Mastering Cloud-Native Microservices

Designing and implementing Cloud-Native Microservices for Next-Gen Apps

Chetan Walia

bpb

Mastering Cloud-Native Microservices

Designing and implementing Cloud-Native Microservices for Next-Gen Apps

Chetan Walia

www.bpbonline.com

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-699

www.bpbonline.com

Dedicated to

My mother, who inspired me to pursue my dreams and always supported me at every step of the life. Her love and guidance continue to inspire me. I dedicate this book to her, as a gratitude for her unwavering belief in my potential. Her memory will forever be cherished.

About the Author

Chetan Walia is a senior techno-functional leader and Technology Advisor with over 24 years of experience in Cloud Computing, Digital Transformation, and IT Modernization. He has a proven track record of delivering successful business outcomes through technology innovation, with expertise in Cloud Computing, Digital Transformation, Application Modernization, Telecom Domain, and Financial Planning.

Chetan is an expert in Delivery Management, pre-sales, and client engagement. He has successfully delivered complex Cloud Migration and Digital Transformation programs worth over USD 100 million, and his client portfolio includes Microsoft, AT&T, Amazon Web Services, and more. He is a sought-after speaker, thought leader, and author, with a reputation for grasping the big picture and developing solutions that achieve results.

Chetan has led pre-sales, scoping, and solution-framing sessions and formulated and implemented design principles to strengthen cybersecurity. He has extensive experience in Cloud consulting to build Vision, Strategy, Architecture, and Roadmap.

Chetan's delivery experience includes driving Enterprise Cloud Transformation at Scale, devising and facilitating the biggest Azure Cloud Migration, and establishing Managed Services engagement. He has managed accounts for leading companies in North America, Latin America, and the Asia Pacific regions.

In his personal life, Chetan is a passionate traveler who enjoys exploring the far corners of the world. He is an accomplished photographer with self-published coffee table books based on his own expeditions. He is also passionate about mountains and has experience in high-altitude trekking and mountain-biking.

About the Reviewer

Karthikeyan Shanmugam is an experienced Solutions Architect professional with about 22+ years of experience in the design & development of enterprise applications across industry domains. Currently he is working as Senior Solutions Architect at Amazon Web Services where he helps customers build scalable, secure, resilient and cost-efficient architectures on AWS. Prior to that, he has worked in companies like Ramco Systems, Infosys, Cognizant and HCL Technologies.

He is author of Kubernetes book ' **IoT Edge computing with MicroK8s**' published with Packt. His specializations include cloud, cloud-native, containers and container orchestration tools such as Kubernetes, IoT, digital twin and microservices domains and has obtained multiple certifications from various cloud providers.

He is also contributing author in leading journals such as InfoQ, ContainerJournal, DevOps.com, TheNewStack & Cloud Native Computing Foundation (CNCF.io) blog.

His articles on emerging technologies (includes Cloud, Docker, Kubernetes, Microservices, Cloud-native development, etc.) can be read on his blog upnxtblog.com.

Acknowledgement

I would like to express my heartfelt gratitude to the following individuals who have played a significant role in the creation of this book.

First and foremost, I want to thank my father, Sh. Sarabjit Walia, for his firm belief in thinking big and pushing me to reach for the stars. His visionary mindset has been a constant source of inspiration.

I would like to take this opportunity to express my heartfelt gratitude to my sister, Dr. Jyoti, and her husband, Sangram Singh Sandhu. They are avid readers and have extensive collections of books that inspired me to write a book and become part of their book collection.

I am deeply thankful to my brother, Amit Walia, and his wife, Manmeet, for their firm support, encouragement, and motivation at every step of this endeavour. Their belief in my abilities and their constant presence has been invaluable.

Lastly, I would also like to express my appreciation and love to my nephew, Sanmay Walia, a vibrant and energetic four-year-old, whose infectious enthusiasm and boundless energy have kept me motivated. His presence has brought joy and light into my life.

To all those mentioned above and to those who have been there for me, I am truly grateful. Thank you all from the bottom of my heart.

Preface

Microservice architecture is at the heart of cloud-native application architecture, and it has become a crucial tool for companies deploying cloud-based applications. Microservices-based cloud applications are becoming increasingly popular, and enterprises are looking for experienced architects, and DevOps experts who can build, run and develop them. In this book, you will learn how to break down a monolith, create microservices, overcome challenges, and strategize for cloud adoption. '**Mastering Cloud-Native Microservices**' is a guide to help you understand design and implementation steps using industry best practices and design patterns. In a practical case study approach, we will review challenges and solutions faced while identifying and implementing Cloud-native Microservices design patterns.

In this book, readers will learn how to break down a monolithic application into smaller, independent microservices, which can be developed and deployed separately. One of the key benefits of microservices-based cloud applications is that they are designed to take advantage of the elasticity, resiliency, and flexibility of the cloud. The book explores how microservices-based cloud applications can achieve these goals, and provides readers with a comprehensive understanding of the cloud-native concept.

The book is written in an example-driven approach, which makes it easier for readers to understand complex concepts. The book includes case studies that demonstrate how microservices-based cloud applications can be used in real-world scenarios, and provides readers with practical guidance on how to develop and deploy these types of applications.

Key Features:

Comprehensive Coverage: The book covers a wide range of topics related to cloud-native microservices adoption, including modern application design principles, microservice adoption frameworks, design patterns for microservices, cloud-powered microservices, inter-service communication, event-driven data management, the serverless approach, security by design, and cloud migration.

Case study-based approach: The book uses case studies to provide real-world examples of microservices implementation and best practices. This approach helps readers understand how to apply the concepts to their own projects.

Practitioner View: The book provides a practitioner's perspective on cloud-native microservices adoption, making it useful for solution architects, solution experts, pre-sales, and techno-functional roles. It helps readers to understand the challenges and benefits of adopting cloud-native microservices, and how to apply these principles in real-world scenarios.

This preface provides an overview of the chapters you will explore throughout this book, offering a glimpse into the valuable knowledge and insights you will gain.

Chapter 1: Cloud-Native Microservices- In this chapter, we delve into the world of cloud-native microservices, discussing their adoption in modern application architecture. We explore key principles, challenges, and the adoption framework for cloud-native microservices. Five industry success stories demonstrate the transformative power of cloud-native microservices.

Chapter 2: Modern Application Design Principles- The Chapter focuses on the design principles necessary for building resilient, scalable, and performant modern applications. We delve into the Twelve-Factor App methodology and explore design principles for availability, observability, security, and more.

Chapter 3: Microservice Adoption Framework- This chapter provides a structured approach to adopting microservices, covering strategies for breaking down monolithic applications, designing microservices, and building resilient systems. We explore enabling technologies such as Docker and Kubernetes, emphasizing the importance of technology adoption and DevOps processes.

Chapter 4: Design Patterns for Microservices- This chapter delves into essential design patterns for microservices, including integration, database management, observability, and cross-cutting concerns. By understanding and implementing these patterns effectively, you can build scalable and maintainable microservices that meet modern application architecture requirements.

Chapter 5: Cloud-Powered Microservices- In this chapter, we explore the powerful combination of microservices and cloud services. We discuss key design patterns that enhance the capabilities of cloud-powered microservices, such as data management, design and implementation, messaging, and reliability.

Chapter 6: Monolith to Microservices Case Study- The Chapter takes a deep dive into the practical aspects of transitioning from a monolithic architecture to microservices. It explores the challenges faced by legacy systems and provides

effective strategies for updating them. The chapter also covers successful database migration and showcases case studies of practitioners who have implemented microservices.

Chapter 7: Inter-Service Communication- In this chapter, the core concepts are of inter-service communication in microservices architecture. It covers different communication models, including synchronous and asynchronous communication, event-driven communication, and service mesh. The chapter highlights the importance of effective communication patterns for building complex microservices architectures.

Chapter 8: Event-Driven Data Management- The Chapter provides an in-depth discussion of event-driven data management for microservices. It explores technologies like event sourcing and CQRS, event-based data replication, validation, integration, access control, and lineage. The chapter explains how event-driven architectures enable communication between decoupled services and how events can be used to implement business transactions.

Chapter 9: The Serverless Approach- The Chapter explores the serverless approach to microservices development. It covers serverless architecture, frameworks, function-as-a-service platforms, edge computing, monitoring and logging, security, and best practices for serverless microservices development. The chapter showcases case studies of successful serverless microservices implementations.

Chapter 10: Cloud Microservices - Security by Design - The Chapter focuses on building secure microservices through a security-by-design approach. It covers practices for authentication, communication, and data security, container security, monitoring, compliance, infrastructure security, threat detection, and continuous security monitoring. The chapter addresses common security concerns and provides guidance on ensuring the confidentiality, integrity, and availability of microservices-based architectures.

Chapter 11: Cloud Migration Strategy- The Chapter serves as a comprehensive guide to the cloud migration journey. It covers the goals, principles, strategy, and lifecycle stages of cloud migration. The chapter provides an overview of the assessment, planning, design, execution, testing, cutover, and post-cutover stages, highlighting best practices for a successful migration.

I hope that this book will serve as a valuable resource, equipping readers with the knowledge and practical guidance needed to adopt and implement microservices successfully. Each chapter aims to provide in-depth insights, real-world examples, and best practices to ensure a comprehensive understanding of microservices architecture and its related concepts.

Let's embark on this exciting journey into the world of microservices and discover the immense potential it holds for modern application development.

Coloured Images

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/bqagct1

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

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 **business@bpbonline.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 **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights 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.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Chapter 1
Cloud-Native Microservices

Innovate at Scale with Cloud-Native Microservices

Introduction

Microservices have gained popularity due to their agility and cloud adoption readiness. The combination of microservices and cloud computing has become a key in enterprise architecture, and enterprises continue to reap the benefits of cloud computing. This chapter will cover the industry trends (with short case studies) and how Cloud-native Microservices adoption helping next generation applications. We will discuss the biggest challenges faced by organization when adopting microservices right from culture to technical complexity. Best practices to resolve these challenges and to have a seamless cloud adoption and microservices architecture implementation will be covered throughout this book.

- *According to Technavio the cloud microservices market share is expected to increase by USD 1.59 billion from 2021 to 2026, and the market's growth momentum will accelerate at a CAGR of 25.1%.*
- *According to the Microservices Architecture Market Research Report the Microservices Architecture industry is projected to grow from USD 5.49 Billion in 2022 to USD 21.61 Billion by 2030, exhibiting a compound annual growth rate (CAGR) of 18.66% during the forecast period (2022 – 2030).*
- *According to the 2021 Cloud Native Computing Foundation (CNCF) survey, 92% of respondents reported that they use containers in production, up from 84% in 2020.*

The Cloud Microservices market is growing worldwide, and that is no secret anymore. A large number of companies are upgrading their product portfolios so that they are not only cloud-ready but cloud-native with microservices architecture. The trend is in favour of cloud-native microservices, and organizations that move quickly will have an advantage. There have been tremendous success stories across all industry verticals, including retail and e-commerce, healthcare, telecom/media and entertainment, BFSI, government, and manufacturing. Certainly, there are challenges with respect to investment OPEX/CAPEX, Competitor Landscape, Timelines, ROI for such migrations and modernizations. It is very important for a company to comprehend all the aspects including the risk involved in order to strategize better. Therefore, the key question we should ask yourself would be:

"Cloud-native Microservices" Is it the right choice for your next application?

Companies have shifted from bare-metal infrastructures and monolithic architectures to microservices and container-based architectures in today's fast-paced environment. A better, more flexible, and scalable way to work is being established with cloud native microservices. Organizations can add new application components without revising entire applications or waiting for a release window. In other words, we are able to create new microservices quickly to achieve better results.

In this book, we will examine all aspects of cloud-native microservices, including scalability, flexibility, and resilience. By breaking down applications into small, loosely-coupled services that can be developed, deployed, and scaled independently, cloud-native microservices architecture can help organizations deliver applications faster and with greater reliability. However, adopting cloud-native microservices requires a significant shift in application development, deployment, and management practices, and may not be the best choice for every application. It is imperative to evaluate your specific needs and goals before making a decision. The purpose of this book is to provide you with the right tools, knowledge, and resources to plan, build and implement an optimal roadmap for your business.

Structure

In this chapter we will discuss following topics:

- Understanding the cloud native microservices
- Adopting cloud-native microservices
- Capability maturity level model
    - **Focus area:** People, Process and Knowledge to achieve End-to-end accountability

    - **Focus area:** Technology and Design Maturity for enabling Zero-touch operations
- Play Book for cloud-native microservices adoption
- Key principles of microservices
- **Short case study 01:** Snap on AWS
- **Short case study 02:** Wynk Music App
- The biggest challenges with microservices adoption
- **Short Case Study 03:** UPWARD, Inc.
- **Short Case Study 04:** The Government of India Powers a Population-Scale Vaccine Drive
- SWOT analysis for your application stack
- **Short Case Study 05:** IMDb Video Team Builds Strategies for the Future
- Conclusion

Objectives

The purpose of this chapter is to provide you with an overview of the 'Cloud-Native Microservices' adoption framework and why it's the right choice for your next application. To bring the focus back to the three pillars of any transformation, we will discuss the capability maturity level model I devised. We will also talk about how we can move to the next level, whether it's through people, process, or technology. We will discuss the Cloud-native Microservices adoption framework playbook for successful digital transformation in detail. We will be covering five industry success stories to understand the importance of Cloud-Native Microservices and how they can change the game.

Understanding the cloud native microservices

So the key question to start with would be "Is it the right choice for your next application or enterprise?". Rather than being a monolithic entity, cloud-native microservices architectures develop large applications using loosely coupled microservices. A microservice is a small, autonomous, self-contained software component, organized around a business domain, which allows each part to be easily monitored, tested, and updated without affecting the others, enabling greater speed and agility in business and operations.

For any organization to make it a right choice, it is important to develop a strategy with strong governance before moving forward in terms of guiding principles, migration/modernization goals, and business priorities. The technology stack, cloud provider, partners, tools, and so on will be determined by the core strategy and target business value.

In microservices, pieces of code are logically separated so that they can run independently with as few dependencies as possible. However, individual microservices can still communicate with one another and work together to create a complex application. Using self-contained microservices reduces dependencies and the need to coordinate changes across services. As a result, one failed service will have less impact on the entire application.

Containers go well with cloud native microservices and we must plan accordingly when defining a key strategy for such a transformation. The use of containers allows developers to work without constraints imposed by hardware components because they are software-only solutions. Microservices can be designed and tested independently, each performing a specific function.

Likewise, microservices also offer many advantages, but for a successful digital transformation, we need to look beyond technology. It is only through understanding current capability maturity levels and how to reach desired maturity levels encompassing people, process, and technology that we will be able to answer our main question: Is it the right choice for your next application? Let us start with the cloud adoption framework and capability maturity level model.

Adopting cloud-native microservices

Cloud adoption framework required to help clients/organizations build, migrate, modernize, operate, expand, and optimize their applications, infrastructure, data, and analytics in the Cloud. Technology upgrades are only one aspect of the business value equation, but delivering the right value is the key. Hence, we need to consider all aspects like people, process, and technology in our framework.

Next generation application and digital transformation requires industry leaders to rethink their business priorities, strategies, and operations to ensure continued success and value-based delivery. For example, as illustrated in *Figure 1.1:* Key drivers for Cloud-native Microservices adoption can be achieved through a planned adoption.

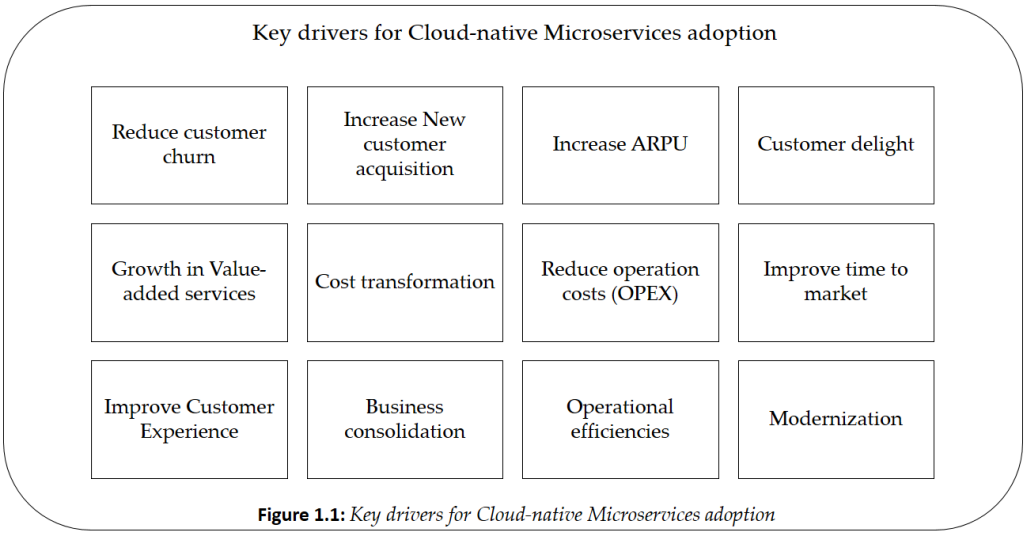


Figure 1.1: *Key drivers for Cloud-native Microservices adoption*

*ARPU is a metric that stands for average revenue per user

*Opex (operational expenditure) is the money an organization spends on an ongoing, day-to-day basis to run its business.

Cloud-native microservices adoption can help achieve the following key drivers:

- **Reduce customer churn:** By leveraging cloud-native microservices, organizations can develop and deploy new features and functionality faster, leading to a better customer experience and reduced churn. For example, a mobile banking application can use microservices to provide customers with a seamless experience across multiple devices and platforms, enabling them to easily access their accounts, make payments, and view their transaction history.
- **Increase new customer acquisition:** Cloud-native microservices can help organizations deliver new products and services faster, leading to increased customer acquisition and market share. For instance, a streaming service can use microservices to create personalized content recommendations for new users, helping to increase engagement and retention.
- **Increase ARPU:** By delivering more personalized and relevant services, organizations can increase **average revenue per user** (**ARPU**) and drive growth. For example, a retail company can use microservices to create targeted promotions and loyalty programs for customers, driving repeat purchases and increasing ARPU.
- **Customer delight:** By leveraging cloud-native microservices, organizations can create innovative, user-friendly experiences that delight customers and

differentiate themselves from competitors. For instance, a healthcare provider can use microservices to offer a virtual health platform that allows patients to easily schedule appointments, access medical records, and connect with doctors in real-time.

- **Growth in value-added services:** Cloud-native microservices can enable organizations to develop and offer new value-added services that enhance their existing offerings. For example, a telecommunications provider can use microservices to create an AI-powered virtual assistant that helps customers troubleshoot technical issues and provides personalized recommendations for new services.
- **Cost transformation:** By adopting cloud-native microservices, organizations can reduce costs and improve efficiency by leveraging automation, scaling on demand, and reducing infrastructure overhead. For instance, a logistics company can use microservices to automate shipping and logistics processes, reducing the need for manual intervention and saving on operational costs.
- **Reduce operation costs (OPEX):** Cloud-native microservices can help organizations reduce operational costs by providing greater visibility, control, and automation. For example, a financial services company can use microservices to automate fraud detection and prevention processes, reducing the need for manual intervention and saving on operational costs.
- **Improve time to market:** Cloud-native microservices can enable organizations to deliver new products and services faster, reducing time to market and improving competitiveness. For instance, a software company can use microservices to create a cloud-based platform that enables customers to quickly and easily deploy and manage applications.
- **Improve customer experience:** By leveraging cloud-native microservices, organizations can create more personalized, intuitive, and responsive experiences that improve customer satisfaction and loyalty. For example, an e-commerce company can use microservices to provide personalized product recommendations, real-time inventory updates, and streamlined checkout processes, improving the overall customer experience.
- **Business consolidation:** Cloud-native microservices can enable organizations to consolidate disparate systems and applications, improving data management, reducing complexity, and enhancing agility. For instance, a financial services company can use microservices to integrate and consolidate customer data from multiple sources, enabling more accurate and efficient risk management.
- **Operational efficiencies:** Cloud-native microservices can help organizations streamline operations, automate manual tasks, and reduce complexity,

leading to greater efficiency and cost savings. For example, a healthcare provider can use microservices to automate patient intake and registration processes, reducing wait times and improving the overall patient experience.

- **Modernization:** Cloud-native microservices can help organizations modernize legacy systems and applications, enabling them to leverage new technologies and stay competitive. For instance, a manufacturing company can use microservices to modernize its production processes, improving efficiency and quality while reducing costs.

We have mentioned business value several times, but how to achieve the same or step up capability maturity in terms of end-to-end accountability for people, process, and knowledge should be part of our strategy. Furthermore, achieving zero-touch operations with technology design maturity will be a key investment in architecture, operations and monitoring, delivery, provisioning, site reliability engineering, and security and compliance. We will be covering all these aspects throughout this book and how you can enable your organization or application to gain these benefits through Cloud-native Microservices adoption.

Capability maturity level model

As illustrated in *Table 1.1: Capability Maturity Level Model* will help and provide direction on how to improve overall capability maturity level in any of the organization covering all key aspect i.e. People, Process, and Technology.

Focus Area		**Capability Maturity Level**			**Value Delivered**
		1	**2**	**3**	
People, Process and Knowledge	**Team**	Cross-functional teams	DevOps	DevSecOps	**End-to-end accountability**
	Process	Agile	Design thinking / process blueprint	Process automation	
	FinOps	Workload cost predictability	Cost Optimization	Cloud ROI	

Technology and Design Maturity	**Architecture**	Client Server	Microservices	Cloud-native or Cloud Agnostic	**Zero-touch operations**
	Operations and Monitoring	Alerting and Monitoring	Self-healing, and Preventive AI/ML	Data insights on cloud and Observability	
	Delivery	Periodic Release	CI/CD Automation workflow	Complete Automation	
	Provisioning	Scripted	Infrastructure as a code	Shift left -- Optimization	
	Site Reliability Engineering	Service availability	Scalability and reliability	Performance Matrix	
	Security and Compliance	Monitor the Environment	Control the Access, Protect the Data, and Secure the network	Security by Design	

Table 1.1: *Capability Maturity Level*

Our basics should be clear at a high level or to begin with. Every organization would like to be next generation ready and to achieve the same we need to upgrade/upskill all three key pillars i.e. people, process and technology. In the following chapters, we will discuss technology-side design patterns in detail to help you implement and modernize your application stack. The purpose of this chapter is to understand the big picture and the approach one should take even before writing the first line of code.

Capability maturity level model (*Table 1.1*) created for explaining organization maturity levels for cloud adoption is different from the **Capability Maturity Model** (**CMM**) methodology which is used to develop and refine an organization's software development process. CMM talks about five maturity levels that can be developed incrementally from one level to the next. Skipping levels is not allowed/feasible.

- Level 1 - Initial
- Level 2 - Repeatable
- Level 3 - Defined
- Level 4 - Managed (Capable)
- Level 5 - Optimizing (Efficient)

Logically these steps are very much applicable to every activity we perform. Right now we do not intend to go in details of CMM but it is recommended to review them.

Let's review the process / best practice to improve our Capability Maturity level for each of these focus area (refer: Table 1.1: Capability Maturity Level)

Focus area: people, process and knowledge to achieve End-to-end accountability

- Team maturity is critical aspect and idea is to fast-forward from traditional waterfall development to a DevOps | DevSecOps model to have end to end accountability. An automation strategy for application and infrastructure development and management needs to be adopted and accordingly team structure, roles and responsibilities will need to be changed. Similarly, moving to cloud will bring in some of the cybersecurity challenges and hence eventually we should adopt DevSecOps practice. For example, some of the basics things like knowledge management, resource training and upskilling for future needs will help in moving to next level.

- **FinOps:** It is about optimizing cloud spending by workload cost predictability, and cost Optimization. Following definition is from FinOps.org. We will be discussion more about FinOps in next chapter. As illustrated in *Figure 1.2:* Cloud FinOps Defined as maximizing the business value from the resources deployed and to and maintain financial accountability for cloud services.

Cloud FinOps Definition

"FinOps is an evolving cloud financial management discipline and cultural practice that enables organizations to get maximum business value by helping engineering, finance, technology and business teams to collaborate on data-driven spending decisions."

At its core, FinOps is a cultural practice. It's the way for teams to manage their cloud costs, where everyone takes ownership of their cloud usage supported by a central best-practices group. Cross-functional teams in Engineering, Finance, Product, etc work together to enable faster product delivery, while at the same time gaining more financial control and predictability.

