keeping the Kubernetes control plane up to date with the latest version and security patches. Finally, EKS simplifies the Kubernetes integration with the rest of the AWS ecosystem such as ELB, AWS IAM, Amazon **Virtual Private Cloud** (**VPC**), and Amazon CloudWatch through AWS-managed add-ons. Kubernetes is a complex topic and going to production requires a thorough understanding of the platform, even when using a managed service. Operated well and it will provide world-class resilience and scalability.
- **AWS Fargate**: Fargate is a serverless compute engine for containers that works with both ECS and EKS. It allows you to run containers without having to manage the underlying infrastructure, eliminating the need to provision and manage servers or clusters. Fargate automatically provisions and scales the required compute resources, making it a convenient option for running containerized applications without managing the underlying infrastructure. Fargate gives you the flexibility of the container runtime (EKS and ECS) and most of their mechanisms but takes away the pain of managing low-level infrastructure.
- **AWS Lambda**: Lambda has the possibility to deploy functions using container mechanisms. At the end of the deployment, Lambda will still expect the Lambda runtime, and syntax to be

respected as a regular function. So, this is not exactly a container runtime platform but a way to provision functions easily.

Regardless of the runtime, if we visualize immutable deployments using containers, *Figure 9.1* will become like *Figure 9.4*. The result of a CI/CD pipeline will now become a container image, properly tagged and versioned. A deployment will consist of replacing appropriate containers with the newer image following a deployment strategy by the orchestration platform.

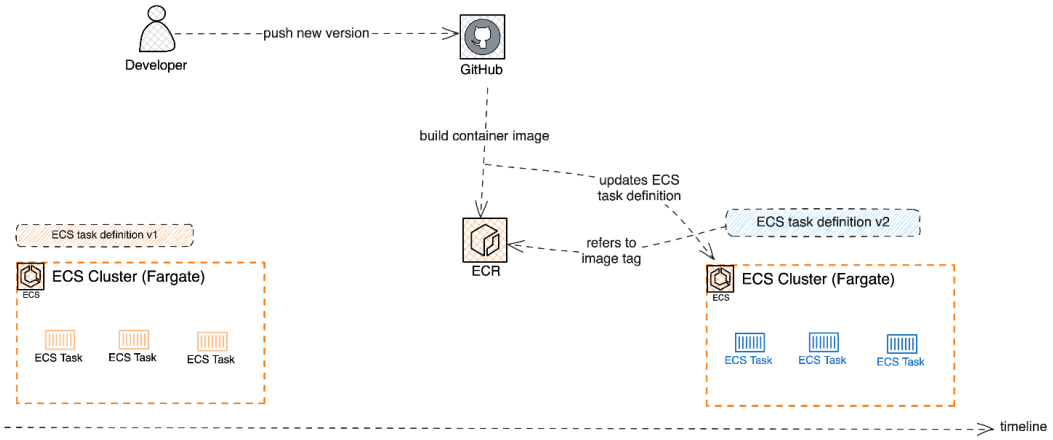

Figure 9.4 – New deployment process with ECS

Immutable infrastructure allows for uniform and stable deployments. Containers standardize packaging, testing, and production deployment, making it easier to scale. In the next section, we'll see how to scale containers to respond to customer demands.

Scaling and load-balancing containerized applications

Scaling and load balancing are critical aspects of ensuring the resiliency and performance of any application, including applications running inside containers. One of the important mechanisms for resilience is scaling and spreading the load over multiple copies of the systems. Scaling involves adjusting the resources allocated to an application based on demand, ensuring that it can handle fluctuations in traffic or workload. Effective monitoring is essential for making informed scaling decisions. You need to collect and analyze metrics such as CPU utilization, memory usage, network traffic, and sometimes, custom metrics from your applications. By setting appropriate alarms and thresholds based on these metrics, you can trigger scaling actions to maintain the desired performance and resiliency of your applications. Services such as AWS CloudWatch provide comprehensive monitoring capabilities for containerized applications doing heavy lifting and providing automatic visualizations for performance.

We have covered **horizontal scaling** in *Chapters 2* and *6*, as it's a core recipe of resilience. Still, let's touch on the minor specificities when it comes to containers, especially because there are two main types of scaling: horizontal and **vertical scaling**. Depending on the runtime, vertical scaling can become tricky with containers.

Horizontal scaling

Horizontal scaling involves adding or removing instances of an application to match the current demand. In the context of containerized applications, this typically means scaling the number of container instances, Kubernetes pods, or ECS tasks running the application. The process of horizontal scaling is generally similar across different compute platforms, such as AWS App Runner, ECS, and EKS. However, the specific mechanisms and configurations may vary.

App Runner makes it very easy to scale applications based on concurrency. You can provision multiple auto scaling configurations and associate them with your services depending on their traffic patterns. App Runner allows you to specify the minimum and maximum number of container copies you can afford to have, and lets you scale in and out based on the request concurrency.

With Amazon in ECS, you can configure a service to maintain a desired number of tasks, and ECS will automatically replace any stopped or failed tasks. You can also set up Auto Scaling policies to dynamically adjust the number of tasks based on metrics such as CPU utilization or memory usage. At the time of writing this book, ECS propose two types of scaling policies:

- **Target tracking**: Here, you give ECS a target CloudWatch metric and metric value you want to maintain. You also give a cooldown period, so the scaling (in and out) is done at intervals and is not too aggressive.

  Here's an example of a target tracking of *CPU utilization at 70%*. If you have two containers running (at minimum) and their average container utilization is above 70%, ECS will initiate a scaling activity, might add one or two more containers (the number can vary depending on your patterns), and will reassess until the target is met, while respecting the minimum and maximum number of containers you have pre-set.

- **Step scaling**: This is a more traditional approach whereby you provide any CloudWatch alarm in which you will define your own thresholds. When the alarms are triggered, ECS will increase the number of tasks by increments that you specify.

  Here is an example. If you have an alarm on a high number of visible messages in an SQS queue for five minutes, ECS can add three more containers when an alarm is triggered to process the queue.

With EKS or Kubernetes on EC2, you can leverage the **Horizontal Pod Autoscaler** (**HPA**) (`https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`) to automatically scale the number of pods (containers) based on observed CPU utilization or other custom metrics for a Kubernetes deployment. In general, for all these services, you can always force the scaling and make it aggressive. For example, if you face sudden traffic spikes, you can modify the minimums (instances and/or containers) to absorb traffic. However, you need strong observability to make sure scaling doesn't create constraints on other parts of the system, for example, maxing out the allowed number of database connections.

With containers, scaling can happen both on the number of nodes in a cluster and on the number of containers in a single node. You can ask to scale to 100 containers, but you need to ensure that you have enough capacity in your cluster nodes (EC2 instances) to run those containers. Depending on the platform chosen, one level can be abstracted for you, for example, with AppRunner. For ECS (non-Fargate), you will need to manage this yourself using metrics and alarms on container reservation capacity metrics. EKS has many options to resolve this and relies on the customer to manage this two-dimensional scaling. Solutions such as **Karpenter** (`https://karpenter.sh/`) would automatically adjust the number of worker nodes in your Kubernetes cluster based on the resource demands of your applications. You can go further in application scaling with solutions such as **Kubernetes Event-Driven Autoscaling** (**KEDA**) (`https://keda.sh/`), which extends HPA to create event-driven workflows for auto scaling using multiple event sources such as Apache Kafka, ActiveMQ, SQS, DynamoDB, and many more.

You can visualize two dimensions of horizontal scaling with containers by looking at the following figure, where the capacity requested needs to be scheduled on virtual nodes. When node management is offloaded to AWS (for example, with Fargate), you don't have to worry about the node's capacity.

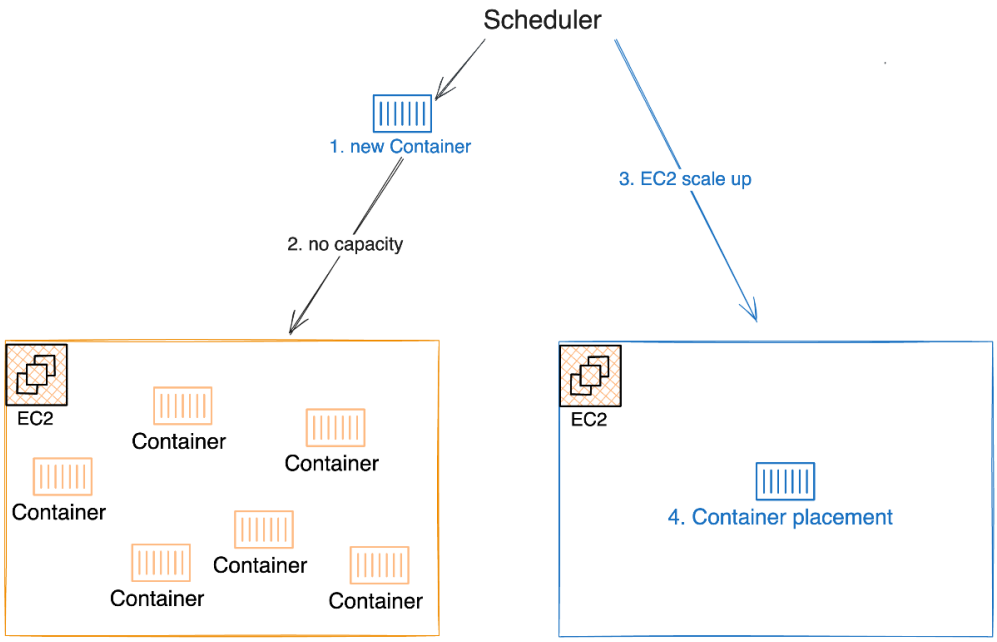

Figure 9.5 – Two-dimensional container horizontal scaling

Vertical scaling

This type of scaling involves increasing or decreasing the resources (CPU, memory, etc.) allocated to an existing instance of an application. With AWS Fargate, you can specify the desired CPU and memory configurations for your tasks, and AWS will automatically provision the necessary resources.

For non-serverless container platforms such as ECS or EKS running on EC2 instances, vertical scaling may require more manual intervention, such as modifying the instance types or resource limits of your worker nodes or tasks. Vertical scaling can provide more control and granularity in fine-tuning how scaling works. For example, you can assign a group of containers to a specific set of instance types that are faster for I/O operations on SSD (storage-optimized instances) for a set of services that manipulate data on disk. You can assign memory-intensive containers to another node group that is memory optimized.

While we have covered scaling a bit more, the mechanisms of load balancing do not really change from what we have discussed previously, including integrations with AWS load balancers and health checks. On the other hand, communication between containerized services can be challenging. We'll explore ways to communicate in a cluster between services in the following sections.

Inter-service communication with containers

In a microservices architecture, where applications are decomposed into multiple independent services, efficient and reliable inter-service communication is crucial. Containerized environments add an additional layer of complexity, as services are often distributed across multiple containers and hosts. Ensuring seamless communication between these services is essential for maintaining the resiliency and performance of the overall application.

There are several approaches to facilitating inter-service communication in containerized environments, each with its own advantages and trade-offs.

Service discovery

Service discovery mechanisms enable services to locate and communicate with each other dynamically, without relying on hardcoded IP addresses or ports. AWS Cloud Map and Kubernetes DNS are examples of service discovery solutions that can be used in containerized environments. For ECS and EKS, AWS provides different service discovery mechanisms to facilitate inter-service communication. Really, the role of service discovery is, from within a cluster, to call a remote service with a friendly name and let the system worry about how to find that service, regardless of its placement in the cluster.

ECS service discovery with AWS Cloud Map

AWS Cloud Map is a cloud resource discovery service that allows you to define custom names for your application resources and associate them with the corresponding resource endpoints. With ECS, you can use Cloud Map to register your services and discover their endpoints dynamically.

Here's an example of how you can use the AWS CLI to create a Cloud Map namespace and register an ECS service:

```
aws servicediscovery create-private-dns-namespace \
    --name my-namespace \
    --vpc VPC_ID
```

Then, you would register your ECS service with the following command:

```
aws servicediscovery create-service \
    --name my-service \
    --dns-config NamespaceId=NAMESPACE_ID,RoutingPolicy=MULTIVALUE \
    --health-check-custom-config FailureThreshold=1 \
    --namespace-id NAMESPACE_ID
```

Once registered, other services can discover and communicate with the registered service using the Cloud Map DNS name or API calls. ECS does provide **Service Connect** (`https://docs.aws.amazon.com/AmazonECS/latest/developerguide/service-connect.html`), which provides both a service discovery and service mesh for ECS clusters.

EKS service discovery with Kubernetes DNS

In Kubernetes (and EKS), service discovery is a first-class citizen, and DNS is an integral part of the service discovery mechanism. Kubernetes automatically assigns a DNS name to every service and load balances traffic across the service's pods. The DNS schema for services in Kubernetes follows the `<service-name>.<namespace>.svc.cluster.local` pattern. For example, if we deployed our example on EKS in the `default` namespace, our `example-chapter-9` Kubernetes service would have an `example-chapter-9.default.svc.cluster.local` DNS name. Additionally, Kubernetes supports different types of services, such as **ClusterIP**, **NodePort**, and **LoadBalancer**, each with its own DNS schema for service discovery. To know more about those services and deployment types, please read the Kubernetes services documentation (`https://kubernetes.io/docs/concepts/services-networking/service/`) as it's a vast topic. Once the service is created, other pods within the same cluster can communicate with it using the assigned DNS name or cluster IP.

By leveraging these service discovery mechanisms, containerized applications running on ECS or EKS can achieve resilient inter-service communication, enabling seamless scaling and load balancing across multiple instances of a service. DNS can fail, and service communication can be impacted, so always monitor Kubernetes DNS, usually **CoreDNS**. Also, if you are using CloudMap, leverage CloudWatch metrics for CloudMap.

Next, let's see how load balancing can help with service communication

Load balancing

Load balancing distributes incoming traffic across multiple instances of a service, ensuring high availability and efficient resource utilization. AWS ELB and Kubernetes-native services are commonly used for load balancing in containerized applications. Counterintuitively to this section, load balancing is utilized more to expose your service outside of the clusters and to communicate with the external world. Sometimes, it's the easiest way to start, but the round trip from outside and going back inside can generate some additional slowness. See the following figure for how load balancing compares to service discovery:

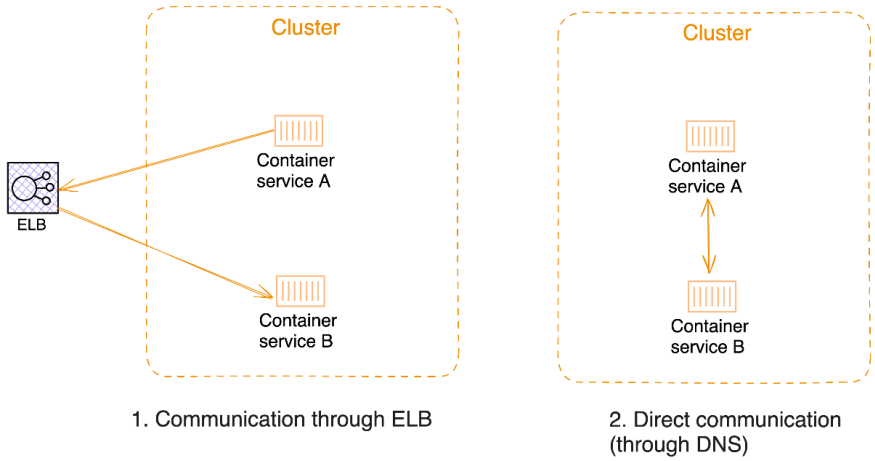

Figure 9.6 – Inter-service communication styles

In communication through ELB, Container services A and B in the preceding figure will reach out to each other through an external load balancer. In direct communication, they will resolve a DNS name that corresponds to a local address of containers in the same cluster. The first scenario will add a slight additional lookup time and longer round-trip. Depending on the use case, the additional lookup time can be *non-negligible*.

That said, Kubernetes services by default act as load balancers and enable communication between different components within the cluster through multiple mechanisms (`https://kubernetes.io/docs/concepts/services-networking/`).

Sometimes, we want fine-grain control over how communication routing between services works that go beyond simple load balancing or service discovery. Let's talk about **service meshes** in the next section!

Service mesh

A service mesh is a dedicated infrastructure layer that handles service-to-service communication, providing features such as traffic management, observability, and security enforcement. Service

meshes act as a transparent proxy layer, intercepting and controlling the traffic between services without requiring changes to the application code. Service meshes, when implemented well, can help in improving resilience as they aim to provide the following:

- **Service discovery and load balancing**: Service meshes provide a centralized way of discovering and managing the network of microservices within a cluster. They handle service registration, load balancing, and routing traffic between services, reducing the need for each service to implement these complex functionalities.
- **Resilience and fault tolerance**: Service meshes offer features that enhance resilience, such as the following:
  - **Circuit breaking**: They can automatically detect and isolate failing services, preventing cascading failures and enabling graceful degradation.
  - **Retries and timeouts**: Service meshes can automatically retry failed requests and enforce timeouts, improving the overall reliability of the system.
- **Fault injection**: They support injecting faults into the system for testing purposes, allowing you to validate the resilience of your services under various failure scenarios.
- **Observability and monitoring**: Service meshes provide detailed insights into service-to-service communication, capturing metrics, logs, and traces. This visibility aids in troubleshooting, performance optimization, and monitoring the overall health of the system.
- **Traffic management**: Service meshes enable advanced traffic management capabilities, such as canary deployments, traffic shifting, and traffic mirroring. These features facilitate safe and controlled rollouts and testing of new service versions.
- **Security**: Service meshes can enforce secure communication between services by enabling mutual TLS encryption, authentication, and authorization policies. This helps protect your microservices from unauthorized access and potential attacks.

Although this looks like a silver bullet that solves all problems, it also comes with additional complexity as it's another layer to manage, scale, and monitor. Here are a few popular solutions for implementing service mesh on AWS:

- **AWS App Mesh**: An AWS fully managed service mesh that integrates with ECS, EKS, and Fargate, providing features such as traffic routing, circuit breaking, retries, and observability through integration with AWS Cloud Map and AWS X-Ray.
- **Istio**: An open source service mesh that can be deployed on EKS, offering a comprehensive set of features for traffic management, security, observability, and policy enforcement.
- **Consul**: A service mesh solution from HashiCorp that can be used with both ECS and EKS, providing service discovery, configuration management, and secure service-to-service communication.

- **Linkerd**: A lightweight, open-source service mesh that can be used with EKS, focusing on providing a simple and efficient service mesh solution with features such as traffic management, observability, and secure communication.

Async communications with message brokers

We talked about this in *Chapter 8* and this is also a very resilient way to communicate data between services. Message queues decouple services by providing an asynchronous communication mechanism, where messages are sent to a queue and processed independently by consumers. Amazon **Simple Queue Service** (**SQS**), Amazon Managed Streaming for Apache Kafka (`https://aws.amazon.com/msk/`), and Amazon MQ (`https://aws.amazon.com/amazon-mq/`) are examples of message queues that can be used in containerized environments.

The choice of inter-service communication approach depends on factors such as the complexity of the application, performance requirements, observability needs, and desired level of control and customization. Service meshes, in particular, have gained popularity due to their ability to provide a consistent and centralized way to manage service-to-service communication, enabling features like traffic management, observability, and security enforcement.

Regardless of the approach chosen, ensuring efficient and reliable inter-service communication is crucial for maintaining the resiliency and performance of containerized applications, especially in large-scale microservices architectures.

Let's now see in which ways preventing security issues can improve resilience.

Security considerations for container resilience

Security is a critical aspect that can significantly impact the resilience of containerized applications. In the containerized world, security vulnerabilities or misconfigurations can lead to various issues, such as data breaches, service disruptions, and system compromises. These security concerns can directly affect the availability, reliability, and overall resilience of your containerized applications. Therefore, it's essential to adopt robust security practices to ensure the resilience of your container-based infrastructure. The first security layer is offered by AWS core infrastructure with VPCs, subnets, and security groups to provide a natural boundary between container services. Next, we'll look at some additional considerations for improving the security for containers.

Securing container images and registries

Container images are the building blocks of containerized applications, and their security is paramount. Compromised or vulnerable container images can introduce security risks and potentially lead to system breaches or service disruptions. To mitigate these risks, it's crucial to follow best practices for securing container images and registries. To avoid these risks, there are a few best practices to adopt:

- **Image scanning**: As a container image should be already small, automated image scanning processes help identify and remediate vulnerabilities in your container images by scanning all

packages involved for known vulnerabilities. As a best practice, an image should include only necessary packages and be built as minimally as possible, just like we did in our example above. Tools such as AWS ECR Image Scanning, Docker Scout, and many others will scan as soon as you push a new image version and give you insights into known vulnerabilities.

- **Registry authentication and access control**: Secure your container registries by implementing strong authentication mechanisms and access control policies. AWS ECR provides features such as AWS IAM integration, resource-based permissions, and encryption at rest to ensure the security of your container images.
- **Image signing and verification**: Digitally sign your container images to ensure their integrity and authenticity. Tools such as cosign (`https://github.com/sigstore/cosign`) and Docker Content Trust (`https://docs.docker.com/engine/security/trust/`) can help you sign and verify container images, preventing unauthorized modifications and ensuring that only trusted images are deployed.
- **Least privilege principle**: Build and run your container images with the least privilege principle in mind. Avoid running containers as root, and limit the capabilities and permissions granted to your containers based on their specific requirements. Limit IAM permissions for a container or any AWS application in general.
- **Secure supply chain**: Implement secure practices throughout your container image build and deployment pipeline. This includes using trusted base images, validating third-party dependencies, and maintaining a secure and auditable build process.

By following these best practices for securing container images and registries, you can significantly enhance the security and resilience of your containerized applications, reducing the risk of security breaches, data leaks, and service disruptions.

In the next subsection, we can discuss additional security best practices, such as network security, runtime security, and security monitoring for containerized environments.

Securing container runtimes

During runtime, containers can be exposed to various security risks, such as resource exhaustion, privilege escalation, and unauthorized access. Implementing robust runtime security measures can help mitigate these risks and enhance the overall resilience of your containerized environment. Applying security on runtimes involves the following:

- **Container isolation**: Implement secure workload configurations by running containers with non-root user privileges, dropping unnecessary capabilities, and applying security contexts (e.g., **SELinux** and **AppArmor**) to restrict container access to system resources. This helps prevent container breakouts and limits the impact of a compromised container on other components of your application.

- **Resource limits**: Set appropriate resource limits (CPU, memory, and disk) for your containers to prevent resource exhaustion and ensure fair resource allocation. This helps mitigate the risk of denial-of-service attacks and ensures that critical components of your application have access to the necessary resources.
- **Immutable infrastructure**: As we saw in the first section, adopting an immutable infrastructure approach by treating containers as immutable and disposable entities, instead of modifying running containers, and spinning up new containers can ensure a consistent and secure runtime environment.
- **Monitoring**: Implement secure logging and auditing mechanisms to capture and analyze container activity logs. EKS offers audit logs that capture all Kubernetes control plane actions. These can be centralized in CloudWatch logs and leveraged for analysis. Additionally, audit logs can help you maintain compliance with regulatory requirements. Application logs, metrics, and potentially distributed traces need to be collected to get a comprehensive state of the health of the cluster. Services such as CloudWatch (metrics and logs), AWS X-Ray (traces), Amazon Managed Services for Prometheus (metrics), and Amazon OpenSearch (logs and traces) can help manage your containers and clusters' telemetry data.

By following these runtime security best practices, you can significantly enhance the resilience of your containerized applications by mitigating the risk of security breaches, resource exhaustion, and unauthorized access during runtime. Services such as AWS WAF and GuardDuty can help elevate your security posture and should be considered when exposing your container applications on the internet. To go further on runtime security for Kubernetes, explore the AWS EKS best practices guide page (`https://aws.github.io/aws-eks-best-practices/security/docs/runtime/`).

Secrets management and encryption

Containerized applications often require access to sensitive information, such as database credentials, API keys, and other secrets. Proper management of these secrets is crucial for maintaining the security and resilience of your containerized environment. Failure to securely handle secrets can lead to data breaches, unauthorized access, and potential service disruptions.

AWS provides a couple of services and mechanisms to help you manage secrets securely in your containerized applications:

- **AWS Secrets Manager**: AWS Secrets Manager is a service that allows you to securely store, rotate, and retrieve secrets, such as database credentials, API keys, and other sensitive data. ECS and EKS both integrate with Secrets Manager, enabling you to securely inject secrets into your containers during runtime. This avoids hardcoding passwords and secrets into the code or container configurations.

- **AWS Systems Manager Parameter Store** (**SSM Parameter Store**): This is like the preceding service, with fewer options (for example, there is no managed secret rotation). Parameter Store is simply a key-value store with sensitive data protection. It has a free tier and can be integrated with ECS, EKS, App Runner, or AWS Lambda.

As you might have guessed by now, Kubernetes has its own internal secrets management mechanisms (`https://kubernetes.io/docs/concepts/configuration/secret/`) to inject secrets into pods and containers. EKS makes it easy to integrate with AWS services for secret management.

Regardless of the system you use, it's essential to follow best practices for secrets management, such as regularly rotating secrets, limiting access to secrets based on the principle of least privilege, and implementing secure communication channels for transmitting sensitive data.

Summary

In this chapter, we examined how containers provide a lightweight and efficient method for packaging and deploying applications, making them ideal for microservices architectures. We covered the advantages of containers over traditional EC2 instances, such as faster spin-up and tear-down times, immutability, and horizontal and vertical scalability. We saw the particularity of inter-service communication, scaling, and security for containers. This chapter helps builders understand the lifecycle of containers and AWS platforms that can help run them with great resiliency. In addition to *Chapter 8*, where we covered serverless, we can always combine serverless and containers to pick the best tool for the job.

In the next chapter, we will see how to leverage multiple AWS regions to achieve greater resilience.

10
Resilient Architectures Across Regions

In today's interconnected world, where businesses rely heavily on digital infrastructure, even a brief outage can have severe consequences, including financial losses, reputational damage, and customer dissatisfaction. By distributing resources across multiple geographic regions, organizations can mitigate the risk of localized failures, such as natural disasters, power outages, or network disruptions. If one region experiences an issue, the application or service can failover to another region, ensuring continuous operation and minimizing downtime.

However, designing and implementing multi-region architectures is not a trivial task. It requires careful planning, consideration of various factors, and a deep understanding of the trade-offs involved. Factors such as data consistency, latency, cost, and complexity must be carefully evaluated to strike the right balance between resilience, performance, and operational overhead. By understanding the concepts and best practices of this chapter, you will be able to succeed with architecture design that spans across multiple regions.

This chapter explores the benefits and drawbacks of various deployment modes for cross-region architectures on AWS. It provides insights and best practices to help you make informed decisions when designing resilient architectures across multiple regions. The topics covered include the following:

- Understanding active-passive architectures
- Exploring global versus regional services on AWS
- Delving into active-active architectures
- Introducing cell-based architectures

Understanding active-passive architectures

Active-passive architectures (or primary standby) are a common approach to achieving regional resiliency. In this model, one regional deployment serves as the primary or active region, handling all incoming traffic and requests. Simultaneously, a secondary or passive region remains on standby, ready to take over operations in the event of a regional failure or disaster. The primary advantage of an active-passive architecture is its simplicity and cost-effectiveness. Since only one region is actively serving traffic at any given time, the operational costs and resource utilization in the passive region are minimal. This makes it an attractive option for organizations with limited budgets or those seeking a basic level of regional resilience.

In the following figure, you can see an example of active-passive multi-region deployment. It is key here to leverage **infrastructure as code** (**IaC**) to replicate all the components of the infrastructure in both locations.

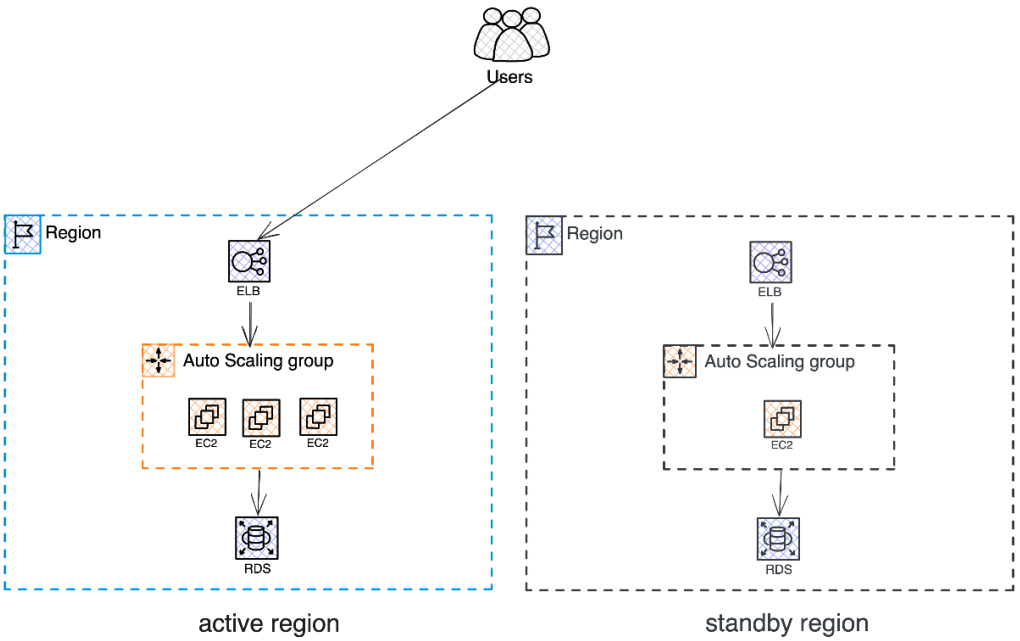

Figure 10.1 – Active passive deployment

Active-passive architectures are often employed for disaster recovery purposes, where the passive region serves as a standby environment. In the event of a regional outage or disaster affecting the primary region, the passive region can be activated, and traffic can be redirected to it, minimizing downtime and ensuring business continuity. This process is typically orchestrated through **failover mechanisms**, often with DNS and/or load balancing configuration. We'll explore the failover mechanisms in more detail in the next section.

Failover mechanisms

In active-passive architectures, the failover mechanism is a critical component that determines when and how the transition from the active region (or primary region) to the passive region (or standby) occurs. The primary goal of the failover mechanism is to detect failures or outages in the active region and initiate the failover process in the passive region in a timely and reliable manner. The overall process looks like the following.

1. **Defining the "working" state**: Before implementing a failover mechanism, it's essential to define what constitutes a *working* state for the application or service. This typically involves monitoring various metrics, such as resource health, availability, performance, and error rates. AWS services such as CloudWatch, CloudTrail, and X-Ray can be leveraged to collect and analyze these signals. For failover correctness, it's best practice for a system to define in advance *what good looks like* as we typically only want to failover when it doesn't look good anymore.
2. **Preparing the passive region**: Before switching over to the passive region, it's crucial to ensure that the passive region is properly configured and ready to handle incoming traffic. This typically involves provisioning and configuring the necessary resources upfront, such as compute instances, databases, storage volumes, and other supporting services. Using infrastructure as code to create these components will help avoid errors or forgetting a component in both active and passive regions. The preparation process may vary depending on the specific application architecture and requirements, including the following:

    - **Compute**: In the passive region, they need to be provisioned and configured to handle the application workload. This includes launching and configuring **Elastic Compute Cloud** (**EC2**) instances or **Auto Scaling groups** (**ASGs**), ensuring they have the latest application code and configurations. If using containerized workloads, deploy and initialize container clusters (e.g., **Amazon ECS** or **Amazon EKS**) in the passive region. For serverless architectures, ensure AWS Lambda functions and related resources are deployed and configured in the passive region.
    - **Databases**: Provisioning and configuring database instances (e.g., Amazon RDS, Amazon Aurora, or Amazon DocumentDB) in the passive region is crucial. Implement data replication strategies to synchronize data from the active region to the passive region. This can be achieved through techniques such as database replication, data snapshots with AWS Backup, or AWS **Database Migration Service** (**DMS**). Ensure that the passive region's database is up to date and ready to handle incoming traffic during failover.
    - **Storage**: Create and configure storage resources, such as **Amazon Elastic File System** (**Amazon EFS**) or Amazon S3 buckets, in the passive region. Replicate or synchronize data from the active region to the passive region's storage resources, if necessary.
    - **Supporting services**: Deploy and configure other supporting services required by the application, such as message queues (**Amazon SQS**), caching layers (Amazon ElastiCache), or load balancers (**Elastic Load Balancing** (**ELB**)), in the passive region. Ensure that any

necessary configurations or data are replicated or synchronized to the passive region's supporting services.