** Above reference is from*
*https://**www.finops.org**/introduction/what-is-finops/*

Figure 1.2: *Cloud FinOps Definition*

For example, many of the organizations end up spending more on cloud services compared to their on-premises footprints cost. Idea is to control costs and deliver business value. It is a financial management side of your Cloud adoption. You should be able to take the decision on underutilized resources, predict and plan spend.

- Process maturity is heart of any transformation success. In order to improve quality, cost, and delivery time, businesses should focus on optimizing processes. Most of the time, there are issues because we view it as a software development process, a business transaction process, or some other improvement to a function. Business processes need to be enhanced end-to-end to achieve the desired competitive edge. Processes should be measured, automated, and continuously improved. For example, we need to have value driven process to achieve year on year continuous improvement. High level of automation to support zero down time, zero touch operation, and fast track deployments. Need to progress from Agile way of working to Design thinking/process blueprint.

Focus area: technology and design maturity for enabling Zero-touch operations

- **Architecture:** Enterprise architect practice evolved overtime from being almost non-existent to reactive that focused only on ad hoc technical issues. Next stages were functioning that is a business-outcome-driven **Enterprise Architect** (**EA**) practice; Integrated practice delivering business value and is repeatable. Finally, ubiquitous enterprise architecture has become the organization's default mode of operation. We have seen organization adopting Client Server architect for earlier legacy days to now Microservices based Cloud-native or Cloud Agnostic design principles. For example, Modern application have unique requirements to innovate faster, improve performance, security, and reliability, all while lowering their total cost of ownership. We will have a dedicated chapter on modern application design requirement but for now we need to understand it is important for an organization to have a design maturity at architect level.
- **Operations and monitoring:** An operational monitoring strategy needs to be in place starting from types of data to collect to self-healing, and preventive AI/ML Data insights on cloud. Idea is to have system matured enough to provide leading indicators of an outage or service degradation and act to prevent the same. For example, we need to have a system to capture metrics, traces, and logs for enabling effective observability to understand why part of the incident and with that we will be able to eliminate its reoccurrence. Data insights on cloud will help us for taking timely action for Self-healing, and Preventive AI/ML eventually getting to a stage of zero touch operations.

- **Delivery:** We have seen the time of periodic release cycles and in fact it is still operational in many of the organizations. Having code release once a month used to be a normal norm with hotfixes getting released in-between and then having code merge issues. Agile software development approach is to accelerate time to market and improve code quality achieved through CI/CD Automation Workflow. For example, Developers gain the most from continuous integration because it allows them to test their code automatically and continuously integrate it with the code of their colleagues. Business users benefit from Continuous Delivery as soon as code has been accepted at the CI stage and has been tested logically. Continuous deployment allows code that has been accepted in the CI/CD cycle to be seamlessly pushed into production.
- **Provisioning:** Infrastructure as code (IaC) uses the DevOps methodology and versioning for managing and provisioning infrastructure through code instead of through manual processes. Developers can deploy applications without manually provisioning servers, operating systems, storage, and other infrastructure components with IaC, which automates infrastructure provisioning. For example, Chef, Puppet, Ansible, Terraform, AWS CloudFormation are some of the scripting tools used for building infrastructure pipelines. These tools are well suited to CI/CD automation workflows.
- **Site Reliability Engineering:** By implementing **Site Reliability Engineer** (**SRE**) practices, software systems become more reliable in critical areas such as availability, performance, latency, efficiency, capacity, change management, monitoring, emergency response, and incident response. For example, SRE practice will ensure service availability, scalability and reliability with performance matrix for tracking and trend analysis. It is a process of proactively writing code and developing internal tools and applications for services to combat reliability and performance concerns for a seamless production operation.
- **Security and compliance:** With Cloud adoption cyber security took centre stage and we need to have a strong security and compliance mechanism in place to Control the Access, Protect the data, and Secure the network, monitor the environment. We will have a dedicated chapter covering 'Security by Design' to understand the approach one should need to take while planning any such a digital transformation journey. For example, implementing 256-bit AES encryption for your data at rest and TLS 1.2 data in transit by design would enhance security. Encryption at rest and in transit means that your data is fully encrypted in any situation.

Play book for cloud-native microservices adoption

For any successful digital transformation, we need to have a defined road map in place. Whether it's enterprise wide modernization or even if we want to venture into newer tech like Metaverse technology, Blockchain, and the like. Key principles towards the approach or the play book will remain same. Below Play-Book is more aligned to this book and focusing to Cloud adoption and Microservices. In order to stay relevant in today's ever-changing environment of Volatility, Uncertainty, and Complexity, modern businesses need to realign, reconsider, re-plan and re-prioritize their road-map. This play book will give you a direction to move fast and fail fast (in case it's not the right decision). For example, as illustrated in *Figure 1.3:* Play Book for Cloud Native Microservices adoption should cover these aspects. We have discussed them in detail with example for you to have a clarity for creating your own roadmap.

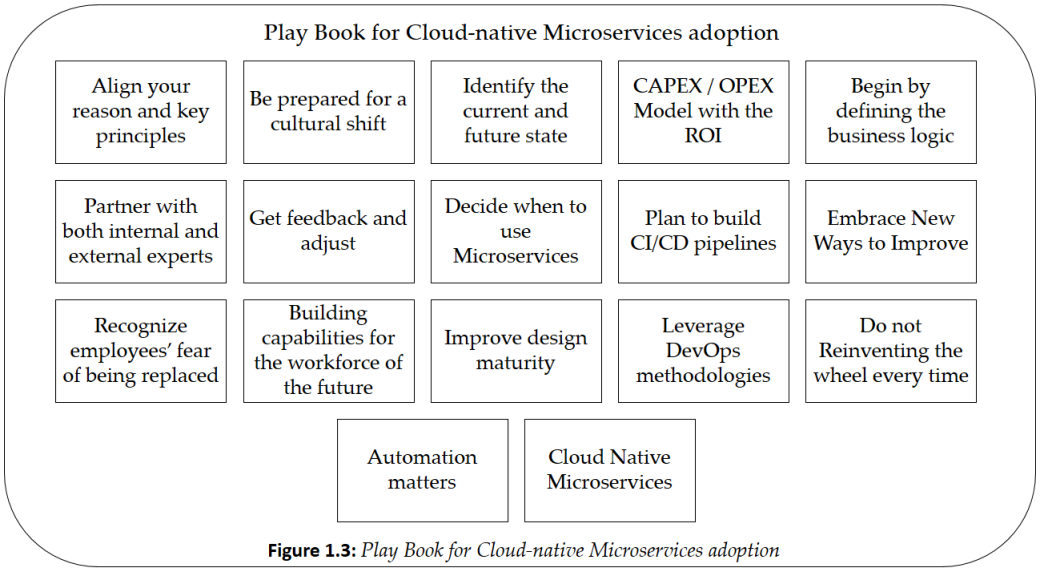


Figure 1.3: *Play Book for Cloud-native Microservices adoption*

- **Align your reason and key principles:** Analyse how digital transformation adds value to your business. Figure out your business strategy (phase-wise approach) before you invest in anything. For example, optimize applications to reduce resource (hardware / software) consumption and provide dynamic elasticity could be one of the key principle. Idea is to have a clear vision and a high level strategy with goals in place even before you start.
- **Be prepared for a cultural shift:** Digital transformation must put people at the centre as it is not just a technology adoption. Repeating it again as it is one of the main miss causing failures, delays, and cost overrun. For example,

Train and certify workforce according to future mode of operations. Good example would be FinOps or Security compliance on Cloud. Idea is to have participation and involvement of your complete workforce and prepare them well for any such change.

- Identify the current and future state of your organization's maturity in terms of people, process, knowledge, and technical design maturity. Phase-wise plan to upgrade one or more components to achieve end-to-end accountability with Zero-touch operations on Cloud. For example, creating phase wise roadmap for each of the components (the way given in previous table) for eventually achieving goals and strategy defined by organization. Ideas is to work in parallel on overall maturity covering people, process, and technology.
- CAPEX / OPEX Model with the ROI needs to be well defined and accordingly, the transformation approach should be finalized (cap and grow, evaluation, and revolution). For example, application do not work in isolation and hence we need to build move groups for specific set of apps related to technical or business domain. We can plan to implement cap and grow methodology where we cap any further expenditure on perm and grow apps on Cloud. Similarly, evaluation or revolution approach might fit better for in certain scenarios/organizations.
- Begin by defining the business logic of the future app and breaking it down into large components which can be split out into more Microservices later using design patterns. For example, Domain-Driven Design should be the starting point for breaking monolith into logical microservices to have failure isolation, decentralization and other benefits. We will be discussing multiple examples and different option to achieve desired results throughout this book.
- **Partner with both internal and external experts:** Do more and do it faster with partners that share your vision. Leverage insider knowledge about what works and what doesn't in their domain. For example, it is critical to get outside in view and vice versa. We need to review industry best practices and success stories while planning any such digital transformation journey.
- **Get feedback and adjust:** Need to design the customer experience from the outside in by obtaining extensive and in-depth input from the customers and other stakeholders. For example, the key advantage of microservices would be the ease of upgrading and adjusting without impacting complete application. Idea is to break your transformation plan in multiple phases and conduct lesson learnt session and improve for next iteration. So you explore, experiment, test and optimize your design for next set of applications.

- **Decide when to use Microservices:** Companies started with a monolith that got too big and was broken up for the scaling demands of its rapidly growing customer base. Sometimes it is a great start for a start-up environment. For example, a detailed discovery and assessment of your application set is required to understand the complexity involved in any such migration. We have seen cases where legacy systems have issues and need to be modernize but the cost and risk involved is too high as legacy (source architecture) is not clear to anyone or in any of the documentation.
- Plan to build continuous integration and deployment pipelines with "one-button" deployment and release setup. To orchestrate services, consider Docker and Kubernetes. For example, build continuous delivery (CI/CD pipelines) and implement DevOps culture for your organization. There are many tools to support but the key is in mind-set for adopting this approach.
- Embrace New Ways to Improve: Bring start-up culture for agile decision-making, rapid prototyping, and flat structures. For example, we recommend POC approach for every new tech stack or for every new group of application you want to upgrade. Decide on the target architect and build POC (without automation or pipelines) to validate the concept and then start building Cloud native microservices. Idea is to fail fast and improve.
- **Recognize employees' fear of being replaced:** Success would be difficult when employees perceive Digital Transformation as a risk to their jobs. For example, partner with them as they know the best about the applications in scope. Their inputs clubbed with the organization's vision which will help in deciding application deposition and dependency with other apps in the ecosystem. Idea is to grow together in a close collaboration.
- **Building capabilities for the workforce of the future:** Train and upskill your workforce. For example, Train your workforce for future business and technology needs. It is much easier to learn newer technology compared to understand the whole business from scratch. Idea is to leverage their years of experience and understanding of issues to build next generation applications. Dedicated effort to upskill your workforce will help for a seamless transformation.
- **Improve design maturity:** Invest in technology solutions that will scale with your long-term goals. For example, Identify application **Bill of material** (**BOM**) for Cloud-native Microservices application. List of approved software's with their minimum version for any application deployment on Cloud while keeping in consideration your security, and scalability goals.
- **Leverage DevOps methodologies:** Plan to use DevOps, agile software development methodologies and that will help significantly in your Cloud transformation journey. For example, improve collaboration across

DevOps, app dev and IT operations teams across your organization and accordingly the tools and technology adoption will progress to support DevOps. Microservices, Cloud native serverless application architectures, and container management goes very well with DevOps approach.

- **Do not reinvent the wheel every time:** We have seen tremendous success when you follow a factory delivery model rather than isolated functional or development teams. Idea is to have a repeatable, standardized processes and technology framework to achieve higher velocity and a predictable outcome. For example, you can have a factory with multiple scrum teams responsible for cloud migration and application modernization to use cloud native microservices architect. They will have a better performance when workload assigned to teams would have similar tech stack (for instance, one set of team focusing on Windows work load and another set on Unix/Linux) that way they will be able to build common code repositories and repeatable processes.
- **Automation matters:** In continuation to the above point. We need to understand that automation matters a lot and it is nothing new to IT industry we have been writing scripts / programs from very initial days of industry to automate as much as possible. For example, the meaning of automation and expectation from automation is a lot different from initial days. Now we need immutable environments, serverless setup, PAAS / SAAS solutions, single click deployment, zero touch operations, multi-cloud setup, AI/ML embedded monitoring, IA Ops solution, and so on.
- **Last but not the least 'Cloud Native Microservices':** Design patterns, inter-service communication, event-driven data management, deployment strategy, and the like, to be well defined. All that will be covered in detail with examples and case studies but right now we need to be clear on why would business invest, what would be ROI, what all RISKS we should plan to mitigate, what would be our approach, roadmap, and timelines and finally, how all three key pillars that is people, process, and technology will progress together.

An enterprise can gain substantial business and operational benefits from microservices, but like any other technology, they are not guaranteed to succeed. Therefore, we need a strategy in place before moving to a microservice architecture and selecting suitable candidate applications and above point gives you a snippet on how to approach any of the large scale transformation.

Key principles of microservices

Microservices architecture is an approach to building software systems that involves breaking down a large monolithic application into smaller, independent services.

As illustrated in *Figure 1.4:* Key Principles of Microservices implementation and the same will be reflected in upcoming examples:

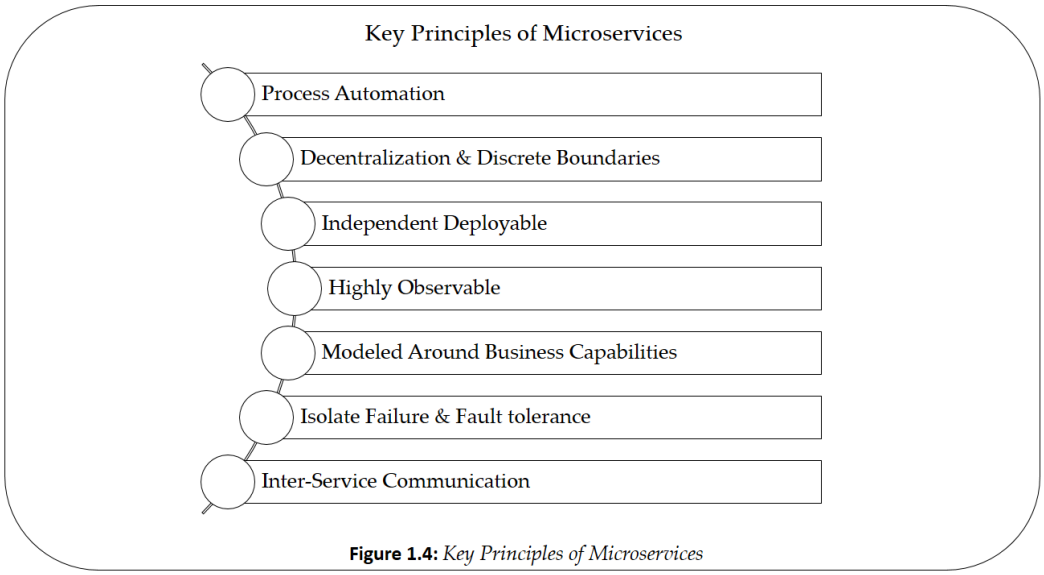


Figure 1.4: *Key Principles of Microservices*

- **Process automation:** Microservices architecture emphasizes automating the deployment, testing, and monitoring of services to streamline the development process. For example, an e-commerce application may automate the deployment of microservices responsible for order processing, inventory management, and payment processing. With DevOps tools such as Jenkins, Git, Docker, Kubernetes, and Ansible, microservice deployment, testing, and monitoring can be automated.
- **Decentralization & Discrete Boundaries:** Microservices are independent and self-contained services that operate within discrete boundaries. Each service should perform a single business function and should not depend on other services. This allows for greater flexibility and scalability, as well as reducing the risk of system-wide failures. For example, a video streaming platform may have separate microservices for user authentication, video encoding, and content delivery. To ensure independent and self-contained services with discrete boundaries, microservices architecture relies on containerization technologies like Docker and container orchestration tools like Kubernetes or Docker Swarm.
- **Independent Deployable:** Microservices should be independently deployable, allowing for faster development and deployment cycles. Each service can be developed, tested, and deployed independently, without affecting other services. For example, a ride-hailing service may have separate microservices for driver matching, ride tracking, and payment processing,

each of which can be updated and deployed without disrupting other services. Microservices can be independently deployed using **continuous integration** and **continuous deployment** (**CI/CD**) tools such as Jenkins, CircleCI, and TravisCI.

- **Highly Observable:** Microservices should be highly observable, meaning that developers and operators should be able to easily monitor and troubleshoot services in real-time. This requires detailed logging and monitoring of service performance and events. For example, a social media platform may monitor microservices for user authentication, post creation, and comment moderation to ensure that the platform is functioning as intended. To ensure highly observable microservices, logging and monitoring tools such as stack **Elasticsearch, Logstash, Kibana** (**ELK**), Prometheus, Grafana, and Jaeger can be used.
- **Modeled around business capabilities:** Microservices should be designed around business capabilities rather than technical concerns. This allows for a more efficient and effective development process that is aligned with business needs. For example, a healthcare application may have separate microservices for patient data management, appointment scheduling, and prescription management. Microservices can be modeled around business capabilities using **Domain-Driven Design** (**DDD**) and **Event-Driven Architecture** (**EDA**) principles. Tools such as Apache Kafka, RabbitMQ, and AWS EventBridge can enable event-driven architectures.
- **Isolate Failure and Fault tolerance:** Microservices should be designed to isolate failures and to be fault-tolerant. This means that if one service fails, it should not affect the entire system, and the system should be able to recover quickly from failures. For example, an online banking platform may have separate microservices for account management, transaction processing, and fraud detection, each of which should be able to function independently in case of a failure in other services. To isolate failures and ensure fault-tolerant microservices, technologies such as circuit breakers (Hystrix), health checks, and service meshes (Istio) can be used.
- **Inter-Service Communication:** Microservices should be able to communicate with each other seamlessly and efficiently. This requires a well-defined API that allows services to interact with each other. For example, a travel booking application may have separate microservices for flight booking, hotel booking, and car rental booking, each of which needs to communicate with the others to provide a seamless booking experience for the user. Inter-Service Communication: Inter-service communication can be facilitated through API gateways such as Kong, Apigee, and AWS API Gateway, and messaging protocols such as REST, gRPC, and GraphQL.

We know success stories from Amazon, Netflix, Uber, Spotify, Etsy, and many more for their Cloud adoption and microservices implementation. Let us discuss some relatively smaller but smart implementations and how they got benefitted for the same.

Short case study 01: Snap on AWS

This case study is one of many examples of start-ups leveraging cloud technologies and scaling up as required. With AWS pay-as-you-grow cloud computing, Snap avoided any upfront capital costs for building the platform. As shown in *Figure 1.5:* Snap on AWS case study snippet explaining their adoption story.

Snap on AWS

"Snap was born in the cloud — launching its flagship app, Snapchat, in 2011 on a cloud-native, monolithic architecture. As the app grew in popularity, Snap migrated to a microservices architecture on Amazon Web Services (AWS) to improve scalability, optimize availability, minimize latency, and reduce costs. On AWS, Snap now supports more than 306 million Snapchat users sending over 5.4 billion Snaps daily with 20 percent less latency than its prior architecture. Freed from managing infrastructure, Snap engineers can focus on developing new, unique offerings, such as Bitmoji TV, which renders users' Bitmoji avatars as the stars of personalized, animated videos in real time with the compute power of Amazon Elastic Compute Cloud (Amazon EC2) G4 instances. Snap continues to innovate on AWS, experimenting with new services and features to enhance visual communication and storytelling for its users."

** Above reference is from*
*https://**aws.amazon.com**/solutions/case-studies/innovators/snap*

Figure 1.5: *Snap on AWS Case Study*

What can we learn from this example?

- Cloud is a great option for a start-up as there is not upfront capital expenditure.
- There is a need for continuous optimization whether it is cost, performance, technology, business.
- Cloud native services will help your organization get maximum advantage of cloud adoption.
- Building low-latency, near real-time messaging architecture that handles over 10 million transactions per second is not efficient on any of the on premise data centre.
- In their phase two: Snap runs on Amazon EKS to evolve from a monolithic architecture to providing a secure, fast, and highly scalable micro services infrastructure.

- Migrating to Amazon DynamoDB is again not just lift and shift operation rather they re-architect the data for better scalability and performance.
- Despite the short timeline and high pressure, Security is ensured, and Continuous Integration and Continuous Delivery are used.

Short case study 02: Wynk Music App

In this case study, Wynk wanted to re-architect and re-build their existing monolithic application stack into a Cloud Native Microservices architecture to support a large customer base and expand it. Re-architecting is meant to create an open and loosely coupled integration landscape that can connect various new features that can be released within 2 to 4 weeks. For example, as illustrated in *Figure 1.6:* Wynk Music App case study snippet explaining their adoption story.

Wynk Music App by Airtel Raises the User Experience Bar with Amazon EKS

"Wynk Music—a leading music streaming service in India with 72 million monthly active users and more than 14 million tracks in its content catalog. To grow its music subscription revenue, Wynk sought to offer a high-quality search and discovery experience where users have easy access to their desired content with a single click. Wynk sought assistance from Amazon Web Services (AWS) to launch its new Content Discovery platform, where users are given real-time tailored music recommendations. To create an agile and horizontally scalable infrastructure to support the new Discovery platform, Wynk moved its monolithic applications to Amazon Elastic Kubernetes Service to easily start, run, and scale Kubernetes applications in the AWS Cloud. By running AWS EKS, the Wynk DevOps team has around 100 microservices live in production at any time, while reducing development time for new customer services from 2 weeks to 4 days. With a microservice-driven approach using AWS EKS, Wynk's DevOps teams can run, scale and test tasks independently and in parallel resulting in faster services creation."

** Above reference is from https://**aws.amazon.com**/solutions/case-studies/wynkmusic-eks*

Figure 1.6: *Wynk Music App by Airtel Case Study*

What can we learn from this example?

- Scalability is a key feature of Cloud and that helps where we need to multiply user base.
- They have implemented microservice (100 microservices in prod at any given point in time).
- Significantly reduced development and release time which is critical factor for time to market.
- No better way to manage such a large amount of events per second.
- Data Analytics used for generation recommendations.

- DevOps implementation for end to end ownership for each of the microservices.
- Used Amazon EKS to easily maintain Kubernetes clusters.
- Significantly saved on infrastructure provisioning because of auto scalability.

The biggest challenges with microservices adoption

By now we have seen some of the good examples of implementing Cloud-native microservices but it is critical to know the possible pitfalls. Let us discuss: **What could possibly go wrong?**

There are a lot more moving parts in microservices than in traditional applications, which requires new skills and new ways of working. An organization won't be able to realize the full potential of microservices unless they develop, deploy, run, and maintain them strategically. Here are some scenarios that could lead to failure.

- Cost of innovation and ROI
    - Certainly, we will need to invest time, money, and people for innovation anything and to take idea from POC stage to Production. ROI would matter for building a business case and the same is true for modernization of your workloads / applications. Usually ROI will be based on reduced costs, greater operational efficiency, and improved software developers' productivity.
- Why not invest in another SAAS product?
    - Application modernization is the process of updating old applications to make them more efficient, reliable, and profitable. We can achieve that by updating, rehosting, or replacing.
    - **Update**: Rewriting or redesigning the entire code or certain parts of the code to bring the app back to life.
    - **Rehost**: Migrating an application from a mainframe/legacy/on-prem datacentre to another infrastructure, usually the cloud.
    - **Replace**: Remove the legacy app and replace it with a brand new SAAS-based software system. Usually, this approach is beneficial when we consolidate and retire multiple legacy applications with a mature SAAS product.
- Increased resource usage and network communication

  - Due to their self-contained nature, microservices rely heavily on networks for communication. As a result, there may be slower response times (network latency) and an increase in network traffic.

- Difficult in Global Testing and Debugging
  - Because microservice-based applications are distributed across multiple servers and devices, testing and debugging them can be challenging. To test and debug an application effectively, you need to be able to access all the servers and devices in the system. The process can be challenging in large, distributed systems.

- Integration between different APIs, communication protocols
  - Communication overhead increases when an application is divided into multiple smaller modules. When handling requests between modules, developers must take extra care. It might be necessary to use an interpreter/converter if different systems communicate in different ways which will increase complexity.
  - Dependency management between services will required additional effort.
  - Microservices can create information silos if not planned properly.
  - Extensive monitoring and logging required.

- Skill gap is another big issue in our industry

  It's true for both legacy and latest technologies. Microservices implementation would require individual with both technical and domain understanding.

Basically, overall management, planning and execution is not an easy task as there are many moving parts right from Monoliths to Microservices design conversion, from Waterfall to DevOps adoption, Microservices + DevOps + Cloud + Cloud Native adoption, technology upgrading (leveraging open-source technologies), API integration.

Let us see some more examples as case studies:

Short Case Study 03: UPWARD, Inc.

In this case study, UPWARD wanted to accelerate its business growth by overhauling its entire infrastructure. They wanted to develop high-quality code, increase scalability, and improve release times while reducing costs. With **Infrastructure as Code** (**IaC**), UPWARD has been able to standardize infrastructure design and streamline all changes by using PaaS (Cloud-Native) and modernize applications

with a Microservices architecture. For example, as illustrated in *Figure 1.7:* **UPWARD** case study snippet explaining their adoption story.

UPWARD, eyeing enterprise markets at home and abroad, migrates service platform to Azure for cloud service innovating last one mile of sales

"UPWARD, Inc., a provider of UPWARD, a cloud service for sales engagement (SaaS) combining GIS and CRM, has migrated part of its service platform to Microsoft Azure. By changing from its IaaS-based architecture to a microservices architecture and combining the new architecture with Azure's PaaS, the company has gained the reliability required of an enterprise system and the speed and agility to respond to changes in the environment. The goal of the new platform is to provide a service that can be used in sales fields without being conscious about the complex functions of CRM."

** Above reference is from https://**customers.microsoft.com**/en-us/story*

Figure 1.7: *UPWARD Case Study*

What can we learn from this example?

- Rebuilding platform from IaaS to Azure PaaS to meet enterprise needs, aiming to improve security, scalability, and operability.
- Many organizations initially prefer IaaS approach to start cloud journey mainly because of their comfort level with somewhat similar to on-premises setup.
- Cloud native and microservices adoption gives much better flexibility, and scalability.
- By automating continuous integration pipelines and environment builds using infrastructure as code, you can achieve a quick and cost-effective development and release cycle.
- Low latency applications are critical to bringing more digital experiences to stakeholders, including end users in this case. In addition, it is essential to monitor across layers of applications.
- Microservices and PaaS solution works well for flexible scaling of applications and platforms.
- Seamless Worldwide scalability can be achieved with ease once you are on Cloud.

Short Case Study 04: The Government of India Powers a Population-Scale Vaccine Drive

In this case study, we will see how technology help in a fight against COVID-19. We have seen significant usage of technology right from contact tracking to seamless vaccination drives. In this scenario need to have a scalable solution that can be rolled out quickly cannot be fulfilled easily without Cloud adoption. As shown in *Figure 1.8:* The Government of India Powers a Population-Scale Vaccine Drive case study snippet explaining their success story.

The Government of India Powers a Population-Scale Vaccine Drive on AWS

"Government of India, needed a highly reliable, scalable, and resilient technical infrastructure to power a large-scale COVID-19 vaccination drive for India's more than 1.3 billion citizens. The result was a microservices-based, cloud-native architecture developed from the ground up. Using the elasticity and agility of AWS-managed solutions, the Ministry launched the Co-WIN application quickly at population scale. It scales in seconds to handle user registrations and consistently supports 10 million vaccinations daily. When application access opened to those in the 18–44 age group, unprecedented traffic volumes saw the solution scale in 1 minute from 6,000 requests per second to 46,000 requests per second.

Delivering fast scaling without the requirement to manage servers, the solution relies on Amazon DynamoDB, a fully managed, serverless, key-value NoSQL database designed to run high-performance applications at virtually any scale. The use of Elastic Kubernetes Service - a managed container service to run and scale applications—also helps to deliver performance and high availability."

** Above reference is from https://**aws.amazon.com**/solutions/case-studies/meity-gov-india-case-study*

Figure 1.8: *The Government of India Co-Win Vaccine Drive Case Study*

What can we learn from this example?

- Cloud native microservices proven to be best solution to develop a secure, resilient architecture and to deliver a solution that required high-performing, reliable, cost-effective infrastructure for rapid scaling to handle huge surges in demand.
- It is a smart user interface and providing a single source of truth for the administration and tracking of vaccinations. "Co-WIN made it easier for people to get vaccinated generate certificates, linking it with Aadhaar (individual identification number).
- Performance is excellent, and the application runs smoothly and flexibly (worked well for 1.3 billion Indians).

SWOT analysis for your application stack

We are going with the assumption that you are familiar with the basics of Cloud if not Cloud native with an understanding of Microservices at a high level. You might be implementing Cloud Native Microservices, or maybe you are looking for ways to start a digital transformation journey.

During the past few years, we have moved from the cloud age, when resources needed to run applications could be rented in the cloud as a service, to the cloud native age, where applications are built to maximize elasticity and resilience in the cloud by being purpose built and optimized. Although the advantages are clear, adoption is not straightforward as it requires re-structuring and re-designing of current applications and infrastructure. Need to build and implement distributed systems, microservices, containers, serverless, and other emerging technologies and architectures.

So we have started with our key question "Is it the right choice for your next application?"

It can be answered once we have a clear understanding of how this technology approach will impact my business objectives, what the investment would be, and when it would be implemented in terms of timelines.

SWOT analysis for an application stack deposition for a cloud-native microservices journey:

Strengths:

- **Scalability**: Cloud-native microservices can be scaled up or down easily to handle traffic spikes or sudden changes in demand.
- **Agility**: Microservices are designed to be modular and independent, making it easier to add new features or modify existing ones without affecting the entire application.
- **Resilience**: Cloud-native microservices are designed to be fault-tolerant, with automated failover and self-healing capabilities.
- **DevOps integration**: The use of DevOps practices in a cloud-native microservices environment enables teams to release software more frequently and with greater confidence.
- **Cost efficiency**: Cloud-native microservices are designed to use resources efficiently, which can lead to cost savings.

Weaknesses:

- **Complexity**: Managing a cloud-native microservices environment can be complex, especially as the number of services and components increases.
- **Inter-service communication**: As microservices communicate with each other via APIs, there may be a risk of performance issues or failure if the API isn't designed properly.

- **Security**: Cloud-native microservices environments can be vulnerable to security threats, such as unauthorized access, data breaches, or API attacks.
- **Overhead**: The use of container orchestration platforms like Kubernetes adds an additional layer of complexity and overhead.

Opportunities:

- **Innovation**: Cloud-native microservices provide an opportunity to innovate and create new services or features quickly and easily.
- **Competitive advantage**: By leveraging cloud-native microservices, organizations can gain a competitive advantage by delivering software faster and more efficiently.
- **Collaboration**: Cloud-native microservices enable teams to work together more closely and collaborate on code, which can lead to increased productivity and efficiency.

Threats:

- **Vendor lock-in**: If an organization relies heavily on a specific cloud provider or container orchestration platform, it may be difficult to switch to another provider or platform.
- **Adoption challenges**: The adoption of cloud-native microservices may face challenges, such as resistance from legacy systems, lack of expertise, or difficulty adapting to new processes and workflows.
- **Integration issues**: Integrating cloud-native microservices with existing systems and applications may be challenging, especially if they were not designed to work together.
- **Compliance and governance**: Cloud-native microservices may be subject to regulatory compliance requirements, such as GDPR or HIPAA, which can be complex to manage.

The goal of cloud native applications is to maximize the potential of cloud computing by optimizing their environments to achieve transformational business and digital outcomes and microservices go hand in hand when it comes to actual implementation. To understand better, we need to perform a SWOT analysis for the applications in scope and decide on their deposition in terms of refactor, re-platform, repurchase, re-host, relocate, retain, retire.

We need to understand that there is a big difference between application being Cloud ready and Cloud native. Cloud ready application is the one which can be deployed on cloud as Software used currently will work as is on Cloud environment at least as a IAAS option. Whereas Cloud native application will be utilizing economies of scale and designed for distributed computing. It is elastic, easy to deploy and

manage on demand, and resilient to failures. It is recommended to refer '*Table 1.1: Capability Maturity Level*' while evaluating your own setup / applications / data centre in scope.

As an example, let us take a look at a relatively simple case where moving workload and storage to the cloud seems obvious. In reality, it is still a complicated process involving multiple applications and hundreds of interfaces. So we need to perform a detailed discovery exercise for deciding eventual deposition of our set of applications.

Short case study 05: IMDb Video Team Builds Strategies for the Future

In this case study, IMDb needs a reliable, secure and a mechanism to search and recommend fast for a consistent engagement with user base. Cloud base Infrastructure and storage is in any case best fit solution and now they can introduce this customer base to newer services just by plug and play approach (adding new functionality as and when required). In *Figure 1.9:* IMDb Video Team Builds Strategies case study snippet explaining their adoption.

IMDb Video Team Builds Strategies for the Future Using AWS

"IMDb is a popular and authoritative source worldwide for movie, TV, and celebrity content, designed to help fans discover and decide what to watch. The IMDb consumer site has a combined web and mobile audience of more than 200 million monthly visitors.

The IMDb Originals & Events team of producers, editors, and motion designers creates extensive original video content each year. The team needed an infrastructure that could support access to its millions of digital raw media assets and archive them securely. The team was previously using on-premises content storage and web-based file hosting services for storage and distribution."

** Above reference is from*
*https://**aws.amazon.com**/solutions/case-studies/imdb-video-case-study*

Figure 1.9: *IMDb Video Team Case Study*

What can we learn from this example?

- **Amazon Simple Storage Service** (**Amazon S3**) used for a cost-effective, scalable archive solution to future-proof these assets.
- In this case it is sound like an easy decision but look at the volumes, number of events per day you will understand complexity.
- It is even more mission critical as fans are waiting for certain movie or supports event and hence timely delivery is crucial factor to manage.

- Redundantly store data on a number of devices across multiple Availability Zone is another design factor.

Let us go back to our discussion on how to decide on your applications deposition?

Cloud-first strategy and migration are not a destination, but a step towards innovation-friendly environments. Following section is from Cloud migration perspective as modernizing application to be Cloud native and microservices is one of the deposition category for you cloud strategy.

This scenario would sound similar once you analyse your application stack and on-premises data centre. We will experience following situations, once we start analysing our workload for categorization under eventual application deposition.

- A renewed digital strategy requires modernizing IT infrastructure provisioning as **infrastructure as code** (**IaC**).
- It involves re-designing and fixing decades of legacy architecture with limited documentation.
- Due to multiple other dependencies, timelines would be another issue.
- Re-platforming and refactoring accounted for XX% usually higher percent of all applications.
- Additionally, a X% of all legacy app were decommissioned, reducing technical debt.
- Modern SAAS applications were also used to replace legacy applications.
- Usually the existing workloads will be categorised under one of the popular 6R category i.e. Re-host, Re-platform, Re-factor/Re-architect, Re-purchase, Retire and Retain.
- While analysing workload / applications we need to define its complexity in term of t-shirt sizing and again that process needs to be mathematically designed based on number of interfaces, volumetric, technology stack, disaster recovery requirements and the like.
- Applications do not work in isolation and hence we need to create move groups of application that work as a unit or have strong dependencies between them. Therefore, their deposition plan, timelines for their modernization, migration, and cutover needs to be aligned.

Cloud migration is not just about moving your application stack to the cloud; it is an iterative process of optimization to decrease costs, improve performance and flexibility and reach the full potential of the cloud. The journey would be unique for

every organization, as there is no one-size-fits-all migration plan / strategy available. We can discuss industry best practices and success stories to define a roadmap for our own journey. Both aspects of Cloud-Native Microservices are covered in this book, whether it is application modernization or building/designing a new application based on the Cloud-Native Microservices design pattern.

Conclusion

Investing in People, Process, and Technology:
Will Drive Success in the Cloud Native and Microservices Landscape

Innovation and modernization are more needed than ever to meet rising demands and expectations from IT / Software teams. No matter whether you are modernizing your existing application stack or migrating / optimizing workloads to use cloud native or you are building a new set of applications directly on the cloud utilizing cloud native microservices principles. Another key aspect we have discussed it to invest in all three key pillars i.e. People, Process, and Technology as they have hard dependency on each other.

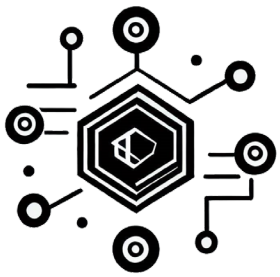
We will need to deep dive into many other areas in and around Cloud Native and Microservices for analysing, and finalizing an efficient solution. For example, Hybrid and Super Cloud (multi-cloud) application platform and now to balance between cloud-native and cloud agnostic. Similarly, other key area for a deep dive would be DevOps, Serverless, API management and Mainframe modernization. From this chapter take away would be the clarity around Capability Maturity Levels, Play Book for Cloud-native Microservices adoption framework, applications deposition strategy, and What all can we learn from those short case studies.

In next chapter we will deep dive into Modern Application Design Principles covering 12-Factor application methodology for building Cloud-native Microservices applications, Modern application design requirements.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



CHAPTER 2
Modern Application Design Principles

Building the Future with Modern Application Design Principles

Introduction

Modern application design principles refer to the set of best practices, techniques, and strategies used to design and develop software applications that can meet the evolving needs and requirements of today's enterprises. The goal of modern application design is to create software applications that are scalable, reliable, secure, and can be easily maintained and updated.

During uncertain times like the Covid crisis and war situations, modern application design principles become even more critical for businesses. Companies need to adapt quickly to changing market conditions and customer needs. Modern applications can help businesses to respond rapidly to changing market dynamics and customer requirements, enabling them to remain competitive and profitable. For example, during the Covid crisis, many companies shifted their operations online to reach customers at their doorsteps. This required modern applications that could provide seamless online experiences for customers, such as e-commerce platforms, video conferencing, and online collaboration tools. Companies that had invested in modern application design principles were better equipped to handle the sudden shift to online channels and were able to continue serving their customers.

Technologies such as Artificial Intelligence, IoT, 5G, Cloud Computing, and cybersecurity are critical components of modern application design principles.

These technologies help businesses to build applications that are scalable, reliable, secure, and can be easily maintained and updated.

In this chapter, we will review the 12-Factor application methodology for building Cloud-native microservices applications that offers portability, agility, and scalability. In addition to that we will discuss about additional key aspects / X-factors based on industry experience and modern application needs. In order to increase innovation and create new customer experiences, organizations must modernize how they build and operate applications. The latest application development paradigm enables you to rapid innovation by using cloud-native architectures, loosely coupled microservices, DevOps, managed databases, as well as built-in monitoring and self-healing mechanisms. So the key question for this chapter would be **What are the modern application design requirements and how can we achieve them**.

Structure

In this chapter we will discuss following topics:

- Modern application design requirements
    - Availability
    - Scalability
    - Performance
    - Observability
    - Security
    - Resiliency
    - Cost optimization
    - Portability, being Cloud-Agnostic
    - Cloud-native
    - AI/ML enabled
    - DevOps delivery
    - Sustainability
- The Twelve-Factor App methodology
    - **Codebase:** One codebase tracked in revision control, many deploys
    - **Dependencies:** Explicitly declare and isolate dependencies

    - **Config:** Store config in the environment
    - **Backing services:** Treat backing services as attached resources
    - Build, release, run: Strictly separate build and run stages
    - **Processes:** Execute the app as one or more stateless processes
    - **Port binding:** Export services via port binding
    - **Concurrency:** Scale out via the process model
    - **Disposability:** Maximize robustness with fast start-up and graceful shutdown
    - **Dev/prod parity:** Keep development, testing, staging, and production as similar as possible
    - **Logs:** Treat logs as event streams
    - **Admin processes:** Run admin/management tasks as one-off processes
- Going beyond the twelve factors:
    - API First
    - Security
- Conclusion

Objectives

The purpose of this chapter is to provide you with the details of modern application design requirements along with the Twelve-Factor App methodology. The following principles will come in handy whether you have to design an entirely new application directly on the cloud, or if you want to modernize your legacy workload using microservices. These will be your design principles for building complex mission critical production systems.

Modern application design requirements

Firstly, let us list the core requirements for a modern application, and then we will try to understand the way to achieve them (will share some examples from the major Cloud service providers). There can be addition design requirements based on the type of workload but in any case would be most recommended set. For example, as

illustrated in *Figure 2.1: Modern Application Design Requirement* will have these as key components.

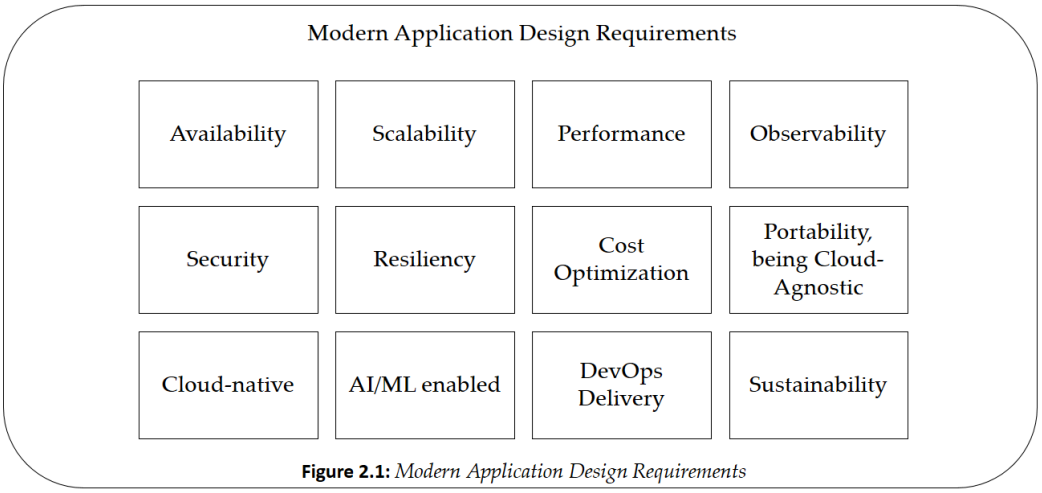


Figure 2.1: *Modern Application Design Requirements*

Availability

Typically, availability is planned and configured between 99.9% to 99.999%, for Production systems. Modern application design principles dictate that there should be no single point of failure, and continuous monitoring should be in place to detect failures and trigger failover mechanisms. It's important to carefully evaluate the need for high availability on a per-application basis, as implementing HA solutions can add to the overall cost and complexity of operations. To ensure high availability, modern applications should be designed to be fault-tolerant, meaning that they are able to continue functioning even in the event of hardware or software failures. This can be achieved through techniques such as redundancy, load balancing, and automatic failover. In addition, continuous monitoring and proactive management are essential for detecting and addressing issues before they impact users.

When evaluating the need for high availability, it's important to consider the potential impact of downtime on the business. Applications that are critical to business operations or customer experience may require higher levels of availability, while others may be able to tolerate more downtime. It's also important to consider the cost and complexity of implementing high availability solutions, as well as the ongoing maintenance and management required to ensure they continue to function properly.

- **For example**, let us review some of the Azure Cloud availability options (all other major Cloud provider will have similar options).

- We should select multiple Availability Zone (AZ is a physically separate location within an Azure region). There are three Availability Zones per available per Azure region.
- Azure **virtual machine scale sets** (**VMSS**) allows you to create and manage a group of load-balanced virtual machines. The number of VM instances can auto-scale in response to demand or a schedule.
- Azure's availability sets enable it to understand how your application is designed to provide redundancy and availability.
- The Azure Load Balancer that distributes traffic between multiple virtual machines and can be combined with an availability zone or availability set to get the most application resilience.
- Multiple copies of your data are stored in Azure Storage to safeguard against planned and unplanned events, such as hardware failures, network outages, and massive natural disasters.
- Azure Site Recovery is used to replicates workloads running on a **virtual machines** (**VMs**) from one primary site to another, ensuring business continuity and disaster recovery.

Scalability

Scalability is usually a concern when volumes are large or subject to sudden spikes. In the previous section, we mentioned VMSS and it is more of a scalability solution for VMs. Cloud scalability refers to the ability to scale up or down cloud resources according to demand. Cloud computing allows companies to better manage their resources and costs, which is one of the main benefits of using it. Another benefit of cloud scalability is that it enables businesses to make architectural decisions that are more dynamic and adaptable to changing business requirements. Unlike traditional static architectural decisions that may hinder a system's ability to meet changing business needs, cloud scalability allows for more dynamic decision-making that can better support evolving business requirements.

For example, for a major launch or event, organizations do not have to overhaul their infrastructure as they would with on premise solutions. Traditionally, organizations that have not yet adopted the Cloud have gone through days or even months of preparation before a major release, like the iPhone launch, or before a large sporting event, like the World Cup. Traditionally, architectural decisions are implemented as static, one-time events, with a few major versions of a system over its lifetime. As a business continues to evolve, these initial decisions might hinder the system's ability to deliver changing business requirements.

Performance

Performance is a critical consideration for mission-critical applications in every organization, as these applications can have a direct impact on revenue or customer experience. In today's fast-paced digital world, users expect high performance and quick response times from the applications they use. As a result, organizations need to design and implement applications that can handle high volumes of traffic and requests without experiencing performance degradation or downtime.

Cloud-native microservices architecture is well-suited for high-performance applications. With microservices, applications are broken down into smaller, independent services that can be scaled and optimized individually. This allows organizations to isolate and address performance bottlenecks in specific services without affecting the performance of the entire application. Moreover, cloud-native microservices architecture is highly scalable and can handle large volumes of traffic and concurrent users without impacting performance. Cloud providers offer a range of services and tools that can be used to optimize application performance, such as load balancers, auto-scaling, and **content delivery networks** (**CDNs**).

In addition to this, cloud-native microservices architecture allows for the use of modern programming languages and frameworks that are optimized for performance. For example, many microservices are built using container technologies like Docker and Kubernetes, which provide low overhead and fast startup times. These technologies allow for efficient use of resources and can reduce the overall infrastructure requirements needed to support high-performance applications.

For example, there is another category of use cases that require extensive calculations and high speed such as weather forecasting and risk assessment. These compute-heavy and data-intensive workloads that need to process complex simulations (some including terabytes of data) would gain a significant advantage in Cloud. On-premises solution for such a scenario would be much more expensive because of the initial high procurement cost, secondly, aging hardware will impact the total cost of ownership.

Observability

The ability to observe a system's current state based on its logs, metrics, traces, application processes, data processes, and hardware processes is what defines observability. You cannot analyse and fix problems at the speed you need if you cannot monitor your servers, containers, and data in the cloud.

Monitoring dashboards used in earlier scenarios are usually configured so that they alert you to performance issues you may encounter/expect to see later. It is assumed that you can predict what problems will arise before they occur in these dashboards and then it is configured to monitor the same possible scenario. On the other

hand, cloud-native observability does not just mimic what logging, tracing, and monitoring apps did before the cloud. Observability is where an environment has been fully designed to capture complete observability data. With this data, you can investigate what is going on and identify the root cause of issues you may not have been able to anticipate. It helps accelerate DevOps adoption because it allows you to identify and rectify issues occurring between different projects and containers.

For example, observability is extremely helpful to cross-functional teams in enterprise environments in figuring out what is happening in highly distributed systems. It is possible to improve performance if you are able to observe what is slow or broken. When teams implement an observability solution, they can receive alerts about issues and resolve them proactively before users are affected.

Security

Security is a critical aspect of modern application design, and it is a top priority for most organizations when moving or expanding to the cloud. Adopting cloud-native microservices design patterns can significantly enhance an organization's cybersecurity posture. Here are some tips to keep your cloud data secure:

- **Application Gateway with WAF**: Application Gateway is a web traffic load balancer that helps manage traffic to your web applications. It can make routing decisions based on additional attributes of an HTTP request, such as URL path or host headers. You can deploy **Web Application Firewall** (**WAF**) on Azure Application Gateway or WAF on Azure Front Door Service to make it more secure.
- **Protect data using encryption**: Data encryption at rest and in motion is crucial. For example, AES-256 encryption should be used, and enhanced data protection should be implemented with encryption at all transport layers, file shares, and communication channels.
- **Key Vault:** It should be used to securely store and tightly control access to tokens, passwords, certificates, API keys, and other secrets. For example, **Azure Key Vault** (**AKV**) is a recommended solution for key management on Microsoft Azure Cloud. AKV makes it easy to create and control the encryption keys and other certificates.
- **Multi-factor Authentication (MFA)**: MFA is an excellent way to secure logins, and it is commonly used these days in financial transactions, connecting to official websites, and more. It usually includes passwords, security questions, captcha, apps that authenticate, OTPs via SMS or calls, and even biometric data.
- **DevSecOps and Automation**: Organizations that have adopted the highly automated DevOps CI/CD culture must ensure appropriate security controls

are identified and embedded in code and templates as early as possible in the development process.