The preparation process for the passive region should be thoroughly tested and validated to ensure that it can seamlessly handle incoming traffic and workloads during a failover event. Additionally, it's crucial to maintain consistency between the active and passive regions' configurations, resource specifications, and application versions to minimize potential issues during failover.

- **Triggering the failover**: Once the monitoring systems detect that the active region is no longer in a *working* state, based on predefined thresholds or conditions, the failover process can be triggered. This can be done manually or through automated systems, depending on the requirements and criticality of the application. In general, everything should be tested, but it's a best practice to perform the switch manually; the action of switching must be in total confidence to avoid a disaster.

One of the most common failover mechanisms for active-passive architectures is DNS failover using **Amazon Route 53**. Route 53 allows you to configure primary and secondary DNS records, pointing to the active and passive regions, respectively. When a failover is triggered, the DNS records can be updated to reroute traffic to the passive region.

Route 53 also supports advanced failover mechanisms, such as health checks and weighted routing policies, which can automate the failover process based on predefined conditions. For example, the next figure shows a standby region that is ready to receive traffic when Route 53 health checks detect a failure in the active region and perform the switch automatically.

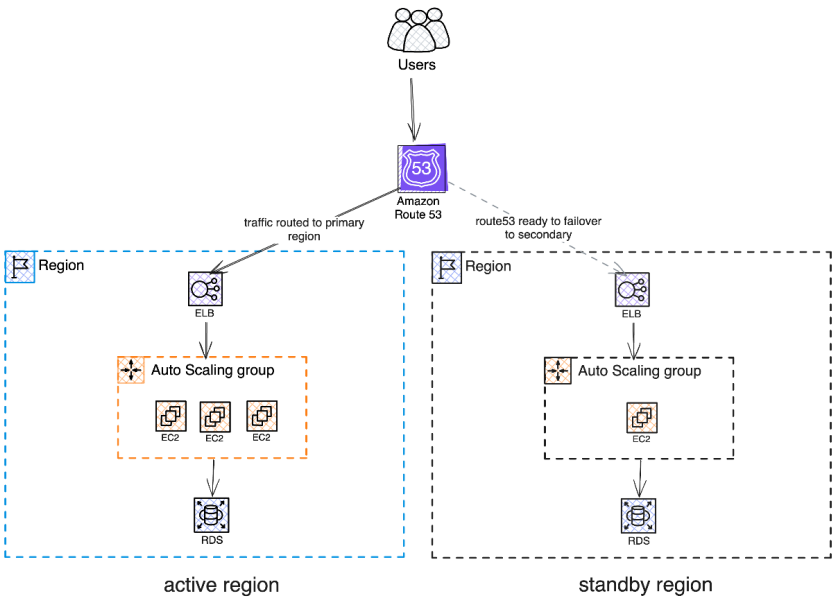

Figure 10.2 – Example routing scenario schema with R53

AWS **Application Recovery Controller** (**ARC**), which we saw in *Chapter 7* for regional redundancy, can also simplify the failover process across multiple AWS regions. ARC allows you to define recovery plans and orchestrate the failover process, including resource provisioning, data replication, and DNS updates. It can be integrated with other AWS services, such as AWS Lambda, AWS Systems Manager, and AWS CloudFormation, to automate various failover tasks. An example of using ARC for cross-region failover could involve the following steps:

1. Define a recovery plan in ARC, specifying the resources, data replication strategies, and DNS updates required for failover.
2. Configure monitoring and alerting systems to detect failures in the active region.
3. When a failure is detected, trigger the ARC recovery plan, which will orchestrate the failover process, including provisioning resources in the passive region, replicating data, and updating DNS records.

It's important to regularly test and validate the failover mechanisms to ensure they function as expected during an actual failover event. This can be achieved through periodic failover drills or by leveraging AWS services such as **AWS Fault Injection Simulator** to simulate failures and validate the failover process.

Let's now see how some AWS services can help facilitate failover scenarios.

Simplified failover with serverless

AWS serverless services, such as AWS Lambda, Amazon API Gateway, and Amazon DynamoDB, can simplify the process of building resilient architectures across multiple regions. These services are inherently distributed and can be easily replicated across regions, providing built-in regional failover capabilities. Additionally, the pay-per-use pricing model of serverless services can help optimize costs when running resources in multiple regions.

For example, with AWS Lambda, you can deploy your functions in multiple regions and configure AWS API Gateway to route traffic to the appropriate region based on predefined rules or health checks. If one region experiences an outage, traffic can be automatically routed to a different region, ensuring continuous availability of your serverless applications.

The example we saw in *Figure 10.1* with EC2 instances becomes even easier when using serverless (Fargate, serverless containers) compute as the capacity doesn't need to be managed. In *Figure 10.3*, we illustrate how the API gateway handles API traffic, routing requests to Lambda functions, and using DynamoDB as the database. DynamoDB allows you to enable **point-in-time recovery** (**PITR**) for your tables, which provides continuous backups and allows you to restore to a new table at any point in time. By exporting the backup to S3 and replicating it to another region, you can create the table in the new region with data, allowing you to resume activities in the passive region when a disaster occurs.

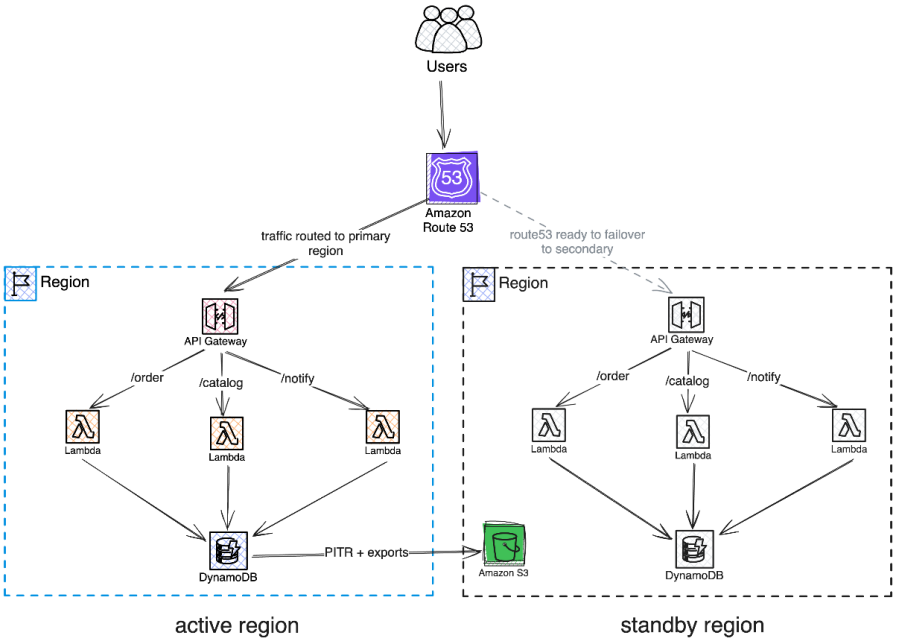

Figure 10.3 – Failover with serverless-based architectures

It's important to note that active-passive architectures can lead to longer failover times and possible data inconsistencies during the transition. Data synchronization between the active and passive regions is crucial to ensure that the passive region has the most up-to-date data when it needs to take over. Strategies such as asynchronous replication, database backups, and periodic data snapshots can be employed to maintain data consistency across regions.

Additionally, active-passive architectures may not be suitable for applications or services that require continuous availability or have strict **recovery time objective** (**RTO**) and **recovery point objective** (**RPO**) requirements. In such cases, active-active architectures, which distribute traffic across multiple regions simultaneously, maybe a more appropriate choice, albeit at a higher operational cost and complexity.

Overall, active-passive architectures offer a cost-effective and relatively simple solution for achieving regional resilience and disaster recovery capabilities. They are particularly well suited for non-critical applications or scenarios where occasional downtime during failover is acceptable, and where the primary goal is to protect against regional outages or disasters.

Before diving into other multi-region architectures, let's revisit the distinction between global and regional AWS services and how they can complement multi-region deployments.

Exploring global versus regional services

When designing multi-region architectures on AWS, it's essential to understand the distinction between regional and global AWS services, as they play different roles in enabling resilience and ensuring consistent behavior across regions.

Regional services are deployed and operate within a specific AWS region. Examples of regional services include Amazon EC2, Amazon **Elastic Block Store** (**EBS**), Amazon **Relational Database Service** (**RDS**), and Amazon ELB. These services are designed to be highly available within a region but are isolated from other regions. In a multi-region architecture, regional services must be deployed and managed separately in each region where the application or service needs to operate.

Global services, on the other hand, are designed to operate seamlessly across multiple AWS regions. These services provide consistent experience and functionality regardless of the region from which they are accessed. Examples of global services include Amazon **CloudFront** (**content delivery network** (**CDN**)), **AWS Global Accelerator** (**AGA**), AWS **Identity and Access Management** (**IAM**), **Amazon Route 53** (DNS service), and **AWS Organizations**.

When designing multi-region architectures, it's crucial to understand the interplay between regional and global services. Regional services provide the core infrastructure and compute resources for running applications and services, while global services enable consistent management, security, and operational practices across regions. By leveraging both regional and global services effectively, organizations can build resilient, high-performance, and scalable architectures that span multiple AWS regions.

Multi-region doesn't always mean that you operate all the regions by yourself. With global services, there are easier ways to have a global presence with a single region. Let's see the example of CloudFront.

Using CloudFront

Amazon CloudFront, AWS's global CDN, can play a crucial role in enhancing the resilience and performance of applications, even when operating from a single region. CloudFront's globally distributed edge locations can cache and serve static and dynamic content, offloading traffic from the origin and bringing content closer to end users worldwide. By leveraging CloudFront, organizations can achieve a global presence without the need to deploy and manage infrastructure in multiple regions. The application's origin can reside in a single AWS region, while CloudFront's **edge locations** (points of presence) handle the delivery of content to users around the world. This approach not only improves performance by reducing latency but also increases resilience by distributing traffic across CloudFront's highly available and redundant network.

In the event of an outage or failure in the origin region, CloudFront can continue to serve cached content from its edge locations, providing a level of resilience and ensuring that end users can still access at least a portion of the application or website.

CloudFront supports advanced features such as **Lambda@Edge** and **CloudFront Functions**, which allow the running of lightweight compute functions at the edge, enabling dynamic content generation, personalization, and other compute-intensive tasks closer to the end users. **CloudFront KeyValueStore** provides a global, data store fully operated by AWS that can be used in read-only mode by CloudFront Functions to perform directly at the edge locations, advanced CDN programming, such as the following:

- **Custom-logic redirections**: The ability to route dynamic content (APIs) based on the request attributes or switch traffic from an active region to a passive (which then becomes active) with just a status change in the KeyValueStore.
- **A/B testing or feature flags**: The ability to segment the user base and provide different behaviors and experiments.
- **Authorization**: The ability to control request flows based on a certain value.

Furthermore, CloudFront integrates seamlessly with other AWS services, such as S3, EC2, and ELB, allowing the building of resilient architectures that leverage the strengths of both regional and global services. For example, an application can be deployed in a single region, with CloudFront serving static content and CloudFront Functions handling dynamic content generation, while the core application logic runs on EC2 instances or containers within the same region, as shown in the following figure:

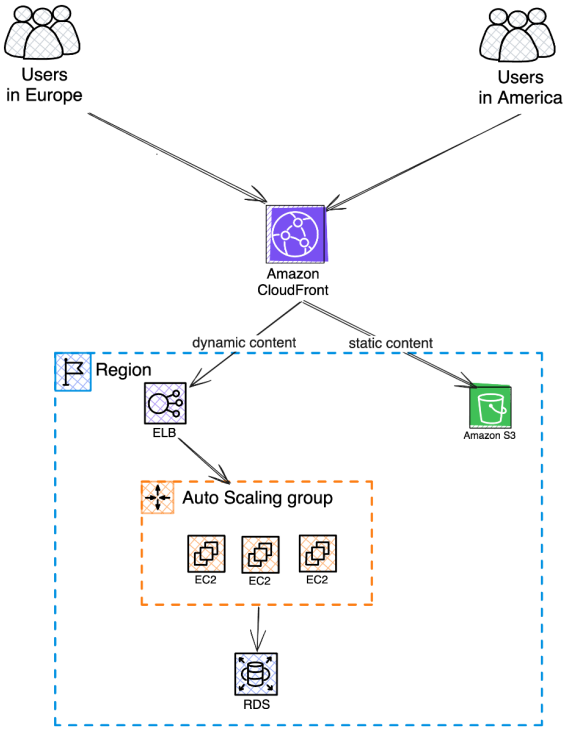

Figure 10.4 – Global deployment, single architecture with CloudFront

While a **single-region architecture** with CloudFront may not provide the same level of resilience as a multi-region, active-active deployment, it offers a balance between resilience, performance, and operational complexity. Organizations can start with a single-region deployment and gradually evolve to a multi-region architecture as their requirements and workloads grow, leveraging the global reach and capabilities of CloudFront throughout their journey.

Another advantage of CloudFront's global presence and massive scale is to absorb **distributed denial of service** (**DDoS**) attacks. Integrating with AWS Shield and Shield Advanced can give even more threat protection safety.

Next, let's cover another service that can help go global even when operating in a single region.

Application performance and availability with AWS Global Accelerator

AGA is a global service that improves the availability and performance of applications by optimizing the path for internet traffic to AWS regions and resources. It leverages AWS's global network infrastructure and edge locations to route traffic through the most optimal path, reducing latency and improving the overall user experience. With AGA, users' traffic quickly enters the AWS network backbone, which, by itself, improves stability (as the AWS network is highly redundant, operated by AWS, isolated from random cable cuts, and multiple network hops) and jitter compared to the public internet. By entering as quickly as possible into the AWS backbone, we can lower the chances of being a victim of internet attacks, such as man-in-the-middle and packet sniffing. It then provides a more stable and consistent experience to end users.

AGA is designed to work both with resources deployed across multiple AWS regions and from a single region. Global Accelerator uses redundant, highly available static IP addresses as entry points to your application. If one entry point experiences an issue, traffic is automatically routed to the next available entry point, ensuring continuous availability. At the time of writing this book, AGA supports routing traffic to Load Balancers (network and application), EC2 instances, and Elastic IPs through **Transmission Control Protocol** (**TCP**) or **User Datagram Protocol** (**UDP**) connections. AGA accelerates networking speed with faster handshakes, leveraging the AWS backbone, jumbo frames, TCP buffers, and larger TCP windows, leading to up to 60% improvement over the public internet.

AGA can also be configured to automatically failover to a secondary region and minimize downtime. Similar to CloudFront, AGA can scale to absorb malicious attacks and can be integrated with Shield and Shield Advanced for more security events protection.

We will see in the next section more details about route traffic to provide fast failover.

Delving into active-active regional architectures

Active-active architectures take regional resilience to the next level by distributing traffic and workloads across multiple regions simultaneously. In this approach, each region is actively serving requests and processing data, providing a higher level of fault tolerance and continuous availability compared to active-passive architectures. One of the primary advantages of active-active architectures is their ability to maintain service availability even in the face of regional failures or outages. Since traffic is distributed across multiple regions, if one region experiences an issue, the other regions can continue to serve requests without interruption. This level of resilience is particularly valuable for mission-critical applications or services that require minimal downtime and strict RTOs and RPOs.

Active-active architectures also offer performance benefits by leveraging resources across multiple regions. We'll see in load balancing across regions how to make use of active-active architectures to improve performance.

The following figure shows the evolution of *Figure 10.2* in an active-active scenario.

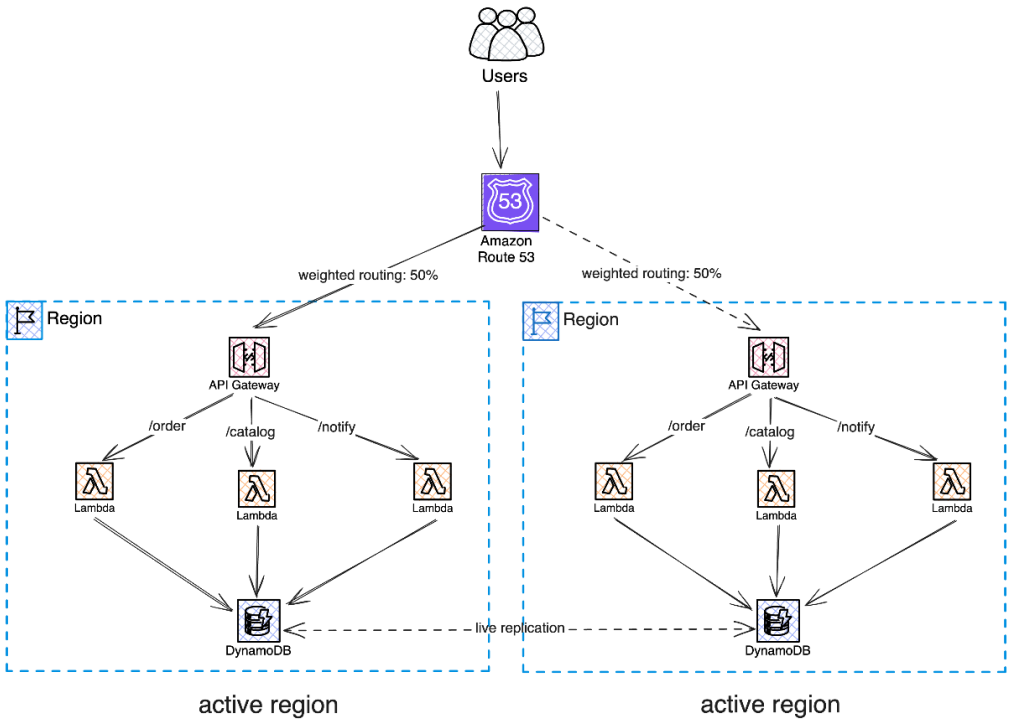

Figure 10.5 – Active-active deployment across regions

In the preceding diagram, we can see that the requests are processed in both locations. There are many factors for deciding how to route traffic or a specific user in a region or another in this scenario. Let's see a few examples in the next section.

Load balancing across regions

AWS provides several services and features that can be leveraged to implement intelligent traffic distribution strategies based on various factors, such as geographic proximity, latency, and application-specific requirements. By distributing workloads across regions, active-active architectures can provide increased scalability and handle higher traffic volumes more effectively. Let's explore the most common patterns.

Using Route 53

Based on DNS, Route 53 allows multi-region traffic to be routed with multiple options, including the following:

- **Geolocation routing**: Route 53 can direct traffic to the nearest AWS region based on the user's geographic location, reducing latency and improving performance.
- **Latency-based routing**: This routes traffic to the region with the lowest latency for the user, providing the best possible user experience.
- **Weighted routing**: This allows for distributing traffic across multiple regions based on predefined weights, enabling controlled traffic splitting and gradual deployments.

Using AWS Global Accelerator

As seen in the *Exploring global versus regional services* section, AGA can greatly help not only within a single region but also can be quite useful for high-performance traffic management with multiple regions. It can be combined with Route 53 and ELB to route traffic to the nearest AWS edge location and then to the appropriate regional endpoint. AGA provides dial knobs to break down traffic to multiple endpoints across different regions, as shown in the following figure:

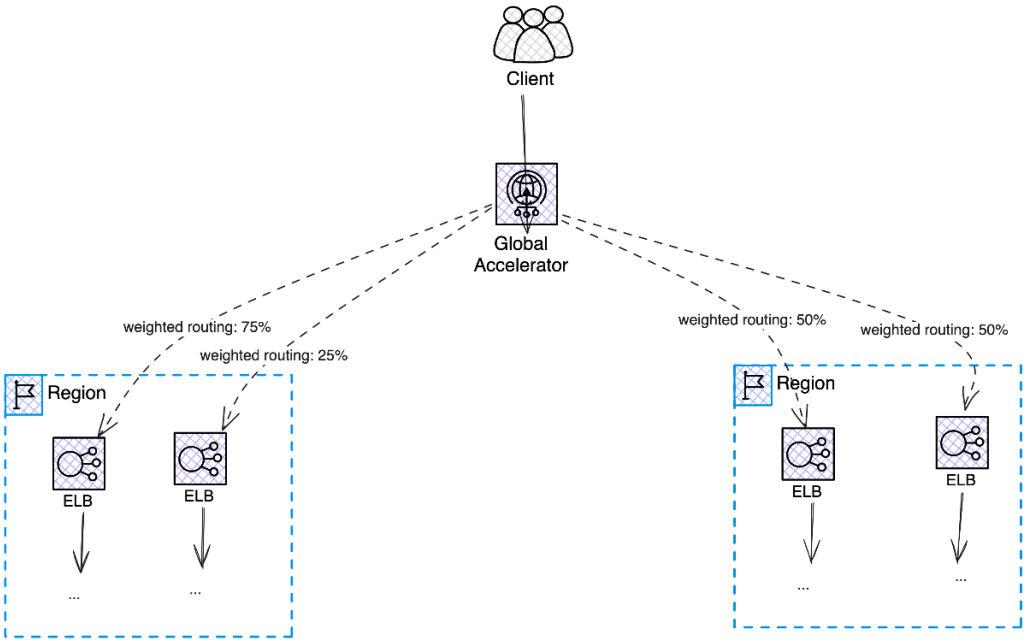

Figure 10.6 – Cross-region weighted traffic with AGA

Unlike Route 53, AGA doesn't use DNS-based failover, which can solve an important problem. DNS has been around for a while and some of its concepts are not always fully respected by its clients. Oversimplifying, DNS will give you a set of IP addresses (or just one) when you issue a request with a name. With the response comes a **time-to-live** (**TTL**) value, which instructs the resolver how long to cache to query before requesting a new one. Java, for example, has a historically bad reputation for caching TTLs for too long; some routers and firewalls suffer from this too.

With AGA using **anycast** IP addresses, when a failover happens, there's no need to reconfigure the client and no need to wait for DNS TTLs to be renewed, which at a very high scale can avoid traffic loss or failures.

Application-specific patterns

Sometimes, applications have custom business logic to route traffic to users, which could be dependent on regulations. For example, a company operating in Europe and America might have a policy to handle European customer data only in Europe regions, regardless of the current geolocation of the user. Applications can implement a thin routing layer to handle these complex rules. We can think about these general patterns:

- **Client-side load balancing**: For applications with a large number of globally distributed clients, client-side load balancing can be implemented to direct traffic to the nearest region based on client-side logic.

- **Service discovery**: In microservices architectures, service discovery mechanisms can be used to dynamically route requests to the appropriate service instances across regions.
- **CDN**: For content distribution and caching, CDNs such as Amazon CloudFront can be leveraged to serve content from the nearest edge location, reducing latency and improving performance with programmatic capabilities, as we saw earlier.

The choice of load balancing and traffic routing strategy depends on the specific requirements of the application, such as the need for low latency, geographic affinity, or failover capabilities. In some cases, a combination of these strategies may be employed to achieve the desired level of resilience, performance, and flexibility.

It's important to note that while these mechanisms enable intelligent traffic distribution across regions, they may introduce additional complexity in terms of configuration, monitoring, and operational overhead. Careful planning and testing are essential to ensure seamless failover, failback, and optimal traffic routing in various failure scenarios.

Another consideration when running workloads over multiple regions simultaneously is how to synchronize data and make multi-region transparent for the final users, which we will cover in the next section.

Data consistency and synchronization

The complexity of data synchronization varies depending on the application's requirements, data models, and the level of consistency needed. As you can see by now, there's no one way to run multi-region deployments. The choice of data synchronization strategy should be driven by the application's specific requirements, such as the level of consistency needed, the volume and velocity of data updates, and the criticality of the data. In some cases, a hybrid approach combining different strategies for different data types or workloads may be necessary to strike the right balance between consistency, performance, and operational complexity.

Here are several scenarios and approaches to consider, which can be categorized into two – centralized versus decentralized.

Using centralized data sharing

The centralized or **hub-and-spoke model** allows one region to act as the central hub, where all data updates and writes are processed. The hub region then replicates data to the other spoke regions, ensuring data consistency across regions. This model simplifies data synchronization and conflict resolution in scenarios where data strong consistency is critical, and the write workload is relatively low compared to the read workload. This is summarized in the following figure, where all the writes to the RDS database happen in the hub region. The spoke regions get a replica of the data for strongly consistent read operations.

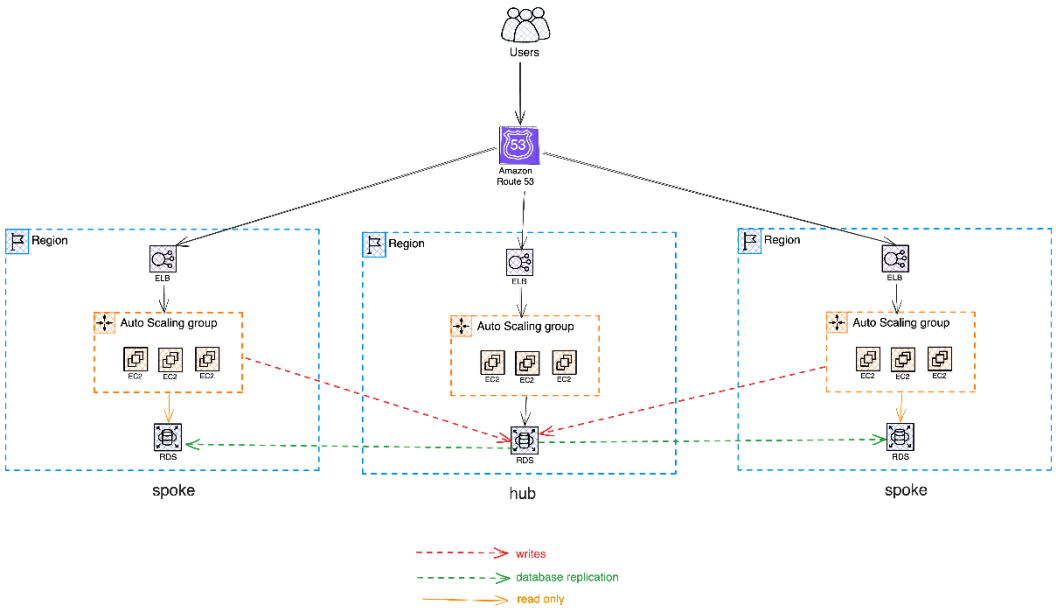

Figure 10.7 – Hub and spoke models

The hub region can become a potential bottleneck or single point of failure, impacting write performance and availability. To mitigate this risk, the hub region can be designed with high availability and failover mechanisms, such as multi-**availability zone** (**AZ**) deployments or cross-region failover capabilities. To go further, services such as Amazon Aurora, with Global Database, provide a centralized, highly available database solution with a primary instance in the hub region and read-only replicas in the spoke regions.

Using decentralized data sharing

In a decentralized model, each region operates as a peer, capable of processing data updates and writes independently. Data changes are then propagated and synchronized across all regions using various replication mechanisms. This approach offers better scalability and fault tolerance, as there is no single point of failure, but it increases the complexity of conflict resolution and data consistency management. However, there are a few nuances to this, such as the following:

- **Stateless or ephemeral data applications**: For applications that do not maintain a persistent state or rely solely on ephemeral storage, data consistency is not a concern. These applications can be deployed across multiple regions without the need for data synchronization mechanisms, as each instance operates independently without shared data.
- **Segmented user base with regional affinity**: In scenarios where transactions or user sessions must be confined to a single region, a segmentation approach can be employed. This involves partitioning the user base or workload based on specific criteria (e.g., geographic location,

customer ID) and routing requests to the appropriate region. Each region maintains its own data store, and transactions are isolated within that region, simplifying data consistency requirements. However, this approach may require a custom routing layer or load balancing logic to ensure consistent regional affinity.

- **Performance-based routing (geo, latency)**: For applications that prioritize low latency and optimal performance, data can be replicated across multiple regions using asynchronous replication mechanisms. Requests are routed to the nearest region based on geographic location or latency, providing low-latency access to data. However, this approach may result in eventual consistency, where data updates are propagated across regions with a slight delay. Conflict resolution mechanisms may be required to handle concurrent updates across regions. AWS offers managed services such as Amazon Aurora Global Database and Amazon DynamoDB global tables that simplify cross-region data replication and synchronization (see *Figure 10.5*). These services handle the complexities of data replication, conflict resolution, and failover scenarios, providing a more straightforward approach to achieving data consistency across multiple regions.

When designing data synchronization strategies for active-active multi-region architectures, it's essential to consider the trade-offs between consistency, availability, performance, and cost. Stronger consistency guarantees often come at the cost of increased latency and potential performance bottlenecks, while eventual consistency approaches prioritize availability and low latency but may require additional mechanisms for conflict resolution and data reconciliation. Additionally, a live active-active deployment will have an impact on the overall AWS bill.

The choice between centralized (hub-and-spoke) and decentralized (peer-to-peer) approaches depends on various factors, such as the application's data consistency requirements, write-to-read ratio, scalability needs, and the criticality of data. In some cases, a hybrid approach combining elements of both models may be necessary to strike the right balance between consistency, availability, and performance.

In the next section, let's look at another deployment for resilient architectures across regions that allows for limiting blast radius and massive scale for large-scale applications.

Introducing cell-based architectures

Cell-based architectures represent an advanced approach to building scalable and resilient systems, whether within a single region or across multiple regions. The core philosophy is to divide the application into smaller, isolated units called cells, each with its own resources and responsibilities. The main motivations are scalability and limiting the blast radius of failures. As applications grow in complexity, monolithic architectures become difficult to scale. By breaking down the application into independent cells, each cell can be scaled based on its resource requirements, enabling efficient resource utilization and controlled scaling. Additionally, isolating components into separate cells contains the impact of failures, preventing cascading issues and improving overall resilience.

What is a cell?

A **cell** is a self-contained, isolated unit or component of an application or system. It is designed to operate *independently*, with its own dedicated resources, such as compute instances, databases, and other services. A cell typically encapsulates a specific set of functionalities or responsibilities within the overall application or system. A cell is typically comprised of multiple microservices.