- **Zero Trust**: In cloud security, zero trust means not automatically trusting anyone or anything within or outside the network and verifying (i.e., authorizing, inspecting, and securing) everything. It promotes a least privilege governance strategy where users/individuals are only given access to the resources they need to perform their duties.
- **Implement Email Security System**: Many cyberattacks, such as phishing and malware attacks, begin with an email. Having an effective email security system and user awareness training in place will help prevent such situations.
- **Ensure Software and OS Patches are updated:** It is a very common and obvious measure that is often overlooked. In recent years, the majority of large-scale attacks have taken advantage of systems and devices that have been left vulnerable due to ignored updates. The purpose of patches is to address critical vulnerabilities.

Resiliency

It is true that cloud resilience cannot be achieved unless it is designed, implemented, and maintained. The question is, how can an organization ensure that they get it right from the start?

- **Define Resilience**: Before embarking on the four-step process, it's important to have a clear understanding of what resilience means for your organization. Define what resilience means in the context of your cloud workloads and what outcomes you are looking to achieve. This will help ensure that the steps you take are aligned with your goals.
- **Consider Risk Assessment**: In addition to assessing business resilience requirements, it's also important to perform a risk assessment. Identify potential risks that could impact the availability, performance, or security of your cloud workloads. This will help inform your resilience strategy and ensure that you are prepared to address potential issues.
- **Implement Automated Failover**: One key aspect of cloud resilience is the ability to quickly recover from failures. Implementing automated failover can help reduce downtime and ensure that your applications remain available. This can be achieved through the use of load balancers, auto-scaling, and other automation tools.
- **Monitor and Continuously Improve**: Resilience is not a one-time effort, but an ongoing process. It's important to monitor your cloud workloads and resilience strategy on an ongoing basis and make adjustments as needed.

This can help you identify and address issues before they become major problems.

- **Consider Redundancy and Replication**: Resilience can also be achieved through redundancy and replication. This means having multiple copies of your data and applications in different locations to ensure availability in the event of a failure. Consider implementing a multi-region deployment strategy to ensure that your applications remain available even if an entire region goes down.
- **Have a Disaster Recovery Plan:** In addition to implementing a resilience strategy, it's also important to have a disaster recovery plan in place. This plan should outline the steps to take in the event of a major outage or disaster, including how to recover your data and applications and restore service to your customers.

Cost optimization

By using cloud cost optimization, you can reduce your overall cloud spend by identifying mismanaged resources whether it is VM's, memory or storage, eliminating waste or ideal capacity, reserving capacity for higher discounts, and right-sizing computing services. By designing your application for scalability and consuming services as you go, the cloud gives your organization unlimited scalability and lower costs. However, we need to understand that **Amazon Web Services** (**AWS**) and Microsoft Azure, or any other cloud for that matter will charge customers for the resources they order, whether they use them or not. There is solution to fine tune your expense by going serverless, or implementing other cost optimization tools.

Cloud Cost Optimization strategy would include:

- **Find Unused or Unattached Resources**: There are unused or unattached resources that can simply be removed to optimize cloud costs. Many a time your cloud admin or developer might "spin up a resource" a temporary server or attach some storage or memory to perform a task or test and then forget to remove it afterward. You will be surprised at the number of development and testing environments per application. We can save significantly once we develop governance and accountability mechanisms (easy to do for non-prod env's).
- **Identify and Consolidate Idle Resources**: There are idle resources sometimes as your consumption forecast or consumption trends changed and that would be the next step in optimizing cloud computing costs. It is possible for an idle computing instance to have a CPU utilization level between 1 to 10%. Enterprises waste a significant amount of money when they receive a bill for 100% of their computing instance which is actually almost ideal most

of the time. This can be one of the drawbacks of implementing microservices without considering resource consumption overhead.

- **Right Size Computing Services**: Computing services are right-sized by analysing them and modifying them accordingly. Provides cloud optimization, which means getting the most out of the resources you are paying for. For example, cloud cost optimization relies heavily on heat maps. It is a visual tool that shows how computing demand peaks and valleys over time. Since cloud elasticity allows us to expand on demand, we no longer need to provision for peaks.
- **Invest in Reserved Instances:** Investing in reserved instances is a smart idea for enterprises that intend to use the cloud for the long term especially when we know our consumption trends. In exchange for an upfront payment and time commitment, these discounts are large. It is crucial to optimize cloud costs through RI, as savings can be as high as 75%.

Portability, being cloud-agnostic

The portability of cloud services refers to the ability to move applications or data from one cloud service provider to other **cloud service provider** (**CSP**) without the need to rewrite the code or restructure the application. Enterprises using public cloud services are increasingly relying on cloud-agnostic development strategies. Being cloud-agnostic is a key feature that businesses are looking for to manage cloud costs. As a result of this portability, they will be able to switch between providers based on performance and cost-efficiency. It is possible to gain edge computing benefits by using different providers based on the region. Flexible, reliable, and avoiding vendor lock-in are some of the benefits of this approach.

For solution to be cloud agnostic all its tools, platforms, applications, and pipelines should be compatible with any other cloud service provider infrastructure and can be moved between cloud environments without any code change. However, there are certain drawbacks to this approach as it limits the use of Cloud-Native options and to maintain this level of portability you will need to invest more while migration and maintaining these applications.

- **Costs & Complexity upfront**: Being cloud-agnostic can save time, money, and stress in the long run, but it also requires more upfront work. Initial development costs and complexity are likely to increase if you develop an agnostic cloud strategy from scratch, regardless of which workloads you plan to run in the cloud.

- **Inability to use vendor-specific features**: In a strictly cloud-agnostic strategy, it may be difficult to take advantage of new features introduced by AWS, Microsoft Azure, Alibaba, Google Cloud Platform, or another large provider. For example, the **Amazon Relational Database Service** (**Amazon RDS**) is a cost-effective, easy-to-set-up, operate, and scale managed database service in the AWS cloud environment, although it does not support cloud-agnostic features.
- **Terraform (HashiCorp)**: With HashiCorp Terraform, infrastructure can be defined as code (for example, instead of cloud-specific scripting for ARM temples in Azure). Additionally, the organization must maintain these scripts if it wishes to switch providers at any time. Scripts for different cloud providers require developers to have a certain level of knowledge about each platform. This is because they should know about what type of instance is spun up and what size the instance should be. There are other configuration tools, such as ansible, chef, and puppet, that have similar abilities while having specifics of their own.
- **Interface Layers and Design Patterns**: Even though multiple vendors provide similar services, the interface to these services may differ. As a result, the application design can have separate service layers to reduce the friction in porting services between them. Similarly, vendors recommend various design patterns that are cloud-native, but their implementations are different from one cloud provider to another. Therefore, in a way we have to limit ourselves to the lowest common denominator.

Cloud-native

For this section, let us not discuss Cloud-native in detail as we will be going in depth of Cloud-native design principles, patterns with scenarios in coming chapters. At high level Cloud-native refers to an approach to software development and deployment that leverages cloud computing services, infrastructure, and platforms to build and run applications that are optimized for the cloud environment. Cloud-native applications are designed to be scalable, resilient, and flexible, making them well-suited for modern, dynamic, and distributed environments.

Microservices, on the other hand, are an architectural style that structures applications as a collection of small, independent, and loosely coupled services. Each service is responsible for performing a specific function and communicates with other services using lightweight protocols such as RESTful APIs or messaging queues.

For example, as illustrated in *Figure 2.2:* Cloud Native official definition from 'Cloud Native Computing Foundation'.

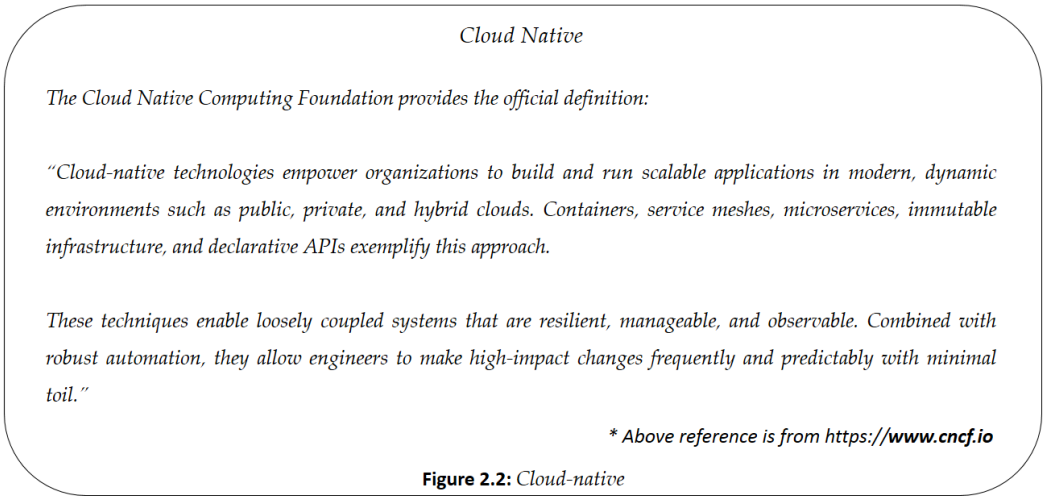


Figure 2.2: *Cloud-native*

Cloud-native and Microservices: They are often used together to create highly scalable and resilient applications. For example, a company might use a cloud platform such as AWS to host a microservices architecture for their e-commerce site. Each service could be designed to handle a specific task, such as processing orders, managing inventory, or handling payments. By using a cloud-native approach, the company can take advantage of the platform's scalability, availability, and elasticity features to ensure that the application can handle large amounts of traffic and can scale up or down as needed. The use of microservices also makes it easier to manage the application, since each service can be developed, tested, and deployed independently.

AI/ML enabled

With artificial intelligence and machine learning, applications can perform many tasks that previously required humans to perform by themselves. Many businesses are embarking on the next generation of digital transformation journey by utilizing **Machine Learning** (**ML**) and **Artificial Intelligence** (**AI**). A business's nature and the way that AI/ML are implemented determine the benefits we can drive.

For example, from simple product recommendations to product design optimization, many different use cases can be implemented with the help of AI/ML. Machine learning in a way is a subset of AI that uses deep learning with neural network algorithms. It can recognize patterns and perform complex computational tasks with precision. During the Covid-19 outbreak widely available chest X-ray were used with artificial intelligence for diagnosis. COVID-19 caused pneumonia, pneumonia, and normal CXRs as a dataset to train Microsoft CustomVision. The

model performed with high sensitivity and accuracy. We will not be able to discuss the AI/ML approach, technologies, and modules in detail. However, you can view the AI/ML document on your preferred cloud platform. There are solutions and jumpstart kits available to implement and experiment with advanced techniques like Reinforcement learning, Data preparation and analytics, AIOps, Build and train models and the like.

DevOps delivery

Ability to deliver quality applications and services at high velocity can be achieved by adopting DevOps practices which is a combination of culture, process, and tools. It aims at merging development, quality assurance, and operations (deployment and integration) into a single, continuous set of processes. In many ways, this methodology is an extension of the Agile and continuous delivery approaches.

As mentioned earlier it is a change in culture and mind-set. It is about breaking down barriers between traditionally siloes development and operations teams. The two teams work together through DevOps to optimize both developer productivity and operations reliability.

Communication, efficiency, and quality improvement all come naturally with the adoption of DevOps. In addition to that, quality assurance/testing and cyber-security teams may become tight-knit as well.

Sustainability

In most cases, cloud computing is more environmentally friendly than traditional data centres as they have greater focus and investment towards reducing the carbon footprint. Public clouds are a greener option when compared with more traditional on-premises computing. However, the resource your application going to consume will be based on how efficiently it has been designed and the governance around it. If it is not done properly then the cloud may not be green for you.

Power-optimized application development is usually not the focus area. The ability to code and deploy an application that requires the minimum amount of power to carry out ongoing operations. Optimized workloads would require less cloud resources and accordingly will consume less power and less monthly expense for cloud resources. All major public clouds are working towards lower carbon footprints by efficiently designing their data centres and using renewable energy.

The Twelve-Factor App methodology

The Twelve-Factor App methodology for building software-as-a-service applications. These best practices are the base designed for enable applications to be built with

portability and resilience. For example, as illustrated in *Figure 2.3:* The Twelve-Factor Application methodology as defined by *12factor.net*.

The Twelve-Factor Application Methodology

"In the modern era, software is commonly delivered as a service: called web apps, or software-as-a-service. The twelve-factor app is a methodology for building software-as-a-service apps that:

- *Use declarative formats for setup automation, to minimize time and cost for new developers joining the project;*
- *Have a clean contract with the underlying operating system, offering maximum portability between execution environments;*
- *Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration;*
- *Minimize divergence between development and production, enabling continuous deployment for maximum agility;*

And can scale up without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc)."

** Above reference is from https://**12factor.net***

Figure 2.3: *The Twelve-Factor Application*

Code base

We should track code changes for the app in a version control system such as Git or SVN. For any changes, code should be checked out into a local development environment. Storing code in a version control system will ensure basic hygiene and enable our team to work together. We will have an audit trail of changes to the code that will help in resolving merge issues, and the ability to roll back (if req.) the code to a previous version. It also provides a place from which to do **continuous integration** (**CI**) and **continuous deployment** (**CD**) pipelines will pick latest source code. For example, as illustrated in *Figure 2.4:* Codebase is key component for application design and deployment.

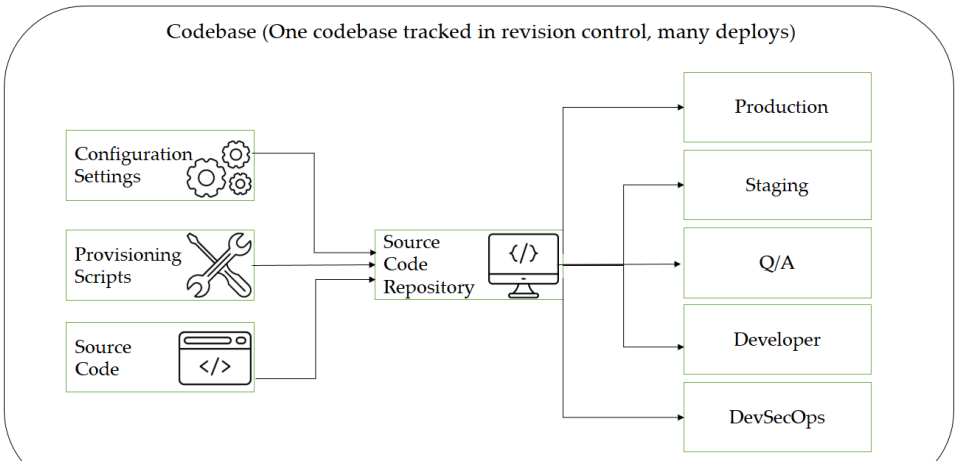


Figure 2.4: Codebase

For example, while developers might be working on different versions (branch) of the code in their development environments, at any given time the single source of truth is the code in the version control system (usually the main branch). Using the repository code, a single build is produced, which is then combined with an environment-specific configuration to produce an immutable release for the target environment. Any changes/modifications should be built and deployed in a new release.

Dependencies

In 12-factor apps, dependencies should never be implicit, and they should always be declared explicitly and checked into version control. That way, our code can be started quickly in a repeatable manner and dependencies can be tracked easily. Cloud-native applications cannot rely on implicitly existing system-wide packages. Explicitly declaring and isolating application dependencies is what this factor focuses on. As a result, it simplifies the setup for new developers and supports portability between cloud platforms, enabling consistency between development and production environments.

For example, package managers such as sbt and maven can be used to manage all application packages. Configuration management tools, such as chef, ansible, and so on., can be used to install system-level dependencies in non-containerized environments. Similarly, the Docker file can be used for containerized environments.

Configurations

Almost every modern app requires some level of configuration. Each environment, such as development, testing, and production, has its own configuration. We usually have a build (source code) + configuration specific to environments. These configurations usually include service account credentials, environment-specific settings (if any) and resource handles for backing services such as databases. One of the key principles is that the configurations for each environment should be external to the code and not checked into version control. You have multiple configurations, but everyone only works on one version of the code. The configuration depends on the deployment environment. Using this approach, a single binary can be deployed to all environments, with only the runtime configuration differing. Checking whether the configuration can be made public without revealing credentials is an easy way to determine whether it has been externalized correctly.

For example, a better approach is to store configuration in environment variables. A developer can easily change these at runtime for different environments, they are unlikely to be checked into version control, and they are not dependent on the programming language or development framework.

Backing services

All services that the app consumes in its normal operation, such as file systems, databases, caching systems, and message queues, should be accessed as services and externalized in the configuration. These backing services are abstractions of the underlying resources. As an example, decoupling storage from the app allows you to seamlessly change the underlying storage type when writing data to storage. Without changing the app code, you can switch from a local PostgreSQL database to Cloud SQL for PostgreSQL. For example, as illustrated in *Figure 2.5*: Backing Services is an important application design element.

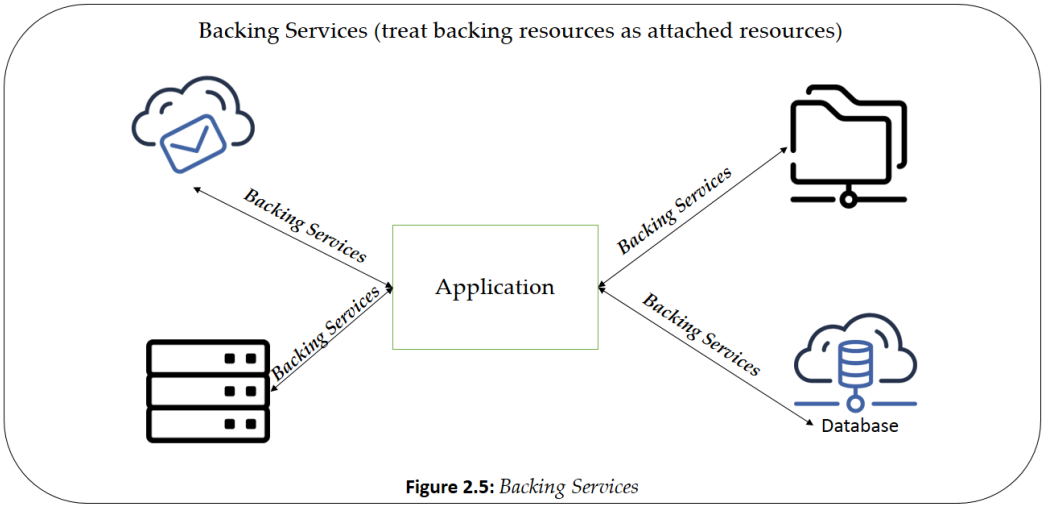


Figure 2.5: *Backing Services*

For example, the purpose of this factor is to treat these backing services as bound resources. You can use bound resources to connect your application to a backing service. Your application can consume a database resource by providing a username, a password, and a URL.

Build, release, run

The software deployment process should be divided into three distinct stages: build, release, and run. Every stage should produce an artifact that can be uniquely identified. An environment's configuration and a build should be associated with every deployment. The result is an easy rollback process and a visible audit trail of every release / deployment.

- **Build**: We should focus on building everything needed for our application during the build stage; packaging builds with configuration into release artifacts and running applications should not involve additional build steps. A build artifact (for example a WAR or JAR file) is created by combining the

dependencies declared during the design phase. After a build is complete, an environment-independent server is packaged, containing all of the application's configuration. Especially for cloud-native applications that need to be deployed in multiple environments, this is crucial.

- **Release**: During the release stage, output from the build stage is combined with configuration values (both environmental and application-specific) to produce another release. Labelling these releases with unique IDs gives the ability to rollback to previous versions in case anything goes wrong, and allows for historical auditing.
- **Run**: In the run stage, the application is launched using tools like containers and processes on the cloud.

For example, as illustrated in *Figure 2.6:* Build, Release, and Run needs to be strictly separated.

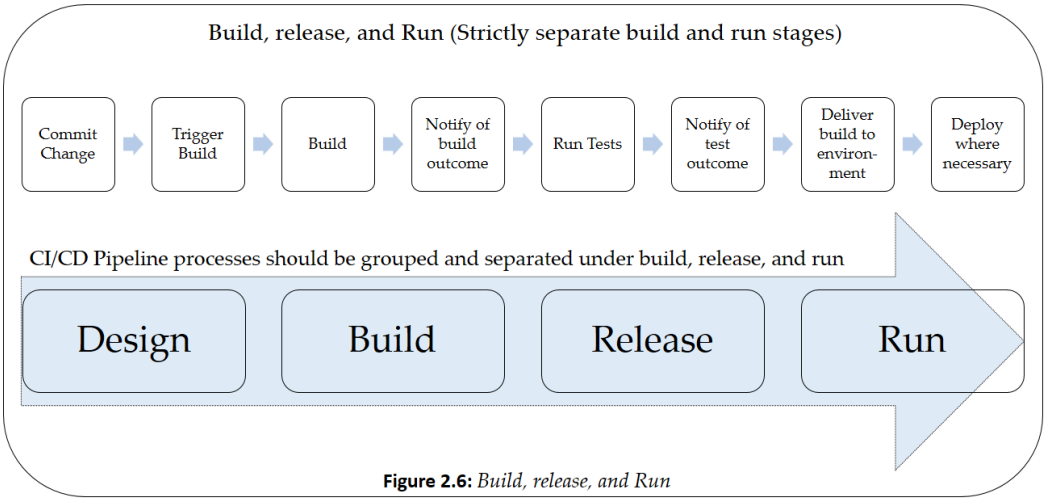


Figure 2.6: *Build, release, and Run*

For example, it is a standard process that each release must enforce by defining a strict separation across the build, release, and run stages. When you commit code that has passed all required tests, the build stage is usually triggered automatically or we can manually run the build. During the build stage, the code is compiled, files are fetched, and assets are packaged into self-contained binaries. During the build stage, an artifact is created. Release stages combine the build artifacts with environment configurations once the build stage is complete. As a result, a release is created. Continuous deployment applications can automatically deploy the release into the environment. Alternatively, you can use the same continuous deployment app pipeline to trigger the release. Finally, the run stage launches and starts the release.

Processes

You run twelve-factor apps in the environment as one or more processes. These processes should be stateless (which will be useful while scaling up or down) and should not share data with each other. Processes can also be transported across different computing infrastructures when they are created as stateless apps.

This requires a change in how you think about handling and persisting data if you are used to sticky sessions. Despite the fact that processes can disappear at any time, local storage contents are not guaranteed to remain available (there are workaround solutions like persistence storage but it is not a recommended solution). In order to reuse data, you must explicitly persist it in a database or external backing service. For example, as illustrated in *Figure 2.7*: Processes needs to be stateless.

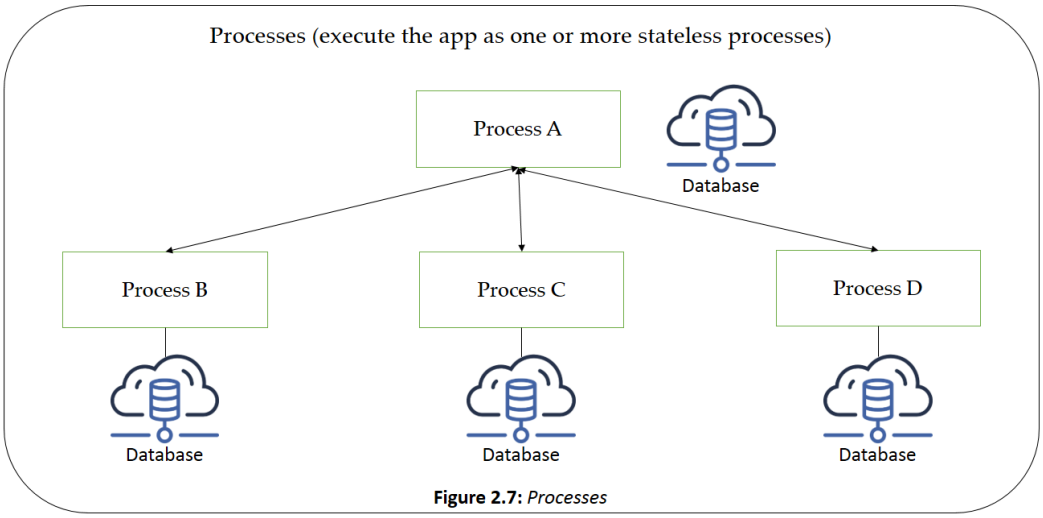


Figure 2.7: *Processes*

Memory stores can be used as a backing service if you need to persist data. Your apps will be able to cache state and share common data between processes so that loose coupling is encouraged.