Cells have a fixed maximum bounded size and should be balanced. Typically, growing capacity with cell-based architectures involves adding more cells. Traffic to cells is routed through a thin layer, which should know where to send a request based on a **sharding** algorithm depending on the use case. Sharding, in this context, is a technique used to distribute and manage data across multiple cells or partitions. It determines how data is divided and assigned to different cells within a distributed system. A high-level architecture of cell-based architectures looks like the following:

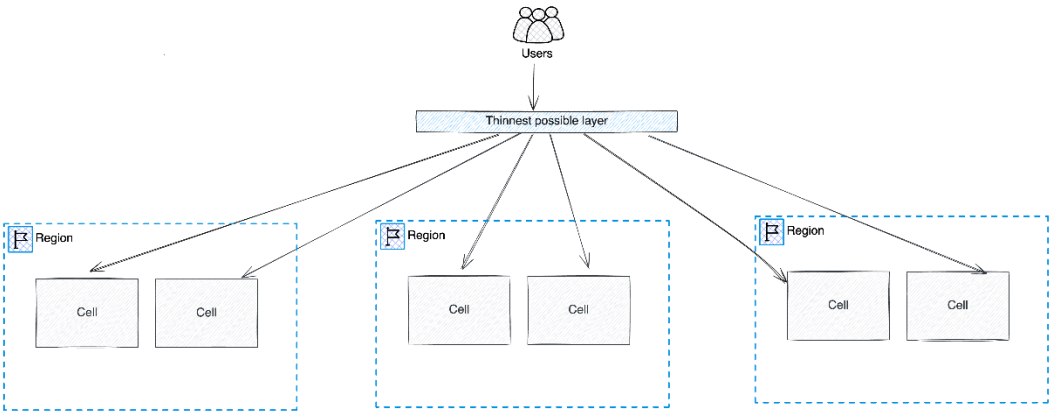

Figure 10.8 – Cell-based architectures

Cells are best suited in scenarios where the following is the case:

- Applications are handling multiple tenants and want to avoid noisy neighbors (larger customers impacting other customers)
- Horizontal and vertical scalability; growing too large and hitting too many limits
- Applications that are ultra-sensitive to failure

Cells typically span across multiple accounts and can be designed for *single* or *multiple* regions.

Advantages of using cells

When implemented properly, here are the benefits we can gain from cells:

- **Blast radius reduction**: Breaking a service up into cells reduces blast radius. When properly isolated from each other, cell failures can be contained to a minimum. It can even be a business

feature where a high-paying customer can have his own cell for example. When bounded in size, they are also faster to recover compared to a very large deployment.

- **Scalability**: Ideally, cells should be equal in infrastructure sizing, which means more horizontal scaling, which is much faster than vertical scaling.
- **Availability**: Cells provide regular availability as any well-architected system. However, a cell failure will have less impact on the overall application.
- **Testability**: Cells are typically bound in size, and this allows for well-understood and testable maximum scale behavior. We can easily run production experiments with chaos engineering by targeting experiments to a few cells and limiting customer impact.
- **Safer deployments**: Cells give multiple stop points to ensure a rollout works as intended before being applied globally.

Considerations when using cells

The following are additional considerations when building cell-based architectures:

- **Increasing complexity**: Adopting a cell-based architecture introduces additional complexity in terms of design, implementation, and operations. It is crucial to ensure that the decision to use cells is driven by valid business reasons and requirements, such as scalability, resilience, or fault isolation, rather than technological curiosity alone. The increased complexity and costs associated with cell-based architectures should be carefully weighed against the potential benefits and justified by the specific needs of the application or system. Additionally, the thin layer routing component responsible for distributing traffic across cells should be designed to be as simple and resilient as possible, avoiding potential single points of failure.
- **Costs**: Duplicating the entire stack to form a cell will be more expensive, but not exponentially. For example, instead of having 100 instances in a region, we could have 10 cells with 10 instances, which is the same thing ultimately. However, some other indirect costs will appear, such as routing layers, observability, and deployment pipelines.
- **Balancing cells**: While cells are designed to have a capped size, it is essential to strive for balanced cells to maximize the benefits of fault isolation and resilience. If one cell becomes disproportionately larger than others, a failure or outage in that cell could impact a significant portion of the user base or workload. Regularly monitoring and rebalancing cells based on resource utilization, traffic patterns, and growth projections can help maintain an even distribution of workloads across cells.
- **Continuous improvement and tuning**: Cell-based architectures require a culture of continuous improvement and tuning to ensure optimal performance, scalability, and resilience. As workloads and traffic patterns evolve, cells may need to be resized, rebalanced, or reconfigured to adapt to changing demands. Regularly reviewing and optimizing cell configurations, resource allocations, and scaling policies can help maintain efficiency and prevent resource wastage or bottlenecks.

- **Deployment pipelines and sub-cell units**: Implementing robust deployment pipelines and strategies is crucial in cell-based architectures to ensure seamless updates and minimize downtime. Considering sub-cell deployment units, where components within a cell can be independently deployed and rolled back, can help isolate failures and minimize the impact on the entire cell. Blue/green or canary deployment strategies can be employed to gradually roll out updates and validate changes before fully deploying them across all cells. It's better to fail fast in one sub-component of a cell than an entire cell or worse, multiple cells at the same time.
- **Observability**: You cannot escape this for any kind of architecture. Implementing centralized logging, metrics collection, and tracing mechanisms can provide visibility into the health and performance of each cell, enabling rapid identification of affected cells and impacted users. It should encompass not only the application components but also the underlying infrastructure and resources within each cell, such as compute instances, databases, and network components.

Overall, cell-based architectures offer a powerful approach to building scalable, resilient, and highly available systems, whether within a single region or across multiple regions. By embracing the principles of isolation, fault containment, and balanced distribution, organizations can achieve greater resilience and operational flexibility while minimizing the impact of failures or outages. Cells come with high operations discipline to be successful and are not meant for all use cases. To read more about how to build cellular architectures, read the AWS whitepaper on cell-based architectures (`https://docs.aws.amazon.com/wellarchitected/latest/reducing-scope-of-impact-with-cell-based-architecture/reducing-scope-of-impact-with-cell-based-architecture.html`).

Summary

In this chapter, we explored various architectural patterns and strategies for achieving regional resilience and high availability across multiple AWS regions. This includes active-passive architectures, the distinction between global and regional AWS services, active-active architectures, and the concept of cells-based architectures.

It's important to remember that a single AWS region is highly performant and capable of handling most workloads with high availability and fault tolerance. An AWS region comprises multiple AZs, which are physically separate data centers, allowing for resilient and redundant deployments within the same region. Before attempting multi-region architectures, it's crucial to leverage the basic concepts and best practices within a single region, such as multi-AZ deployments, auto scaling, load balancing, and data replication strategies. Once these foundational concepts are solidified, organizations can then consider expanding to multi-region architectures based on their specific requirements for disaster recovery, data residency, or global presence.

In the next chapter, we will discuss resilient architectures on AWS and revisit some of these aforementioned concepts through examples.

Part 3: Validating Your Architecture for Resiliency

This part covers different architecture examples and will guide you through different ways to validate the effectiveness of the architecture. AWS has a suite of services that will help users create a plan and test their resilience posture. We will explore these services in this part.

This part has the following chapters:

- *Chapter 11*, *Examples of Resilient Architecture*
- *Chapter 12*, *Observability, Auditing, and Continuous Improvement*
- *Chapter 13*, *Performing Chaos Engineering Testing*
- *Chapter 14*, *Disaster Recovery Planning and Testing*
- *Chapter 15*, *Finalize Building Resilient Architecture Using AWS Resilience Services*

11
Examples of Resilient Architecture

In this chapter, we will explore several resilient architecture examples on the AWS cloud. These examples will demonstrate how to design and implement systems that can withstand failures and maintain their functionality in the face of unexpected events. By the end of this chapter, you will have a deeper understanding of how to create resilient architectures on the AWS cloud and how to use AWS services to enhance the reliability and availability of their applications.

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

- Introducing single-Region architecture
- Multi-Region architecture deployment
- Multi-site architecture deployments
- Designing DDoS/security resilient architecture

Introducing single-Region architecture

Amazon has multiple data centers localized in a Region, with each data center referred to as an **Availability Zone** (**AZ**). These zones are separated from each other. Customers can choose to deploy their infrastructure in any AZ present in a Region (single-Region deployment), based on the location of their users. In this section, we will review the reliability requirements for different components while deploying applications in a single Region. This can be a **single AZ deployment** or deployment across multiple AZs. We will explore some sample architecture for the workloads that have **single-Region deployment requirements**.

Why customers choose single-Region architectures

Customers prefer to deploy applications in a single Region for various reasons, including easier management, reduced complexity, lower costs, and improved performance. A single-Region deployment can also provide better data locality and reduced latency. Additionally, it can be more secure, with fewer configurations to manage; hence, it has a smaller attack surface. Compliance and data sovereignty may also be factors, as some customers may be required to store and process their data within a specific geography or Region.

> **Important note**
> This is only applicable if AWS has only one Region in the geographical location of operation.

Furthermore, deploying in a single Region can be a **cloud migration strategy** (this means starting the testing of your first workload in the cloud, which may be migrating an existing on-prem workload or testing a new workload). It's also worth noting that single-Region deployments can still provide high availability and fault tolerance, but if there are natural disasters, there is a possibility of multiple AZs getting affected simultaneously within a single Region. This is when **multi-Region deployments** can protect your business from downtime.

Different configurations in single-region architecture

A Region has more than one AZs. An AZ is one or more discrete data centers with redundant power, networking, and connectivity in an AWS Region. All AZs in an AWS Region are interconnected with high-bandwidth, low-latency networking over fully redundant, dedicated network connections, providing high throughput and low-latency networking between AZs. AZs are physically separated by a meaningful distance, many kilometers from any other AZ, although all are within 100 km (60 miles) of each other.

AWS provides multiple choice for Regions, and you can decide the Region based on proximity to your user and also consider the cost of deployment. You can choose to deploy your applications in a single AZ or across multiple AZs. Let's review both of these deployments in detail.

Single-AZ deployments

Single-AZ deployments can be quite common, although it is not a recommended reliable architecture. In this sub-section, you will learn what it means to deploy applications in a single AZ and the reasons for doing so. We will explore architecting reliability for different components that will be deployed in a single Region, and we will discuss the limitations/concerns while deploying to a single AZ.

What are single-AZ deployments, and why do customers use them?

A single-AZ deployment in the AWS cloud means that all the resources and instances that make up the application or service are located in the same AZ, and they are not replicated or distributed across multiple AZs.

Customers can choose to deploy a single AZ architecture in the AWS cloud for various reasons, including cost-effectiveness, simplicity, testing and development purposes, small-scale applications, security concerns, compliance requirements, data sovereignty, and cloud migration strategy. A single-AZ architecture can be more cost-effective than deploying a multi-AZ architecture, as it requires fewer resources and less infrastructure, and the absence of a data transfer cost within AZs contributes to a lower cost. Additionally, it can be easier to manage and maintain, with fewer moving parts and less complexity. If an application or service does not experience high traffic, a single-AZ architecture may be sufficient to handle the workload.

For testing and development purposes, a single-AZ architecture can be quickly and easily spun up, without incurring the additional cost of multi-AZ deployments. Small-scale applications or services may not require the redundancy and failover capabilities of a multi-AZ deployment, making a single-AZ architecture a suitable choice. Data residency may also be a factor, as deploying an application or service in the same AZ as the data it needs to access can reduce latency and improve performance.

Reliability configurations in single AZ

The following are different reliability considerations while deploying your workload in a single AZ:

- **AWS Managed Services**: AWS Managed Services, such as Amazon **Elastic Container Service** (**ECS**), Amazon **Elastic Container Service for Kubernetes** (**EKS**), and Amazon **Relational Database Service** (**RDS**), provide built-in redundancy and failover capabilities within a single AZ. For example, ECS and EKS provide automatic instance replacement, which means that if an instance fails or is terminated, a new instance will be automatically launched to replace it. RDS provides automated backups, snapshots, and database instance replicas for redundancy.
- **Compute instances**: When you deploy applications on Amazon EC2 instances or containers on AWS, you will need to make sure that the instances can be replaced on failure. In other cases, you may need a number of instances to be scaled based on the traffic pattern. Amazon **Auto Scaling** can help in replacing an instance of failure, or it can be configured to scale up/down to the number of instances based on average metrics (e.g., the CPU threshold) across the pool of **Auto Scaling groups** (**ASGs**).
- **Networking**: Networking plays a crucial role in ensuring the reliability of applications and services in the AWS cloud. One way to configure networking for reliability in a single AZ is to use redundant network connections. This can be achieved by attaching multiple **Elastic Network Interfaces** (**ENIs**) to each instance and configuring them to use different subnets and routers.

- **Storage**: It is critical to protect data stored on your storage, and you have different storage options to achieve the desired reliability. Amazon **Simple Storage Service** (**S3**) is a scalable and durable object storage service that provides high availability with automatic redundancy. Another option is Amazon **Elastic Block Store** (**EBS**), a block-level storage volume service that provides persistence for block-level storage, with features such as snapshots. Amazon **Elastic File System** (**EFS**) and the Amazon **FSx** family of storage offer another choice, a file-level storage service that provides a highly available and scalable file system with automatic file replication.
- **Database**: You have different options to ensure reliability in a single-AZ architecture. We have different types of database types:
    - **SQL**: One option is Amazon RDS, which provides automated backups, point-in-time recovery, and high availability features. Another option is Amazon Aurora, A fully managed relational database service that is compatible with MySQL and PostgreSQL, providing high availability, durability, and performance. Aurora offers features such as automatic backups, multi-master replication, and crash-consistent backups. For RDS and Amazon Aurora, you can configure an additional read-only database that can be promoted to be read/write databases when the primary write database fails.
    - **NoSQL**: Amazon DynamoDB is another choice, a fully managed NoSQL database service that provides high availability and scalability. DynamoDB automatically scales to handle large workloads and provides features such as automatic backups and point-in-time recovery.
    - **Cache**: Amazon Elastic Cache is also an option, a fully managed, in-memory data store service that provides high availability and performance.

    In addition, monitoring database performance and troubleshooting issues quickly is important, minimizing downtime and ensuring the reliability of your databases and applications.

Now, let's look at how to configure a reliable single-AZ architecture. We will explore a sample single-AZ architecture and how it is designed to protect from failure.

A single-AZ architecture example

The following is a single-AZ architecture diagram.

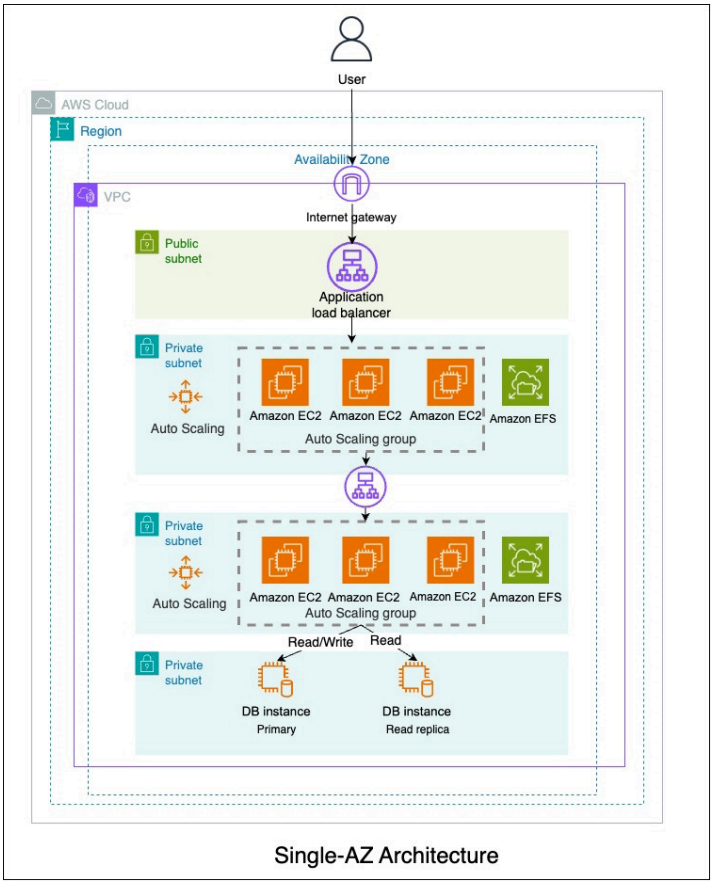

Figure 11.1 – Single-AZ architecture

In the preceding architecture, we have a three-tier **web-application** setup. Users will access the application by reaching the **application load balancer** first, which will forward traffic to a web tier to manage the frontend user behavior. All business logic will be handled by the application tier, and the database will store all transactional data.

On the AWS cloud, an application load balancer is a managed service, and its AWS responsibility is to make sure that uptime `service level agreements` (**SLAs**) are met for the **application load balancer** (**ALB**). Servers in the web and app tiers are configured to auto scale to handle the redundancy and reliability requirements. You can choose which instance family should run your workload, and based on your choice, a launch template can be created that will be used by the ASG to spin up the servers when the need arises. We use Amazon EFS as shared file storage across the web and app tiers.

For database, you have a primary and replica database, and you can promote the replica to be primary if the primary database goes down. With EFS One Zone file systems, your data is stored in a single AZ with redundant copies, and Amazon EFS can handle device failures by detecting and repairing lost redundancy quickly.

The limitations of Single-AZ deployment

Although you have different use cases where single-AZ deployment is used, it has a most significant limitation in that it can be affected by AZ availability issues, such as planned maintenance, unplanned outages, or natural disasters. If an AZ experiences issues, your applications and services may become unavailable. Due to this, the recommendation is to use multi-AZ architecture.

Now that we have reviewed single-AZ architecture deployments, let's now review how you can improve your reliability by deploying across multi-AZs.

Multi-AZ deployments

We have learned about reliability issues with single-AZ deployments, so in this subsection, we will learn what the architectural considerations are for deploying your applications across multiple AZs in a single Region. We will explore a sample multi-AZ architecture that will ensure reliability if an AZ fails.

What does multi-AZ mean, and why do we need it?

Architecting for multiple AZs in the AWS cloud means designing and deploying your applications and services across multiple AZs to ensure high availability, redundancy, and resilience. You need to configure your applications to span multiple AZs to mitigate a single-AZ failure causing application downtime.

Reliability considerations in multi-AZ architecture

The following are reliability considerations while deploying across multiple AZs:

- **Compute**: In multi-AZ, you will follow the best practices shared in single-AZ architecture and, in addition, deploy compute instances across multiple AZs. This will ensure that you have instances available for your applications in different AZ when one of the AZ goes down.
- **Storage**: When you have data shared across multiple AZs, you will choose either EFS or the Amazon FSx storage family to ensure that data is replicated and accessible in multiple AZs. For other storage needs, you can continue to follow the process for single-AZ deployment.
- **Networking**: For networking reliability, you will need to ensure that each tier is set up with a VPC containing multiple subnets in different AZs. Application connections within the same subnet should be preferred to avoid unnecessary network charges.
- **Database**: For multi-AZ setups, you will use the best practices shared in single-AZ setups. For Amazon RDS, It is recommended to set up AWS-managed multi-AZs in RDS, which will have two writers in synchronous mode, and AWS will take care of failing over the writer instance if

the primary writer goes down. In Amazon Aurora, you can set up a read replica, as mentioned when we discussed single-AZ architecture, and AWS will take care of switching to the read replica. In the case of Amazon RDS, if you only have a read replica as a failover mechanism, you will need to monitor to detect writer failure and promote the read replica as the writer instance.

Let's review how you can design a multi-AZ architecture to improve your reliability. We will go over a sample multi-AZ architecture and how it is configured to protect from failures.

An example of multi-AZ architecture

The following is a sample multi-AZ architecture; here, we have expanded on the three-tier architecture that we discussed previously in the single-AZ architecture section:

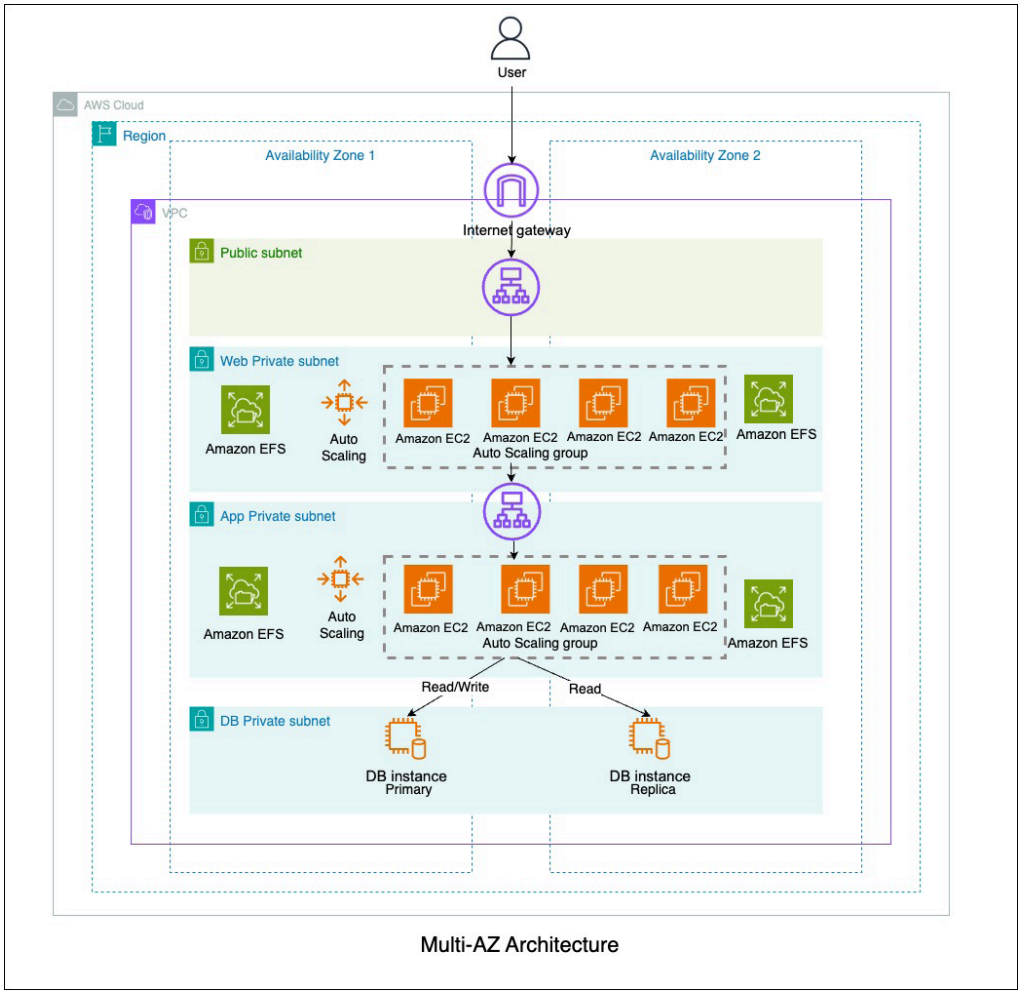

Figure 11.2 – Multi-AZ architecture

In a multi-AZ configuration, you will distribute your web and app servers across multiple AZs, which will ensure that you have servers in an AZ available to handle requests if one of the AZs is impacted. With respect to storage, we use Amazon EFS, which replicates data across multiple AZs and makes it available to all the servers in the web and app tiers. In terms of databases, we will have a primary database in one AZ while the replica will be set up in another AZ. This will allow us to promote the replica to be primary if the primary database goes down or if the AZ where the primary database resides goes down.

The limitations of multi-AZ deployments

Multi-AZ mostly provides the redundancy and availability required for your business applications and compliance needs. However, in some cases, your compliance requirement may need your **disaster recovery** (**DR**) deployments to be far from each other, which is not met by the multiple AZs, and if you are concerned about a catastrophic event impacting the whole Region, you will need to consider a multi-Region strategy.

We have reviewed how to deploy resilient architecture in a single Region using single-AZ or multi-AZ deployments. Now, let's look at deploying the same architecture across multiple Regions and the considerations in doing it.

Multi-Region architecture deployment

In this section, you will learn what it means to set up a workload in a **multi-Region architecture** configuration. We will explore some of the patterns in a multi-Region configuration, followed by a review of sample multi-Region architecture.

When to utilize a multi-Region architecture

AWS customers typically meet resilience goals for workloads within a single Region using multiple AZs or Regional services. However, some customers deploy multi-Region architectures for three key reasons:

- Their mission-critical workloads have exceptionally high availability and continuity requirements that may exceed a single Region's capabilities.
- Data sovereignty laws, regulations, and compliance standards mandate locating workloads in specific jurisdictions.
- Improving performance and user experience requires running workloads near end users across Regions.

You have learned about the need for multi-Region architecture. Now, in the following section, let's look at what the different multi-Region architecture configurations are.

Different multi-Region configurations

In a multi-Region architecture, you will have one of the following configurations:

- **Active-passive read/write traffic**: In this configuration, one Region is actively serving traffic, but you have another Region available to handle traffic if the primary Region goes down. You can configure the switch from one Region to another using DNS-based routing. Depending upon the **Recovery Time Objective** (**RTO**) and **Recovery Point Objective** (**RPO**) requirements of your application, one of the following modes of configuration can be applied:
    - **Backup/restore**: This approach mitigates regional disasters by replicating data across AWS Regions. As well as data, the infrastructure, configuration, and application code must be redeployed in the recovery Region. **Infrastructure as Code** (**IAC**) expedites deployment to enable faster disaster recovery.
    - **Pilot light**: In a pilot light approach, you replicate your data to your secondary Region and have core infrastructure deployed. Only infrastructure required to replicate data is turned on but with minimal infrastructure.
    - **Warm standby**: In this approach, you will have a scaled-down, fully functional copy of the production environment in another Region.
- **Active-active read traffic and active-passive write traffic**: In this configuration, both Regions have all components that are set up with identical infrastructure or configured with the size of infrastructure based on the traffic distribution. The read-only component of the content is managed by your CI/CD pipeline and does not contain any user-generated data. All user-generated data will be considered as write traffic. Due to the latency in replicating data between regions, customers may choose to direct write traffic – a small portion of the overall traffic – to a single region. In the event of a regional failover, the write traffic can then switch to an alternate region..
- **Active-active read/write traffic**: In this case, both regions always accept both read and write traffic. You can choose to split the traffic by a certain ratio. This can be done based on the geolocation of your users by using Route53 geo-based DNS routing.

We have reviewed what are the different multi-Region deployment options, and now we need to look into how the different services need to be configured to improve reliability.

Reliability configurations in a multi-Region setup

In a multi-Region setup, to keep data in both Regions in sync, you will need to use additional services as well as basic infrastructure components:

- **Web/app content**: This content is managed by your content team and can be managed by a CI/CD pipeline by integrating with a content management system.

- **User-generated content**: This is a data replication mechanism, which can be a feature of your storage solution or implemented through services such as DataSync, which will keep your files between Regions in sync. Amazon S3 provides replication across Regions.
- **Compute**: Here, you will have different auto scaling configurations for each region, which will distribute the compute resource across multi-AZs for regional redundancy/reliability.
- **Database tier**: If you manage your own data, you will need to set up a replication strategy between databases that syncs data across the Region, using secure network traffic. You can also choose to use AWS-managed services such as Aurora or DynamoDB that allow you to replicate data across multiple Regions.
- **Storage**: For storage, you have the option to choose AWS services that have an in-built cross-Region file replication feature, such as EFS, or you can leverage S3 object storage with cross-Region replication. Amazon EFS's replicated file system is only available in read-only mode.
- **Networking**: If you have access to the AWS Regions from your corporate or remote office, you need to ensure you have redundant secure paths to reach the Region. Depending on your application requirements, you can leverage AWS Transit Gateway to peer between the Regions. Amazon Route 53 DNS service is critical in routing your traffic to the right Region. You will use a combination of a health check of your Region and low **time to live** (**TTL**) to switch your users to the Region of choice.

Let's look at a multi-architecture example and review how the different components are used in the configuration.

An example of multi-Region architecture

The following is a multi-Region sample architecture that is set up so that your write traffic goes to your primary Region of choice, while the read traffic is distributed across both Regions.

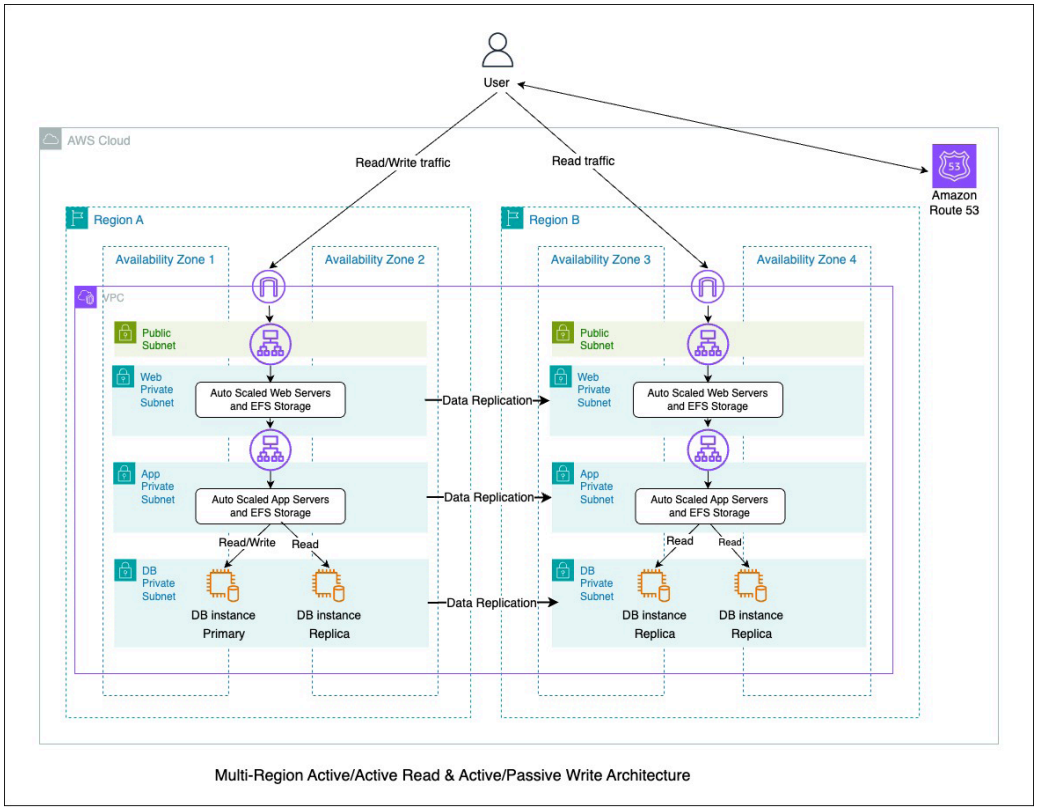

Figure 11.3 – Multi-Region architecture

In the preceding architecture, we have two Regions set up with a three-tier web-application-database architecture. We follow the same setup as the multi-AZ setup, and we have a similar setup in both Regions. The primary Region has a primary database that will handle all the write traffic and replicate data to the replicas in the primary Region, as well as the secondary Region. When the primary Region has an outage, you will promote one of the replicas in the secondary Region as master, adding replicas to sync up with this new master and handle the traffic from both Regions. Similarly, for file storage, we use Amazon EFS and enable an in-built replication feature to replicate files from the primary Region to the secondary Region. If there is a primary Region failure, you can fail over to the replica Region. The Amazon Route 53 DNS service helps in managing the redirection of the traffic.

The limitations of multi-Region architecture

Although it sounds inviting to set up a multi-Region architecture, it comes with the added cost of maintaining the infrastructure, the complexity of the architecture, keeping the systems in sync, and resolving any conflict issues. So, before you plan on setting up a multi-Region architecture, review the business priority and the implications of building and maintaining multi-Region architecture.

We have reviewed architecture in a single Region and across two Regions, but companies may need to deploy across multiple Regions. The following section will look at how to achieve it.

Multi-site architecture deployments

In this section, we will discuss architecture where the application is deployed in more than two Regions. We will review a sample multi-site architecture and how different components can be configured to achieve the required resiliency.

When to utilize multi-site architecture

Multi-site architecture allows customers to deploy their web applications across multiple Regions, providing improved performance and availability for their users. By distributing their application across multiple sites, customers can reduce the latency and buffering that can occur when users are located far from an application's servers. Additionally, multi-site architecture can provide built-in disaster recovery and business continuity, as the application can continue to function even if one site goes down. This can also help to reduce the risk of a single point of failure. Furthermore, multi-site architecture can also help to increase the scalability of the application, as more sites can be added as necessary to handle increased traffic. Customers can also benefit from the ability to serve their users from the location that is closest to them, which can improve the overall user experience. Finally, multi-site architecture can also provide a more robust security posture, as customers can use different security measures at each site to protect their applications.