For example, each microservice having its own process, isolated from other running services. Data stores or distributed caches can be used to externalize required state.

Port binding

The port-binding factor states that cloud-native applications should export services using port binding. The principle of port binding asserts that services and applications are identified by their port numbers, not their domain names. By using manual manipulation domain names and associated IP addresses can be assigned at run time. Therefore, it is unreliable to use them as a point of reference. For example, as illustrated in *Figure 2.8:* Export services via Port binding.

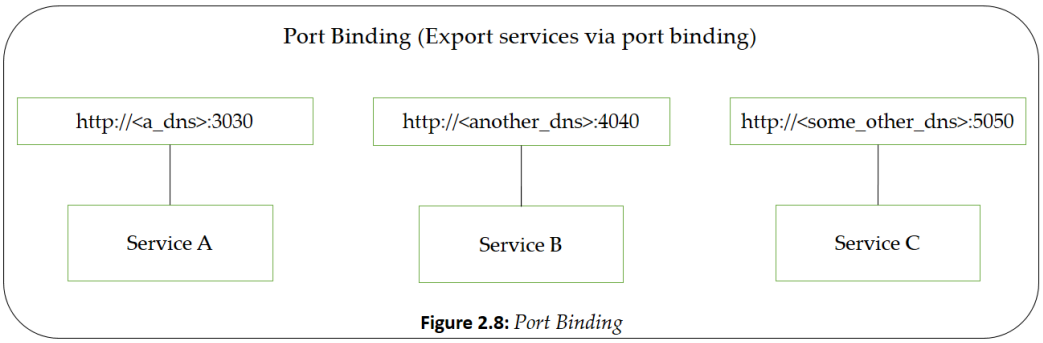


Figure 2.8: *Port Binding*

A service or application can be exposed to the network more reliably and more easily by exposing its port number. Port binding relies on the idea that a process should be accessible to the network uniformly by means of its port number.

For example, 0 through 1023 are the well-known ports (also called system ports).

Number	Assignment
20	File Transfer Protocol (FTP) Data Transfer
21	File Transfer Protocol (FTP) Command Control
22	Secure Shell (SSH) Secure Login
23	Telnet remote login service, unencrypted text messages
25	Simple Mail Transfer Protocol (SMTP) email delivery
53	Domain Name System (DNS) service
67, 68	Dynamic Host Configuration Protocol (DHCP)
80	Hypertext Transfer Protocol (HTTP) used in the Internet
443	HTTP Secure (HTTPS) HTTP over TLS/SSL

Above list is an example only for your understanding as there are more commonly used ports available.

Concurrency

Idea is to scale out horizontally across multiple identical processes rather than scaling up one large instance on a higher capacity server. Scale out the application seamlessly in cloud environments by designing it as a concurrent application. Apps should be decomposed into separate processes based on their types, such as background, web, and worker processes. This allows your app to scale up and down based on individual workload demands. The principles of disposability and statelessness are at the core of apps that are ideally suited to horizontal scaling. For

example, as illustrated in *Figure 2.9*: Concurrency should be designed as Scale out via the Process model.

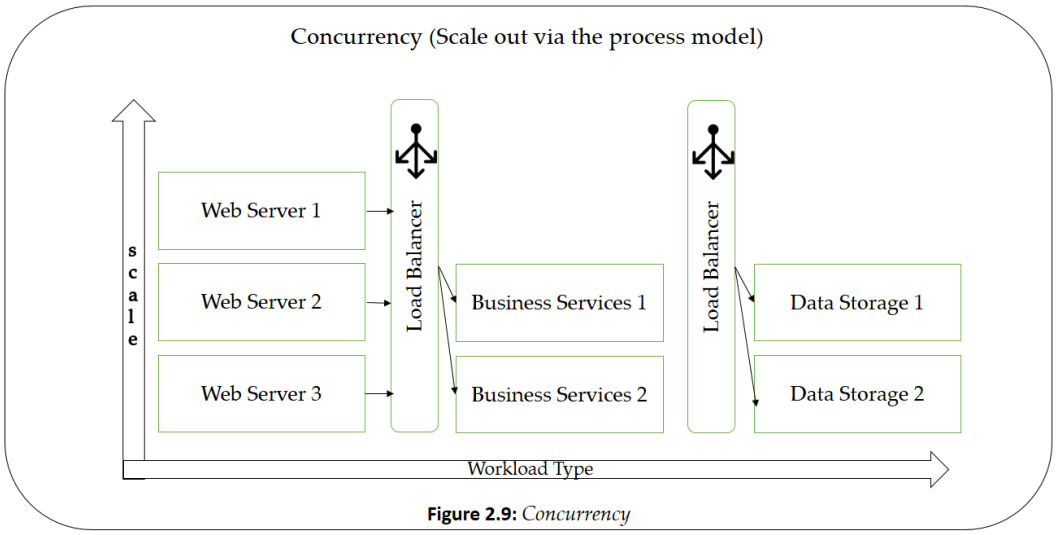


Figure 2.9: *Concurrency*

For example, Using App Engine, you can host your apps on Google Cloud's managed infrastructure if you are using GCP. You can have one or several instances running at the same time, with requests being distributed among them based on the current load. Similarly, if you are on Azure, VMSS provides scalability to add VM as per the application needs. Automatic scaling allows you to balance performance and cost by setting target CPU utilization, target throughput, and maximum concurrent requests.

In addition, we will cover serverless in a later chapter. Serverless solutions such as AWS Lambda, Azure Functions, GCP Cloud Functions, and others can be fully managed by cloud providers and will scale as per need.

Disposability

You should treat application infrastructure / cloud infrastructure as disposable resources. Your apps should be able to handle the temporary loss of the underlying infrastructure and should be able to gracefully shut down and restart. Cloud-native applications have disposable processes that can be started and stopped quickly. In order to scale, deploy, release, or recover quickly, an application needs to be able to start quickly and shut down gracefully.

While your application is starting, hundreds or thousands of requests may be denied if it takes time to reach a steady state. Therefore, there are some key design elements to be considered:

- Utilize backing services to decouple state management and storage of transactional data.
- Managing environment variables outside of the app allows them to be used at runtime.
- Minimize start-up time. The start-up process will take a significant amount of time if you are downloading and initializing several packages or binaries (this process can be tuned).

Dev/Prod parity

It is critical to keep environments across the application lifecycle (Prod and non-prod env's) as identical as possible and avoid costly shortcuts. Every application will have different environments (Prod, and Non-Prod) during its development lifecycle. These environments are development, testing, staging, and production. We have seen applications with 10 or more non-production environments. Therefore, it becomes critical to keep these environments as similar as possible otherwise debugging and environment management becomes a big overhead.

As developers have embraced source control, configuration management, and a separate build, run, and deploy process, maintaining environment parity has become easier. As a result, it is easier to deploy an app to multiple environments consistently.

Dev, test, and production parity can be achieved with tools like Docker. Containers provide a uniform environment for running code, which is one of the benefits. In order to eliminate the differences between development, testing, and production environments, it is important to be able to lock down every detail of the environment.

Logging

Using logs, you can monitor the health of your apps. You should decouple log collection, processing, and analysis from the core logic of your apps and treat them as a separate stream of events. We have seen many enterprises that process logs using another 3rd party software for observability. It is especially beneficial to decouple logging when you are running your applications in public clouds. This is because

it eliminates the overhead of managing log storage locations and aggregating distributed virtual machines.

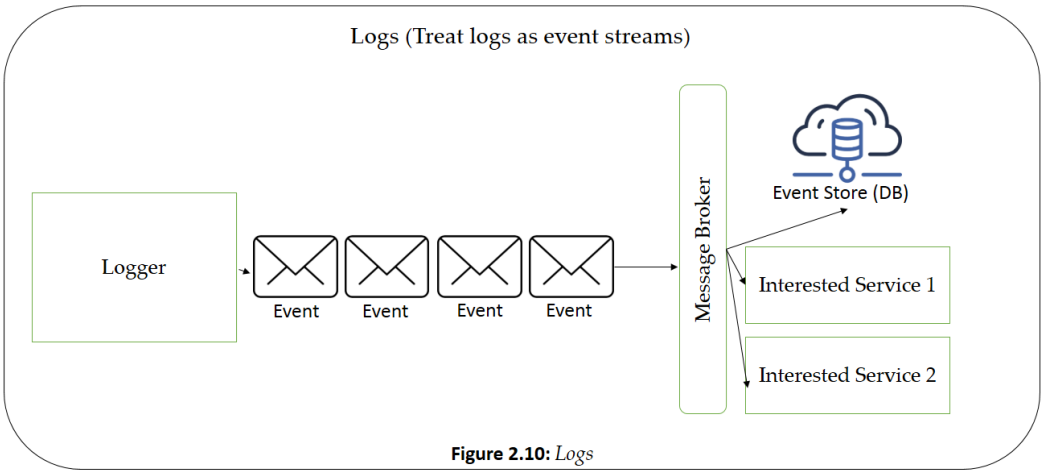


Figure 2.10: *Logs*

Admin processes

Administrative processes include one-off tasks or timed, repeatable tasks, such as generating reports, running batch scripts, backing up databases, and migrating schemas. Admin processes was written with one-off tasks in mind in the twelve-factor manifesto. The importance of this factor becomes apparent when you are creating repeatable tasks for cloud-native apps. Data cleanup and analysis can be run as one-time processes for administrative/management tasks. Therefore, you should decouple the management of admin processes from the app when designing for admin processes. You can invoke these tasks independently from the application using independent tools for example CronJobs).

Going beyond the twelve factors

There are addition key factors which are of integrated part of today's modern application design should be considered while designing / modernization of your application stack.

API first

Consider your code to be consumed by a client, gateway, or another service and design it as a service. APIs are used to communicate between apps. Start by designing an API strategy when you are building apps for your app's ecosystem and then think about how the app will be used. The API should be designed in a way that makes it easy for external stakeholders and app developers to consume. API endpoints

should isolate and decouple the consuming applications from the app infrastructure that provides them. As a result, the app's consumers will not be affected by changes to the underlying service or its infrastructure.

Let's say you are designing a mobile application for a retail company that sells clothes online. In this case, you should design an API strategy that makes it easy for external stakeholders and app developers to consume. For example, you can create APIs for product catalogue, shopping cart, payment gateway, and order tracking. These APIs should be designed in a way that isolates and decouples the mobile app from the app infrastructure that provides them. This way, even if the underlying service or infrastructure changes, the mobile app's users won't be affected.

Security

Considering the increasing number of touchpoints within modern applications, as well as the risk of cyber-attacks, it is crucial that you adhere to standard procedures to secure the network, protect the data while it is in transit and at rest with encryption, control access based on need only, and monitor your environment for any security breaches. Security covers a wide range of topics, including operating systems, networks and firewalls, data and database security, application security, and identity management. In an enterprise's ecosystem, security must be addressed in all its dimensions. APIs allow an app to access apps within your enterprise ecosystem. Thus, when designing and building your app, make sure security considerations are taken into account. Use **Transport Layer Security** (**TLS**) to help protect data in transit. Use app-level security to control access to your app based on who the consumer is. In order to enforce security, you can use API keys (for apps consuming them), certification-based authentication, **JSON Web Tokens** (**JWTs**) exchange, or **Security Assertion Markup Language** (**SAML**).

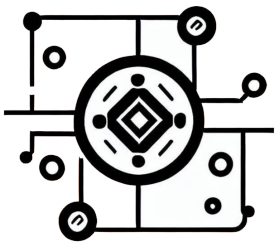
Continuing with the same example of a mobile application for a retail company, you should also consider security while designing and building your app. For instance, you can use **Transport Layer Security** (**TLS**) to help protect data in transit between the mobile app and the backend services. You can also use app-level security to control access to your app based on who the consumer is. This can be done by requiring users to log in with a valid username and password or using other forms of authentication such as biometrics. Furthermore, you can use API keys to ensure that only authorized apps consume your APIs. Finally, you can monitor your environment for any security breaches and take corrective actions if needed.

Conclusion

From Complexity to Simplicity:
Designing Modern Applications with Efficiency and Excellence

Modern application is becoming more and more capable, and complex as well unless you design them well based of principles we have discussed. In modern applications there are multiple points of access to the open internet, and they must also be reliable, secure, cost-efficient, and high performing. In this chapter we have explored many system design concepts and we will continue exploring detail around the Cloud Native and Microservices design patterns for designing a modern application. Concepts we have discussed in this chapter will come in handy when you plan to refactor your application from a monolith into microservices. As we have discussed the following application design requirements are critical in today's applications: availability, scalability, performance, observability, security, resiliency, cost optimization, portability, Cloud-native, AI/ML enabled, DevOps, and sustainability. We have discussed the Twelve-Factor App methodology in details covering some of the examples related to implementation approach.

In next chapter we will discuss, Cloud Native Microservice Adoption Framework. We will review Microservice architecture style and its key components in detail. We will review their benefits and challenges from a practitioner view point using a short step wise case study on 'Breaking the Monolith'.



Chapter 3
Microservice Adoption Framework

Adopt Microservices with Confidence: A Framework for Success

Introduction

In production today, many applications run on n-tier, monolithic architectures. However, these architectures are not ideal for cloud-based systems. Cloud adoption is driven primarily by the need for agility and flexibility to keep pace with accelerating innovation and disruption from competitors. Many companies, find that simply moving their legacy systems to the cloud does not adequately meet their needs. They cannot achieve their goals because of their monolithic systems. We need an architecture optimized for the cloud, and microservices fit the bill.

In this chapter we will discuss microservice architecture style and its enabling technologies. We will review its benefits and challenges from a practitioner view point using a short case study 'Breaking the Monolith'. Monolithic application conversion to microservices is a process of modernizing and providing business value. So, while designing cloud-native microservices we should repeatedly ask ourselves as a checkpoint to ensure that as an end result refactored systems can support the requirements of its modern business. Most of the modern application requirements we have discussed in last chapter can be achieved by refactoring the monolithic application to create microservices and migrating it to the cloud.

When modernizing a monolithic application for the cloud, microservices are likely the best choice. Compared to large monoliths, microservices are relatively lightweight, use fewer resources, and are loosely coupled and distributed, so it is easier to scale

particular services up and out. A second reason is that the overall architecture was designed to work well in the cloud.

In comparing monoliths and microservices architectures, it is important to remember that monoliths were the default option before the internet age and application requirements were different. Previously, applications had access to all the data they needed without having to ask other applications to share information, so we have seen huge ERP systems and enterprise applications based on monolithic architectures. Microservices, on the other hand, are designed to communicate with one another in a cloud computing environment. Microservices complete their own tasks and then pass them on to the next. In this way, microservices do not become slowed down by a single point of contention, and if a certain service has increased load, auto-scaling up can be performed easily for the targeted workload / service.

Structure

In this chapter we will discuss following topics:

- From Monolith to microservices
    - **Breaking the Monolith:** Strategies for building a microservice design
    - Organizing data into bounded contexts or domains
    - **Building Resilient Microservices:** Techniques for handling failure and faults
    - **Monitoring Microservices:** Best practices for testing and debugging microservices
    - Embracing continuous delivery with DevOps
- Enabling technologies for microservices
    - **Docker and Microservices:** Use cases for containerization
    - **Using Docker:** Exploring the benefits of containerization
    - Container Orchestration with Kubernetes
    - **Advantages of Using Kubernetes:** Orchestration for scalability and availability
    - Components of Kubernetes
    - **Alternatives to Container Orchestration:** Other tools
- Microservices adoption using Domain Driven Design
    - Domain-driven application decomposition steps
    - **Short Case Study 06:** Insurance Claim Processing

- Characteristics of microservices design
    - **Using Microservices Correctly:** Characteristics
    - **Case Study:** Modernizing Architecture using Microservices
- Conclusion

Objectives

This chapter aims to introduce a Microservice Adoption Framework that enables organizations to adopt microservices in a structured and organized manner. It covers various topics related to microservices adoption, including strategies for building a microservices architecture by breaking down a monolith application, best practices for organizing data into bounded contexts or domains, and techniques for building resilient microservices that can handle failures and faults.

The chapter also covers best practices for testing and debugging microservices through monitoring and emphasizes the importance of embracing continuous delivery with the help of DevOps tools and practices. Enabling technologies for microservices, such as Docker and Kubernetes, are also discussed, including their use cases and benefits for microservices development and deployment.

Additionally, the chapter provides a stepwise approach to Domain Driven Design for microservices architecture and includes a case study on microservices adoption using DDD in Insurance Claim processing. It also covers the characteristics of microservices design and best practices and principles for using microservices effectively, including a case study for modernizing architecture using microservices.

By the end of this chapter, readers will gain a clear understanding of the benefits and challenges of microservices adoption and the key strategies, best practices, and enabling technologies involved in the successful implementation of microservices architecture.

From monolith to microservices

Most big applications today were once monolithic systems, because they are easier to develop, easier to test, easier to deploy, and even scale horizontally. It is difficult to work on individual functions in isolation in a monolithic system due to its numerous interdependent functions. As a result, if even a small change needs to be made, the whole application must be taken down. In a monolithic application, it can also be difficult to understand which code controls which function.

There are poorly designed monolithic applications and there are monolithic applications (with better "layered" or "n-tier" designs) usually with **Presentation Layer, Business Layer, and Data Access Layer**. Layered architectures are defined

by the rule that each layer can only use its layer directly below it. For example, as illustrated in *Figure 3.1*, monolithic architecture request and response sequence represented.

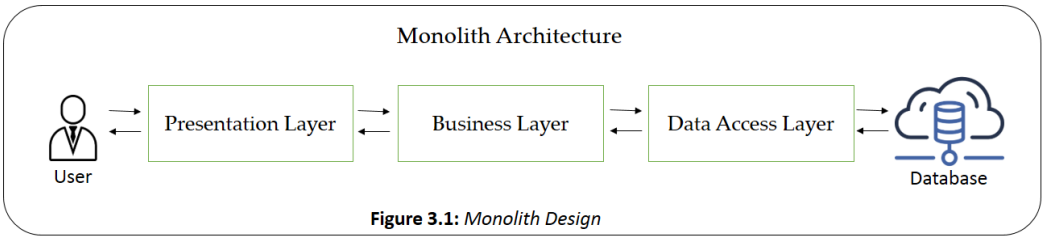


Figure 3.1: *Monolith Design*

Monoliths are large blocks of code with multiple tightly coupled modules in software. Monoliths are deployable binaries that contain both applications and business logic. An application that is monolithic usually has three layers, namely the database, the user interface, and the server-side application. As you can see in the above representation, each layer has a very specific purpose.

Thus, you could have one team working on database changes in the data layer, another team working on **Representational State Transfer** (**REST APIs**) in the business layer, and another team creating the front-end interface. While this architecture sounds great, it does not solve one problem. It remains a single application despite its layered architecture. In order to implement any large changes, you will have to re-deploy the entire application.

- REST, or Representational State Transfer, is a software architectural style that defines a set of constraints and properties based on HTTP. It is often used in the development of web services that allow different applications to communicate with each other over the internet.
    - REST APIs typically use HTTP methods to perform operations on a web server, such as GET, POST, PUT, DELETE, and so on. These methods are used to retrieve data from a server, create or update data on the server, or delete data from the server. REST APIs can return data in a variety of formats, such as XML, JSON, or HTML. In general, REST APIs are designed to be easy to use and flexible, allowing developers to build a wide range of applications that can interact with web services in a consistent and predictable way.

On the other hand, we can increase the speed of software development - time to market - by splitting the application into multiple functions (microservices) by moving from a monolithic to a microservice architecture, allowing us to work independently on each component. Microservices typically consist of an API layer, compute resources, and data storage, and can either be built to scale or leveraged through open-source services.

For example, as illustrated in *Figure 3.2: Microservices Architecture* and the way it is communicating between services represented.

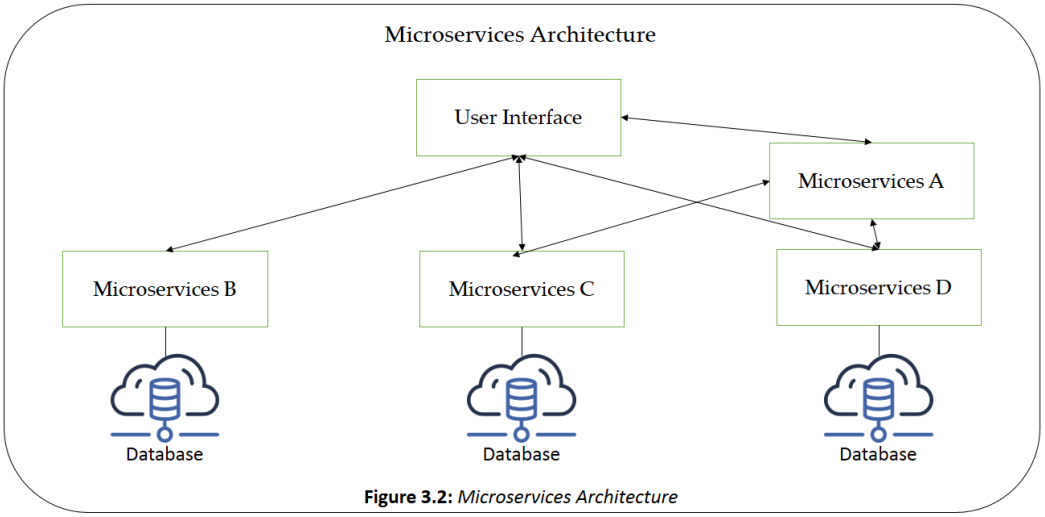


Figure 3.2: *Microservices Architecture*

A microservice architecture provides future-proofing for systems so they can adapt to changing business requirements. In complex networks of systems that require constantly evolving applications, microservices are most beneficial. Systems and applications can benefit from microservice architecture when they become too large and complex and need to be separated into smaller components.

We can start with transforming one service at a time instead of re-factoring complete application end to end. Select high-value functions for migration that can demonstrate improvement through data that is measured against a predetermined baseline. Instead of rewriting applications from the ground up to move microservices, it makes better business sense for large applications to incrementally refactor them into microservices using the following strategies:

Breaking the monolith: Strategies for building a microservice design

Breaking down a monolithic application into microservices requires careful planning and strategy to ensure that the transition is smooth and effective. Here are some additional details to consider when building a microservice design:

- **Identify the boundaries of your services**: Before you can start breaking down your monolith, you need to identify the boundaries of your services. This means understanding how your application is structured and where the different functionalities lie. Once you have identified the boundaries, you can start breaking them down into smaller, independent services.

- **Restructure your teams**: As you start breaking down your monolith into microservices, you'll need to restructure your teams accordingly. This means creating cross-functional teams that are responsible for specific services. These teams should be self-sufficient and have the autonomy to make decisions about their services.
- **Use a RESTful API**: A RESTful API is a common way of communicating between microservices. It provides a standardized way of sending and receiving data, which makes it easier to integrate different services. If you're not already using a RESTful API, consider implementing one.
- **Keep your communication protocol simple**: When designing your communication protocol, keep it as simple as possible. The protocol should be responsible for transmitting data only, without transforming it. This helps to reduce the complexity of your architecture and makes it easier to maintain.
- **Use endpoints to process requests**: Requests should be received by endpoints, which process them and emit responses. Endpoints are responsible for handling a specific type of request and should be designed to be as independent as possible.
- **Avoid tight coupling**: To avoid tight coupling between components, microservice architectures try to keep things as simple as possible. This means that services should be designed to be as independent as possible, with minimal dependencies on other services.
- **Consider an event-driven architecture**: Sometimes, an event-driven architecture can be a better option than a RESTful API. An event-driven architecture relies on asynchronous messaging, which can help to decouple services and improve scalability. However, it can be more complex to implement and maintain than a RESTful API.