Reliability configurations in a multi-site configuration

Customers will use a similar strategy for deploying compute in a multi-site configuration, including approaches such as the following::

- **Compute**: You can follow the same process that applied to the multi-Region setup.
- **Networking**: Multi-site networking can follow the same reliability considerations that you would set up for multi-Region, but in this case, you connect more than two Regions. Therefore, you have an option to create a network mesh or hub-spoke model.
- **Storage**: Amazon S3 allows replication across more than one destination Region, so it is the best storage option for a multi-site use case.
- **Database**: Amazon DynamoDB global tables, which comprise multiple replica tables, allow you to have multi-site read-write ability and are best suited to a multi-site data store. An Amazon Aurora global database can be used to span multiple Regions and replicate data in multiple Regions for fast local reads, with low latency.

Now, let's look at an example of multi-site architecture and review the services that are used for the different tiers of a three-tier web-app architecture.

An example of multi-site architecture

The following is a multi-site architecture diagram.

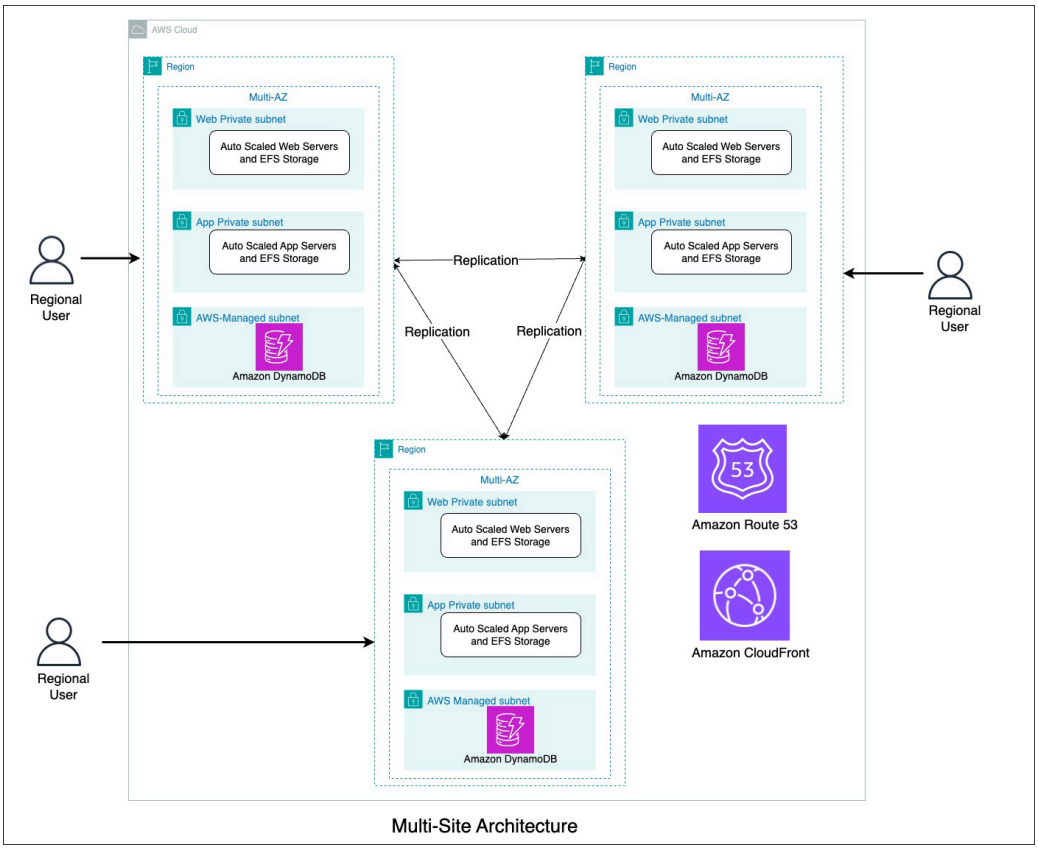

Figure 11.4 – Multi-site architecture

In the preceding architecture, you can see a sample multi-site architecture, which is distributed across three Regions. You can follow the same concept for more than three Regions. We are using the same three-tier use case that we used for multi-Region, but the choice of storage and database are different. For file storage, we use Amazon S3 and serve the content via Amazon CloudFront, which will provide low latency to global users. With S3 multi-Region replication, you can ensure that your data is replicated from one Region to all other Regions. Amazon DynamoDB global tables will ensure that you can have multiple writes across all Regions, which will be replicated in other Regions. This will allow us to make an application available to the users in a Region closer to them. Route 53 DNS service will help to redirect users to the Region closest to them.

The limitations of multi-site architecture

Multi-site architecture helps to keep your environment resilient from any regional failures, as well as providing performance benefits to your global customers. However, this has the highest complexity in terms of designing your architecture, as you will need to keep data in sync and resolve conflicts between updates in different Regions.

We have reviewed the architecture designs that allow you to deploy across single or multiple Regions. Next, let's look at how **Distributed Denial of Service** (**DDoS**)/security can impact the reliability of your environment and how can you architect to protect from it.

Designing DDoS/security resilient architecture

In this section, we will look into how DDoS/security attacks can impact the availability of your applications. We will discuss briefly what DDoS is and what the different types of security attacks are. We will explore some of the mitigation strategies for these attacks and implement them in a sample architecture.

An example of DDoS/security resilient architecture

The following is the AWS-recommended security resilient architecture diagram. It is built around three-tier web applications. Since our focus is on the security aspect of the architecture, we have simplified the web application components.

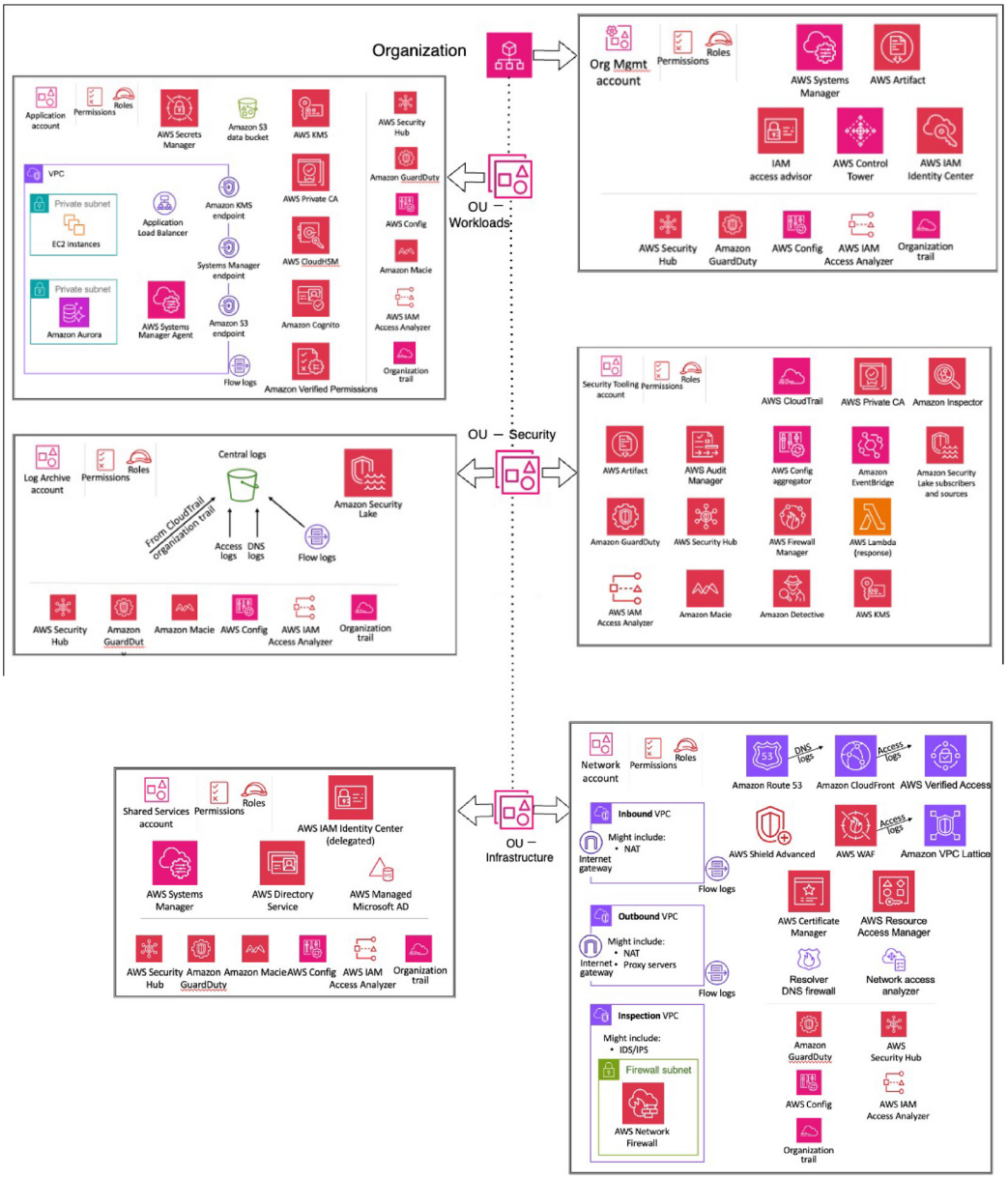

Figure 11.5 – Security reference architecture

The architecture uses AWS organizations to organize and manage your accounts. Accounts are grouped into **Organization Units** (**OUs**), based on their functionalities. We start with the account responsible for managing the organization. We call this the **Org Mgmt account**. This account will have all the security services that are required to manage the security and governance across the organization.

We will keep all the security tooling in the **Security Tooling account**. This will be grouped under **OU – Security**. The Security Tooling account has the specific purpose of running security services, keeping watch over AWS accounts, and enabling automated security alerts and responses. This account acts as the administrator for security services that use an administrative/member structure across AWS accounts.

To ensure that logs are centralized, we will have a **Log Archive account** under OU – Security. The core function of the Log Archive account is to take in and store all security-related logs and backups. By centralizing these logs, you gain the ability to track, audit, and send alerts about critical actions, such as Amazon S3 object access, potentially unauthorized user activity, alterations to IAM policies, and other important changes that involve sensitive data and resources.

For all inbound and outbound network configurations, we will use a dedicated **Network account**, which will be under **OU – Infrastructure**. The Network account oversees the gateway between your application and the public internet. Securing that two-way interface is critical. The Network account separates the networking services, settings, and operations from the specific application workloads, security measures, and other infrastructure components. This isolation protects the network gateway from potential issues arising in those other elements.

All shared services will be grouped under the **Shared Services account**, which will also be under OU – Infrastructure. The purpose of the Shared Services account is to provide services utilized by multiple applications and teams to meet their various goals. Some examples of these shared services are directory services (such as Active Directory), messaging services, and metadata services.

All workloads will ideally be separated into their own account, which will help reduce the blast radius if your workload is impacted by a security incident. We represent one of the workload accounts with the name **Application account**, grouped under **OU – Workload**. The Application account hosts the primary infrastructure and services to run and maintain an enterprise application. You will also run security services that are local to the account, which aggregate the findings and share them with the centralized security account.

The services mentioned in the architecture may not be comprehensive, and you don't need to choose all of them. Based on your security requirements and your risk tolerance, you can design your own architecture.

Organizations bring in standardization and resilience, and we can further protect our workloads by enhancing security with specialized services, such as AWS Shield for DDoS. We will now look at what DDoS is and how to protect from a resiliency impact caused by DDoS.

What is DDoS, and what are security threats?

DDoS and security issues can have a significant impact on the reliability of applications hosted on the AWS cloud. DDoS attacks are designed to flood a network or system with traffic in an attempt to overwhelm it, making it unavailable to legitimate users. This can cause downtime and slow performance, which can negatively impact the user experience and the reputation of an application.

Security issues, such as vulnerabilities in an application or underlying infrastructure, can also impact the reliability of the application. If the application is vulnerable to attack, an attacker may be able to exploit the vulnerability to gain unauthorized access to the application or its underlying resources. This can lead to data breaches, data loss, or other security incidents that can impact the reputation of the application.

What do we mean by DDoS/security resiliency?

In order to implement DDoS/security resilient architecture, you will need to implement security best practices at different stages of your development lifecycle and infrastructure management. This will ensure that your applications are not vulnerable and that application/OS code are updated to fix any vulnerabilities. In addition, you will implement threat mitigation strategies by implementing AWS or third-party security services that will provide an additional level of protection.

Reliability configurations to prevent DDoS/security threats

To mitigate the impact of DDoS and security issues, AWS provides a number of services and tools that can be used to increase the reliability and security of applications hosted on its cloud. For example, AWS Shield is a service that can be used to protect applications from DDoS attacks. It uses automated mitigation techniques to detect and block malicious traffic, and it can also be configured to work with third-party DDoS mitigation services. You should also leverage CloudFront wherever possible, as it provides DDoS protection for your workload in addition to other performance benefits.

To protect applications from security threats, you can use AWS security services such as AWS **WAF** (**Web Application Firewall**). It allows customers to define custom security rules that can be used to block malicious traffic and protect against common web exploits (OWASP: `https://owasp.org/`). You should follow **DevSecOps** practices to secure your applications at each stage of the development lifecycle, and use patch management services similar to AWS Systems Manager Patch Manager to keep your environment updated with security fixes. Don't forget to test the patches first before deploying them in the production environment.

Summary

In this chapter, we looked at some sample architecture designs that can help ensure that your application is resilient within a single AZ. We also explored how to use multi-AZ configurations to provide redundancy against regional outages. Additionally, we considered how to use regional workload distribution to ensure redundancy, as well as how to design multi-site architecture to provide low-latency access to customers globally. We also discussed how DDoS and security threats can impact reliability and the best practices to architect against these threats. In the next chapter, we'll focus on the importance of observability in ensuring reliability and the steps you can take to audit and improve it.

12

Observability, Auditing, and Continuous Improvement

In the last chapter, we learned about different architectural examples for reliability. Now we need to make sure that you have a process to ensure that the environment is working reliably. This can be achieved by verifying you have configured steps to monitor your resources and application. In this chapter, we will learn about designing for observability and the steps to audit it to ensure that the configuration is working as expected. We will also go over the steps to continuously improve your observability process. This chapter is crucial for enhancing reliability, as a comprehensive observability and auditing process is essential to ensure your system operates reliably. Without it, you cannot proactively address reliability issues before they lead to downtime and impact your applications.

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

- Observability is key to resilience
- Designing observability for resilience
- Logging key metrics and events
- Auditing environments for resilience
- Continuous observability improvement

Observability is key to resilience

As we think about the reliability of your services, it means we need to replace a particular service or component of a pool of services that is down in order to bring the system back to normal operation mode. This can only be achieved if we can identify that a failure has occurred. Observability services are crucial for detecting when a service or its components fail. In this section, we'll explore why observability is essential and the consequences of lacking effective observability on system reliability.

Observability is key to resilience for a few key reasons:

- It gives you insight into the health and performance of your applications and infrastructure. By having visibility into metrics, logs, and traces, you can quickly detect when issues occur that could impact resilience, such as application errors or resource saturation.
- It helps you to quickly correlate events and pinpoint the root causes of issues. By providing data from various services and resources, it enables you to link related events rather than viewing individual problems in isolation. This interconnected view accelerates the resolution of outages and performance issues.
- It supports remediation and recovery. By providing insights into where and why failures occurred, observability enables faster remediation. You can also assess the impact of incidents and ensure full recovery across dependencies.
- It informs future improvements. By analyzing observability data over longer periods, you can identify patterns, trends, and weaknesses to proactively improve resilience. This allows you to address systemic issues before they cause major outages down the road.

In summary, having strong observability practices is table stakes for resilience. The visibility and actionable insights in observability allow faster detection, containment, recovery, and learning from issues that can otherwise easily cascade into catastrophic failures if left unobserved. This is why building out robust observability is a foundational best practice.

If you do not set up proper observability, it can severely impact reliability in the following ways:

- **Outages will happen undetected**: Without metrics, logs, and traces actively monitored, outages or performance issues may go unnoticed until customers complain. This increases downtime.
- **Root causes will be hard to determine**: When issues do happen, lack of observability data makes it extremely difficult to pinpoint the source and components impacted and correlate events across distributed services. Due to this, the resolution of issues can take longer, impacting the availability of your application.
- **Reliability can't be measured or managed**: With no visibility, you have no measurable way to set reliability goals, track improvements or failings, or hold teams accountable. Reliability turns more into luck than diligent engineering.
- **Errors can't be categorized or predicted**: Observability data is necessary to track types of errors, failures, exceptions, and so on. Without this aggregate view, you miss out on insights that could prevent future failures.
- **Customer experience can't be quantified**: Lack of end-to-end tracing means there is no way to characterize customers' actual experience even if backend health checks show no problems. This leads to blind spots about reliability from a user perspective.

In essence, running applications and infrastructure on **AWS** without proper observability is a bit like driving very fast on the highway with your eyes closed. Disaster is inevitable unless fundamental visibility and control are established through robust observability practices first.

We have discussed why observability is key and what the impacts of not configuring observability are. If you don't have a thorough observability plan identifying all areas that need to be monitored, it can result in downtime. Here are a few key reasons why configuring proper observability is critical for managing reliability on AWS:

- **Real-time visibility**: Observability gives you real-time visibility into the health, performance, and availability of resources running on AWS. Without observability, you are flying blind, and reliability issues can creep in undetected.
- **Rapid fault identification**: With metrics, logs, and traces all centralized and correlated, observability makes it faster to pinpoint root causes when outages or degradation occur. This speeds up mean-time-to-resolution and recovery.
- **Insights into user experiences**: Observability data shows overall service health from a customer perspective. You can immediately know whether users are being impacted even if backend issues haven't yet triggered alerts.
- **Proactive improvements**: Analyzing trends in observability data allows you to identify recurring issues that might be precursors to larger reliability problems. You can get ahead of outages before they happen.
- **Monitoring distributed systems**: Modern applications often run across containers, microservices, and serverless. Tying all this telemetry data together is essential for reliability management.

In essence, configuring observability equips you with actionable insights to get in front of reliability risks and incidents. Without observability best practices in place, it becomes almost impossible to effectively manage reliability at scale on AWS.

In the next section, we will review strategies to design observability for resilience.

Designing observability for resilience

In the previous section, we learned the importance of observability and logging key metrics to achieve reliability. In this section, we will learn how to design observability for common resources and applications to achieve resilience in your environment.

Steps in designing observability

Designing observability for resilience on the AWS cloud involves several steps and considerations. Here are some best practices to help you achieve this:

- **Define resilience requirements**: Identify the critical services and systems that require high availability and resilience. Determine the maximum acceptable downtime and the **recovery time objectives** (**RTOs**) for each service. This will help you focus your observability efforts on the most critical components.

- **Instrument your applications and services**: Use AWS tools such as AWS X-Ray, AWS CloudWatch, and AWS Lambda to collect metrics, logs, and traces from your applications and services. This will provide you with a comprehensive view of your system's performance, latency, and errors.
- **Monitor AWS services**: Use AWS CloudWatch to monitor AWS services such as EC2, S3, RDS, and Elastic Load Balancer. This will help you detect issues with AWS services that could impact your application's performance and availability.
- **Set up alarms and notifications**: Configure alarms and notifications in AWS CloudWatch to alert your team of potential issues before they become incidents. Use alarm metrics such as CPU utilization, memory usage, and error rates to detect anomalies and potential problems.
- **Implement distributed tracing**: Use AWS X-Ray to track requests and interactions across your microservices and systems. This will help you identify bottlenecks, latency, and faults in your distributed systems.
- **Use machine learning for anomaly detection**: Leverage AWS machine learning services such as CloudWatch Anomaly Detection service (`https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Anomaly_Detection.html`) or Amazon Lookout for Metrics (`https://aws.amazon.com/blogs/machine-learning/connect-to-your-amazon-cloudwatch-data-to-detect-anomalies-and-diagnose-their-root-cause-using-amazon-lookout-for-metrics/`), which specializes in detecting anomalies related to logs and matrices.
- **Implement automated remediation**: Use AWS Automation services such as AWS Lambda, AWS Systems Manager, and AWS Step Functions to automate remediation processes for common issues. This will help you reduce downtime and improve your **mean time to recovery** (**MTTR**). MTTR is a metric that is used to measure the average time it takes to repair a system or piece of equipment after it has failed.
- **Practice continuous monitoring**: Regularly review and analyze your observability data to identify trends, patterns, and potential issues. Use this information to optimize your systems, applications, and processes to improve resilience and performance.
- **Train your team**: Ensure that your team is familiar with the observability tools and processes. Provide regular training and workshops to help them understand how to use the data to identify and resolve issues quickly.
- **Continuous improvement**: Regularly review and update your observability strategy to ensure it remains effective and aligned with your resilience requirements. This will help you stay ahead of the complexity and scale of your AWS environment.

By following these best practices, you can design a robust observability strategy that will help you achieve resilience on the AWS cloud. Remember to continuously monitor and analyze your data to identify potential issues before they impact your users.

Next, we will go over how to set up observability for commonly used resources that were used in our previous chapters.

Observability of common resource

Let's discuss what are the different observability configurations that you will have set up for the architecture discussed in *Chapter 11*. They are as follows:

- **Application Load Balancer** (**ALB**): AWS manages the availability and uptime of ALBs as it is a managed service. However, to effectively monitor the health of the applications running on instances connected to the ALB, you need to configure health checks. Health checks allow the ALB to regularly assess the performance and availability of each instance. Instances that fail these health checks are automatically removed from the pool of servers serving requests, ensuring that only instances running healthy applications handle traffic.
- **EC2 instances**: Basic monitoring of EC2 instances is provided by AWS but you should review whether you need additional monitoring. We have discussed that EC2 instances will be part of the **Auto Scaling group** (**ASG**), but we will need to configure auto scaling to monitor instances for resource utilization (e.g., CPU), which can trigger scale-up/down or replace faulty instances.
- **Route 53**: It should be configured to do a health check of each region, to ensure that the region is available to take traffic, or you can trigger failover. The health check can be in the form of an `http` check of the site URL with the expectation that the site returns an `HTTP 200` response.
- **Databases**: If you set up databases that are AWS-managed services such as RDS or Aurora in multi-AZ mode, AWS will take care of monitoring the database and failover the `db` endpoint to the secondary database. If you only had a write and replica instance, you would need to set up monitoring to identify that the write/read instance is down so you can take action to replace it or failover to another instance.
- **Networking**: AWS-managed networking services will be monitored as part of the service. If you decide to set up third-party software (e.g., Calico on EKS) defined networking, you will need to ensure proper monitoring is set up for failover.
- **Replication**: If you are using multi-region or multi-site and have a process to replicate the environment, you will need to ensure you have proper monitoring to ensure the replication process is redundant and can failover if any of the components in the pipeline are down.

Let's review an example of how to set up monitoring for one of the resources and alert if exceeds a threshold. In this example, we'll use Amazon CloudWatch to monitor an Amazon **Relational Database Service** (**RDS**) instance, which is a managed database service provided by AWS. The following are steps to create the metric:

1. Create an Amazon CloudWatch alarm for high CPU utilization:
    I. Go to the CloudWatch console in the AWS Management Console.
    II. Click on **Alarms** in the left-hand navigation pane, then under the **In alarm** subsection, click **Create alarm**.

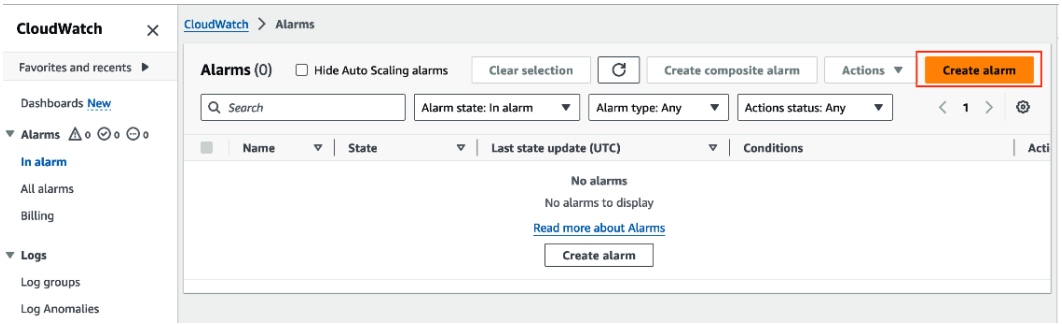

Figure 12.1 – Create Alarm

III. Click on **Select metric** under **Specify metric and conditions**.

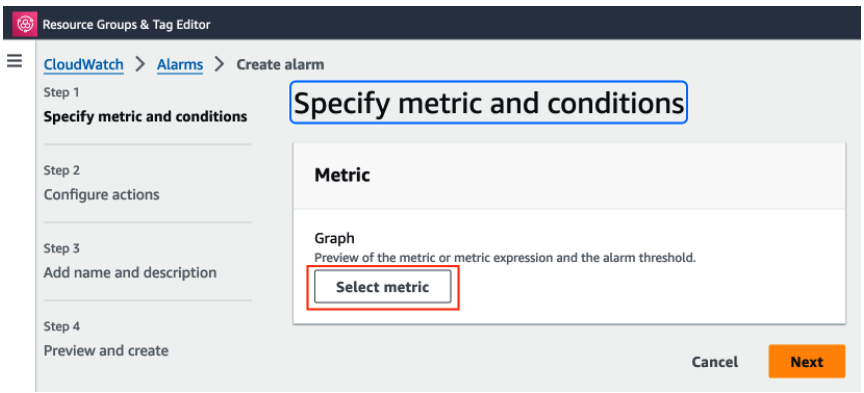

Figure 12.2 – Select metric

IV. Under **Metrics**, select the **RDS** namespace.

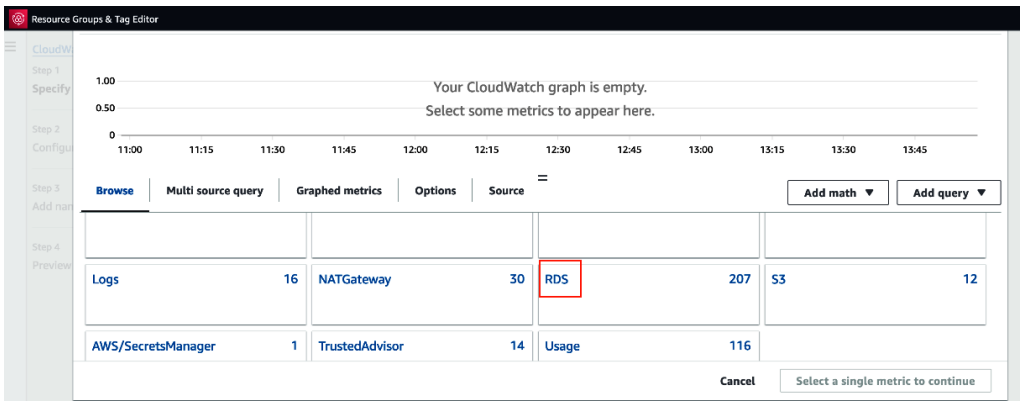

Figure 12.3 – RDS namespace

V. Under the **DBInstanceIdentifier** metric grouping, choose the **CPUUtilization** metric for your RDS instance and click on **Select metric**.

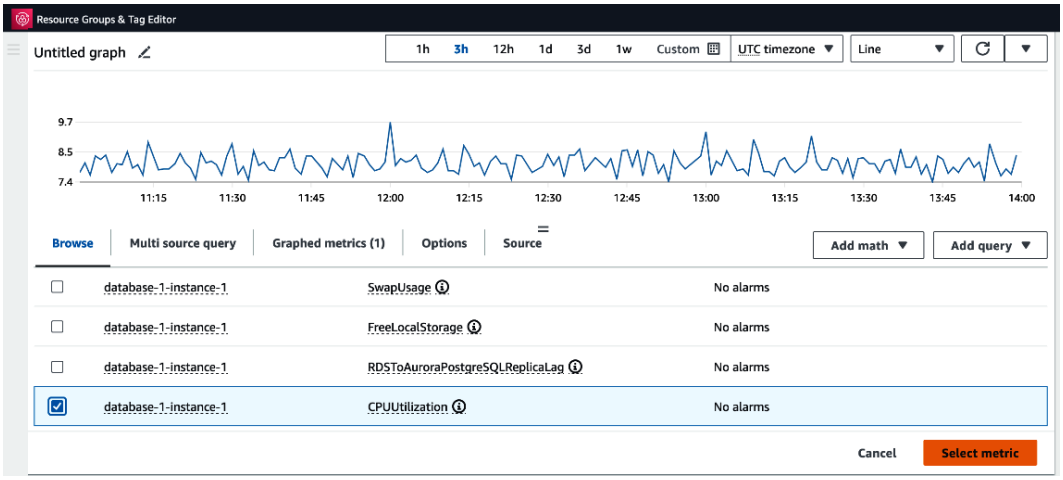

Figure 12.4 – CPU Utilization

VI. Define **Statistic** and **Period** for the metric.

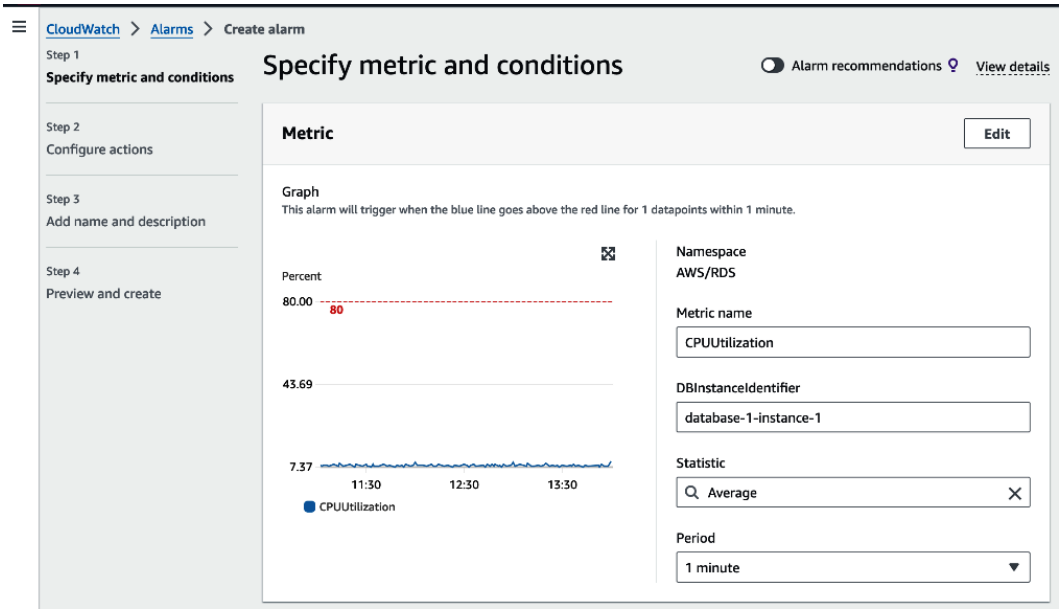

Figure 12.5 – Specifying metrics

VII. Set the **Threshold type** and define the alarm condition based on your requirements. For example, you could set the threshold to 80% CPU utilization.