Organizing data into bounded contexts or domains

In monolithic applications, all business features are stored in a single database. This singular database may not make sense as a monolith is broken up into microservices. A service that accesses the database heavily may interrupt other services' access to the database. Furthermore, multiple teams trying to modify a single database at the same time can lead to a bottleneck in the collaboration process. Depending on the microservice data needs, the database may need to be split up or additional data storage tools added. Monolith applications mostly have a database with multiple schemas for all business functions of the application. As a monolith application is split into microservices, this singular database will not work. A central database can

become a bottleneck for traffic scaling and usually couple high traffic bad performing queries can bring down performance of complete DB. That way, if a particular service accesses the database with peak load can impact the database access of other services. Secondly, a single database can become a collaboration bottleneck to handle modifications at the schema level. Extremely large databases with 100's of tables would require specialized handling and understanding. This may call for the database to be split up to support microservice data needs. Schema planning can be accomplished using bounded contexts, which is a pattern used in DDD. As a result, some data will be duplicated in different databases, posing a new set of problems to keep them in sync. Here are some points to consider while organizing data for a microservice design:

- **Define bounded contexts**: Bounded contexts refer to a set of related functionalities within a microservice architecture that share a common language and business rules. By defining bounded contexts, you can ensure that each service only has access to the data it needs to perform its specific functions. This can help to reduce dependencies between services and improve performance.
- **Use Domain-Driven Design (DDD)**: DDD is a design approach that focuses on modelling the problem domain of a business. It helps to identify and define bounded contexts and can provide a shared understanding of the business domain. DDD also emphasizes the importance of language and terminology, which can help to ensure that teams are using a common language to describe the domain.
- **Implement data isolation**: To ensure that each service only has access to the data it needs, you can implement data isolation techniques, such as using separate databases or schemas. By doing this, you can ensure that each service has its own isolated data store, which can help to reduce the risk of conflicts and improve scalability.
- **Use Event-Driven Architecture (EDA)**: EDA is an architectural pattern that emphasizes the use of events to communicate between microservices. By using events, you can reduce the need for services to access a central database and can help to reduce dependencies between services.
- **Use data replication**: When data needs to be shared across multiple microservices, you can use data replication techniques to keep the data in sync. However, data replication can be complex and can pose challenges in terms of data consistency and synchronization.
- **Monitor data usage**: It is important to monitor data usage across microservices to identify any potential bottlenecks or performance issues. This can help to identify services that are accessing the database heavily and can help to optimize the database schema to improve performance.

- **Choose the right data storage tools**: Depending on the data needs of each microservice, you may need to choose different data storage tools. For example, some services may benefit from using a NoSQL database, while others may require a relational database. It is important to choose the right tool for the job to ensure optimal performance and scalability.

Building resilient microservices: Techniques for handling failure and faults

All the services must interact to complete actions, which makes things more complicated. You need to cater for the fact that your system is now distributed with multiple points of failure. You must be able to deal with both cases where a service is not responding and slow network responses. It can also be tricky to recover from a failure, since you need to ensure that those services that are back online do not become overloaded with pending messages. We will be covering design patterns in detail like bulkhead and circuit breaker in next chapter to see how to design for failure.

- **Bulkheads**: A ship's hull consists of several individual watertight bulkheads, so if any one of them gets damaged, the failure only affects that bulkhead, rather than the entire ship. Partitioning can also be used in software to isolate failures in small parts of the system. Service boundaries (that is, the microservices themselves) serve as bulkheads to isolate failures.

- **Circuit breaker**: When a service fails, this pattern ensures that the failure does not negatively impact the entire system. For each call, we have to wait for a timeout before moving on if the number of calls to the failed service was high. As a result of making the call to the failed service and waiting for it to respond, the overall system would eventually become unstable. In the event of a circuit breaker trip, fallback logic can be initiated instead after a defined threshold of failed attempts is reached.

In addition to the two key patterns that we mentioned above, we must also take into account the following points to build a setup that is failure proof.

- The implementation of redundant and backup components or services means that you have multiple instances of each service running in different locations in order to be able to continue serving requests even if one instance fails.

- A self-healing and recovery mechanism must be implemented. This means that the system should be able to detect and recover from failures automatically, for example by restarting failed services or rolling back failed deployments. Consequently, the system will be able to recover from failures without the need for manual intervention if there are any.

- Developing monitoring and alerting systems. Monitoring systems are systems that are capable of monitoring the health and performance of the services, and that alert the appropriate teams when a problem or failure is detected. By doing so, issues can be identified quickly and addressed before they become serious problems.

As a whole, when it comes to building a microservices architecture for failure, we need to design it to be resilient and self-healing and implement mechanisms for detecting and recovering from failures. Providing a more reliable and stable platform for running an application can ensure that the system can continue to function even when individual components or services fail, as well as providing a more stable platform for operating the application.

Monitoring microservices: Best practices for testing and debugging microservices

Having a test environment up and running for an application built in one codebase does not require much effort. Things are more complicated when it comes to microservices. The unit tests will still be quite similar to those of the monolith and should not be any more painful. However, integrating and testing systems can be more challenging. Your setup might require several services to be started simultaneously, different datastores to be set up, and message queues that you did not need with your monolith. Due to the increasing number of moving parts, it is very difficult to predict the different types of failures that can occur when functional tests are run in this situation. Here are some ways to improve monitoring of microservices:

- **Adopt Continuous Integration and Delivery (CI/CD)**: CI/CD practices can help to automate the process of building, testing, and deploying microservices. This can help to reduce the risk of errors and make it easier to detect issues early on in the development process.
- **Use logging and tracing**: Logging and tracing can provide visibility into the behavior of microservices at runtime. By logging key events and tracing requests across different services, you can gain insights into the performance of the system and identify potential issues.
- **Implement health checks**: Health checks can be used to monitor the status of individual microservices and ensure that they are running properly. By implementing health checks, you can detect issues early on and take action to prevent service outages.
- **Use metrics:** Metrics can provide insights into the performance and behavior of microservices over time. By collecting metrics on key performance indicators such as response times, error rates, and resource usage, you can identify trends and potential issues before they become critical.

- **Use APM Tools**: **Application Performance Monitoring** (**APM**) tools can provide deep insights into the behavior of microservices at runtime. By using APM tools, you can track performance metrics, detect errors, and diagnose issues quickly and efficiently.
- **Implement load testing**: Load testing can be used to simulate high levels of traffic on your microservices architecture. By load testing your system, you can identify potential bottlenecks and performance issues before they impact your users.
- **Monitor third-party services**: Microservices often rely on third-party services such as databases, message queues, and APIs. It is important to monitor the performance of these services and ensure that they are running properly to prevent issues from affecting your microservices architecture.

By monitoring, you can identify issues early and take action accordingly. Different services have different baselines, and you need to understand them, as well as react when they behave unexpectedly. Microservice architectures are resilient to partial failures, which is one of their advantages.

Embracing continuous delivery with DevOps

Continuous delivery is a crucial aspect of microservice architecture, and adopting DevOps tools and practices is essential to make it work effectively. The continuous delivery process involves integrating code changes, building and testing them, and then deploying them to the production environment automatically. This automation process helps ensure that new features and fixes can be released faster and more reliably.

DevOps practices involve collaboration and communication between development and operations teams to automate and streamline the entire software delivery process, from code development to deployment and monitoring. By adopting DevOps practices, organizations can improve their deployment frequency, reduce deployment failure rates, and shorten the time between fixes. One of the key benefits of DevOps in a microservice architecture is that it allows teams to deploy services independently of each other. By breaking down the application into smaller components, teams can work on specific services independently, without disrupting other services. This enables teams to release services more frequently, iterate faster, and get feedback from users more quickly. Another important aspect of DevOps in microservice architecture is the ability to automate the testing process. As the number of services increases, testing them manually becomes impractical. Automated testing can help detect problems early on in the development cycle, reducing the risk of deployment failures.

Overall, embracing continuous delivery with DevOps tools and practices is critical to the success of a microservice architecture. It enables faster and more frequent deployments, greater flexibility, and more resilience to failures.

Enabling technologies for microservices

Microservices architecture relies heavily on containerization technology to encapsulate individual services into self-contained units that can be easily deployed and managed. Docker containers are one such technology that allows developers to package an application and its dependencies into a portable container image. This image can then be deployed to any environment that supports Docker, making it easy to move the application across different platforms without having to worry about compatibility issues.

One of the major benefits of using Docker containers in a microservices architecture is that it allows applications to be independent of the host environment. Each container runs in its own isolated environment, with its own set of resources allocated to it. This makes it easier to manage resources and ensures that each service runs in a consistent and predictable environment.

To manage these containers and deploy them to a production environment, a container orchestration tool is typically used. Kubernetes is one of the most popular container orchestration tools available today, and it provides a robust and flexible platform for deploying and managing containerized applications.

For example, consider an e-commerce application that consists of multiple microservices, such as a product catalog service, a shopping cart service, and a payment processing service. Each of these services can be encapsulated in a Docker container, which is then deployed to a Kubernetes cluster. Kubernetes can automatically scale these services up or down based on demand, ensure that each service is running correctly, and provide a range of other management and monitoring features.

Overall, Docker and Kubernetes are two essential technologies for building and deploying microservices-based applications. By encapsulating services in containers and leveraging container orchestration tools like Kubernetes, developers can build scalable, resilient, and highly available applications that can be easily managed and updated.

Docker and microservices: Use cases for containerization

The Docker platform is an open-source platform for developing, deploying, running, updating and managing containers. The use of containers simplifies the

development and delivery of distributed applications. With the move towards cloud-native development and hybrid multicloud environments, they have become increasingly popular. Linux and other operating systems offer developers the ability to create containers without Docker. However, Docker simplifies, speeds up, and secures containerization.

For example, as illustrated in *Figure 3.3: Docker – Develop faster. Run anywhere* functionality enables us to build and run application efficiently.

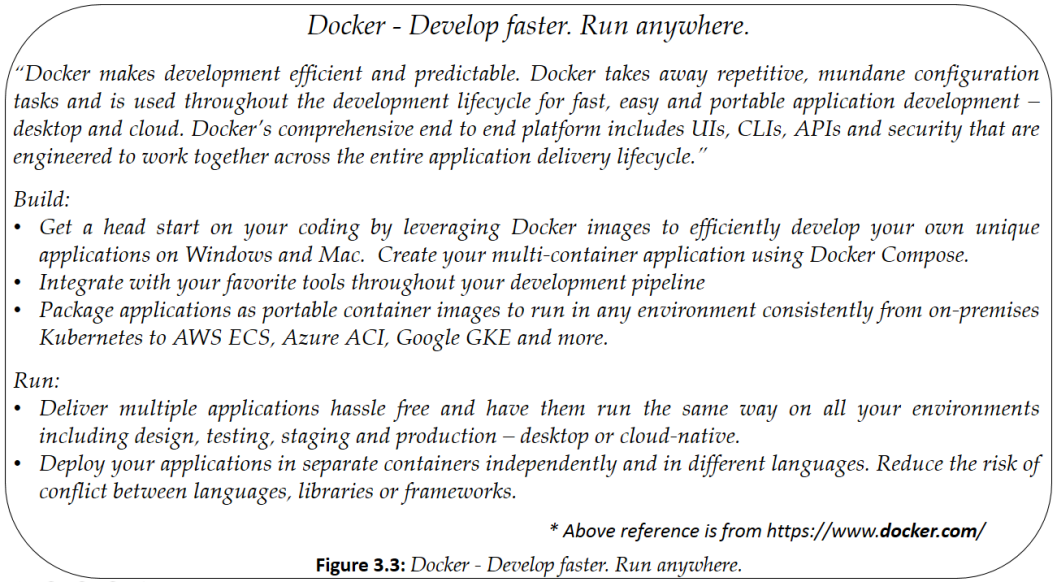


Figure 3.3: *Docker - Develop faster. Run anywhere.*

A number of advantages make Docker containers particularly popular over **virtual machines** (**VMs**). VMs contain complete copies of an operating system, the application, and the necessary binaries and libraries. Storage capacity for this usually amounts to dozens of gigabytes. Additionally, VMs can be slow to boot, unlike Docker containers. On the other hand, Docker containers usually require less storage space because they contain images that are only a few megabytes in size. Docker allows fewer virtual machines and operating systems to be used, allowing more applications to be processed. Containers can also be used on edge devices, such as Raspberry Pis, which are compact single-board computers.

The ability to run different applications on a single operating system instance greatly enhances diverse deployment options. A key advantage of Docker containers is their ability to isolate apps not only from one another, but also from their underlying systems.

Using Docker: Exploring the benefits of containerization

- Containers are lighter than VMs, as shown in the *Figure 3.3*, since they do not carry a full OS instance and hypervisor. They only need OS processes and dependencies to execute code.
- Containerized applications can be written once and run anywhere, resulting in increased developer productivity. Also, containers can be deployed, provisioned, and restarted much faster and easier than virtual machines. As a result, they are ideal for use in **continuous integration and continuous delivery** (**CI/CD**) pipelines, as well as in Agile and DevOps development teams.
- **Resources can be utilized more efficiently with containers**: Containers allow developers to run multiple copies of applications on the same hardware as they can with virtual machines.
- **Versioning of container images**: Docker allows you to track versions of a container image, roll back to a previous version, and even see who built the version.
- Utilizing existing containers as templates for building new ones is an excellent way to reuse containers.

For example, as illustrated in *Figure 3.4: Virtual Machine Vs Docker* the advantages of Docker over VM related to the resource consumption has been highlighted.

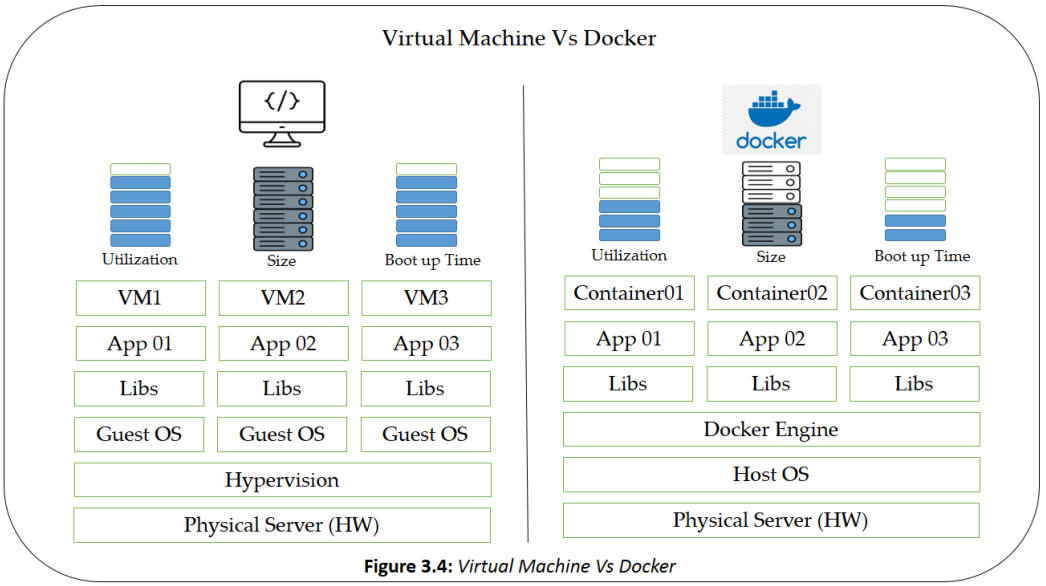


Figure 3.4: *Virtual Machine Vs Docker*

Key components of Docker

These are core components that are commonly used in Docker-based applications. They work together to provide a platform for building, deploying, and running applications in containers.

- **DockerFile**: DockerFile automates the process of creating Docker images. To assemble the image, Docker Engine runs a series of **command-line interface** (**CLI**) instructions.
- **Docker images**: In a Docker image are the executable source code and all the tools, libraries, and dependencies needed to run the code in a container. A Docker image is made up of layers, each corresponding to a version. New top layers are created whenever a developer updates the image, and these top layers replace the previous top layers. Previously saved layers can be rolled back or reused in other projects.
- **Docker containers**: A Docker container is an instance of a Docker image that is running live. Containers are executable files, unlike Docker images, which only exist as read-only files. Administrators can use Docker commands to adjust their settings and conditions.
- **Docker daemon**: Using commands from the client, Docker daemon creates and manages Docker images. A Docker daemon acts as your Docker implementation's control center.
- **Docker registry**: Docker registries are open-source storage and distribution systems for Docker images. Through the registry, you can track images in repositories and identify them by tagging. Version control software, such as git, is used to accomplish this.
- **Docker Engine**: This is the core of the Docker platform and is responsible for managing the containers.
- **Docker Hub**: This is a cloud-based registry service that allows users to share and find Docker images.
- **Docker Compose**: This is a tool for defining and running multi-container Docker applications. It allows users to specify the dependencies between containers in a single file, making it easy to manage complex applications.
- **Docker Swarm**: This is a clustering and scheduling tool for Docker containers. It allows users to create a cluster of Docker hosts and schedule containers to run on them.

In terms of building and running containers, Docker is one of the most popular options. This system is based on a client-server architecture. Docker clients communicate with Docker daemons, which build, run, and distribute Docker containers. Clients

and daemons can run on the same system, or clients can connect to remote Docker daemons. UNIX sockets or a network interface are used to communicate between the Docker client and daemon.

Container orchestration with Kubernetes

The Kubernetes container orchestration platform is also called "K8s" or "Kube" and is used to schedule, manage, and scale containerized applications. In addition to its breadth of functionality, open-source ecosystem, and portability across cloud providers, developers chose and still choose Kubernetes for its breadth of functionality, open-source ecosystem, and wide range of sharing tools. There are several public cloud providers that offer managed Kubernetes services, including **Amazon Web Services** (**AWS**), Google Cloud, IBM Cloud, and Microsoft Azure.

For example, as illustrated in *Figure 3.5: Kubernetes – Production Grade Container Orchestration* explains how K8 will help enterprise to scale.

Kubernetes - Production-Grade Container Orchestration

"Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications."

It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community.

Planet Scale: Designed on the same principles that allow Google to run billions of containers a week, Kubernetes can scale without increasing your operations team.

Never Outgrow: Whether testing locally or running a global enterprise, Kubernetes flexibility grows with you to deliver your applications consistently and easily no matter how complex your need is.

Run K8s Anywhere: Kubernetes is open source giving you the freedom to take advantage of on-premises, hybrid, or public cloud infrastructure, letting you effortlessly move workloads to where it matters to you.

** Above reference is from https://**kubernetes.io/***

Figure 3.5: *Kubernetes - Production-Grade Container Orchestration*

By automating deployments, updates (rolling-updates), and managing our apps and services, Kubernetes can speed up the development process. In addition, it has self-healing capabilities. When a process within a container crashes, Kubernetes can detect and restart it. Kubernetes schedules, monitor, and automates container-related, including:

Advantages of using Kubernetes: Orchestration for scalability and availability

- **Deployment**: Maintain a desired state for a specified number of containers deployed on a specified host.

- **Rollouts**: In Kubernetes, you can initiate, pause, resume, or roll back rollouts.
- **Service discovery**: Using a DNS name or IP address, Kubernetes can expose a container to the internet or to other containers.
- **Storage provisioning**: Your containers can be mounted to persistent local or cloud storage via Kubernetes.
- **Load balancing**: Performance and stability can be maintained by Kubernetes load balancing based on CPU utilization or custom metrics.
- **Autoscaling**: Using Kubernetes autoscaling, you can spin up new clusters when traffic spikes.
- **Self-healing**: To prevent downtime when a container fails, Kubernetes restarts or replaces it automatically. Containers that do not pass the health check can also be taken down.

For example, as illustrated in *Figure 3.6: Container Orchestration with Kubernetes* explains it is key components and how they interact and manage your application.

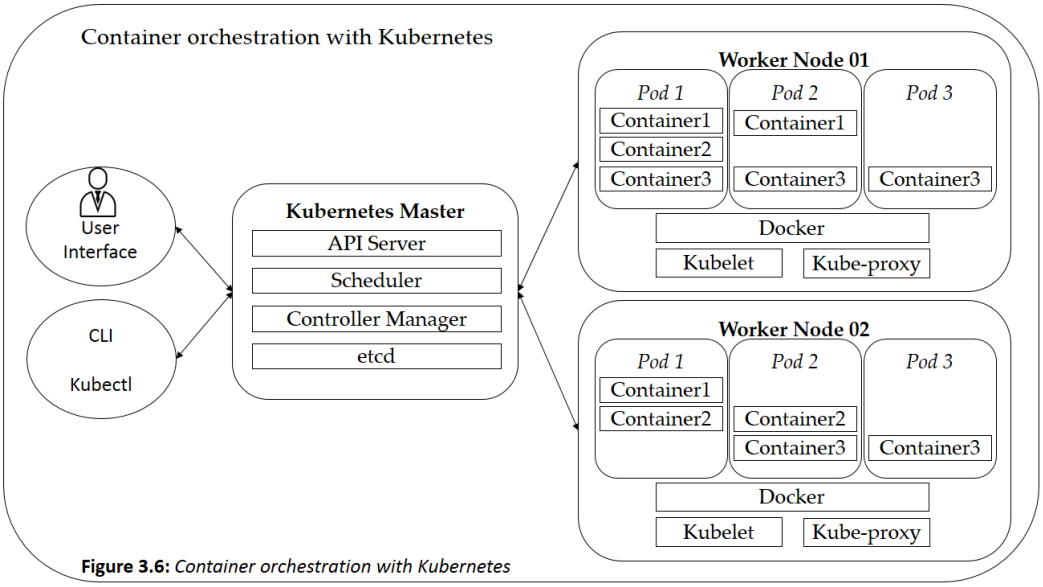


Figure 3.6: *Container orchestration with Kubernetes*

Components of Kubernetes

Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications. It consists of a number of components, including:

- **Kubectl**: To interact with the master node of Kubernetes, Kubectl is a **command line tool** (**CLI**).
- **Kubernetes master:** A Kubernetes Master is a node that manages all Kubernetes clusters. The orchestration of the worker nodes is handled by it.
    - **API Server**: A Kube API Server is a frontend for the Kubernetes control plane, interfacing with APIs.
    - **Scheduler**: Pods are watched by the scheduler and are assigned to specific hosts.
    - **Controller-Manager**: Controller manager runs the controllers in background.
        - ❖ A node controller is responsible for detecting and responding to node failures.
        - ❖ A replication controller determines how many identical copies of a pod should be running on a cluster.
        - ❖ Services and pods are connected by endpoint controllers.
        - ❖ Access management is handled by the Services account and Token controllers.
        - ❖ Pod replication is ensured by ReplicaSet controllers.
        - ❖ Pods and ReplicaSets are updated declaratively by the deployment controller.
        - ❖ The DaemonSets controller ensures that all nodes run a copy of specific pods.
        - ❖ A job controller supervises batch jobs carried out by pods.
        - ❖ Communication is enabled by services.
- **Etcd**: etcd is a key-value store. The data stored in etcd includes job scheduling information, pods, state information, etc.
- **Worker Nodes**: Worker nodes are the nodes where the application actually running in Kubernetes cluster. Kubelet processes are used to control each of these worker nodes.
    - **Kubelet**: Kubelet is the primary node agent runs on each nodes and reads the container manifests which ensures that containers are running and healthy.
    - **Kube-proxy**: Kube-proxy is a process helps us to have network proxy and loadbalancer for the services in a single worker node. It performs TCP/UDP network routing and connection folding.

- **Pods**: A group of one or more containers deployed to a single node.
    - Containers in a pod share an IP Address, hostname and other resources.
    - Containers within the same pod have access to shared volumes.
    - Pods abstract network and storage away from the underlying container. Containers can be moved around the cluster more easily this way. Horizontal pod auto scaling allows pods of a deployment to be automatically started and stopped based on CPU usage.
    - Each Pod has its unique IP Address within the cluster.
- **Deployment**: A deployment is a blueprint for the Pods to be created. Handles update of its respective Pods. This will keep the pods running and allow them to be updated (with rolling updates) in a more controlled manner.
- **Service**: A service is responsible for making our Pods discoverable inside the network or exposing them to the internet.

These are the main components of Kubernetes, but there are many others that are part of the system, such as various CLI tools and various plugins and extensions that can be added to the system to extend its capabilities.

Alternatives to container orchestration: Other tools

Kubernetes may be the default choice for container orchestration, but it is worth exploring alternatives as well.

Using Docker's swarm mode, you can natively manage a cluster of Docker engines. By using Docker CLI, you can create a swarm, deploy application services to it, and manage it. Other container orchestration systems include Apache Mesos and Marathon.