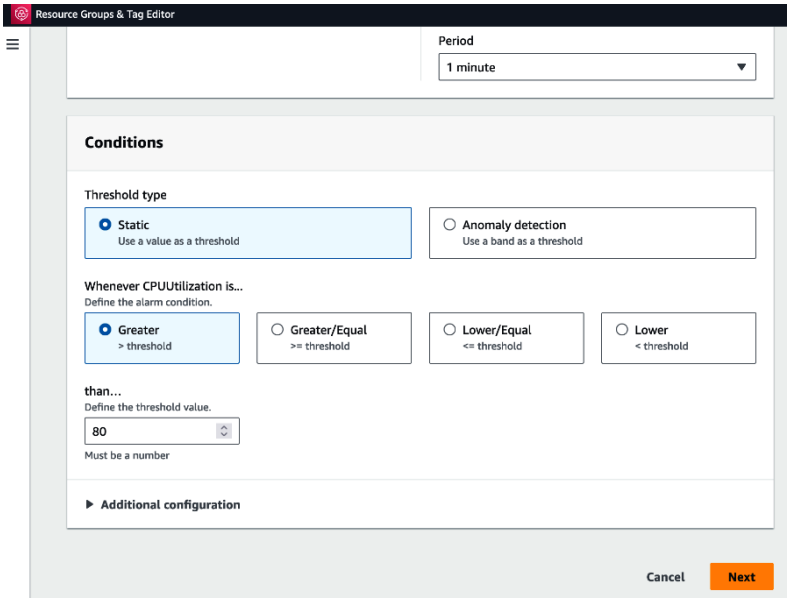

Figure 12.6 – Set threshold

VIII. Configure the alarm to send notifications via email or other channels when the threshold is breached.

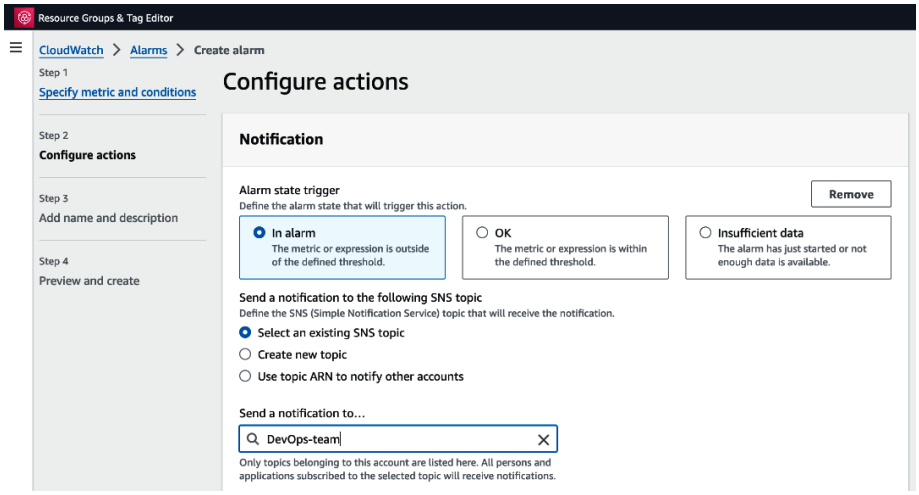

Figure 12.7 – Notification

2. Create an Amazon CloudWatch alarm for high database connection usage:
    I. In the CloudWatch console, click **Create Alarm** again.
    II. Under **Metrics**, select the **RDS** namespace, and choose the **DatabaseConnections** metric for your RDS instance.
    III. Set the threshold for the alarm based on your requirements. For example, you could set the threshold to 80% of the maximum allowed connections.
    IV. Configure the alarm to send notifications when the threshold is breached.
3. Create an Amazon CloudWatch alarm for low free storage space:
    I. In the CloudWatch console, click **Create Alarm** again.
    II. Under **Metrics**, select the **RDS** namespace, and choose the **FreeStorageSpace** metric for your RDS instance.
    III. Set the threshold for the alarm based on your requirements. For example, you could set the threshold to 20% of the allocated storage space remaining.
    IV. Configure the alarm to send notifications when the threshold is breached.
4. Set up Amazon CloudWatch Logs for database logs:
    I. In the AWS Management Console, navigate to the Amazon RDS service.
    II. Select your RDS instance and click on the **Logs & Events** tab.
    III. Enable log exports for the log types you want to monitor, such as errors, slow queries, or general logs.
    IV. Configure CloudWatch Logs to receive and store these logs for monitoring and analysis.
5. Set up Amazon CloudWatch dashboards for monitoring:
    I. In the CloudWatch console, click on **Dashboards** in the left-hand navigation pane.
    II. Create a new dashboard and add widgets to visualize the metrics you want to monitor for your RDS instances, such as CPU utilization, database connections, free storage space, and logs.

By setting up these monitoring configurations in Amazon CloudWatch, you can proactively monitor the health and performance of your database server, which is a critical resource for your application's resilience. CloudWatch alarms will notify you when specified thresholds are breached, allowing you to take appropriate actions to maintain the availability and reliability of your database server.

We have discussed how observability is critical but without having an alerting solution, you won't be able to notify the right personnel or take automated action to rectify the cause impacting reliability. In the following section, we will discuss different ways to configure alerting.

Alerting

We have discussed the importance of monitoring but you will need to also have an alerting component in place. This ensures that all failures are sent to the appropriate operational team so they can take suitable action. You can use AWS services to alert via email or integrate with a third-party paging service.

The following are AWS services that can be used to set up alerting:

- **AWS Simple Notification Service (SNS) topics**: AWS SNS topics allow you to send messages or push notifications to multiple subscribers or clients. You can use SNS topics to send alerts to your team or to integrate with other alerting systems.
- **AWS Simple Queue Service (SQS) queues**: AWS SQS queues allow you to decouple applications and services from each other, enabling you to send messages between them. You can use SQS queues to send alerts to your team or to integrate with other alerting systems.
- **AWS Step Functions**: AWS Step Functions allow you to coordinate the components of distributed applications and microservices. You can use Step Functions to create workflows that trigger alerts based on specific events or thresholds.
- **AWS Config rules**: AWS Config rules allow you to create rules that monitor and enforce configuration compliance across your AWS resources. You can use Config rules to create alerts that trigger based on specific configuration changes or violations.
- **AWS Lambda functions**: You can use AWS Lambda functions to create custom alerting mechanisms that trigger based on specific events or thresholds. For example, you can create a Lambda function that triggers an alert when a specific metric exceeds a certain threshold or when a specific event occurs.
- **Amazon EventBridge**: EventBridge can be used to create alerts that trigger based on specific events or thresholds. EventBridge provides a flexible and scalable way to integrate with other AWS services.

The following steps show an example of how to set up alerting using Amazon SNS for your CloudOps team:

1. Create an SNS Topic:

    I. Go to the Amazon SNS console and under the **Topics** section, click on **Create topic**.

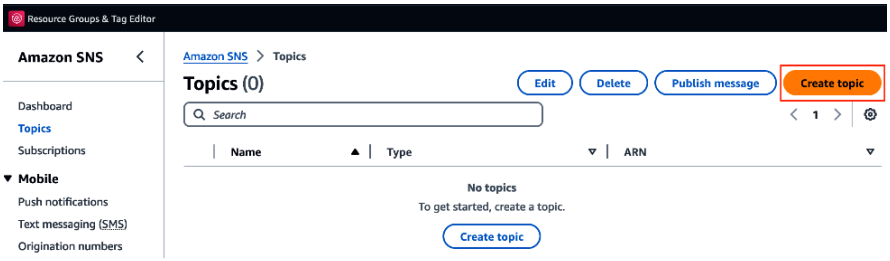

Figure 12.8 – SNS console

II. On the following screen, enter a name for the topic (e.g., `DevOps-team`) and a display name (optional), then click on **Create Topic**.

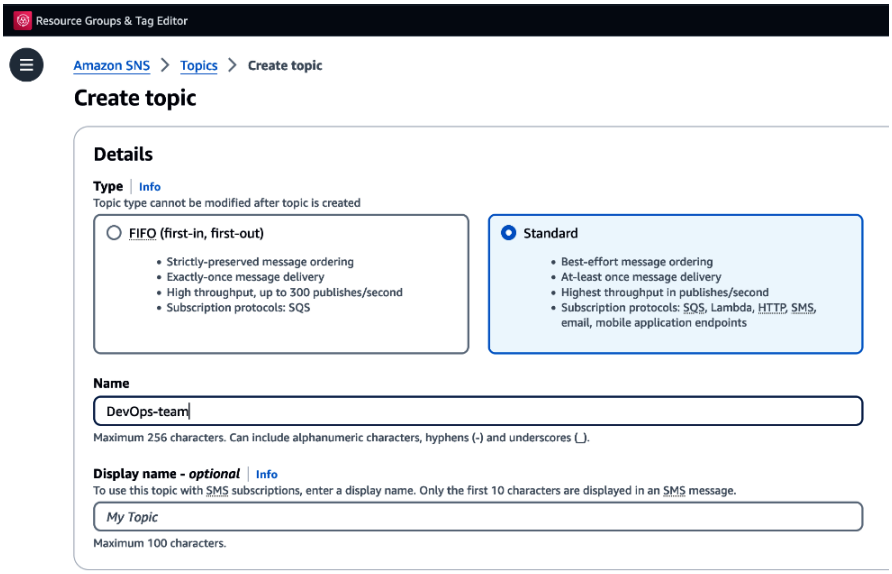

Figure 12.9 – Create an SNS topic

III. On the following screen, you will create a subscription for the **DevOps-team** topic by clicking on **Create subscription**.

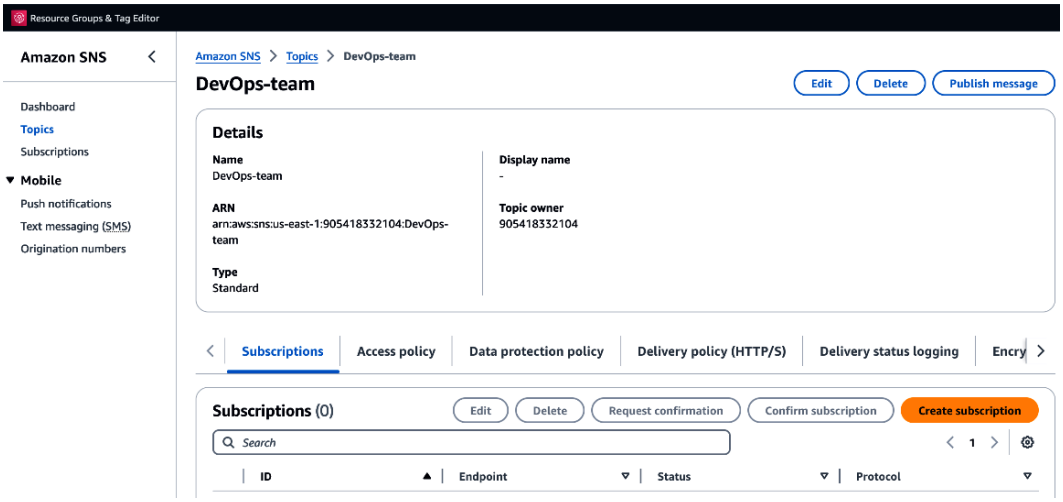

Figure 12.10 – Create Subscription

IV. Provide details for **Create subscription**, as shown in the following figure.

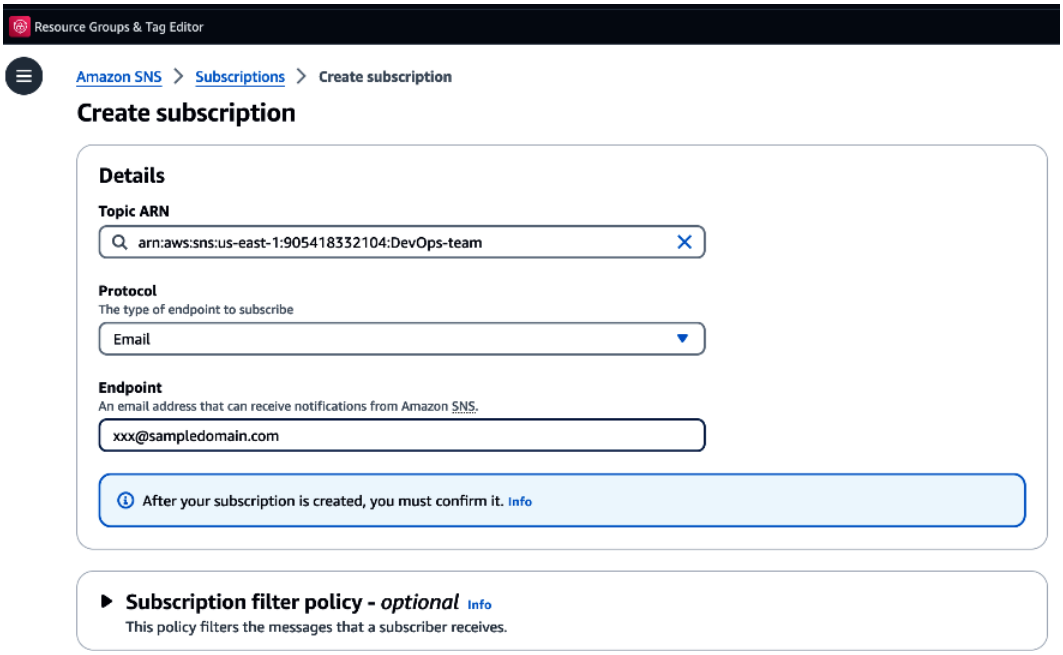

Figure 12.11 – Email Subscription

V. Click on **Create subscription**.

2. Subscribe to the SNS topic:

   I. After creating the topic, on the topic details page, click on **Create subscription**.

   II. Choose the protocol for receiving notifications (e.g., **Email** or **SMS**).

   III. Enter the email address or phone number where you want to receive notifications.

   IV. Click **Create subscription**.

   V. You'll receive a confirmation message or code to confirm the subscription.

3. Test the alarm:

   I. To test the alarm, you can simulate high CPU utilization on the RDS instance that you created before. One way to do this is by running a CPU-intensive query or workload on the database.

   II. Monitor the CloudWatch metrics for your RDS instance's CPU utilization.

III. If the CPU utilization exceeds the configured threshold for the specified evaluation periods, the CloudWatch alarm should be triggered.

IV. You should receive a notification via the SNS topic you subscribed to (e.g., email or SMS).

By setting up an SNS topic and configuring the CloudWatch alarm to send notifications to that topic, you can receive real-time alerts when your RDS instance experiences high CPU utilization or any other metric breaches the defined thresholds. SNS supports various notification channels, such as email, SMS, AWS Lambda functions, and HTTP/HTTPS endpoints, allowing you to integrate alerts with your preferred communication methods or automated response systems.

In the following section, we will go through the services available on AWS, which can be used to implement observability.

AWS observability tooling

Observability tooling on the AWS cloud refers to the set of services and tools that AWS provides to help customers monitor, troubleshoot, and optimize their applications and infrastructure. These tools provide visibility into the performance, health, and security of AWS resources, allowing customers to quickly identify and resolve issues, optimize performance, and improve the overall customer experience.

The following is an example of observability on AWS cloud using AWS observability services:

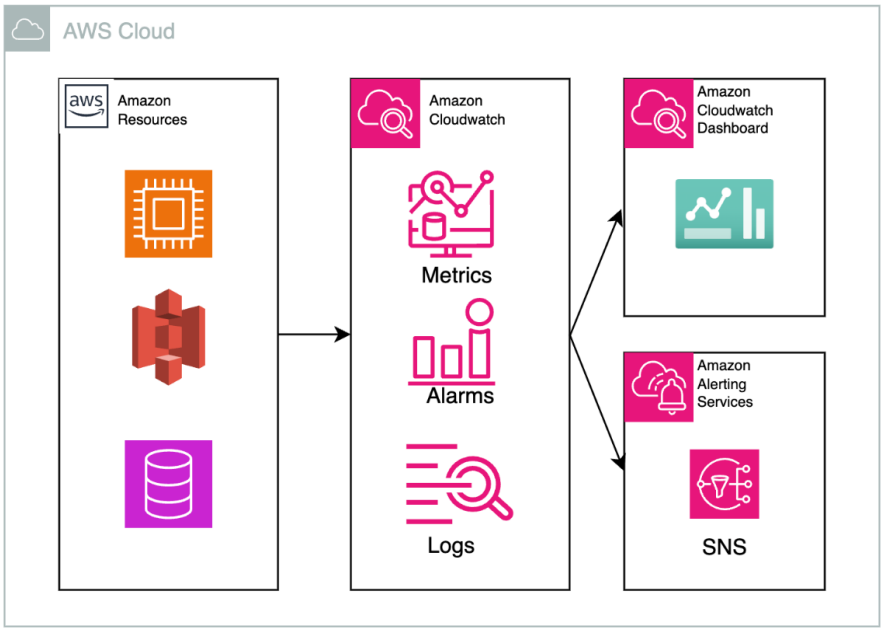

Figure 12.12 – AWS observability architecture

Here is a list of observability tooling on the AWS cloud that is available for you:

- **Monitoring and logging**: AWS offers a variety of services, such as Amazon CloudWatch, Amazon CloudTrail, and AWS X-ray, which can help you monitor and log your AWS resources
- **Metrics and dashboards**: AWS provides a range of services that can help you collect and analyze metrics (e.g., CPU utilization) and dashboards about your AWS resources
- **CloudTrail**: AWS CloudTrail is a service that helps you monitor and track the API calls made to your AWS resources
- **AWS Config**: AWS Config is a service that helps you track the configuration of your AWS resources
- **AWS Trusted Advisor**: AWS Trusted Advisor is a service that can help you optimize your AWS resources
- **AWS Health**: AWS Health is a service that can help you monitor the health and performance of your AWS resources
- **AWS X-Ray**: AWS X-Ray is a service that can help you analyze and debug your applications running on AWS

In the next section, we will discuss key metrics/events and how to log them to improve observability.

Logging key metrics and events

Monitoring key metrics and events is crucial to ensuring reliability on the AWS cloud. Logging of key metrics and events helps in monitoring your resources and applications to ensure they are operating reliably. In this section, we will go over the steps to log key metrics and events for common resources within the AWS environment.

Here are some of the most important ones to consider:

- **CPU utilization**: Monitor CPU utilization for your EC2 instances, containers, and serverless functions. High CPU utilization can indicate performance issues or potential bottlenecks.
- **Memory usage**: Track memory usage for your EC2 instances, containers, and serverless functions. Low memory availability can cause performance issues, crashes, or slowdowns.
- **Disk utilization**: Monitor disk utilization for your EC2 instances, especially for root volumes and data volumes. Low disk space can cause performance issues, crashes, or data loss.
- **Network traffic**: Monitor network traffic to and from your instances, including traffic to and from databases, load balancers, and other services. High network traffic can indicate performance issues or potential bottlenecks and in some cases is an indication of a compromised system.
- **Error rates**: Track error rates for your APIs, microservices, and other critical systems. High error rates can indicate issues with your application or underlying systems and a need to improve your application code.

- **Latency**: Monitor latency for your APIs, microservices, and other critical systems. High latency can indicate performance issues or potential bottlenecks and a need to improve your application code.
- **Instance startup time**: Monitor instance startup time for your EC2 instances. Slow instance startup times can indicate issues with your instance startup process or underlying host.
- **Database performance**: Monitor database performance metrics such as CPU utilization, memory usage, disk utilization, and query latency. Poor database performance can cause issues with your application's performance and availability.
- **Load balancer performance**: Monitor load balancer performance metrics such as request latency, error rates, and CPU utilization.
- **S3 bucket performance**: Monitor S3 bucket performance metrics such as request latency, error rates, and bucket size.
- **DNS performance**: Monitor DNS performance metrics such as request latency and error rates.
- **Route 53 performance**: Monitor Route 53 performance metrics such as request latency, error rates, and DNS response times.
- **CloudWatch alarms**: Monitor CloudWatch alarms for your AWS resources. CloudWatch alarms are triggered when the threshold is reached for the metrics, such as high CPU utilization, low disk space, or high error rates.
- **SNS notifications**: Monitor SNS notifications for your AWS resources. SNS notifications can indicate issues with your resources, such as instance terminations, database issues, or security alerts.
- **AWS service health**: Monitor AWS service health for your region and availability zone. AWS service health can indicate issues with AWS services, such as EC2, S3, or RDS.
- **Scheduled events**: Monitor scheduled events for your AWS resources, such as instance terminations, database backups, or security patches. Scheduled events can help you prepare for potential downtime or performance issues.
- **Compliance and security**: Monitor compliance and security metrics for your AWS resources. Compliance and security metrics can indicate issues with your resources, such as non-compliant instances, unencrypted data, or security breaches.
- **Custom metrics**: Monitor custom metrics that are specific to your application or business needs. Custom metrics can help you identify issues that are not covered by standard AWS metrics.
- **Application logs**: You should also consider monitoring other metrics that are specific to your application, industry, or business needs.

Establishing monitoring and alerting processes for the key metrics mentioned here will furnish a comprehensive observability system, empowering prompt detection and response to potential

issues. Auditing your configuration is crucial to ensuring the reliability, security, and performance of your application. In the next section, we will discuss what steps you need to take to audit your observability configurations.

Auditing environments for resilience

Once you set up observability for your environment, you will need to audit it to ensure it is effective. In this section, we will outline steps you can take to audit your observability process and environment.

Auditing your AWS environment for resilience involves assessing your resources, configurations, and processes to identify potential weaknesses and areas for improvement. Here are some steps to help you conduct a resilience audit in your AWS environment:

1. **Identify critical resources and systems**: Start by identifying the resources and systems that are critical to your business operations. These may include EC2 instances, RDS databases, S3 buckets, load balancers, and other services.
2. **Review resource configurations**: Check the configurations of your critical resources to ensure they are properly configured for resilience. This includes reviewing parameters such as instance types, security groups, subnets, and storage configurations.
3. **Check for redundant resources**: Verify that you have redundant resources in place to minimize the impact of potential outages. For example, ensure that you have multiple EC2 instances running in different availability zones, or that you have a standby RDS database instance.
4. **Review VPC and subnet configurations**: Ensure that your VPC and subnet configurations are properly set up to minimize the risk of network disruptions. Check that your subnets are properly isolated and that your VPC has the necessary CIDR block to accommodate your resources.
5. **Evaluate load balancing and auto scaling**: Review your load balancing and auto scaling configurations to ensure that they are properly set up to handle changes in traffic and resource utilization. Verify that your load balancers are configured to distribute traffic effectively and that your auto scaling policies can handle changes in capacity.
6. **Assess disaster recovery processes**: Evaluate your disaster recovery processes to ensure that they are properly defined and tested. Verify that you have disaster recovery plans in place for your critical resources and systems, and that you have tested them regularly.
7. **Review CloudWatch alarm configurations**: Check your CloudWatch alarm configurations to ensure that they are properly set up to detect potential issues before they impact your environment. Verify that you have alarms configured for critical metrics such as CPU utilization, memory usage, and disk utilization.
8. **Evaluate IAM policies and permissions**: Review your IAM policies and permissions to ensure that they are properly defined and restricted. Verify that your IAM roles and policies grant only the necessary permissions to your resources and users.

9. **Assess compliance and security**: Evaluate your compliance and security posture to ensure that it meets industry and regulatory requirements. Verify that your resources are properly configured to meet security standards, and that you have proper access controls in place.
10. **Review AWS service limits**: Check the AWS service limits for your account to ensure that you are not approaching any limits that could impact your environment's resilience. Verify that you have sufficient resources available to handle traffic spikes and other demands.
11. **Identify single points of failure**: Identify any single points of failure in your environment that could impact your resilience. This may include resources, services, or processes that are critical to your operations.
12. **Develop a remediation plan**: Based on your findings, develop a remediation plan to address any issues or weaknesses in your environment. This may include adjusting resource configurations, implementing redundant resources, or improving your disaster recovery processes.

Using the steps mentioned here, you can conduct a thorough resilience audit of your AWS environment and identify areas for improvement to ensure that your resources and systems are properly protected against potential disruptions.

Observability is not a one-time process, so you need to have steps in place for continuous improvement. We will cover this in the next section.

Continuous observability improvement

As your environment evolves or since you may find gaps in your observability, you need to have a process to continuously improve your observability. In this section, we will go over the different steps that you can take in order to achieve continuous improvement.

Steps to set up continuous observability

Continuous observability improvement on AWS cloud refers to the ongoing process of monitoring, analyzing, and optimizing your AWS resources and systems to ensure they are operating efficiently, effectively, and reliably. This involves collecting and analyzing data from various sources, identifying trends and patterns, and implementing improvements to enhance performance, availability, and resilience. You incrementally make your systems more observable in order to achieve higher reliability on AWS. Here's a closer look at what this can entail:

1. **Identify and prioritize gaps**: Assess your current metrics, logs, and traces to identify critical gaps that could enhance reliability, such as missing APIs that need instrumentation, additional metrics required for each request, or lacking operational insights. Once identified, prioritize the necessary improvements.
2. **Instrument more services**: Expand instrumentation to any supporting services tied to your critical workflows – queues, caches, databases, and third-party APIs. Observability should span across direct dependencies.

3. **Add more context data**: Enrich your observability data with additional dimensions such as customer IDs, device types, and so on. This context can help uncover specific root causes faster.
4. **Streamline dashboards**: Create centralized dashboards tailored to reliability risks – saturation points, node failures, latency spikes, and so on. Quick access to key metrics can accelerate detection.
5. **Automate anomaly detection**: Implement machine learning algorithms to automatically detect anomalies in key metrics and trigger alerts. This reduces dependence on thresholds.
6. **Trace more requests**: If sampling traces, expand the percentage of requests traced. More complete traces improve understanding of the whole system.
7. **Validate monitoring in lower environments**: Ensure your observability also extends to dev and test environments. This enables catching issues in pre-production.
8. **Continually tune alerting**: Adjust alerting thresholds based on trends seen in different deployment sizes, and avoid false positives.

Improving observability incrementally lets you methodically work toward higher reliability while balancing cost and complexity tradeoffs. It should be an ongoing initiative.

Using third-party observability tools

You may need to consider using third-party tools and services to augment your observability capabilities and gain additional insights into your AWS environment. When selecting an AWS observability tool, there are several key considerations to keep in mind. Here are some of the most important ones:

- **Compatibility**: Ensure that the tool is compatible with your AWS environment, including the services and resources you use. Look for tools that support AWS native integrations, such as AWS CloudWatch, X-Ray, and Lambda.
- **Data sources**: Consider the types of data sources the tool can collect data from. Look for tools that can collect data from a wide range of sources, such as logs, metrics, traces, and synthetic data.
- **Data processing and analytics**: Evaluate the tool's data processing and analytics capabilities. Look for tools that can process large amounts of data quickly and provide detailed insights and visualizations.
- **Alerting and notification**: Assess the tool's alerting and notification capabilities. Look for tools that can alert you to potential issues before they impact your environment and provide detailed information to help you resolve issues quickly.
- **Integration with other tools**: Consider the tool's integration with other tools and services, such as ITSM platforms, chatbots, and incident management systems. Look for tools that can integrate with your existing toolset to provide a seamless experience.
- **Scalability**: Evaluate the tool's scalability. Look for tools that can handle large amounts of data and scale with your growing AWS environment.

- **Ease of use**: Assess the tool's ease of use. Look for tools that provide an intuitive user interface and are easy to set up and configure.
- **Cost**: Consider the tool's cost and pricing model. Look for tools that provide a cost-effective solution that aligns with your budget and provides good value for your investment.
- **Security**: Evaluate the tool's security features. Look for tools that provide robust security features, such as encryption, access controls, and auditing.
- **Support and documentation**: Assess the tool's support and documentation. Look for tools that provide comprehensive documentation, tutorials, and support resources to help you get started quickly and resolve issues efficiently.
- **Customization**: Consider the tool's customization options. Look for tools that provide flexibility and customization options to meet your specific needs and requirements.
- **Community and adoption**: Evaluate the tool's community and adoption. Look for tools that have a strong community of users, contributors, and developers who can provide support, feedback, and new features.

By considering these key factors, you can select an AWS observability tool that meets your needs and helps you achieve your goals for performance, availability, and resilience.

Summary

In this chapter, we went through the importance of setting up observability in order to achieve reliability. You also learned how to design your observability and use logging metrics and events. We then focused on the importance and steps to audit the observability and have a process in place to monitor and continuously improve it. The skills you learned in this chapter will enable you to establish comprehensive observability, which is crucial for reliability management.

In the next chapter, we will review how to test the reliability of your environment using the concept of **chaos engineering**.

Further reading

Here, you can find the links to expand your knowledge about the specific concepts referenced in this chapter. This one focuses on resilience observability – `https://aws.amazon.com/solutions/resilience/detection/`.

13
Performing Chaos Engineering Testing

In the last chapter, we learned the importance of observability and how we should use auditing and continuous improvement to maintain it. **Chaos engineering** testing is a critical component in validating the resilience of your environment. In this chapter, we will go through different stages in implementing chaos engineering, which involves introducing different kinds of faults that test the availability of your workload. You will learn how to validate whether the availability of the application under fault conditions matches the expected behavior and address any gaps to meet your business objectives.

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

- What is chaos engineering?
- Stages in chaos engineering
- Chaos engineering guidelines

What is chaos engineering?

In this section, we will define what is meant by chaos engineering, the benefits of using chaos engineering, and how it differs from normal testing or game day testing.

Chaos engineering is the deliberate introduction of faults into a system to assess its resilience. The goal is to uncover potential failure points and address them before they lead to actual outages or disruptions.

Implementing chaos engineering requires a well-thought-out plan. Without a clear plan, you may end up creating more problems than you solve. When creating your plan, you need to determine what you want to test and how you'll do it. Once you have a plan in place, you can start experimenting.

Software developers should integrate chaos engineering into their workflows. By introducing these events in a controlled, non-production environment, you can observe how your system reacts and identify any potential issues.

After identifying potential failure points, you can begin implementing measures to mitigate them. This could involve enhancing monitoring and logging mechanisms to facilitate early detection of issues as they arise. Additionally, you may consider modifying your system's architecture to fortify its resilience against failures, thereby minimizing the impact of such occurrences.

What are the benefits of chaos engineering?

The idea of purposefully introducing faults into a system may seem counterintuitive, but it serves a crucial purpose. Exposing system weaknesses is necessary to make it more resilient and robust.

Chaos engineering enables companies to avoid outages and other disruptions. By identifying potential failure points and addressing them before they cause problems, organizations can proactively prevent service interruptions.

Furthermore, chaos engineering offers a comprehensive range of benefits that span across the customer experience, business objectives, and technical operations of an organization. The primary advantage of embracing chaos engineering is that it enables companies to build stronger, more reliable, and resilient products and services. By proactively identifying and mitigating potential failure points and weaknesses within their systems, businesses can significantly reduce the risk of costly outages, data loss, or service disruptions. This translates to improved customer satisfaction and trust, as customers can rely on the uninterrupted availability and consistent performance of the products they depend on.

From a business perspective, the increased reliability and resilience achieved through chaos engineering practices directly impact the organization's bottom line. Minimizing downtime and service disruptions leads to reduced revenue loss, avoids penalties or contractual breaches, and maintains a positive brand reputation. Additionally, chaos engineering fosters a culture of proactive risk management and continuous improvement, enabling businesses to stay ahead of emerging threats and adapt quickly to changing market demands or technological advancements.

For technical teams, chaos engineering provides a valuable opportunity to gain deep insights into the behavior and failure modes of their systems under various adverse conditions. This knowledge empowers engineers to design and implement more robust architectures, refine operational practices, and improve incident response capabilities. By embracing chaos engineering, teams can stay ahead of potential issues, proactively address vulnerabilities, and continuously enhance the overall quality and reliability of their systems.

Moreover, chaos engineering promotes collaboration and knowledge-sharing across teams, fostering a culture of shared responsibility for system resilience. It breaks down silos and encourages cross-functional communication, enabling teams to learn from failures and collectively contribute to building more resilient and fault-tolerant solutions. Ultimately, chaos engineering not only helps organizations meet customer expectations but also drives continuous improvement, innovation, and a competitive edge in delivering exceptional products and services.

How does chaos engineering differ from traditional testing?

Chaos engineering and traditional testing differ in their core approaches. While testing aims to verify that a system functions as expected, chaos engineering takes a more proactive stance. Chaos engineering focuses on identifying potential failure points before they can cause problems in a live environment. This approach is in contrast to testing, which is primarily reactive, validating the system's behavior after it has been built.

Chaos engineers deliberately introduce and correct controlled failures to understand the system's resilience. By observing how the system responds to these intentional disruptions, they can identify the parts that are more robust and those that require further attention. In contrast, testing can only confirm that the system operates as intended once it has been completed, without actively seeking out weaknesses.

The key distinction is that chaos engineering is a preventative measure, whereas testing is a verification process. Chaos engineers work to anticipate and mitigate failures before they occur, while testing ensures that the system meets its specified requirements. Together, these complementary approaches help organizations build more resilient and reliable systems.

We have introduced you to chaos engineering and in the following sections, we will learn about different stages of implementing chaos engineering.

Stages in chaos engineering

Chaos engineering has several stages through which it is performed. The following figure shows different stages of chaos engineering that will be covered in detail later.

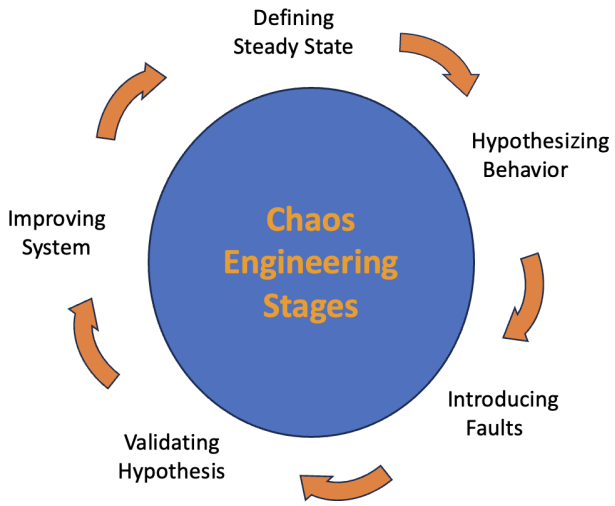

Figure 13.1 – Chaos engineering stages

In this section, we will discuss these stages, starting with defining steady state.

Defining steady state

In this section, you will learn what it means to define **steady state** and about the different examples of baselines you would set to define steady state. In the context of chaos engineering, the concept of *steady state* refers to the normal, expected, and stable operating conditions of a system or application. It represents the baseline or reference state of the system that is considered reliable and functioning as intended.

In chaos engineering, a steady state is the foundation onto which experiments and disruptions are introduced to assess the system's resilience and fault tolerance. The goal is to understand how the system behaves and responds when it is subjected to various types of failures, load spikes, or other unexpected conditions.

The key aspects of steady state in chaos engineering are as follows:

- **Baseline performance**: Steady state represents the normal, expected performance metrics of the system, such as response times, throughput, error rates, and resource utilization.
- **Stability and predictability**: Steady state is a period where the system's behavior is consistent and predictable, allowing for meaningful comparisons and analysis during chaos experiments.
- **Monitoring and observation**: During steady state, the system is closely monitored and observed to establish a baseline understanding of its normal operations and characteristics.
- **Benchmarking**: Steady state serves as a reference point for benchmarking the system's performance, enabling the comparison of its behavior before, during, and after the introduction of chaos experiments.

By understanding steady state, chaos engineers can effectively measure and evaluate the impact of intentional disruptions on the system, identify potential vulnerabilities, and verify the system's ability to withstand and recover from failures. This information is crucial for improving the overall resilience and fault tolerance of the system.

The following are steps to define steady state. It involves creating a stable and predictable environment that maintains consistent performance and availability. Here are the typical steps to define steady state:

1. **Identify the critical components**: This includes identifying the key components of your AWS architecture that are critical to the overall system's performance and availability. This includes EC2 instances, databases, load balancers, applications, and other essential services.
2. **Establish performance baselines**: This involves measuring the performance of your critical components under normal operating conditions. This will include metrics such as CPU utilization, memory usage, network throughput, application response times, application content check, and so on. Establish these baselines to understand the expected behavior of your system.

3. **Implement monitoring and alarms**: This deals with setting up monitoring tools, such as Amazon CloudWatch or third-party tools, to continuously track the performance and health of your critical components. It also includes configuring alarms to notify you when metrics deviate from the established baselines, indicating potential issues.

By following these steps, you can define a steady state for your AWS cloud architecture, ensuring that your system maintains consistent performance, availability, and resilience, even in the face of unexpected events or increased demand.

In the context of a three-tier web application, referenced in *Chapter 11*, you can establish various checks at different stages to define and maintain a steady state. Here are some standard checks that you might set up, although you can add more based on the specific requirements of your web application:

- **Web content check**: This involves verifying the HTTP/HTTPS content of different features within the web application to ensure that it continues to serve the expected content.
- **Web application response time**: Defining the tolerable response time of the application under steady-state conditions is crucial. This helps establish the acceptable performance baseline for the web application.
- **Acceptable error rate**: Determining the acceptable error rate, such as the number of `HTTP 500` errors or other application-specific errors, is important to monitor the overall health of the system.
- **DNS response**: Checking the responsiveness and availability of the **Domain Name System** (**DNS**) is necessary to ensure that users are able to reach the application by correctly resolving the application's domain name.
- **Network connectivity**: Verifying the network connectivity between the different tiers of the application – for example, North-South traffic and East-West traffic, as well as any external dependencies – helps identify potential network-related issues.
- **External access check**: Validating the application's accessibility to external sources, such as the internet, can help detect any connectivity or firewall-related problems.
- **Third-party application checks**: If the web application relies on any third-party services or APIs, it's essential to monitor their availability and responsiveness to ensure that the overall system functions as expected.
- **Database response checks**: Monitoring the database's response times and availability is crucial, as the database tier is a critical component of the three-tier architecture.

By establishing these steady-state checks, you can effectively assess the resilience and reliability of your three-tier web application, enabling you to identify and address potential weaknesses or bottlenecks through chaos engineering experiments.

After you define the steady state of your workload, you will proceed with hypothesizing the behavior of your workload under different conditions. In the next section, we will work on hypothesizing the behavior of a workload and share some examples for the architectures defined in *Chapter 11*.

Hypothesizing behavior

In the context of chaos engineering, *hypothesizing behavior* refers to the process of formulating educated guesses or predictions about how a system or application will respond to specific chaos experiments or disruptions.

The key aspects of *hypothesizing behavior* in chaos engineering are as follows:

- **Defining expectations**: Based on the understanding of the system's architecture, components, and dependencies, chaos engineers formulate hypotheses about how the system is expected to behave when subjected to various failure scenarios or disruptions. These hypotheses help set the expectations for the system's response, which can then be compared to the actual observed behavior during the chaos experiments.
- **Identifying potential impacts**: The hypotheses should consider the potential impacts of the chaos experiments, such as changes in performance metrics, error rates, resource utilization, or the availability of specific system components or services. Chaos engineers aim to anticipate how the system may degrade, fail, or recover during the disruptions.
- **Guiding experiment design**: The hypothesized behavior helps inform the design of chaos experiments, ensuring that the experiments are targeted and meaningful and that the observed results can be effectively analyzed and interpreted. Hypotheses guide the selection of appropriate chaos experiments, the metrics to be monitored, and the expected outcomes that will help validate the system's resilience.

By hypothesizing the behavior of the system, chaos engineers can establish a clear baseline for evaluating the system's response to disruptions, identify potential weaknesses or vulnerabilities, and continuously improve the system's overall resilience through an iterative process of experimentation and learning.

To understand this better, let's look at some sample hypotheses.

- In the event of a single EC2 instance failure, as in the example single-AZ architecture in *Figure 11.1*, in *Chapter 11*, the order system's **Elastic Load Balancer** (**ELB**) health check will detect the failure and redirect requests to the remaining healthy instances. Simultaneously, the EC2 Auto Scaling service will launch a replacement instance, ensuring that the server-side (`5xx`) error rate remains below a 0.01% increase during steady-state operation.
- If an entire **Availability Zone** (**AZ**) experiences a failure, the **Auto Scaling group** (**ASG**) will automatically launch the required number of instances in a different AZ, maintaining an uptime of more than 99% during steady-state operation without any performance degradation.

- In the case of a primary RDS database instance failure, the application workload will seamlessly fail over and establish a connection with the standby RDS database instance. This failover process will result in no more than one minute of database read/write errors during steady-state operation.

This section has provided you with insight into formulating hypotheses about your system's behavior under various fault conditions. In the next section, we will explore the steps involved in intentionally introducing these faults.

Introducing faults

In this section, we will review different fault types that should be introduced and the tools that are available to introduce faults. In chaos engineering, *introducing faults* refers to the deliberate and controlled injection of failures, errors, or disruptions into a system or application to assess its resilience and fault tolerance.

The key aspects of *introducing faults* in chaos engineering are as follows:

- **Fault types**: Chaos engineers identify and intentionally introduce various types of faults, such as network failures, server crashes, database errors, service outages, resource exhaustion, and more. These faults are chosen based on the system's architecture, known vulnerabilities, and the potential impact they may have on the overall system.
- **Fault injection techniques**: Chaos engineers utilize specific fault injection techniques to introduce the desired faults into the system, such as the following:
    - Killing or stopping processes
    - Introducing network latency or packet loss
    - Corrupting or deleting data
    - Exhausting system resources (CPU, memory, storage)
    - Triggering configuration changes or updates
- **Controlled experiments**: Fault injection is performed in a controlled and monitored environment, where the chaos experiments are designed and executed in a way that minimizes the risk of unintended consequences. Chaos engineers ensure that the fault injection is confined to specific components or boundaries and that the overall system can be safely observed and restored.

By introducing faults in a controlled and purposeful manner, chaos engineering aims to uncover hidden vulnerabilities, test the system's ability to withstand and recover from failures, and ultimately improve the overall resilience and reliability of the system.

We have different ways to introduce faults and one such open source tool is Chaos Monkey. For the AWS cloud, you can use **AWS Fault Injection Service** (**AWS FIS**), which makes it easier to introduce faults for AWS components. In the next section, we will review AWS FIS and the different steps involved in using it.

AWS Fault Injection Service

AWS FIS is a managed service that empowers you to conduct fault injection experiments on your AWS-hosted applications. Rooted in the principles of chaos engineering, these experiments deliberately introduce disruptive events to observe how your application responds under stress. By creating controlled disruptions, you can gather valuable insights into your application's behavior and resilience.

With this information, you can then take proactive steps to enhance the performance and reliability of your applications. This ensures they continue to function as expected, even when faced with unexpected challenges or failures. AWS FIS provides a structured approach to testing your application's robustness, helping you build more resilient and dependable systems on AWS

Here are the typical steps involved in using the AWS FIS:

1. **Defining experiment templates**: In AWS FIS, you start by defining experiment templates that describe the types of faults or disruptions you want to inject into your system. These could include things such as shutting down instances, injecting network delays, or simulating resource exhaustion.
2. **Configuring targets**: Next, you select the AWS resources and workloads that you want to target for the fault injection experiments. This allows you to specify the scope and focus of your experiments.
3. **Configuring experiment actions**: Within the experiment template, you configure the specific actions that will be taken to inject the faults or disruptions. This includes details such as the duration, intensity, and triggering conditions for each action.
4. **Running experiments**: Once the experiment template is set up, you can run the fault injection experiment. AWS FIS will then execute the specified actions and disrupt the target resources as defined.
5. **Monitoring experiments**: During the experiment, you monitor the behavior and response of your application or system. This allows you to observe how it handles the injected faults and disruptions.
6. **Analyzing results**: After the experiment is completed, you analyze the results to understand the impacts, identify weaknesses, and determine areas for improvement in your application's resilience and reliability.
7. **Iterating and improving**: Based on the experiment findings, you can make updates to your application architecture, configurations, or operational processes to address any vulnerabilities or issues identified. The insights gained can then inform future iterations of your fault injection experiments.

By following this structured approach, organizations can leverage AWS FIS to proactively test the resilience of their AWS-based applications and systems, ultimately leading to more reliable and fault-tolerant deployments.

In order to better understand what we have discussed so far about FIS, let's look at an example. The following is an example of an AWS FIS template written in JSON format that will stop an EC2 instance with ID `i-0123456789abcdef`:

```
```json
{
  "version": "1.0",
  "title": "Example EC2 Instance Stop",
  "description": "Stops an Amazon EC2 instance to simulate an instance
failure.",
  "tags": ["EC2", "Stop"],
  "targets": {
    "ComputeResources": {
      "ResourceType": "aws:ec2:ec2-instance",
      "ResourceIDs": ["i-0123456789abcdef"],
      "ResourceSelectionMode": "TARGET"
    }
  },
  "actions": {
    "StopInstance": {
      "actionType": "aws:ec2:stop-instances",
      "description": "Stop the EC2 instance",
      "parameters": {
        "Force": true
      },
      "targets": {
        "Resources": ["ComputeResources"]
      },
      "startAfter": ["0 seconds"]
    }
  }
}
```
```

Let's understand this example:

- `version`: This specifies the version of the FIS template.
- `title`: This is a descriptive title for the experiment.
- `description`: This is a brief description of the experiment.

- `tags`: This specifies optional tags to help categorize and filter experiments.
- `targets`: This specifies the AWS resources that will be targeted by the experiment. In this case, it's an EC2 instance with the ID `i-0123456789abcdef`.
- `actions`: This defines the actions to be performed during the experiment. Here, the `StopInstance` action uses the `aws:ec2:stop-instances` action type to stop the specified EC2 instance. The `startAfter` parameter indicates that the action should start immediately after the experiment begins.

This template will stop the specified EC2 instance when the experiment is started, simulating an instance failure scenario. You can customize the template to target different AWS resources and perform various fault injection actions based on your testing requirements.

> **Important note**
>
> This is just an example, and you should always review and test your FIS templates thoroughly before running them in a production environment.

Following are examples of introducing faults for other resources:

- **EBS volume failure**:

```
```json
{
    "description": "EBS Volume Failure Injection",
    "stopConditions": [
        {
            "source": "None"
        }
    ],
    "targets": {
        "aws:ebs:volume": {
            "resourceNames": [
                "YOUR_EBS_VOLUME_ID"
            ]
        }
    },
    "actions": {
        "aws:ebs:volume-failure": {
            "failureMode": "VolumeUnavailable",
            "duration": 900
        }
    },
```
```

```
    "roleName": "FIS-Role"
}
```

Replace `"YOUR_EBS_VOLUME_ID"` with the ID of your EBS volume, and `"FIS-Role"` with the appropriate AWS FIS role in your account.

- **Network latency or packet loss**:

```
```json
{
    "description": "Network Latency and Packet Loss Injection",
    "stopConditions": [
        {
            "source": "None"
        }
    ],
    "targets": {
        "awscloud:ec2:resource-pool": {
            "resourceArns": [
                "YOUR_RESOURCE_ARN_1",
                "YOUR_RESOURCE_ARN_2"
            ]
        }
    },
    "actions": {
        "awscloud:ec2:network-latency": {
            "latencyDuration": 1000,
            "resourceNames": [
                "YOUR_RESOURCE_NAME_1",
                "YOUR_RESOURCE_NAME_2"
            ]
        },
        "awscloud:ec2:packet-loss": {
            "packetLossPercentage": 10,
            "resourceNames": [
                "YOUR_RESOURCE_NAME_1",
                "YOUR_RESOURCE_NAME_2"
            ]
        }
    },
    "roleName": "FIS-Role"
}
```
```

Replace `"YOUR_RESOURCE_ARN_1"`, `"YOUR_RESOURCE_ARN_2"`, `"YOUR_RESOURCE_NAME_1"`, and `"YOUR_RESOURCE_NAME_2"` with the appropriate resource ARNs and names for the resources you want to target, and `"FIS-Role"` with the appropriate AWS FIS role in your account.

- **API Gateway failure**:

```
```json
{
    "description": "API Gateway Failure Injection",
    "stopConditions": [
        {
            "source": "None"
        }
    ],
    "targets": {
        "aws:apigateway:api": {
            "resourceNames": [
                "YOUR_API_GATEWAY_API_ID"
            ]
        }
    },
    "actions": {
        "aws:apigateway:failure": {
            "failureMode": "HttpStatusCodeFailure",
            "httpStatusCode": 500,
            "failurePercentage": 25,
            "duration": 600
        }
    },
    "roleName": "FIS-Role"
}
```
```

Replace `"YOUR_API_GATEWAY_API_ID"` with the ID of your API gateway API, and `"FIS-Role"` with the appropriate AWS FIS role in your account.

- **DynamoDB throttling**:

```
```json
{
    "description": "DynamoDB Throttling Injection",
    "stopConditions": [
        {
            "source": "None"
        }
```
```

```
    ],
    "targets": {
        "aws:dynamodb:table": {
            "resourceNames": [
                "YOUR_DYNAMODB_TABLE_NAME"
            ]
        }
    },
    "actions": {
        "aws:dynamodb:throttling": {
            "throttlingMode": "WriteThrottling",
            "throttlingPercentage": 50,
            "duration": 900
        }
    },
    "roleName": "FIS-Role"
}
```

Replace `"YOUR_DYNAMODB_TABLE_NAME"` with the name of your DynamoDB table, and `"FIS-Role"` with the appropriate AWS FIS role in your account.

- **S3 bucket unavailability**:

```
```json
{
    "description": "S3 Bucket Unavailability Injection",
    "stopConditions": [
        {
            "source": "None"
        }
    ],
    "targets": {
        "aws:s3:bucket": {
            "resourceNames": [
                "YOUR_S3_BUCKET_NAME"
            ]
        }
    },
    "actions": {
        "aws:s3:bucket-unavailability": {
            "duration": 600,
            "failurePercentage": 100
        }
    },
```
```

```
    "roleName": "FIS-Role"
}
```

Replace `"YOUR_S3_BUCKET_NAME"` with the name of your S3 bucket, and `"FIS-Role"` with the appropriate AWS FIS role in your account.

Building upon the examples provided, you now have a foundational understanding of how to construct a template to simulate instance failure scenarios. Leveraging this knowledge, you can explore crafting analogous templates tailored to various other fault injection experiments, allowing you to validate the resilience and fault tolerance of your systems under diverse simulated conditions.

Following are different fault examples that you can inject to test the architecture defined in *Chapter 11*:

- Stop some instances in web/app subnets
- Cause primary database failure
- Network connectivity
- Availability Zone failure
- Database slowness
- Load balancer failure
- Region failure
- Replication failure

The following link will provide you with additional templates that you can use for reference: `https://docs.aws.amazon.com/fis/latest/userguide/experiment-template-example.html`. We have gone through different types of faults that can be injected and learned about mechanisms to introduce faults. In the next section, we will validate the hypothesis that was defined earlier and whether it matches the hypothesis under different fault conditions.

Validating the hypothesis

In this section, we will look into what is meant by *validating the hypothesis*, the steps involved in validating the hypothesis, and why is it important. In the context of chaos engineering, *validating the hypothesis* refers to the process of comparing the observed behavior of the system during chaos experiments to the expected or predicted behavior that was hypothesized earlier.

The process of validating the hypothesis is crucial in chaos engineering because it helps chaos engineers with the following:

- Verifying the effectiveness of the system's resilience mechanisms
- Identifying potential vulnerabilities or weaknesses in the system

- Continuously improving their understanding of the system's behavior under various failure scenarios
- Refining the chaos engineering experiments and approach for future iterations

By validating the hypothesis, chaos engineers can gain confidence in the system's ability to withstand disruptions and make informed decisions to enhance the overall resilience of the system.

The key aspects of validating the hypothesis in chaos engineering are as follows:

1. **Formulating the hypothesis**: We have already covered this earlier in the chapter.
2. **Observing the system's behavior**: During the chaos experiments, the system's behavior is closely monitored and observed, with relevant metrics and data being collected to track the system's performance, error rates, availability, and other key characteristics that represent the system's steady state.
3. **Comparing observed behavior to hypothesis**: The observed behavior of the system during the chaos experiments is then compared to the originally formulated hypothesis. This comparison allows chaos engineers to determine whether the system's actual response aligned with their expectations or if there were any unexpected or divergent behaviors.
4. **Validating the hypothesis**: If the observed behavior matches the hypothesized behavior, the hypothesis is considered validated, confirming that the team's understanding of the system's resilience and fault tolerance was accurate. However, if the observed behavior differs from the hypothesis, it indicates that the team's understanding was incomplete or inaccurate, and the hypothesis needs to be revised or we need to take steps to improve the system by exploring additional aspects of the system's behavior that were not initially considered.

In the next section, we will review the steps that you would take to bring your system to a state that will match your hypothesis.

Improving the system

This is the last step in chaos engineering stages and after this, you can restart the cycle with the first step. In this section, we will review what *improving the system* means and why is it required. After we validate the hypothesis, you may often find that you have gaps in your hypothesis that will result in making changes to your system architecture to fix the gaps. You may also refine and enhance the initial expectations or predictions about how a system will behave when subjected to intentional disruptions or failures.

The following aspects are considered in this stage:

1. **Analyzing discrepancies**: When the observed behavior of the system during chaos experiments does not match the original hypothesis, chaos engineers analyze the discrepancies between the expected and actual outcomes. This analysis helps identify the gaps in the team's understanding of the system's behavior and resilience.

2. **Gathering new insights**: The observed system behavior, as well as the data collected during the chaos experiments, provides new insights that can be used to enhance the original hypothesis. Chaos engineers may uncover previously unknown dependencies, weaknesses, or resilience mechanisms within the system.
3. **Updating the hypothesis**: Based on the new insights gained from the chaos experiments, chaos engineers update the original hypothesis to better reflect the system's actual behavior and characteristics. This may involve modifying the expected outcomes, identifying additional failure scenarios, or considering different system components and their interactions.
4. **Refining the architecture design**: The architecture design will need to be updated to fix the point of failure and meet the expected hypothesis. This can be a multi-step process going through multiple rounds of chaos engineering testing.
5. **Refining experiment design**: The updated hypothesis often leads to a refinement of the chaos engineering experiments themselves. Chaos engineers may adjust the type, intensity, or timing of the introduced faults and disruptions to better align with the improved understanding of the system's behavior.

By improving the hypothesis, chaos engineers can do the following:

- Enhance the relevance and effectiveness of the chaos engineering experiments
- Gain a deeper understanding of the system's strengths, weaknesses, and resilience mechanisms
- Identify and address potential vulnerabilities more effectively
- Ultimately, improve the overall reliability and fault tolerance of the system under study

Chaos engineering guidelines

Chaos engineering guidelines are crucial when conducting chaos engineering tests, as adhering to them ensures a controlled and systematic approach to validating system resilience. Not following these guidelines may lead to unintended consequences, such as causing widespread outages, data corruption, or compromising critical systems. These guidelines help mitigate risks, maintain stability, and provide valuable insights into identifying and addressing potential vulnerabilities within your infrastructure and applications. Let's understand them:

- Before starting any chaos engineering experiments, it's crucial to have a clear understanding of what you're trying to achieve. Define a hypothesis that outlines the expected outcome of the experiment and the potential impact on your system.
- Begin with small-scale experiments that target specific components or subsystems. Gradually increase the complexity and scope of your experiments as you gain confidence and experience.

- Identify the critical components of your AWS infrastructure that are essential to the functioning of your application. These components should be the primary focus of your chaos engineering experiments.
- AWS provides several services and tools that are designed to support chaos engineering, such as AWS FIS and AWS CloudTrail. Leverage these services to ensure that your experiments are executed in a safe and controlled manner.
- Ensure that you have adequate observability and monitoring in place to track the impact of your chaos engineering experiments. This will help you identify any unexpected behavior or failures, and enable you to quickly mitigate any issues.
- Inform relevant teams and stakeholders about your chaos engineering activities, and collaborate with them to ensure that the experiments do not disrupt critical business operations.
- Ensure that your chaos engineering experiments do not put your production environment at risk. Strictly adhere to AWS's best practices and guidelines to maintain the security and availability of your systems. You can choose to maintain a separate replica account or QA account/environment, which will be a replica of production where such chaos engineering experiments can be performed. Once you are happy with these experiments, and if your business permits, you can even do these experiments in a real environment with real traffic/users.
- Develop clear rollback strategies and plan for recovering from any failures or unexpected outcomes during the experiments. This will help you minimize the impact on your production environment.
- Thoroughly document your chaos engineering experiments, including the setup, execution, and outcomes. Use this information to continuously improve your processes and share learnings with your team and the broader community.

Once you have performed tests in the QA environment, you should use **canary deployment** along with chaos engineering when testing in production. Canary deployment is a technique used in software development and deployment to mitigate the risk of introducing new code changes into production environments. It involves gradually rolling out a new version of an application or service to a small subset of users or servers, while the majority still run the previous stable version. By integrating canary deployments with chaos engineering, you can safely test the resilience of your AWS-based applications and identify potential failure points or weaknesses without impacting your entire production environment. This approach helps you build a more robust and reliable system that can withstand unexpected failures or disruptions.

Chaos engineering testing is an iterative process and you will uncover and fix your environment and hypothesis during each iteration. The continuous refinement of the hypothesis is a crucial part of the chaos engineering feedback loop, enabling teams to learn from their experiments and enhance the resilience of the systems they're responsible for.

Summary

Throughout this chapter, you've delved into the concept of chaos engineering and its critical role in testing the resilience of your workloads. You've explored the various stages of chaos engineering, including formulating hypotheses about system behavior, introducing faults that align with the hypothesized fault conditions, and validating the results to implement changes to your environment or refine your hypotheses. Additionally, you've been introduced to AWS Fault Injection Service, a valuable tool for intentionally introducing faults within your AWS cloud environment. Equipping yourself with this tool allows you to gauge the resilience of your existing setup and strengthen the fault-tolerant capabilities of your architecture.

In the next chapter, we will introduce **disaster recovery planning** and testing for different kinds of workloads.

14
Disaster Recovery Planning and Testing

Disaster recovery (**DR**) planning and testing are critical components of an organization's resilience strategy. DR planning involves the development of a comprehensive plan to restore critical business operations and IT systems in the event of a disaster or major disruption. In this chapter, we outline the procedures for an effective **disaster recovery plan** (**DRP**) and steps to incorporate an effective testing strategy to validate your DRP.

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

- Disaster recovery and its significance in cloud computing
- Different disaster recovery strategies in AWS
- Defining and planning your disaster recovery objectives
- Testing disaster recovery plans
- Avoiding disaster recovery pitfalls and misconceptions

Disaster recovery and its significance in cloud computing

DR refers to the process of restoring an organization's critical business operations and IT systems after a disaster or major disruption. In the context of cloud computing, DR is crucial because it ensures that cloud-based applications and data remain available and accessible even in the event of an outage or disaster. A disaster can be anything from a natural disaster, such as a hurricane or earthquake, to a cyber-attack, hardware failure, or human error.

Cloud computing has become an essential part of modern business operations, with many organizations relying on cloud-based applications and services to run their daily operations. However, this reliance on the cloud also introduces new risks and vulnerabilities, such as dependence on internet connectivity, shared infrastructure, and third-party providers. A disaster can have a devastating impact on business operations, leading to loss of revenue, damage to reputation, and even legal liabilities. Therefore, it is

essential for organizations to have a robust DRP in place to ensure business continuity and minimize the impact of a disaster.

In cloud computing, DR is particularly challenging due to the complexity and scale of cloud-based systems. Cloud providers such as AWS offer a range of features and services to support DR, such as **Availability Zones** (**AZs**), **Regions**, and **Edge** locations. However, it is still the responsibility of the organization to design and implement a DRP that meets their specific business needs and requirements. This includes identifying critical applications and data, establishing **recovery point objectives** (**RPOs**) and **recovery time objectives** (**RTOs**), and developing a plan for data replication, backup, and restore. By prioritizing DR, organizations can ensure that their cloud-based operations are resilient, reliable, and always available, even in the face of disaster.

Having emphasized the importance of DR in the cloud, we will now explore the extensive range of DR features and capabilities offered by the AWS cloud. In the following sections, we will delve into the various tools and services that AWS provides to support business continuity, including **backup and restore**, **pilot light**, **warm standby**, and **hot standby**. By examining these features in detail, we will gain a deeper understanding of how AWS can help organizations develop a robust DR strategy that meets their unique needs and requirements.

Overview of AWS's disaster recovery features

AWS provides a robust set of features and services to support DR in the cloud. One key feature is the concept of AZs, which are isolated locations within a Region that are designed to be fault-tolerant and highly available. Each AZ is connected to other AZs in the same Region through low-latency, high-throughput networks, allowing for easy replication and failover of resources. This means that if one AZ becomes unavailable, resources can quickly failover to another AZ in the same Region, minimizing downtime and data loss.

Another key feature is Regions, which are geographic locations around the world where AWS has built clusters of data centers. Each Region is isolated from other Regions, providing an additional layer of redundancy and fault tolerance. This allows organizations to deploy resources in multiple Regions, providing a high level of availability and DR. For example, an organization can deploy a primary application in one Region and a standby application in another Region, allowing for quick failover in the event of a disaster.

Since business-critical data is stored in storage, any impact on the storage will impact the resilience of your application, so it is critical to have a resilience plan for your storage. AWS provides a range of *storage options* that support DR, including **Amazon S3**, **Amazon EBS**, and Amazon **Elastic File System** (**EFS**). These services provide durable, highly available storage that can be used to store backups, snapshots, and other critical data. Under a disaster scenario, when your application is recovered in a different Region, AWS provides features such as **Amazon Route 53**, which provides DNS services that can route traffic to different locations in the event of a disaster, and **Amazon CloudFront**, which provides a **content delivery network** (**CDN**) that can cache and distribute content across multiple locations.

AWS also offers a range of *services* that support DR, including **Amazon CloudWatch**, which provides monitoring and logging capabilities, and **AWS CloudTrail**, which provides auditing and compliance capabilities. These services can be used to detect and respond to disasters and meet compliance and regulatory requirements. Additionally, AWS provides a range of tools and services that support DR, including **AWS CloudFormation**, which provides **infrastructure-as-code** (**IaC**) capabilities, and **AWS Lambda**, which provides serverless computing capabilities. These services can be used to automate DR processes, reducing the risk of human error and improving response times.

Having explored the robust features and capabilities of the AWS cloud that support DR, we will now shift our focus to the various strategies that can be employed to implement effective DR in the AWS cloud. In the following section, we will delve into the different approaches and techniques that can be leveraged to design and implement a comprehensive DRP, tailored to the unique needs and requirements of your organization.

Different disaster recovery strategies in AWS

When it comes to DR in AWS, you have four primary approaches to choose from, each offering a unique balance of cost, complexity, and resilience. At one end of the spectrum, you can opt for a simple and cost-effective approach by making regular backups of your data. At the other end, you can implement more complex strategies that leverage multiple active Regions to ensure high availability and fault tolerance. The following figure highlights different DR strategies with their RTO/RPO values:

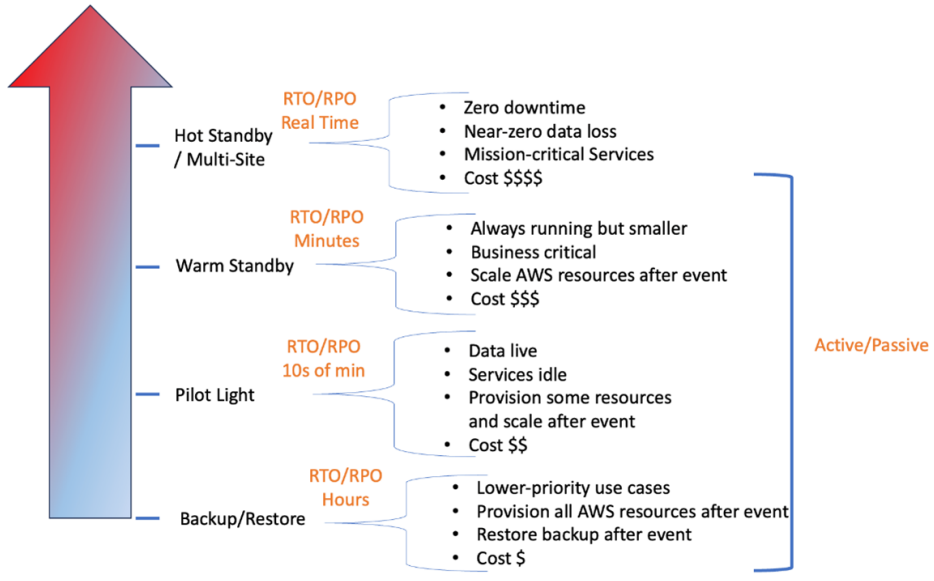

Figure 14.1 – DR modes

We will now review each of the DR strategies shown in the preceding figure, from high RTO/RPO to low RTO/RPO value.