In terms of container orchestration, Kubernetes has emerged as the top choice for enterprises. Containerized applications have the advantage of being portable. On-premises and cloud applications can run simultaneously. There are now managed Kubernetes services offered by three of the most popular cloud service providers - AWS, Microsoft Azure, and Google Cloud. Kubernetes generally has the same core functionality, but each cloud provider may offer different features on top.

- Amazon Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)
- Google Cloud Kubernetes Engine (GKE)

Microservices adoption using Domain Driven Design

Domain-driven design is one of the key design principles for migrating a monolithic application to a microservices architecture. In this process, the application is refactored into smaller services based on domain grouping. The process of completing specific business goals or solving a particular domain problem requires deep understanding of a particular domain. To solve specific business problems, developers work closely with domain experts to design goal-oriented application design models.

- This model offers the tightest alignment with business domains and the most flexible and agile IT model, resulting in a shorter time-to-market.
- The product-centric approach helps drive an engineering culture across the organization by clarifying ownership and squad independence.
- Domain alignment reduces enterprise-wide duplication of capabilities (both applications and data).
- By promoting end-to-end ownership and a continuous improvement culture, this model builds deeper domain and functional skills in squads.

Domain-driven application decomposition steps

There is no one specific set of steps for decomposing an application using **domain-driven design** (**DDD**), because the process varies depending on the specific requirements and characteristics of the application. It is important to note, however, that there are some general steps that are commonly followed when decomposing an application using DDD:

- Identify the core business domains and sub-domains within the application. In order for this to be successful, it is necessary to understand both the business requirements and objectives of the application as well as the various entities, relationships, and processes that are involved in it.
- A domain model provides a conceptual model of the business domain, which illustrates how the various entities and their relationships are connected within that domain, as well as their relationships with each other. This is done by defining the key concepts and entities within each domain, and defining their relationships and interactions between those entities.
- In each domain, define the bounded contexts, which are specific areas within each domain within which certain rules and models apply in order

to achieve certain objectives. As well as ensuring that the domain model remains consistent and cohesive within a single area, bound contexts also allow for flexibility and variation in other areas of the model.

- It is important to identify the interfaces and integration points between the various bounded contexts, as well as the communication and collaboration mechanisms which will be used to ensure seamless integration. In order to integrate the different bounded contexts in the application, it is necessary to define the architecture and design of the application, including all of the components, services, and APIs that will be used.
- Implement the application using the defined architecture and design, following the principles and practices of DDD in order to ensure that the domain model is accurately represented and that the application is aligned with business objectives and requirements. As part of this process, the various components, services, and APIs that make up the application are implemented along with the infrastructure and deployment mechanisms which are required.

The purpose of domain-driven application decomposition is to decompose the application into smaller, more focused, and cohesive modules that are more accurately reflecting the underlying business domains and that can be easily integrated and maintained by the end user. This can be beneficial for improving the design, maintainability, and flexibility of the application, and it will make it much easier to evolve and adapt to changing business requirements as time goes on.

Short case study 06: Insurance Claim Processing

Let us review an example of how logical flow for an insurance claim processing application can be designed as microservices architect.

For example, as illustrated in *Figure 3.7: Logical flow - Insurance Claim* processing explains the business side of the requirement.

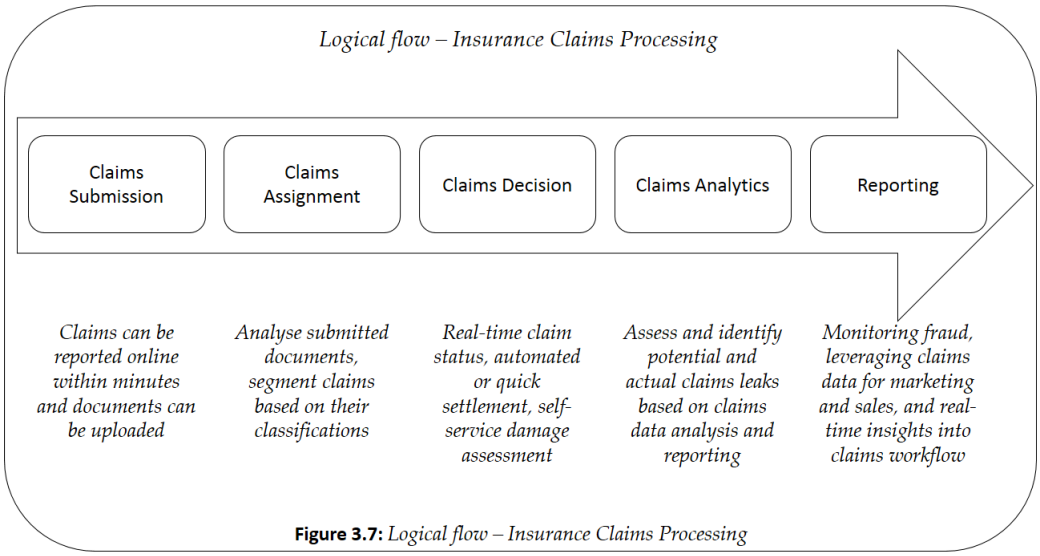


Figure 3.7: *Logical flow – Insurance Claims Processing*

Logical flow of insurance claim processing converted in a high level architecture flow using Domain-driven design principles. For example, as illustrated in *Figure 3.8: Architecture - Insurance Claim* processing explains the microservices design side of the logical flow explained in *Figure 3.7* requirement.

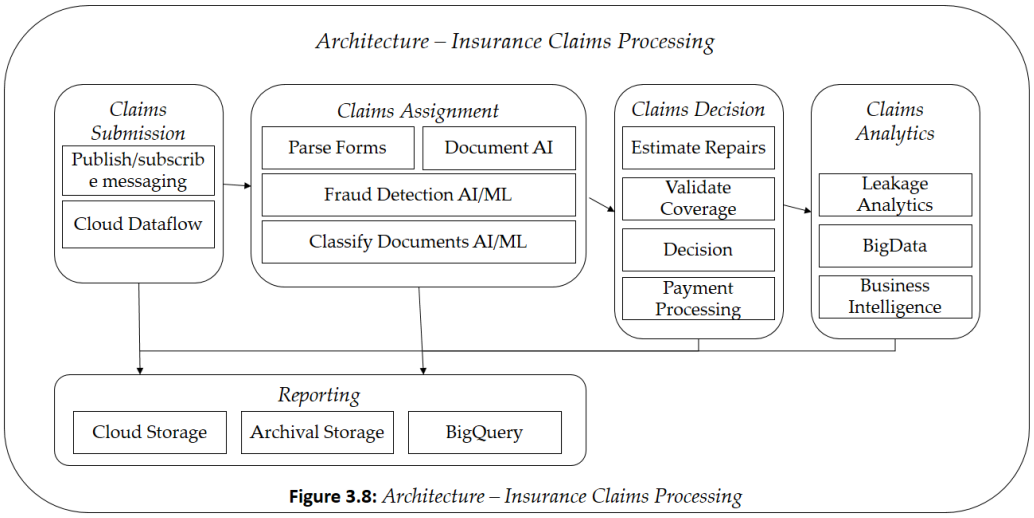


Figure 3.8: *Architecture – Insurance Claims Processing*

As represented in previous architect diagram for Insurance Claims processing functionality has been designed as a set of microservices specifically designed to perform specific task and communication flow designed for achieving the end results as follows:

- **Claims Submission** with two microservices will cover consume submission of claims via any channel i.e. online forms, paper PDFs and scanned images and perform inline data transformation to store data for near and long term analysis.
- **Claims Analysis** will have four microservices performing advanced analytics to intelligently accelerate claims processing, parsing form fields and tables from ingested documents, using AI to classify claims documents, and performing claims segmentation and run fraud detection models.
- **Claims Decision** again will have four microservices making a decision on claims significantly faster and accurately, validating claim, predicting payment and automatically settling claims.
- **Claims Leakage Analytics** will have another set of microservices to analyse and create models to gain insight, to analyse average settlement cost trends, and to understand subrogation recovery trends, and so on.
- **Reporting** will help in analyse and by data visualization models.

Let us expand further as your microservices / applications will need to have Infrastructure, Security, and Operations related services linked with it. For example, as illustrated in *Figure 3.9: Architecture - Insurance Claim* processing explains the microservices design along with key components related to Security and Infrastructure completing the Architect explained in previous *Figure 3.8.*

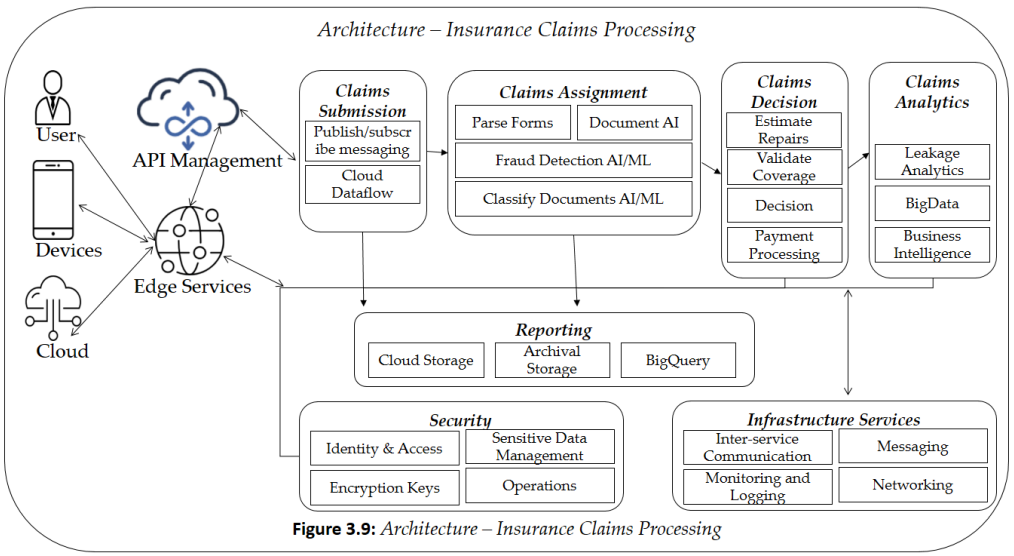


Figure 3.9: *Architecture – Insurance Claims Processing*

- **API Management**: API management will manage APIs across clouds and on-premises. It will include API gateway for traffic optimization.
- **Edge Services**: Edge services (if applicable) will be another touchpoint for your application egress and ingress connections directly from end users.
- **Infrastructure services**: Cloud environments complement microservices by providing networking, messaging, microservice communication, logging and monitoring, virtualisation, service discovery and proxying, resiliency features, and more.
- **Security**: We will not have application / microservices on cloud without security framework integration. It will have services related to identify and access management, sensitive data management, encryption keys, and security related operations.

Using microservices correctly: Characteristics

A microservices design typically consists of the following characteristics:

- **Fine-grained and focused**: Rather than being monolithic and handling multiple unrelated functions, microservices are typically designed to be fine-grained and focused on a specific business capability or function. The overall architecture becomes more modular and adaptable by allowing each service to be developed, tested, and deployed independently of the others. For example, the document submission microservice can be updated without affecting the payment processing microservice.
- **Loosely coupled and highly cohesive**: Each microservice has a clear and well-defined responsibility and interface for communicating with other services, so microservices are typically loosely coupled. As a result, the services can be developed, tested, and deployed independently of one another, improving overall flexibility and adaptability. For instance, the payment processing microservice can communicate with the document submission microservice via APIs or other communication mechanisms.
- **Scalable and resilient**: Typically, microservices are built to handle large volumes of traffic and requests and to recover from failures and disruptions without affecting the application's overall functionality. In order to achieve this, redundant and backup services are used, circuit breakers and retry mechanisms are used, and self-healing and recovery mechanisms are used. For example, in the case of insurance claim processing, the system must be built to handle large volumes of traffic and requests, and to recover from failures and disruptions without affecting the overall functionality of the application.

- **API-driven and event-driven**: It is typical for microservices to be API-driven and event-driven, which means that services communicate with one another via APIs or other communication mechanisms, and events trigger actions and reactions within the system. As a result, the services are loosely coupled and can interact dynamically and flexible. For example, when a new claim is submitted, an event can be triggered, which in turn can trigger the document submission microservice to start processing the documents.
- **Built and deployed independently**: In general, microservices are designed to be built and deployed independently of one another, with each service having its own codebase, build pipeline, and deployment process, all of which are separate from the others. By doing this, it makes it possible to develop, test, and release the services independently of each other, which in turn makes the architecture a lot more flexible and agile. For instance, the document submission microservice can be updated and deployed without affecting the payment processing microservice.

In breaking down a target system into constituent services, there is a real risk of decomposition occurring along existing boundaries within the organization. There are more independent parts to manage, and parts that do not integrate well. This is despite the fact that the resulting services can communicate with one another, which makes the system potentially more fragile.

Business objectives must be taken into account when designing microservices. Rather than using microservices to route calls between teams, this might require teams across organizational boundaries to collaborate and design services together.

For example, as illustrated in *Figure 3.10: Architecture Using Microservices with flaws* as each organization has its own activation and order management system, and each view of the customer is different, so the ordering system cannot scale and from business perspective as well customer management will have issues.

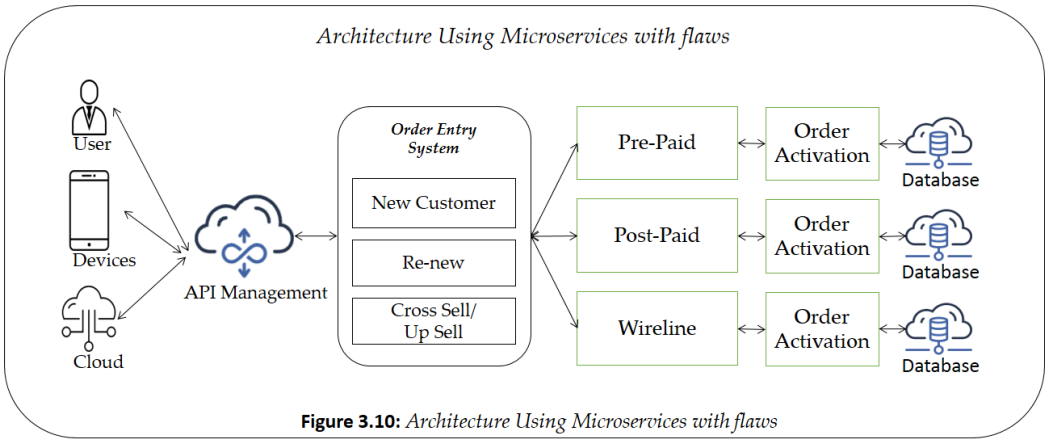


Figure 3.10: *Architecture Using Microservices with flaws*

For example, as illustrated in *Figure 3.11: Re-designed Architecture Using Microservices*. A single unified view of the customer is provided by the database once an order has been received. Orders are sent to an orchestration service, which contacts each individual ordering system as necessary. It is important to understand that we are not splitting customer database in this case just because we are implementing microservices design. For this scenario keeping order management data for all service lines for a customer will be stored together for better overall management and analytics.

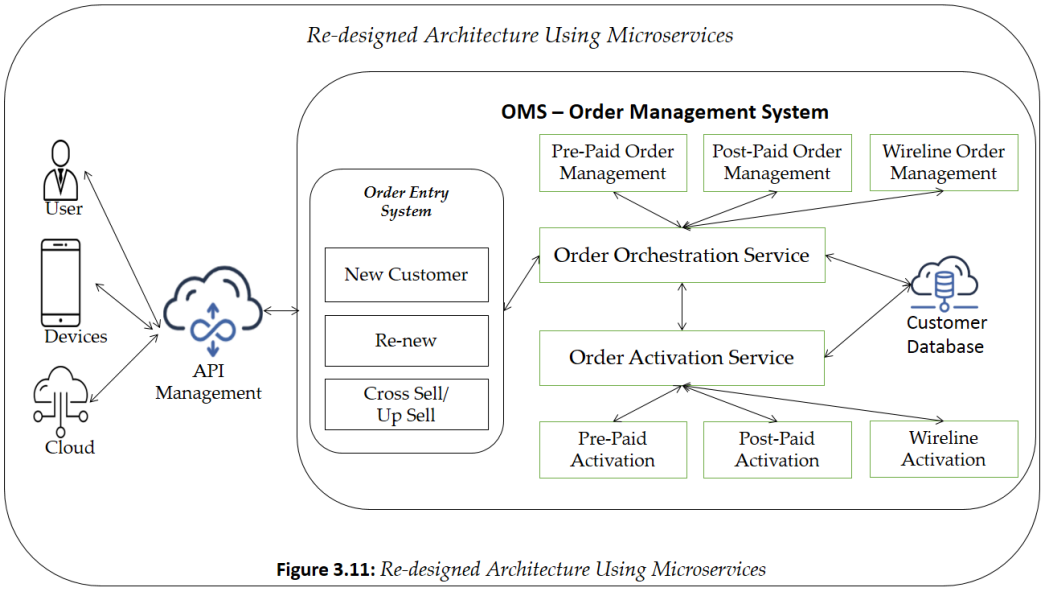


Figure 3.11: *Re-designed Architecture Using Microservices*

To make microservices as loosely coupled as possible, sharing between microservices should be kept at a minimum. It is important to carefully consider how to share the database across microservices, whatever the reason may be. It does violate some of the principles of a microservices-based architecture to share a database. In this case, the two services sharing a database must coordinate changes in the shared database, as the context is not bounded. Basically, for the functional logical from of the application sharing or keeping Customer DB intact makes more sense in above case discussed in *Figure 3.11*.

Short case study 07: Modernizes Architecture Using Microservices

In this case study, volume of players' and their data stored in a RDBMS causing issues. During peak times, the monolithic SQL architecture was not scalable or efficient, causing downtime. As part of its strategy to improve backend efficiency, increase developer productivity, and reduce costs, the company decided to modernize its

architecture using microservices on AWS. For example, as illustrated in *Figure 3.12: Modernizes Architecture Using Microservices.*

MovieStarPlanet Modernizes Architecture Using Microservices

"MovieStarPlanet Modernizes Architecture Using Microservices on AWS, Saving Over 40% on Hosting Costs. With over 400 million registered players, MovieStarPlanet has a lot of data to store. But scaling was a struggle on the company's aging SQL-based monolithic architecture, and servers could take up to 20 minutes to boot. To create a better experience for its players, they decided to modernize its architecture. By modernizing using microservices on Amazon Web Services (AWS), MovieStarPlanet improved engagement and retention, reduced dynamic hosting costs by over 40 percent, and increased developer productivity, helping create new experiences for players."

** Above reference is from*
https://aws.amazon.com/solutions/case-studies/moviestarplanet-case-study

Figure 3.12: *MovieStarPlanet Modernizes Architecture Using Microservices*

Learning from the above example

- Their solution is Amazon Neptune, a serverless graph database that provides fast, reliable, and fully managed service that is easy to build and run applications on.
- Its purpose-built databases allow storing and navigating relationships, and one of its main advantages is that for different microservices you can choose the appropriate database. Using nodes instead of tables allows nodes to store data, and there's no limit to the number and kind of relationships a node can have-which makes them perfect for social applications.
- A microservices architecture is employed in Amazon EC2, a web service that provides resizable, secure compute capacity in the cloud.
- Kubernetes Service (Amazon EKS) implemented as a managed container service to run and scale Kubernetes applications in the cloud or on premises.
- Consequently, the company on longer is concerned about outages that can occur during peak times or during game launches due to server overload.

Using microservices correctly: Characteristics

Organizations can benefit from a number of key practices and considerations that can help them make use of microservices correctly and effectively. Prior to microservices becoming trendy, many companies built monoliths that could not be managed effectively. It was a real challenge to scale, manage, and evolve such a large application effectively in the cloud, which clashed with modern management and scaling practices. Here are some situations where microservices are not appropriate.

- **Microservices should not be your first step**: The point here is that you shouldn't jump straight into a microservices architecture without understanding the requirements of your application. It's better to start with a monolithic architecture and only move to microservices when you need to scale. For example, a small e-commerce startup might start with a monolithic application for their online store and only switch to a microservices architecture when they need to handle more traffic and scale up their backend.

- **DevOps is essential for microservices**: Managing a microservices architecture can be complex, so you need a DevOps team that can automate deployment and monitoring. For example, a company that uses microservices for its financial services platform might have a DevOps team that uses tools like Kubernetes and Prometheus to deploy and monitor the microservices.

- **Managing your own infrastructure is not a good idea**: When you're dealing with multiple databases, message brokers, and data caches in a microservices architecture, it's helpful to use a **Platform as a Service** (**PaaS**) solution to manage your infrastructure. For example, a healthcare provider that wants to develop a patient management system using microservices could use a PaaS provider like **Amazon Web Services** (**AWS**) or Microsoft Azure to manage their infrastructure.

- **Avoid creating too many microservices**: It's important to strike a balance between breaking down your application into small, manageable microservices and keeping the complexity under control. For example, a social media platform that wants to implement microservices should group related functionalities together and create a manageable number of microservices instead of creating a separate microservice for every feature.

- **Value identification is the key step**: Before implementing microservices, you should clearly understand the business requirements and objectives that the architecture will support. For example, a financial institution that wants to implement microservices should first identify the business goals it wants to achieve and then design the microservices architecture accordingly.

- **Determine the granularity level**: Microservices should be finely tuned and focused on a single business capability or function, instead of being monolithic and handling a wide variety of unrelated functions simultaneously. For example, an e-learning platform that wants to implement microservices should identify the appropriate granularity level for each service, such as a microservice that handles course registration or a microservice that handles course content delivery.

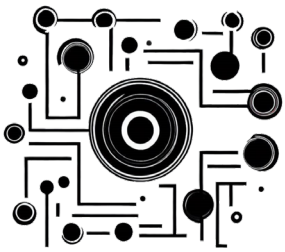
- **Well defined boundaries between microservices**: Each microservice should have a clear responsibility and a well-defined interface for communicating with other microservices. By doing so, the microservices will be loosely coupled, so they can be developed, tested, and deployed independently. For example, a service that handles user authentication should not overlap with a service that handles product search.
- **APIs based communication mechanisms**: In a microservices architecture, communication between microservices is crucial. APIs are a great way to ensure that microservices can communicate with each other in a robust and scalable manner. For example, a microservices architecture might use RESTful APIs or message-based communication systems like gRPC or Kafka.
- **Observability is essential**: Monitoring and observability systems help identify and diagnose issues and failures by providing visibility into the performance and behaviour of microservices. For example, a company that uses microservices for its travel booking platform might use tools like Jaeger and Grafana to monitor and observe their microservices architecture.
- **Keep an eye out for potential latency issues**: When microservices are dependent on each other, it's possible to introduce latency. Performance testing is important to identify any sources of latency in the microservices architecture. For example, a video streaming service that uses microservices for encoding and delivery might run performance tests to ensure that the latency between the two microservices is kept to a minimum.
- **Security should be a top priority**: With microservices, there are multiple entry points to a system that must be secured. Each microservice should be designed with security in mind, and the communication between them should be secured as well. It is also important to have strong authentication and authorization mechanisms in place. For example, a financial institution that uses microservices for its online banking system should implement secure communication protocols such as HTTPS, use encryption for sensitive data, and enforce access control policies for each microservice.
- **Plan for testing and continuous integration**: Microservices require a significant amount of testing, especially when making changes or introducing new services. Implementing a testing strategy that includes unit tests, integration tests, and end-to-end tests is essential. **Continuous integration** (**CI**) should also be used to ensure that changes made to one microservice do not negatively impact other microservices or the system as a whole. For example, an e-commerce company that uses microservices for its online ordering system should have a well-defined testing strategy that includes automated tests for each microservice and CI/CD pipelines that ensure changes are thoroughly tested before being deployed.

Conclusion

Unlocking Agility and Scalability:
Refactoring Monoliths to Microservices with Precision

A monolithic architecture can be migrated to microservices in a systematically organized manner but it is a time-consuming task to execute. Microservices have their own advantages over monolithic architecture, but a transition from monoliths to microservices must be justified. It may be fatal to make an unnecessary switch. When a system becomes too complex to maintain and operate, it should be refactored into microservices. As we have seen in this chapter, there are some steps you can take to make refactoring more sensible and manageable. We have discussed, Monolith vs Microservices application and how to build a microservice architecture by splitting the monolith. We have reviewed bounded contexts and failure resistant design and the importance of DevOps adoption. A successful Microservices implementation works well with technologies like Docker and Kubernetes for container orchestration. Domain Driven Design helps in logical split and we have seen scenarios to do it systematically to modernizes our architecture.