Backup and restore

At the heart of any effective DR strategy lies a robust backup and restore process, which serves as a cornerstone for both normal recovery and DR scenarios. This process enables you to restore your data within a single-Region architecture or replicate it to another Region for a multi-Region architecture, ensuring business continuity in the face of disruptions. While backup addresses the data aspect, it's equally crucial to ensure that your infrastructure can be quickly recovered.

To achieve this, we recommend adopting an IaC approach, which allows you to version, track, and reproduce your infrastructure configuration. AWS provides its own IaC framework, CloudFormation, or you can opt for third-party solutions such as **Terraform**. To take your DR to the next level, consider automating the recovery process using single-click deployment pipelines powered by **continuous integration/continuous deployment** (**CI/CD**) services such as **CodePipeline** or **Jenkins**, enabling rapid and reliable restoration of your infrastructure and applications.

AWS offers a comprehensive backup solution, **AWS Backup**, which simplifies the process of orchestrating backups across your AWS cloud infrastructure. Moreover, many AWS services come equipped with built-in point-in-time recovery capabilities, enabling you to meet your RPO with ease. For instance, database services such as **DynamoDB** and **RDS**, as well as storage services such as **EBS** and **Amazon FSx**, provide this feature out of the box. Additionally, Amazon S3, the object storage service, offers versioning and cross-Region replication capabilities, allowing you to maintain a history of older data and seamlessly replicate it to other Regions without the need for a separate backup service. This integrated approach streamlines your backup and recovery processes, ensuring that your data is protected and easily recoverable in the event of an outage or disaster.

When implementing a backup process, it's essential to establish a robust setup that guarantees the recoverability of your backups. In addition to scheduling regular backups, you'll need to develop a retention strategy that aligns with your business requirements. This involves creating a plan to systematically purge outdated backups that are no longer critical by using lifecycle policies, ensuring that your storage resources are optimized and your backup repository remains organized and efficient. By striking a balance between data retention and storage management, you can ensure that your backups are not only complete but also easily recoverable in the event of a disaster or data loss.

While the backup and restore process offers a cost-effective means of enabling DR for your environment, it may not be sufficient to meet the stringent RTO and RPO requirements of your workload. In situations where near-instant recovery and minimal data loss are critical, a more advanced approach is necessary. To bridge this gap, we'll explore the next solution in our DR approach: **pilot light**. This approach offers a more robust and responsive way to ensure business continuity, providing a significant improvement in RTO and RPO over traditional backup and restore methods.

Pilot light

Pilot light is a DR strategy that significantly improves your RTO and RPO by eliminating the need to restore data in the recovery Region. This approach involves continuously replicating your data to a secondary Region, ensuring that your data is always available and up to date, thereby meeting your

RPO requirements. By doing so, you can bypass the time-consuming data restore process, which is a major bottleneck in traditional DR approaches, and significantly reduce your RTO.

While this approach requires provisioning infrastructure in the secondary Region to store the replicated data, which can increase costs compared to the backup and restore process, it provides a more robust and responsive DR solution. AWS services such as **Aurora Global Database**, DynamoDB global tables, and S3 facilitate ongoing replication in the secondary Region, making it easier to implement a pilot light strategy and achieve near-instant recovery and minimal data loss.

In a pilot light DR strategy, you maintain a minimal infrastructure in the secondary Region to ensure immediate data availability during recovery. However, to handle application traffic, you need to provision additional infrastructure, which can be a time-consuming process. To expedite this process, leveraging IaC and version control systems, such as GitHub or CodeCommit, that store code in the secondary Region can help facilitate the rapid provisioning of required infrastructure.

Despite this, the time required to provide the necessary infrastructure can still impact your RTO. To further improve your RTO, we'll explore an alternative approach in the next section: warm standby. This strategy offers an even more robust solution, designed to minimize downtime and ensure rapid recovery in the event of a disaster.

Warm standby

The warm standby approach builds upon the foundation of the pilot light strategy, further enhancing DR capabilities. To minimize infrastructure setup time, a scaled-down version of the infrastructure is provisioned in the secondary Region, alongside replicated data that's ready for consumption. This approach enables periodic testing of the DR process, ensuring readiness in the event of an outage.

While this approach is more expensive than pilot light due to the additional infrastructure provisioning, it significantly improves the RTO by reducing the infrastructure provisioning time. However, to support live traffic, the scaled-down environment requires adjustments, such as upsizing database instances, adding replicas, and increasing throughput. You can also use the serverless option, which will scale based on traffic demand. Horizontal scaling of the infrastructure is also necessary to handle peak traffic, which can be achieved through auto scaling configurations or using serverless options such as AWS Fargate. By adopting this approach, you can ensure the rapid and seamless recovery of your applications and services.

While the warm standby approach significantly improves the RTO, it still requires accounting for the time needed to scale up the infrastructure to handle live traffic. For mission-critical workloads that demand near-instant recovery, this RTO may not be sufficient. In such scenarios, we recommend adopting the hot standby approach to DR. This approach enables a very low RTO, often measured in minutes, but at a higher cost that is closer to that of the live production environment. By investing in a hot standby solution, you can ensure that your most critical workloads are always available and responsive, even in the event of a disaster or outage.

Hot standby

The hot standby approach involves provisioning a duplicate environment that mirrors the production environment, which can be configured in either active/active or active/passive mode. As discussed in *Chapter 11*, this approach enables a multi-Region and multi-site architecture, providing unparalleled resilience and availability. With an identical environment ready to handle production traffic in active/active mode or redirect traffic in active/passive mode, this approach achieves near-zero RTO and RPO.

However, this level of redundancy comes at a significant cost and requires increased management overheads. As such, the hot standby approach is typically reserved for mission-critical applications that demand the highest levels of availability and warrant additional investment.

In the subsequent section, we will delve into the crucial process of planning and defining your DR objectives. This includes establishing clear goals and requirements for your DR strategy, ensuring that your organization is well prepared to respond to disruptions and minimize downtime. By following the steps outlined in this section, you will be able to define a comprehensive DRP that aligns with your business needs and ensures the continuity of your critical operations.

Defining and planning your disaster recovery objectives

Defining and planning DR objectives are crucial steps in developing a comprehensive DRP. The following diagram shows the flow of the DRP:

Figure 14.2 – DRP

The following are the steps to define and plan the DR objectives shown in the preceding figure:

1. **Identify critical business processes**: Identify the critical business processes that are essential to the organization's operations and revenue generation. These processes should be prioritized for recovery in the event of a disaster.
2. **Determine RTO**: Determine the maximum amount of time that critical business processes can be down before significant financial or operational impact occurs. This will help determine the RTO for each process.
3. **Determine RPO**: Determine the maximum amount of data that can be lost during a disaster. This will help determine the RPO for each process.
4. **Identify dependencies**: Identify dependencies between critical business processes and supporting infrastructure, such as IT systems, networks, and facilities.
5. **Determine service-level agreements (SLAs)**: Determine the SLAs for each critical business process, ensuring they align with specific RTOs and RPOs, as well as application availability, performance, and security objectives.
6. **Assess risk**: Evaluate the potential risks and threats to these critical assets. This could include natural disasters, cyber-attacks, hardware failures, or human errors. The goal is to understand the likelihood and potential impact of each risk.
7. **Establish key performance indicators (KPIs)**: Establish KPIs to measure the success of DR efforts, such as the RTO, RPO, and SLAs. For example, you can have a KPI to achieve the set RTO, RPO, and SLA by simulating the DR scenario.
8. **Involve stakeholders**: Involve stakeholders, including business leaders, IT staff, and other relevant teams, to ensure that DR objectives align with business objectives and requirements.
9. **Prioritize objectives**: Prioritize DR objectives based on business requirements, risk assessments, and resource availability.
10. **Develop recovery strategies**: Based on your RTOs and RPOs, develop strategies for how you will recover each system or asset in the event of a disaster. This could involve things such as data backups, redundant systems, or failover sites.
11. **Document objectives**: Document DR objectives, including the RTO, RPO, SLAs, and KPIs, in a DRP or business continuity plan.
12. **Outline resource requirements**: Identify the resources needed for DR, including hardware, software, data storage, network infrastructure, and personnel.
13. **Test your plan**: Regularly test this plan to ensure it is effective and make any necessary adjustments based on the results of your tests.
14. **Training and awareness**: Conduct regular training about the process and make everyone responsible aware of their role and responsibilities.

15. **Communication plan**: Develop a communication plan to ensure effective and timely communication with stakeholders, employees, customers, and partners during a disaster event.
16. **Review and update**: Regularly review and update your DR objectives to ensure they remain relevant and effective in the face of changing business requirements and risk landscapes.

By following these steps, organizations can develop a comprehensive and effective DRP that minimizes the impact of disruptive events, protects critical business functions, and ensures business continuity.

Just as any development process requires rigorous testing to ensure its effectiveness, a DRP is no exception. A well-crafted DRP is only as good as its ability to perform under pressure, and the only way to confirm its efficacy is through thorough testing.

In the following section, we'll delve into the essential steps for testing your DRP, providing you with a roadmap to validate its readiness and identify areas for improvement, so you can have confidence in your organization's ability to respond to disasters and disruptions.

Testing disaster recovery plans

Testing DRPs in the AWS cloud is paramount to ensuring the resilience and availability of critical business applications and data. In the event of an outage or disaster, a tested DRP enables organizations to quickly recover and restore operations, minimizing downtime and data loss. By testing their plans, organizations can identify and remediate potential weaknesses, such as inadequate backup and restore procedures, insufficient resource allocation, and inadequate personnel training.

Moreover, testing in the AWS cloud allows organizations to take advantage of the cloud's scalability and flexibility, ensuring that their DRP can scale to meet the needs of their business. Regular testing also helps to ensure compliance with regulatory requirements and industry standards, providing an added layer of assurance for organizations. Ultimately, testing DRPs in the AWS cloud provides organizations with the confidence that their business can withstand disruptions, protecting their reputation, customer trust, and, ultimately, their bottom line.

There are different types of DR testing you can do to validate your DR process:

- **Document review**: This refers to a thorough examination of the DRP to ensure it is up-to-date, complete, and accurate. It must be signed off by all IT business teams.
- **Dry run**: This is a simulated test where team members walk through the DRP to identify gaps and areas for improvement.
- **Mock disaster**: A simulated disaster scenario is created to test the DRP, without affecting actual systems or data.
- **Parallel processing**: This is a real-world test where the DR system runs simultaneously with the production system to ensure seamless failover.

- **Full system failover**: This is a test where the production system is intentionally shut down, and the DR system takes over to ensure business continuity.
- **Partial system failover**: This is a test where a subset of production systems is shut down, and the DR system takes over to ensure partial business continuity.
- **Procedure verification**: This is a test where a checklist of DR procedures is reviewed and verified to ensure all steps are complete and accurate.
- **System functionality test**: This is a test that verifies the functionality of specific systems or applications in the DR environment.
- **Integration verification**: This is a test that verifies the integration of multiple systems or applications in the DR environment.
- **Network connectivity test**: This is a test that verifies the network connectivity and performance in the DR environment.
- **Data loss test**: This is a test measuring the amount of data loss that occurs in a disaster.

Once you have set up your DR environment for testing the DR process, you need to do additional testing to validate that the application works as expected.

The following are some of the types of testing you will need to do to ensure your new environment is secure and performant for production.

Functional testing

In a DR environment, **functional testing** refers to the process of verifying that the recovered systems, applications, and infrastructure can perform their intended functions and meet the required business and operational requirements after a disaster or outage.

Functional testing in a DR environment involves testing the following aspects:

- **Application functionality**: Verifying that the recovered applications can perform their intended functions, such as processing transactions, generating reports, and interacting with users
- **Data integrity**: Ensuring that the recovered data is accurate, complete, and consistent with the primary system
- **System integration**: Testing that the recovered systems can integrate with other systems, services, and infrastructure components as expected
- **User access and authentication**: Verifying that users can access the recovered systems and applications with the correct permissions and authentication
- **Performance and scalability**: Testing that the recovered systems can handle the expected workload and traffic and can scale to meet business demands

- **Business process continuity**: Ensuring that the recovered systems and applications can support critical business processes and operations

The goal of functional testing in a DR environment is to ensure that the recovered systems can support the business operations and meet the required service levels, thereby minimizing the impact of a disaster or outage on the organization.

Data loss testing

There are several techniques that can be used to verify the completeness and integrity of the recovered data during a data loss test. Here are some common examples:

- **File and directory listing**: Generate a list of all files and directories present in the recovered data. Compare this list with the known file structure and contents before the simulated data loss. Ensure that all expected files and directories are present and accounted for.
- **File checksums or hashes**: Calculate checksums or cryptographic hashes (e.g., MD5 or SHA-256) for each file before the simulated data loss. After recovery, calculate the checksums or hashes for the recovered files. Compare the pre-loss and post-recovery checksums or hashes to verify that the file contents are identical and have not been corrupted.
- **File size verification**: Record the file sizes of all files before the simulated data loss. After recovery, compare the file sizes of the recovered files with the original sizes. Any discrepancy in file size may indicate data corruption or incomplete recovery.
- **Metadata verification**: Capture file metadata (e.g., creation date, modification date, and permissions) before the simulated data loss. After recovery, verify that the metadata for the recovered files matches the original metadata.
- **Application-specific integrity checks**: For critical applications or databases, use built-in tools or utilities to verify the integrity of the recovered data. For example, database management systems often provide utilities to check the consistency and integrity of recovered databases.
- **Sample data verification**: Manually inspect a representative sample of recovered files or data records. Verify that the content and formatting of the sample data are correct and consistent with the original data.
- **End user testing**: Involve end users or subject matter experts in testing the recovered data. Have them perform typical tasks or operations using the recovered data to ensure its completeness and accuracy.

It's important to note that the specific methods used for verifying data integrity and completeness may vary depending on the type of data, the storage systems involved, and the recovery procedures employed. Additionally, a combination of multiple verification techniques may be necessary to gain a higher level of confidence in the recovered data.

Performance testing

During DR testing, **performance testing** refers to the process of evaluating the ability of the recovered systems, applications, and infrastructure to handle the expected workload and traffic, and to measure their responsiveness, throughput, and scalability under various conditions.

The goal of performance testing in a DR environment is to ensure that the recovered systems can support the business operations and meet the required service levels, even in the event of a disaster or outage.

Performance testing typically involves testing the following aspects:

- **Response time**: Measuring the time it takes for the recovered systems to respond to user requests, such as login, transaction processing, and data retrieval
- **Throughput**: Evaluating the number of transactions, requests, or operations that the recovered systems can handle within a given time period
- **Scalability**: Testing the ability of the recovered systems to handle increased workload, traffic, or user activity without a significant decrease in performance
- **Resource utilization**: Monitoring the usage of system resources such as CPU, memory, disk space, and network bandwidth to ensure that they are within acceptable limits
- **Network performance**: Evaluating the performance of the network infrastructure, including latency, packet loss, and throughput

Performance testing is crucial to ensure that the recovered systems can do the following:

- Handle the expected workload and traffic
- Meet the required service levels and response times
- Support critical business operations and processes
- Minimize the impact of a disaster or outage on the business

Some common performance testing scenarios in a DR environment are as follows:

- **Peak hour simulation**: Testing the recovered systems during peak usage hours to ensure they can handle the increased workload
- **Stress testing**: Pushing the recovered systems to their limits to evaluate their performance under extreme conditions
- **Soak testing**: Testing the recovered systems over an extended period to evaluate their performance and stability
- **Spike testing**: Testing the recovered systems with a sudden increase in workload or traffic to evaluate their ability to handle unexpected spikes

By carrying out performance testing, organizations can identify and address any performance-related issues, ensuring that their recovered systems can support business operations and meet the required service levels.

Security testing

In DR testing, **security testing** refers to the process of evaluating the security controls and measures in place to protect the recovered systems, applications, and data from unauthorized access, use, disclosure, modification, or destruction.

The goal of security testing is to ensure that the recovered systems and data are secure and protected from cyber threats, even in the event of a disaster or outage.

Security testing typically involves testing the following aspects:

- **Authentication and authorization**: Verifying that only authorized personnel can access the recovered systems and data and that access controls are in place to prevent unauthorized access.
- **Data encryption**: Ensuring that data is properly encrypted during transmission and storage, and that encryption keys are properly managed.
- **Network security**: Evaluating the security of the network infrastructure, including firewalls, **intrusion detection systems**, and **virtual private networks** (**VPNs**).
- **System hardening**: Verifying that recovered systems are properly hardened, with unnecessary services and ports disabled, and that security patches are up to date.
- **Incident response**: Testing for security incidents, such as data breaches or ransomware attacks. A ransomware attack refers to a malware security attack designed to deny access to files on an organization's servers.
- **Access control**: Evaluating the access control mechanisms in place, including role-based access control, to ensure that only authorized personnel have access to sensitive data and systems.
- **Compliance**: Verifying that the recovered systems and data meet relevant compliance requirements, such as HIPAA, PCI-DSS, or GDPR.

Security testing is crucial to ensure that the recovered systems and data are protected from cyber threats and that the organization can maintain the confidentiality, integrity, and availability of its data and systems.

Some common security testing techniques used in a DR environment are as follows:

- **Vulnerability scanning**: Identifying vulnerabilities in the recovered systems and applications
- **Penetration testing**: Simulating cyber-attacks to test the defenses of the recovered systems and applications

- **Compliance testing**: Verifying that the recovered systems and data meet relevant compliance requirements
- **Configuration testing**: Evaluating the security configuration of the recovered systems and applications

By conducting security testing, organizations can identify and address security vulnerabilities, ensuring that their recovered systems and data are secure and protected from cyber threats.

The Heartbleed Bug (`https://heartbleed.com/`) that occurred in April 2014 was a consequence of insufficient security testing and code review practices in the OpenSSL project. The code responsible for the vulnerability was introduced in 2012 but went unnoticed until 2014. This oversight highlighted the importance of rigorous security testing, code audits, and vulnerability assessments, especially for critical software components such as OpenSSL, which are widely used to protect sensitive data and communications. The bug allowed attackers to read sensitive data from the memory of systems protected by vulnerable versions of the OpenSSL software, potentially exposing sensitive information such as passwords, encryption keys, and other private data. The vulnerability affected a significant portion of the internet, including popular websites, web servers, email servers, and other services. The potential impact was massive, as the bug could have allowed attackers to eavesdrop on communications, impersonate services and users, and compromise the security of numerous systems and applications. Based on such real-world incidents, the following are some examples of lessons learned during disasters:

- Even with robust DRPs in place, human error can still occur. In addition to a disaster plan, additional safeguards to prevent human errors can help prevent disasters. In October 2018, GitHub experienced a massive service outage that lasted for several hours, affecting millions of developers and businesses worldwide, which was a result of human error (`https://github.blog/news-insights/company-news/oct21-post-incident-analysis/`).
- Many incidents have occurred as the application was not adequately tested, leading to a slower-than-expected recovery. This requires companies to invest in more comprehensive testing and simulation exercises to improve their response to future outages. One such example happened in August 2012, when the Knight Capital Group experienced a devastating trading disruption that resulted in massive financial losses, which could have been avoided by adequate testing (`https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html`).
- Security incidents can also result in disasters, so you need to make sure adequate simulation exercises to improve cybersecurity posture are part of the DR testing.
- In some cases, gaps are identified in the simulation of DR testing. Companies now use mechanisms such as **chaos engineering** (covered in detail in *Chapter 13*) to intentionally introduce failures into their systems to test their response.
- Some disasters can happen due to glitches in your application, so ensure you have a recovery process and a plan so that your team is prepared to respond quickly and effectively in the event of a critical system failure.

While we've covered the essential steps to create an efficient DR and testing plan, it's crucial to acknowledge that common misconceptions and pitfalls can still compromise your availability.

In the next section, we'll examine some of these potential mistakes, including unrealistic expectations, inadequate testing, and insufficient resources.

Avoiding disaster recovery pitfalls and misconceptions

When it comes to DR planning, even the most well-intentioned organizations can fall victim to common pitfalls and misconceptions that can have devastating consequences. By being aware of these common mistakes, you can proactively mitigate risks and ensure that your DRP is robust, reliable, and effective in the face of unexpected disruptions.

Pitfalls

Here are some common pitfalls in DR testing:

- **Inadequate testing scope**: Failing to test all critical systems, applications, and data, leading to incomplete recovery.
- **Insufficient testing frequency**: Not testing DRPs regularly, leading to outdated plans and unprepared teams.
- **Inadequate testing environments**: Not replicating production environments accurately, leading to unrealistic test results.
- **Lack of realistic scenarios**: Not testing DRPs with realistic scenarios, leading to unpreparedness for real-world disasters.
- **Inadequate team training**: Not providing adequate training to teams involved in DR testing, leading to confusion and delays during actual disasters.
- **Inadequate communication**: Not communicating DR testing results and plans to all stakeholders, leading to confusion and lack of awareness. It is important to integrate liveliness checks with automatic email, Slack, and phone communications.
- **Overreliance on technology**: Relying too heavily on technology and not considering human factors, leading to incomplete recovery.
- **Not testing for data integrity**: Not testing for data integrity and consistency, leading to data corruption or loss during recovery.
- **Not relying on a single information source**: Not relying on a single source of information may lead to flaws in the recovery process, which can delay the recovery time and recovery expectations.

By recognizing and addressing these pitfalls, organizations can avoid the most common mistakes and ensure that their DRP is robust, reliable, and capable of supporting business continuity in the face of uncertainty.

In the next section, we'll look at the misconceptions.

Misconceptions

Here are some of the most frequent misconceptions that you may come across in DR testing:

- **We've got a DRP, so we're covered**: Having a DRP is not enough; it must be regularly tested and updated to ensure effectiveness.
- **DR testing is too expensive**: While DR testing may require resources, the cost of not testing can be much higher in terms of downtime, data loss, and reputation damage.
- **We can just restore from backups**: Backups are not a substitute for DR testing; they may not be complete, up to date, or restorable.
- **Our cloud provider will handle DR**: While cloud providers may offer some DR capabilities, it's still important to test and validate your own DRPs.
- **We don't need to test DR because we have redundant systems**: Redundant systems can still fail, and DR testing ensures that you can recover quickly and efficiently. Redundant systems can't help with the recovery of a system in the case of security attacks such as ransomware.
- **DR testing is only for IT**: DR testing involves all teams and stakeholders, not just IT, to ensure a comprehensive recovery.
- **We can just test DR once and be done**: DR testing is an ongoing process that requires regular testing and updating to ensure continued effectiveness.
- **DR testing is too complex**: While DR testing can be complex, it's essential to break it down into manageable components and prioritize testing based on risk and impact.

By being aware of these misconceptions, organizations can avoid common mistakes and ensure that their DR testing is comprehensive, effective, and aligned with their business needs.

Summary

In this chapter, we have provided a comprehensive guide to developing a robust DR strategy, crafting an effective DRP tailored to your organization's unique needs, and understanding the crucial role of testing in validating the plan's effectiveness.

We've also explored the various types of testing processes that can be employed to ensure the reliability of your DRP. Additionally, we've highlighted common misconceptions and pitfalls that can compromise your DRP and availability, enabling you to avoid these potential mistakes.

Building on this foundation, the next chapter will delve into the array of AWS offerings designed to simplify and enhance your DRP and process, providing you with a deeper understanding of how to leverage these services to bolster your organization's resilience.

15

Finalize Building Resilient Architecture Using AWS Resilience Services

As we've explored throughout this book, building resilience in modern systems is crucial for ensuring the availability, reliability, and performance of critical applications and services. In today's cloud-first world, many organizations are turning to cloud providers such as **Amazon Web Services** (**AWS**) to host their applications and data.

AWS offers a wide range of services and tools that can help organizations build resilient systems, from infrastructure and database services to security and analytics tools. In this final chapter, we'll delve into the world of AWS resilience services, exploring the various tools and features that can help you design, build, and operate resilient systems in the cloud.

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

- Backing up using the AWS Backup service
- Following the resilience lifecycle framework
- Utilizing AWS Resilience Hub
- Recovering using AWS Elastic Disaster Recovery

Backing up using the AWS Backup service

AWS Backup is a fully managed backup service offered by AWS that provides a centralized and automated way to back up and restore data across AWS services. We covered more details on the importance of backing up critical data in *Chapter 3*. With AWS Backup, users can easily create backup plans that automatically protect their data across AWS resources such as Amazon **Simple Storage Service** (**S3**), Amazon **Elastic Block Store** (**EBS**), Amazon **Relational Database Service** (**RDS**), and Amazon **Elastic**

Compute Cloud (**EC2**). The following is a link to all the supported resources: `https://docs.aws.amazon.com/aws-backup/latest/devguide/whatisbackup.html#supported-resources`.

AWS Backup provides a scalable, durable, and secure way to store backups, and allows users to set retention periods, manage backup frequencies, and track backup status. It also provides features such as automatic backup validation, data encryption, and access controls, ensuring that data is protected and secure.

Additionally, AWS Backup integrates with other AWS services, such as **Amazon CloudWatch** and **AWS CloudFormation**, to provide a comprehensive data protection and management solution. By using AWS Backup, users can ensure business continuity, meet compliance requirements, and reduce the risk of data loss, all while minimizing administrative burdens and costs.

Next, we'll delve into the inner workings of the AWS Backup process, examining the step-by-step workflow and the key services that come together to enable seamless data protection, including the storage, compute, and security components that power this robust backup solution.

AWS Backup process

Since we have introduced you to the AWS Backup process, let's now review how the AWS Backup process works and examine the different components/stages that are involved in the process. The backup process, as illustrated in the following figure, begins by creating a backup plan for AWS resources through the AWS Backup portal.

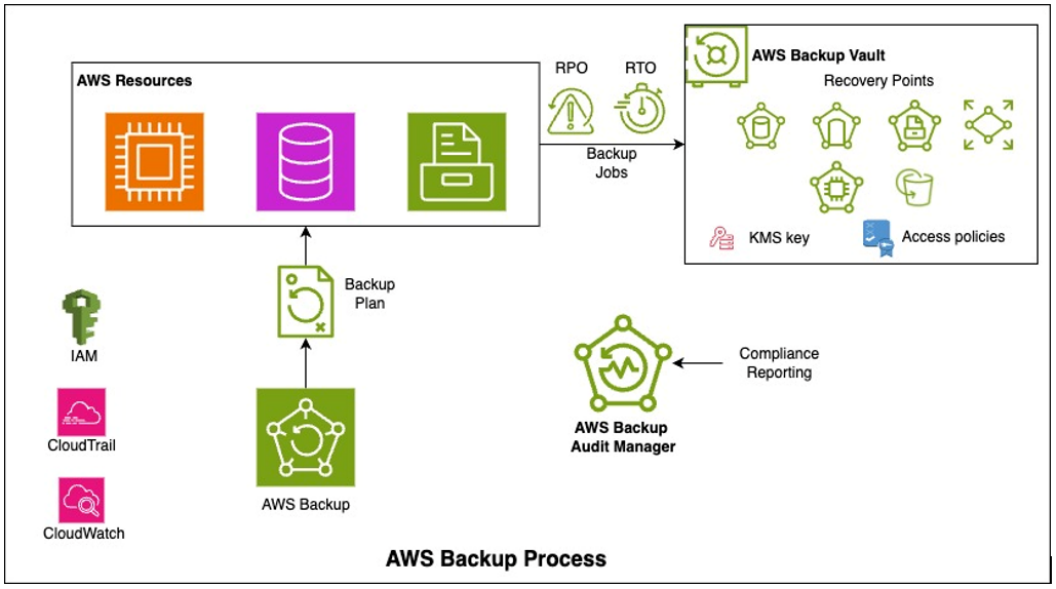

Figure 15.1 – AWS Backup process

This backup plan initiates the backup job and stores all backups in the AWS Backup vault. During the creation of the backup vault, you will specify a **Key Management Service** (**KMS**) key, which will be utilized to encrypt certain backups stored within the vault. For AWS services that do not support independent encryption, AWS Backup will use the data source key to encrypt the data rather than the AWS Backup vault's KMS key. Access policies will be assigned to the backup vaults and the resources they contain, defining who can create backups and limiting access to delete them. **AWS Backup Audit Manager**, depicted in the diagram, is responsible for monitoring the compliance of the backups against the controls that define your specific requirements.

The following are the steps to create a cloud-native backup with an AWS Backup vault:

1. **Create an AWS Backup vault**:

    I. Go to the AWS Management Console and navigate to the **AWS Backup** dashboard.

    II. Click on **Create a backup vault** and enter a name and description for your vault.

    III. Choose the AWS Region where you want to create the vault. Choosing the region will depend on where you want the backup to end up, which depends on the recovery requirements of your workload. Examples of recovery requirements are whether the backup should be restored in the same region to a previous point in time, or whether it should be restored to a different region for **disaster recovery** (**DR**) purposes.

2. **Define a backup plan**:

    I. Go to the **AWS Backup** dashboard and click on **Create a backup plan**.

    II. Enter a name and description for your backup plan.

    III. You have the option to include all resources or choose the resources you want to back up, such as Amazon S3 buckets or Amazon EBS volumes.

3. **Configure backup settings**: Configure the backup settings for each resource, such as the following:

    - **Backup frequency:** Choose how often you want to back up your data. The following are the different values to choose from:

        - Hourly
        - Every 12 hours
        - Daily
        - Weekly
        - Monthly
        - Custom Cron expression

- **Retention period:** Choose how long you want to retain your backups. The following are the different values to choose from:
  - Days
  - Weeks
  - Months
  - Years
  - Forever
- **Backup window:** Choose the time window during which backups can occur. The following are the default settings:
  - Start Time: 12:30 AM local to your system's time zone (0:30 in 24-hour systems)
  - Start Within 8 hours
  - Complete Within 7 days

4. **Assign identity and access management (IAM) roles**:
   I. Assign IAM roles to the AWS Backup service to allow it to access your resources.
   II. Create a new IAM role or use an existing one that has the necessary permissions.
5. **Create a backup job**:
   I. Go to the **AWS Backup** dashboard and click on **Create a backup job**.
   II. Select the backup plan you created and choose the resources to back up.
   III. Choose the backup vault you created and configure any additional settings.
6. **Run the backup job**:
   I. Go to the **AWS Backup** dashboard and click on **Run backup job**.
   II. The backup job will run and create a backup of your data in the backup vault.
7. **Monitoring and reporting**: You learned, in *Chapter 12*, the importance of monitoring and how it is critical to make sure your process is working reliably. You can monitor your backups using AWS Backup's built-in monitoring and reporting features. You can use Amazon CloudWatch to track the backup job status and receive notifications.
8. **Restore data**:
   I. When you need to restore data, go to the **AWS Backup** dashboard and click on **Restore data**.
   II. Select the backup job you want to restore from and choose the resources to restore.
   III. Choose the restore destination and configure any additional settings.

9. **Test and validate**:

    I. Test and validate your backups regularly to ensure that they are complete and recoverable.

    II. Use AWS Backup's built-in testing and validation features to verify the integrity of your backups.

By following these steps, you can create a cloud-native data protection solution using AWS Backup that provides automated, scalable, and secure backups of your data in the cloud.

With a solid grasp of the AWS Backup service under your belt, we're now ready to dive deeper into the advanced features that take data protection to the next level. In the following section, we'll explore the powerful capabilities of **AWS Backup Vault Lock**, which enables you to create immutable backups that are tamper-proof and compliant with regulatory requirements.

Immutable backups with AWS Backup Vault Lock

AWS Backup Vault Lock is a feature of AWS Backup that allows you to create immutable backups, which means that once a backup is created, it cannot be deleted or modified for a specific period of time. This ensures that your backups are protected from accidental deletion or tampering and provides an additional layer of data protection.

Here are some key features of AWS Backup Vault Lock:

- **Immutable backups**: AWS Backup Vault Lock creates immutable backups, which means that once a backup is created, it cannot be deleted or modified for a specified period of time.
- **Retention period**: You can specify a retention period for your backups, which determines how long the backups will be retained in the vault.
- **Lock mode**: AWS Backup Vault Lock provides two lock modes:
    - **Compliance mode**: This mode locks the vault for a specified period of time and prevents any modifications or deletions during that time.
    - **Governance mode**: This mode locks the vault for a specified period of time and prevents any modifications or deletions during that time but allows you to extend the lock period.
- **Tamper-evident**: AWS Backup Vault Lock provides a tamper-evident record of your backups, which means that any attempts to modify or delete a backup will be detected and logged.
- **Audit trail**: AWS Backup Vault Lock provides an audit trail of all backups and restore operations, which helps you to track and monitor access to your backups.
- **Integration with AWS services**: AWS Backup Vault Lock integrates with other AWS services, such as Amazon S3, Amazon EBS, and Amazon RDS, to provide a comprehensive data protection solution.

By using AWS Backup Vault Lock, you can ensure that your backups are protected from accidental deletion or tampering and provide an additional layer of data protection for your organization. To address ransomware concerns and enhance data security, AWS Backup provides a logically air-gapped vault. These immutable backup copies are locked by default and further protected through encryption using AWS-owned keys.

In the subsequent sections, we'll delve into the advantages of leveraging AWS Backup over traditional backup solutions, exploring how its cloud-native approach, seamless integration with AWS resources, and unified management capabilities can help you simplify data protection, reduce costs, and improve overall resilience. With a solid understanding of the AWS Backup process under your belt, we'll now shift our focus to another critical service offered by AWS: the resilience lifecycle framework.

Following the resilience lifecycle framework

The **AWS resilience lifecycle framework** is a set of guidelines and best practices provided by AWS to help organizations design, implement, and operate resilient cloud architectures. The framework is designed to help organizations build and maintain highly available, fault-tolerant, and scalable systems that can withstand disruptions and failures.

The AWS resilience lifecycle framework is based on six pillars:

- **Foundational services**: This pillar focuses on the core AWS services that provide the foundation for a resilient architecture (additional details in *Chapters 10* and *11*), such as Amazon S3, Amazon EC2, and Amazon RDS.
- **Compute resilience**: This pillar covers strategies for building resilient compute resources, including instance types, Availability Zones, and auto scaling.
- **Storage resilience**: This pillar focuses on designing and implementing resilient storage solutions, including data replication, backup, and recovery.
- **Database resilience**: This pillar covers strategies for building resilient databases, including database clustering, replication, and backup and recovery.
- **Security, Identity, and Compliance**: This pillar focuses on securing and protecting data and resources, including IAM, encryption, and compliance.
- **Recovery and Restore**: This pillar covers strategies for recovering and restoring systems and data in the event of a failure or disruption, including DR, backup and restore, and incident response.

The AWS resilience lifecycle framework also provides a set of best practices and design patterns for building resilient architectures, including the following:

- **Design for failure**: Anticipate and plan for failures and design systems to recover from them.
- **Implement redundancy**: Use redundancy to ensure that systems can continue to operate even if one or more components fail.

- **Use automation**: Automate processes and workflows to reduce the risk of human error and improve response times.
- **Monitor and test**: Continuously monitor systems and test for failures to identify and address potential issues before they become incidents.
- **Learn from failures**: Analyze and learn from failures to improve the resilience of systems and processes.

By following the AWS resilience lifecycle framework, organizations can build and operate highly resilient cloud architectures that can withstand disruptions and failures and provide a better experience for their customers.

Now that we've laid the foundation with an introduction to the AWS resilience lifecycle framework, we'll take a deeper dive into the compelling reasons why organizations such as yours should adopt this framework.

Why do you need the AWS resilience lifecycle framework?

Here, we'll explore the pressing need for a structured approach to resilience, and how the AWS resilience lifecycle framework can help you address common pain points, such as inadequate DR planning, insufficient visibility into system vulnerabilities, and ineffective incident response strategies.

By understanding the benefits and motivations behind using the AWS resilience lifecycle framework, you'll be better equipped to make a strong business case for implementing this critical capability within your organization.

You need the AWS resilience lifecycle framework for several reasons:

- **Minimize downtime**: The framework helps you design and implement systems that can withstand failures and disruptions, minimizing downtime and ensuring high availability.
- **Improve customer experience**: By building resilient systems, you can ensure that your customers can access your applications and services consistently, leading to improved customer satisfaction and loyalty.
- **Reduce revenue loss**: Downtime and outages can result in significant revenue loss. The AWS resilience lifecycle framework helps you avoid these losses by ensuring that your systems are always available.
- **Meet compliance requirements**: Many industries have strict compliance requirements, such as adhering to the **Health Insurance Portability and Accountability Act** (**HIPAA**), **Payment Card Industry Data Security Standard** (**PCI-DSS**), and **General Data Protection Regulation** (**GDPR**), which mandate high availability and resilience. The AWS resilience lifecycle framework helps you meet these requirements.

- **Protect brand reputation**: System failures and outages can damage your brand reputation and erode customer trust. The AWS resilience lifecycle framework helps you avoid these risks and maintain a positive brand image.
- **Reduce operational costs**: By implementing resilient systems, you can reduce the operational costs associated with troubleshooting, repairing, and recovering from failures.
- **Improve security**: The AWS resilience lifecycle framework includes security best practices that help you protect your systems and data from cyber threats and other security risks.
- **Scale efficiently**: As your business grows, the AWS resilience lifecycle framework helps you scale your systems efficiently, ensuring that they can handle increased traffic and demand without compromising availability.
- **Enhance DR**: The framework provides guidance on designing and implementing effective DR strategies, ensuring that your systems can recover quickly in the event of a disaster.
- **Stay competitive**: In today's digital economy, system availability and resilience are critical differentiators. By implementing the AWS resilience lifecycle framework, you can stay competitive and ahead of your competitors.
- **Reduce mean time to recovery** (**MTTR**): The framework helps you reduce the time it takes to recover from failures, ensuring that your systems are back online quickly and minimizing the impact of downtime when a disaster strikes.
- **Increase mean time between failures** (**MTBF**): By implementing resilient systems, you can increase the time between failures, reducing the frequency of outages and downtime.

By adopting the AWS resilience lifecycle framework, you can ensure that your cloud-based systems are designed and implemented to withstand failures and disruptions, providing a better experience for your customers and a competitive advantage for your business.

In the subsequent section, we'll delve into the inner workings of the AWS resilience lifecycle framework, examining the key components, processes, and best practices that underpin this comprehensive approach to building and maintaining resilient systems. By gaining a deeper understanding of how the framework operates, you'll be able to effectively leverage its guidance and tools to improve the availability, durability, and responsiveness of your critical applications and services.

How does the AWS resilience lifecycle framework work?

The AWS resilience lifecycle framework is a structured approach to designing, implementing, and operating resilient cloud architectures on AWS. Here's an overview of how it works:

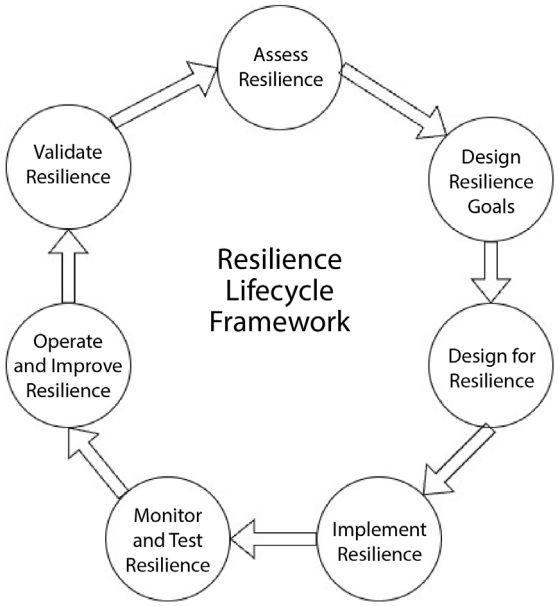

Figure 15.2 – Resilience lifecycle framework

By following this framework, you can design, implement, and operate highly resilient cloud architectures that meet your business needs and provide a better experience for your customers.

Here's a breakdown of the framework, as shown in *Figure 15.2*:

- **Assess resilience**: Identify the critical components of your system, assess their resilience, and prioritize areas for improvement.
- **Define resilience goals**: Establish clear resilience goals and objectives, such as uptime, response time, and data durability
- **Design for resilience**: Apply resilience principles and patterns to your system design, including the following:
  - **Distributed systems**: Design systems to distribute workloads across multiple **Availability Zones** (**AZs**) and Regions.
  - **Redundancy**: Implement redundant components and services to ensure high availability.
  - **Auto Scaling**: Use auto scaling to dynamically adjust capacity to meet changing workloads. In *Chapter 2*, we provided a more detailed explanation of why auto scaling is essential for maintaining resilience.

    - **Fault isolation**: Isolate faults to prevent cascading failures. In *Chapter 7*, we covered how to architect fault-tolerant applications that will help you create isolated failure points.
    - **Self-healing**: Implement self-healing mechanisms to automatically recover from failures. In *Chapter 9*, you learned how to create self-healing container environments.

- **Implement resilience**: Implement resilience features and services, such as the following:
    - **AWS services**: Leverage AWS services such as Amazon S3, Amazon RDS, and **Amazon DynamoDB**, which are designed for high availability and resilience.
    - **Load balancing**: Use Amazon **Elastic Load Balancer** (**ELB**) to distribute traffic across multiple instances and AZs.
    - **Database replication**: Implement database replication and clustering to ensure high availability and data durability.
    - **Backup and recovery**: Implement regular backups and recovery processes to ensure data integrity and availability.
- **Monitor and test resilience**: Continuously monitor and test your system's resilience, including the following:
    - **Monitoring**: Use AWS services such as Amazon CloudWatch and **AWS X-Ray** to monitor system performance and identify potential issues.
    - **Chaos engineering**: Perform controlled experiments to test system resilience and identify areas for improvement. We covered chaos engineering in detail in *Chapter 13*.
    - **DR simulation game days**: Conduct regular DR simulation game days to simulate failures and test incident response processes.
- **Operate and improve resilience**: Continuously operate and improve your system's resilience, including the following:
    - **Incident response**: Establish incident response processes to quickly respond to and resolve outages and failures.
    - **Post-incident review**: Conduct post-incident reviews to identify root causes and implement improvements.
    - **Continuous improvement**: Regularly review and update resilience designs and implementations to ensure they meet evolving business needs.
- **Validate resilience**: Validate your system's resilience through regular audits and assessments, including the following:
    - **Resilience assessments**: Conduct regular resilience assessments to identify areas for improvement.

- **Compliance and audits**: Ensure compliance with regulatory requirements and industry standards.

Now that we've explored the foundational principles and benefits of the AWS Resilience lifecycle framework, we're ready to take the next step in our resilience journey.

In the following section, we'll introduce AWS Resilience Hub, a powerful, centralized dashboard that simplifies the implementation and management of the resilience lifecycle framework.

Utilizing AWS Resilience Hub

AWS Resilience Hub is a service offered by AWS that helps organizations design, implement, and manage resilient architectures for their applications and workloads. With Resilience Hub, you'll be able to easily leverage pre-built templates, automated workflows, and actionable insights to drive meaningful improvements in your organization's ability to withstand disruptions and maintain business continuity. It provides a centralized platform for assessing, improving, and maintaining the resilience posture of AWS resources and applications.

Resilience Hub offers a set of features and tools that enable organizations to do the following:

- **Assess resilience**: Identify potential weaknesses and vulnerabilities in their AWS resources and applications and receive recommendations for improvement.
- **Implement resilience**: Use guided workflows and templates to implement resilient architectures and configurations for their AWS resources and applications.
- **Monitor and optimize**: Continuously monitor the resilience of their AWS resources and applications and receive alerts and recommendations for optimization.
- **Collaborate and govern**: Collaborate with teams and stakeholders to manage resilience and establish governance policies and procedures to ensure compliance.

Resilience Hub supports coverage for a wide range of AWS services, including Amazon EC2, Amazon RDS, Amazon S3, and more. It also integrates with other AWS services, such as the **AWS Well-Architected Framework**, AWS CloudFormation, and Amazon CloudWatch.

By using Resilience Hub, organizations can build more resilient and reliable applications and workloads, reduce downtime and errors, and improve overall business continuity.

How does AWS Resilience Hub work?

AWS Resilience Hub is a service that helps you design, implement, and manage resilient architectures for your applications and workloads. Here's an overview of how it works:

1. **Describe**: Build a comprehensive view of your application by importing resources from multiple sources, such as AWS CloudFormation stacks, Terraform state files, AWS resource groups, and

Amazon EKS clusters. You can also leverage pre-existing applications from the AWS Service Catalog AppRegistry to get started.

2. **Define**: Establish customized resilience policies for your applications, outlining specific **recovery point objective** (**RPO**) and **recovery time objective** (**RTO**) targets to ensure business continuity in the event of disruptions to applications, infrastructure, AZs, or Regions. These targets serve as a benchmark to assess whether your application meets its designated resiliency standards.
3. **Assess**: You provide AWS Resilience Hub with information about your application or workload, including its architecture, dependencies, and performance requirements. Resilience Hub assesses your application or workload and identifies potential weaknesses and vulnerabilities. It uses AWS's Well-Architected Framework to analyze the application components and expose potential gaps in your application's ability to withstand disruptions.
4. **Recommend**: Based on the assessment, Resilience Hub provides recommendations for improving the resilience of your application or workload. Recommendations may include changes to architecture, configuration, or deployment strategies.
5. **Implement**: You implement the recommended changes to your application or workload using AWS services such as Amazon EC2, Amazon RDS, and Amazon S3. Resilience Hub provides guidance and tools to help you implement the changes.
6. **Monitor and optimize**: Resilience Hub continuously monitors your application or workload to identify potential issues and opportunities for optimization. You receive alerts and recommendations for optimization and can use Resilience Hub to implement changes and improve performance.
7. **Govern and comply**: Resilience Hub provides features to help you govern and manage your application or workload, including compliance with regulatory requirements such as HIPAA, PCI-DSS, and GDPR. You can use Resilience Hub to establish policies, procedures, and controls to ensure compliance and governance.

AWS Resiliency Hub can be accessed via the following means:

- **Resilience Hub console**: A web-based interface that provides visibility and control over your application or workload
- **Resilience Hub APIs**: A set of APIs that allow you to integrate Resilience Hub with your existing tools and workflows
- **Resilience Hub agents**: Software agents that run on your instances and provide real-time monitoring and data collection

AWS Resiliency Hub helps to do the following:

- Uncover potential weakness
- Protect mission-critical applications
- Meet contractual and regulatory requirements

By using AWS Resilience Hub, you can improve the overall resilience, performance, and security of your applications and workloads, while reducing risk, cost, and complexity.

Having established a solid foundation in the AWS Resilience lifecycle framework and Resilience Hub, we're now poised to explore the next critical component of our resilience strategy. In the following section, we'll delve into the capabilities of **AWS Elastic Disaster Recovery** (**AWS DRS**), a service that enables you to orchestrate seamless failovers to new infrastructure and failbacks, ensuring minimal downtime and data loss in the event of an outage or disaster. By leveraging AWS DRS, you'll be able to automate your DR processes, reduce recovery times, and improve the overall resilience of your applications and services, giving you the confidence to weather even the most unexpected disruptions.

Recovery using AWS DRS

AWS DRS is a service offered by AWS that helps organizations recover from outages and disasters by replicating their applications and data in a secondary location. It provides a managed service for DR, allowing customers to easily set up, manage, and orchestrate the recovery of their applications and data in the event of an outage or disaster. We covered how to plan for DR in *Chapter 14*, and you can use AWS DRS to implement it.

AWS DRS architecture

The following is the architecture of AWS DRS:

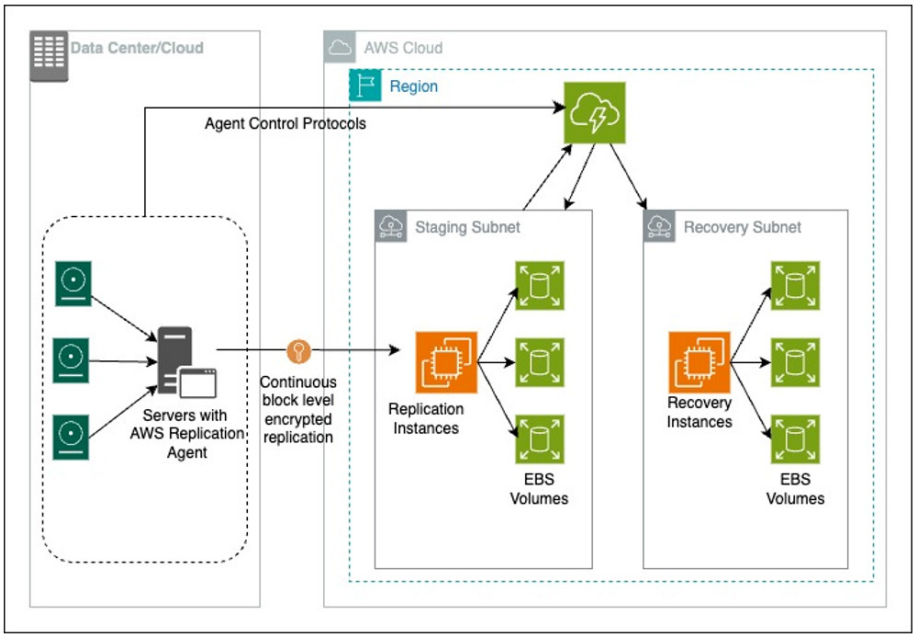

Figure 15.3 – AWS DRS architecture

The following is an overview of how AWS DRS works:

1. **Source environment setup**:

   I. Set up your source environment, which includes your applications, data, and infrastructure.

   II. Install the DRS agent on your source environment, which enables data replication and monitoring.

   III. Configure the agent to collect data about the source environment, including the following:

   - **Server configurations**: Collect data about server settings, such as network configurations, storage, and operating system settings
   - **Application data**: Collect data about applications, such as database configurations, filesystems, and application settings
   - **Storage**: Collect data about storage systems, such as **storage area network** (**SAN**), **network attached storage** (**NAS**), and filesystems

2. **Replication**:

   - DRS replicates your data from the source environment to a target staging environment, which can be another AWS region.
   - Block-level replication is done in near-real time, ensuring that your data is up to date and consistent across both environments.

3. **Monitoring**:

   - DRS continuously monitors your source environment for any issues or outages.
   - If an issue is detected, DRS automatically triggers a failover to the target environment.

4. **DR failover**:

   - DRS creates a new environment in the target location that is identical to the source environment, and it will automatically deploy the replicated data.
   - During a failover, DRS redirects traffic from the source environment to the target environment.
   - The target environment takes over as the primary environment, ensuring minimal downtime and data loss.

5. **Failback**:

   - Once the issue is resolved, DRS can be used to trigger a failback to the source environment.
   - The source environment is updated with any changes made during the failover, and traffic is redirected back to the source environment.

AWS DRS components

The following are the key components of AWS DRS:

- **DRS agent**: A software agent installed on your source environment that enables data replication and monitoring.
- **DRS console**: A web-based interface that provides visibility and control over your DR environment.
- **Staging environment**: The staging environment keeps a copy of the replicated data from the source environment.
- **Target environment**: The secondary environment that takes over as the primary environment during a failover.
- **Recovery plans**: Customizable plans that define the failover and failback processes, including the order of application startup and shutdown.

By using AWS DRS, you can improve your organization's resilience and ensure that your mission-critical applications are protected from disruptions, whether they are caused by natural disasters, cyber-attacks, or other unforeseen events.

You should consider using AWS DRS in the following scenarios:

- **Mission-critical applications**: When you have mission-critical applications that require high availability and cannot afford to experience downtime or data loss.
- **High-risk environments**: When you operate in high-risk environments, such as finance, healthcare, or government, where data loss or downtime can have severe consequences.
- **Large-scale deployments**: When you have large-scale deployments with multiple applications and databases that require a comprehensive DR solution.
- **Complex IT environments**: When you have complex IT environments with multiple dependencies and interconnections, which make it challenging to implement a DR solution in-house.
- **Cloud migration**: When you're migrating applications and data to the cloud and need a DR solution that integrates with your cloud infrastructure. Some customers also use this strategy in migrations to fail over the on-premises traffic to the cloud and then never fail the environment back to on-premises.
- **DR testing**: When you need to test your **disaster recovery plan** (**DRP**) regularly, AWS DRS provides a convenient and cost-effective way to do so.
- **Compliance requirements**: When you need to meet regulatory and compliance requirements, such as HIPAA, PCI-DSS, or GDPR, that mandate DR and business continuity plans.
- **Data center outages**: When you experience frequent data center outages or have concerns about the reliability of your data center infrastructure.

- **Cybersecurity threats**: When you're concerned about cybersecurity threats, such as ransomware or DDoS attacks, which can compromise your data and applications.
- **Business continuity planning**: When you need to develop a business continuity plan that ensures your organization can continue to operate in the event of a disaster or outage.
- **Multi-region deployments**: When you have applications and data deployed across multiple AWS Regions and need a DR solution that can handle multi-region deployments.
- **Customized recovery plans**: When you need customized recovery plans that meet specific business requirements, such as rapid recovery of critical applications or data.

In general, if you have critical applications, data, or systems that require high availability and you want to ensure business continuity in the event of an outage or disaster, you should consider using AWS DRS.

AWS DRS best practices

Here are some of the best practices when using AWS DRS:

- **Implement a DRP**: Develop a comprehensive DRP that outlines your recovery objectives, data protection strategies, and recovery procedures. This plan should be regularly reviewed and updated.
- **Replicate critical workloads**: Identify your critical workloads and ensure they are being replicated to AWS DRS. This will ensure your most important applications and data can be quickly recovered in the event of a disaster.
- **Test regularly**: Regularly test your DRP by performing test failovers and failbacks. This will help you identify any issues or gaps in your recovery process and ensure your plan is effective.
- **Set appropriate RPOs and RTOs**: Establish appropriate RPOs and RTOs for your workloads based on your business requirements. Configure AWS DRS accordingly to meet these objectives. Some people have unrealistic expectations that every application should have an instant RTO and RPO; therefore, appropriate and realistic RTOs and RPOs must be considered.
- **Automate recovery processes**: Leverage AWS DRS features such as recovery plans and automation to streamline the recovery process and reduce manual intervention during an incident.
- **Secure your environment**: Ensure that your AWS DRS environment is properly secured, with appropriate IAM permissions, network configuration, and data encryption measures in place.
- **Protect point-in-time snapshots**: AWS DRS depends on your point-in-time snapshots. In the case of a breach, bad actors can delete the snapshots, which will prevent you from recovering; this is why you need to take extra precautions in protecting them.
- **Monitor and audit**: Continuously monitor your AWS DRS environment for any issues or anomalies. Regularly audit your DR setup to ensure it aligns with your organization's policies and compliance requirements.

- **Maintain documentation**: Maintain detailed documentation on your DR setup, including configuration details, recovery procedures, and contact information for key personnel. It is also important to define the **RACI** (which is short for **Responsible, Accountable, Consulted, and Informed**) matrix with appropriate personnel assigned to it.
- **Integrate with other AWS services**: Leverage other AWS services, such as AWS CloudFormation, AWS Lambda, and Amazon EventBridge, to enhance your DR capabilities and improve overall resilience.
- **Stay up to date**: Keep your AWS DRS and related AWS services up to date with the latest releases and security patches to ensure optimal performance and protection.

By following these best practices, you can ensure that AWS DRS is effectively protecting your critical workloads and enabling a robust, reliable, and efficient DR strategy.

Advantages of AWS resilience services

Now, let's detail the advantages of AWS resilience services. Some of these advantages are common across multiple services while some are unique to the services:

- **Scalability**:
  - AWS Backup is designed to scale with your workload, providing a highly available and durable backup solution that can handle large amounts of data.
  - Resilience Hub is designed to scale with your business, providing a flexible and adaptable solution for managing resilience and performance.
- **Security**:
  - AWS Backup provides encryption at rest and in transit, ensuring that your data is protected from **man-in-the-middle attacks** during transit and unauthorized access at rest.
  - Resilience Hub provides a comprehensive view of your security posture, helping you identify and remediate security vulnerabilities and weaknesses.
- **Compliance**:
  - AWS Backup helps meet compliance requirements by providing features such as data retention, access controls, and auditing.
  - Resilience Hub provides features to help you govern and manage your applications and workloads, including compliance with regulatory requirements such as HIPAA, PCI-DSS, and GDPR.

- **Integration with AWS services**:
  - AWS Backup integrates seamlessly with other AWS services, such as Amazon S3, Amazon EBS, Amazon RDS, and Amazon EC2, making it easy to protect data across multiple services.
  - Resilience Hub integrates with other AWS services, such as Amazon EC2, Amazon RDS, and Amazon S3, providing a comprehensive platform for managing resilience and performance.
  - AWS DRS integrates with other AWS services, such as Amazon EBS, Amazon RDS, and Amazon ELB, to provide a comprehensive DR solution.
- **Cross-region support**:
  - AWS Backup provides support for cross-region backup, allowing you to store backups in multiple Regions for added redundancy and DR.
  - AWS DRS supports DR across multiple AWS Regions, allowing customers to replicate data and applications to different Regions for added resilience.
- **Fast recovery**:
  - AWS Backup provides fast recovery options, allowing you to quickly restore data in the event of a failure or data loss.
  - AWS DRS provides automated recovery capabilities, enabling you to quickly recover from outages and downtime.
- **Monitoring and reporting**:
  - AWS Backup provides monitoring and reporting capabilities, allowing you to track backup status, identify issues, and optimize your backup strategy.
  - Resilience Hub provides real-time monitoring and visibility into your applications and workloads, enabling you to identify issues and opportunities for improvement.
  - AWS DRS provides continuous monitoring of the primary and secondary locations, detecting issues and automatically triggering failover to the secondary location if necessary.
- **Customizable**:
  - Resilience Hub provides customizable dashboards, reports, and alerts, enabling you to tailor the solution to your specific needs and requirements.
  - AWS DRS allows customers to create customizable recovery plans that meet their specific business needs and requirements.
- **Centralized management**: AWS Backup provides a single pane of glass for managing backups across multiple AWS services, making it easier to track and manage backups using a fully managed policy-based service.

- **Automated process**: AWS Backup automates the backup process, reducing the risk of human error and ensuring that backups are performed consistently and reliably.
- **Cost-effective**: AWS Backup provides a cost-effective solution for data protection, reducing the need for on-premises infrastructure and minimizing storage costs.
- **Versioning**: AWS Backup provides versioning, allowing you to store multiple versions of your data and recover to a specific point in time.
- **Support for long-term archiving**: AWS Backup provides support for long-term archiving, allowing you to store data for extended periods of time at a lower cost.
- **Simplified administration**: AWS Backup simplifies backup administration, reducing the complexity and administrative burden of managing backups
- **Improved data protection**: AWS Backup provides improved data protection, reducing the risk of data loss and ensuring that your data is protected from corruption or deletion.
- **Enhanced business continuity**: AWS Backup enhances business continuity, ensuring that your business can continue to operate in the event of a disaster or data loss.
- **Improved resilience**: Resilience Hub helps you design and implement resilient architectures that can withstand failures and outages, reducing downtime and improving overall availability.
- **Reduced risk**: By identifying and remediating potential weaknesses and vulnerabilities, Resilience Hub helps you reduce the risk of outages and downtime.
- **Improved performance**: Resilience Hub provides recommendations for optimizing performance and responsiveness, helping you improve the overall user experience.
- **Cost optimization**: Resilience Hub helps you identify underutilized resources and opportunities for cost savings, reducing waste, and improving overall efficiency.
- **Improved collaboration**: Resilience Hub provides a centralized platform for collaboration and governance, enabling teams to work together to design, implement, and manage resilient architectures.
- **Automated replication**: AWS DRS automatically replicates applications and data from a primary location to a secondary location, ensuring that data is up-to-date and consistent across both sites.
- **Orchestration**: AWS DRS provides automated orchestration of the recovery process, including failover and failback, to minimize downtime and data loss.
- **Testing and validation**: AWS DRS provides features for testing and validating DRPs, ensuring that they are effective and up-to-date.

We have gone through different AWS managed Services that will enable you to manage resilience and build a resilient environment. Now, let's summarize everything that you have learned.

Summary

In this final chapter, we've explored the comprehensive suite of resilience services offered by AWS, including the AWS Backup service, Resilience Hub, and AWS DRS, all of which are underpinned by the AWS Resilience lifecycle framework.

These powerful tools and guidelines are designed to help you build a robust resilience process that protects your environment from disruptions and ensures business continuity. By mastering these services and frameworks, you now possess the knowledge and expertise to design, implement, and operate highly resilient systems that can withstand even the most unexpected challenges.

As you conclude this journey through the world of resilience on AWS, you're equipped with the guidance, tools, and best practices necessary to achieve success and drive business excellence in today's fast-paced, always-on, digital landscape.

Further reading

Here, you can find the links to expand your knowledge about the specific concepts referenced in this chapter:

- AWS Resilience Hub: `https://aws.amazon.com/resilience-hub/`.
- AWS Elastic Disaster Recovery: `https://aws.amazon.com/disaster-recovery/`.
- AWS Backup: `https://aws.amazon.com/backup/`.
- Resilience lifecycle framework: `https://docs.aws.amazon.com/prescriptive-guidance/latest/resilience-lifecycle-framework/introduction.html`.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Mastering Amazon EC2

Badri Kesavan

ISBN: 978-1-80461-668-0

- Discover how to create, manage, and select the right EC2 AMIs
- Explore load balancing and auto scaling with Elastic Load Balancing (ELB) and Auto Scaling Groups (ASGs)
- Study EC2 storage options and performance optimization
- Master monitoring and maintenance with Amazon Web Services (AWS) tools
- Understand containerization, serverless computing, and EC2 automation
- Get up to speed with migration, modernization, and compliance in EC2

AWS Cloud Projects

Ivo Pinto, Pedro Santos

ISBN: 978-1-83588-928-2

- Develop a professional CV website and gain familiarity with the core aspects of AWS
- Build a recipe-sharing application using AWS's serverless toolkit
- Leverage AWS AI services to create a photo friendliness analyzer for professional profiles
- Implement a CI/CD pipeline to automate content translation across languages
- Develop a web development Q&A chatbot powered by cutting-edge LLMs
- Build a business intelligence application to analyze website clickstream data and understand user behavior with AWS

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Building Resilient Architectures on AWS*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

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

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

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

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

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

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/978-1-83588-710-3`

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