In next chapter, we will cover the design patterns for Microservices architecture and how to use them efficiently. Implementing the correct design pattern for your use case can help increase component reusability, thus reducing development time and effort. Each of these patterns will be reviewed using use cases with diagrammatic representations of their components and interconnections.



Chapter 4
Design Patterns for Microservices

Design Patterns that Power Microservices Architecture

Introduction

In a business environment, selecting the right architectural style and implementing it can save capital and human resources. When discussing software architecture, there are usually a lot of questions that arise, such as alternative patterns, what can happen if we adopt these patterns, and how they might negatively affect us. However, there are a wide array of needs in every business, which adds to the confusion, especially with the ever-increasing number of microservice patterns that are on the market. By using the right combination of patterns, you will be able to create services that are more scalable, secure, and maintainable.

In this chapter we will cover design patterns for microservices. The term "design pattern" refers to a general solution to a common software design problem that can often be repeated to resolve the problem. It is important to realize that design patterns are not plug-and-play blocks, but rather approaches to addressing particular situations that can be used for a wide range of issues as a description or template of how to deal with them. It is not a code block, and it does not plug-and-play. Using these patterns can help you speed up your development by applying tested, proven structured approaches to specific problems. As a result, we minimize known bottlenecks and minimize issues that may not be apparent until later in the implementation process. In addition to preventing subtle issues that can lead to major problems, coders and architects can improve code readability by reusing design patterns.

As an example, Google was affected by an outage on December 14, 2020 for approximately 45 minutes around the world. Due to the service-oriented architecture of Google, all of its services, such as Gmail, YouTube, Google Drive, and the like, were still working despite the issue affecting only one service, which is why the outrage went viral. It is clear that if the architecture had been monolithic, these Google services would not have been available as well.

We will be taking a look at how proven design patterns can be used to achieve the essential microservice design requirements for a modern application in this chapter.

Structure

In this chapter we will discuss following topics / design patterns:

- Design Patterns for Microservices
    - Decomposition Pattern
    - Decompose by Business Capability
    - Decompose by Subdomain
    - Decompose by Transactions
    - Decompose by Service per Team
    - Bulkhead Pattern for Resiliency
    - Sidecar Pattern for Service Mesh
    - Strangler Pattern for Legacy Systems
- Integration Pattern
    - API Gateway Pattern for API Management
    - API Aggregator Pattern for Composite Services
    - Gateway Offloading Pattern for Performance
    - Gateway Routing Pattern for Traffic Shaping
    - Asynchronous Messaging Pattern for Loose Coupling
    - Branch Pattern for Parallel Processing
    - Chained Microservices Pattern for Sequencing
- Database Management Pattern
    - CQRS (Command Query Responsibility Segregator) Pattern for Separation of Concerns
    - Database per Service Pattern for Decoupling
    - Shared Database per Service Pattern for Consistency

  - Event Sourcing Pattern for Auditing and Reconciliation
  - Saga Pattern for Long-Running Transactions

- Observability Pattern
  - Distributed Tracing Pattern for Root-Cause Analysis
  - Health Check API Pattern for Self-Healing
  - Log Aggregation Pattern for Centralized Logging
  - Application Metrics Pattern for Performance Monitoring
  - Audit Logging Pattern for Compliance
  - Exception Tracking Pattern for Debugging
  - Monitoring Vs Microservices Observability

- Cross-Cutting Concern Pattern
  - Blue-Green Deployment Pattern for Zero-Downtime
  - Canary Pattern for Incremental Rollouts
  - Canary Vs Blue-Green Deployment Pattern for Deployment Strategies
  - Circuit Breaker Pattern for Fault Tolerance
  - External Configuration Pattern for Dynamic Configuration
  - Service Discovery Pattern for Service Registration and Discovery

- Conclusion

Objectives

It is no secret that microservices present an entirely new set of challenges as a result of their distributed service-oriented architecture. In this chapter, we are going to discuss design patterns related to application decomposition into microservices, integration, data management, observability, as well as other cross-cutting concerns in order to achieve essential microservices design principles.

A number of widely used microservice design patterns will be explored in this chapter that will assist in the implementation of microservice architectures efficiently. You should always remember that if you follow the correct design pattern for your use case, you will be able to increase component reusability, resulting in a reduction in development time and effort as well. The use of microservice design patterns, however, can cause some problems that cannot be solved. In order to make the right choice, it is important to understand the caveats and advantages of each design pattern, in addition to the scenarios in which they are most appropriate, in order to make the best choice.

Design patterns for microservices

A pattern is a way for us to design an optimized structure for your microservices. There are many ways we can design and build microservices. As part of this course, we will review key design patterns used with an understanding of workflows, advantages, disadvantages, and scenarios best suited to each design pattern. When you use the correct design pattern, you can help to increase component reusability, which in turn can result in shortened development times.

For example, as illustrated in *Figure 4.1: Essential Microservice Design Principles* that we need to achieve using design patterns described in this chapter:

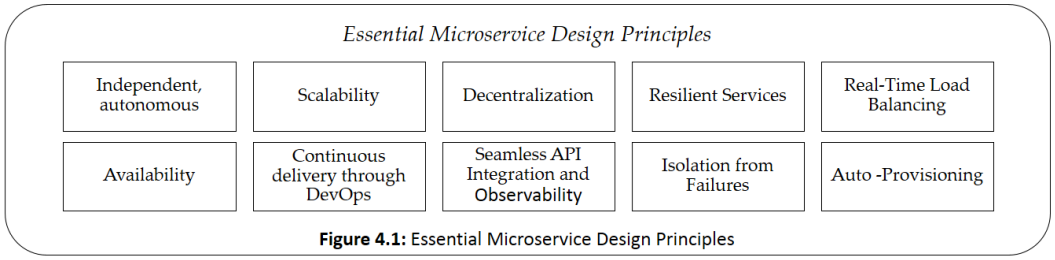


Figure 4.1: Essential Microservice Design Principles

For example, as illustrated in *Figure 4.2: Design Patterns for Microservices* is a list of key design patterns used for decomposition, integration, data management, observability, and other misc. cross cutting concerns. To help you choose the right design pattern for your application modernization, let's examine design patterns in detail.

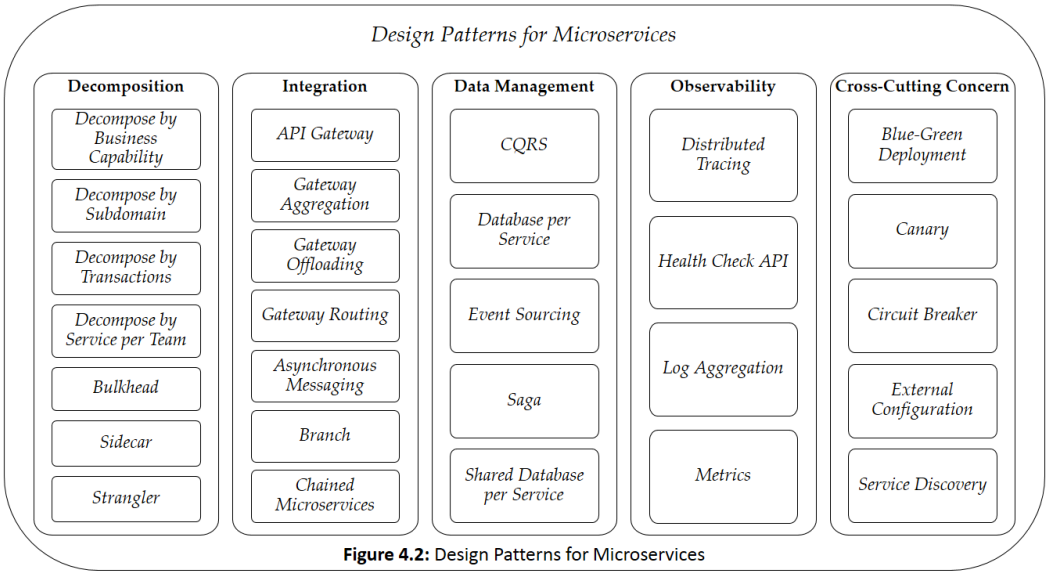


Figure 4.2: Design Patterns for Microservices

For example, as illustrated in *Figure 4.3: Microservices Design patterns within an Application* will give as a perspective how multiple patterns will be deployed in any of the microservice and the way they communicate with end users, internally with other services and with external interfaces and legacy systems.

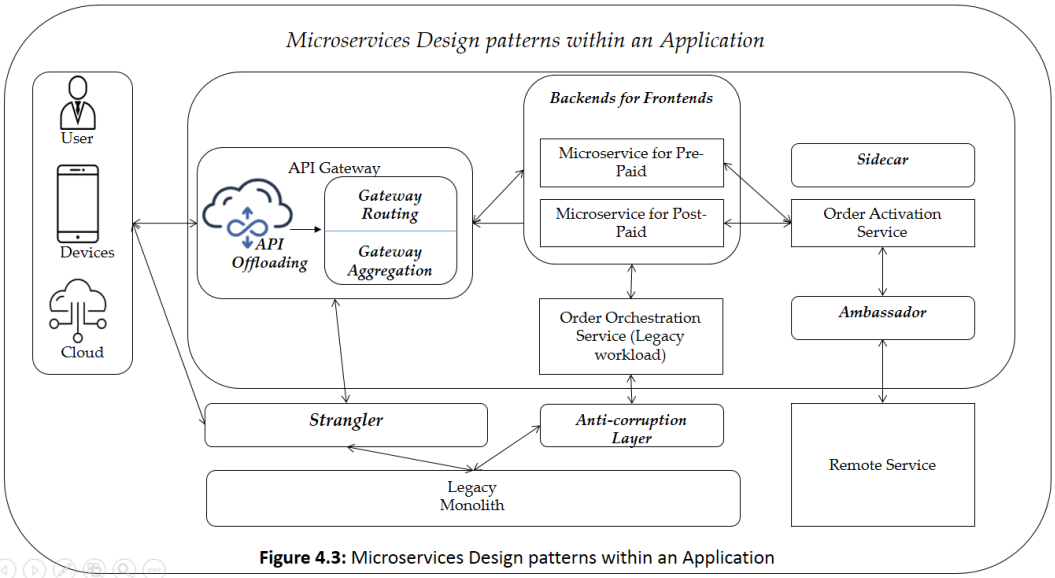


Figure 4.3: Microservices Design patterns within an Application

Decomposition pattern

It has been discussed previously in this chapter as well as the importance of domain driven design. One of the most critical steps is establishing the foundation of your new architect and how microservices will be designed, so it needs to be planned with care. In order for your application to scale and perform properly under higher workloads, decomposition strategies play a crucial role in determining how well it can scale and perform. As a rule, we do not want to decompose large monolithic applications and create another distributed monolithic system with its own set of issues and challenges.

Decompose by business capability

You can decompose a monolith using your organization's business capabilities. Business capabilities refer to what a business does to generate value (for example, sales, customer service, compliance, or marketing). As each industry and sector has its own set of capabilities, these capabilities differ from organization to organization. As long as your team has sufficient insight into your organization's business units and you have subject matter experts for each of them, you can use this pattern. Rather than creating development teams around technical features, these teams are created to deliver business value as opposed to technical features alone.

Advantages

- Services can be scaled independently of one another in order to meet the unique requirements of the business capability they provide. This can enable efficient resource utilization as well as the ability to handle a large number of users or transactions.
- An application can be made more resilient by decomposing it into a set of small services. When a service goes down, it will only affect a subset of the overall application, rather than the entire application.
- In addition to providing faster development cycles, increased flexibility in selecting the appropriate technology for the job, and better support for DevOps practices such as continuous integration and delivery, services can be developed, tested, and deployed independently of one another.
- Produce a stable microservices architecture when business capabilities are relatively stable.
- Rather than focusing on technical features, development teams deliver business value.

Disadvantages

- Communication between services may introduce latency if they are deployed in separate containers or on separate machines, which can negatively impact system performance.
- The design of the application is tightly coupled with the business model.
- Identifying business capabilities and services often requires a deep understanding of the overall business.

When to use this pattern

Businesses generate value by utilizing their business capabilities. Each business type has its own set of capabilities, which is what makes them unique. A team that has a good understanding of the organization's business units, along with additional experts who have expertise in specific business units, should be able to use a decomposition method by business capability. Insurance companies, for example, have a wide range of capabilities, including sales, marketing, underwriting, claims processing, billing, and compliance, among others. As a rule of thumb, we usually use this pattern as a starting point and then decompose it further.

Decompose by subdomain

It makes use of a **domain-driven design** (**DDD**) subdomain to decompose monoliths. A domain model is broken down into different subdomains that can be divided into different categories as core (a key differentiator for the organization), supporting (possibly connected to the organization but not a differentiator), and generic (not business-specific).

It is appropriate for existing monolithic systems that have clearly defined boundaries between the subdomain-related modules to use this pattern. The scope of the subdomain's model is called a bounded context; microservices are developed around this context, so you can repackage existing modules as microservices without significantly rewriting existing code. Consequently, microservices can be developed around existing modules without rewriting code significantly.

Similarly, decomposition by business capability provides the same benefits since the architecture is stable, since the subdomains are relatively stable. Furthermore, services are cohesive and loosely coupled which ensures maintainability.

For example, as illustrated in *Figure 4.4: Decompose by Business Capability and by Subdomain*.

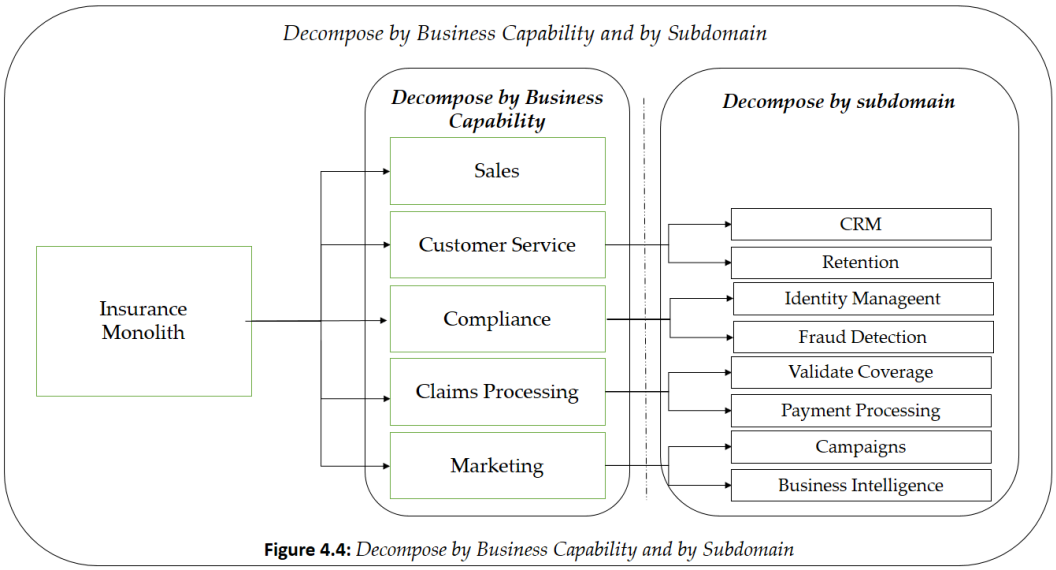


Figure 4.4: *Decompose by Business Capability and by Subdomain*

Advantages

- The benefits of loosely coupled architecture include scalability, resilience, maintainability, extensibility, location transparency, protocol independence, and time independence.

- As a result, systems become more predictable and scalable.

Disadvantages

- Too many microservices complicate service discovery and integration.
- An in-depth understanding of the overall business is necessary to identify business subdomains.

When to use this pattern

This pattern expands on the decomposition by business capability pattern. Identifying subdomains requires an understanding of the business and its organizational structure. It is best suitable approach when we have well defined boundaries for subdomains and teams responsible for single services present in an organization.

Decompose by transactions

When a business transaction is completed in a distributed system, multiple microservices are usually called by the application to accomplish it. It is a good idea to group your microservices based on transactions if you are concerned about response times and your modules do not form a monolithic system. You can avoid latency and two-phase commits if you use this pattern.

The purpose of microservices is to communicate with one another in order to complete a single transaction. A two-phase commit problem can occur if any of the services are unavailable at the same time during the transaction. The idea behind this pattern is to group microservices that are involved in the same transaction together in order to solve this problem.

In this case, the purchase service includes many other services. Once you click "checkout," your shopping cart will be retrieved, and then the order will be created. After payment has been received, an email will be sent, inventory status will be updated, and the shipping process will begin. In the event that a payment is complete, then the Validation Service will be used, followed by a new update for the Inventory Service.

Advantages

- Each service is designed to handle a specific transaction type, making it easier to understand and maintain the system, particularly when there are many services or when they are interdependent.
- It is possible to reuse transactions between different systems, allowing for the faster development of new systems based on common transactions.

- Data consistency is not a problem, and response times is faster.

For example, as illustrated in *Figure 4.5: Decompose by Transactions:*

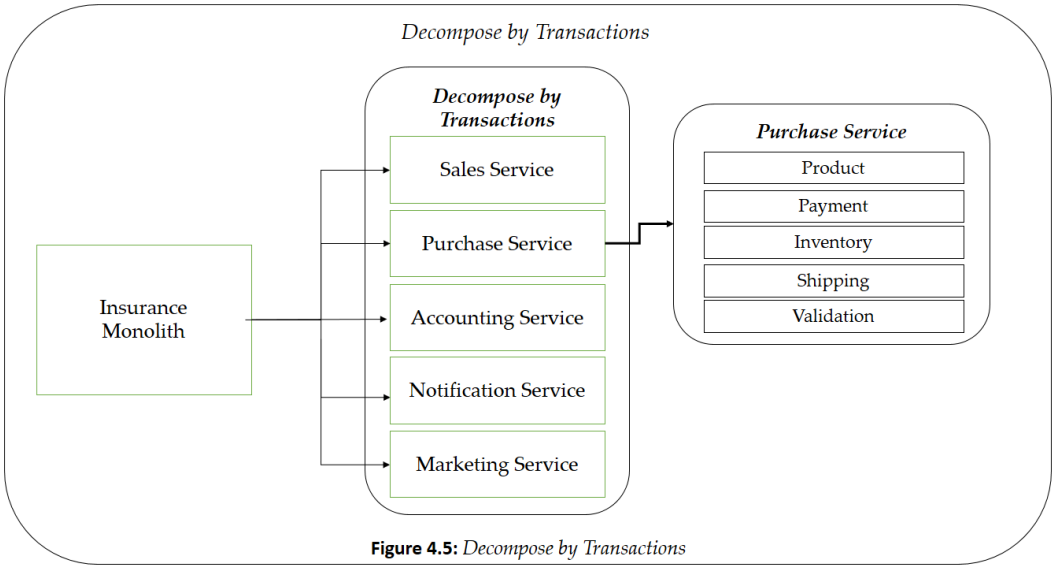


Figure 4.5: *Decompose by Transactions*

Disadvantages

- Due to the fact that multiple functionalities are implemented in a microservice rather than as separate microservices, cost and complexity are increased.
- When updating data, services that handle transactions may share data between them, which could lead to consistency issues and require additional coordination.
- The number of business domains and dependencies among them can affect the growth of transaction-oriented microservices. Example we have seen above will have issues to scale as multiple key payment process and customer experience services have been grouped together.

When to use this pattern

An organization, should utilize this pattern if response times are crucial for customers (for example, in a reservation service). We should not use this pattern as default choice rather it is a strong candidate for specific scenarios with in your application.

Decompose by service per team

A service-based decomposition pattern, as opposed to previous decomposition patterns, breaks a monolith into microservices that are managed by individual

teams. Each team is responsible for maintaining a business capability's code base. It is the responsibility of each team to develop, deploy, test, and negotiate APIs with other teams. Each microservice should be owned by only a single team. Multiple sub-teams within a large team may own different microservices within the same team, if there is a larger team.

For example, as illustrated in *Figure 4.6: Decompose by Service per Team* shows how a monolith can be broken into microservices that each team can manage, maintain, and deliver independently.

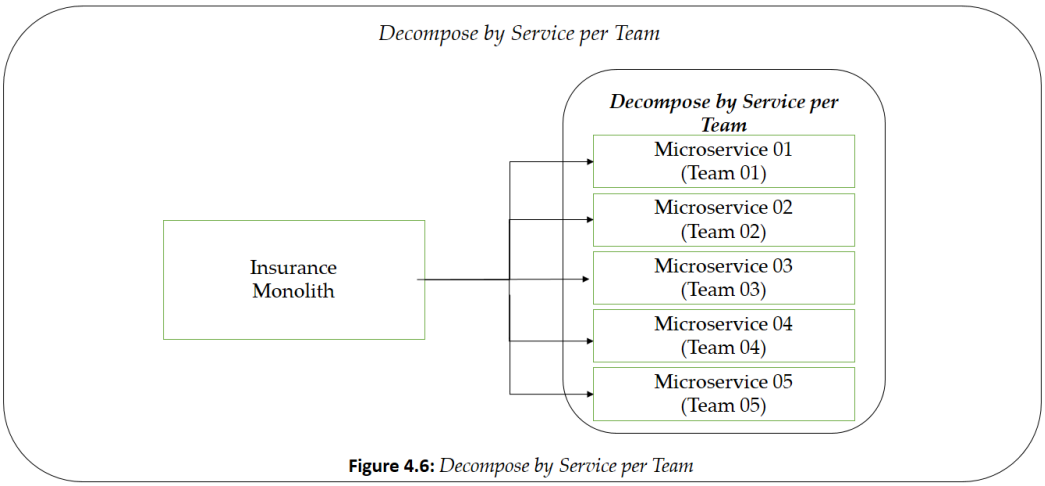


Figure 4.6: *Decompose by Service per Team*

Advantages

- Code bases and microservices are not shared among teams, and they operate independently.
- Each team may use a different technology, framework, or programming language that is most appropriate for the particular microservice.

Disadvantages

- Team alignment with end-user functionality or business capabilities can be challenging.
- Circular dependencies between teams make it difficult to deliver large, coordinated application increments.

When to use this pattern

A team that is responsible for a particular business capability/function owns a code base that is deployed as a service. The Decompose by Service by Team pattern

requires a high level of coordination and communication between teams, as well as a clear definition of service boundaries in order to ensure that services are loosely coupled and easy to maintain. The management of multiple services also requires an overall organization structure that can support the autonomy of teams.

Bulkhead pattern for resiliency

As a software design pattern, bulkheads are primarily used to isolate components from each other in order to improve system resiliency. In situations where multiple interconnected components are present in a system, and a single component's failure may affect the performance or availability of the entire system, this pattern is often used.

Bulkhead patterns create compartments or bulkheads for each component of the system. These compartments are designed to be isolated from one another in order to prevent the failure of one component from impacting the other components.

For example, as illustrated in *Figure 4.7: Bulkhead Patterns*. The solution isolates Service B and Services C from cascading failures by isolating each issue within its own bulkhead, preventing the entire solution from failing.

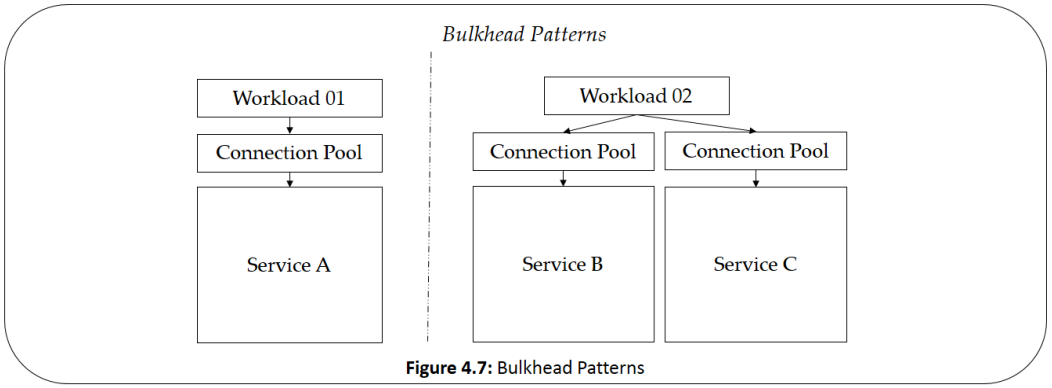


Figure 4.7: Bulkhead Patterns

As a general rule, you will use a combination of techniques to implement the bulkhead pattern, such as resource pooling, circuit breakers, and timeouts. As an example, you might create a resource pool for each component of the system, and use circuit breakers to prevent one component from overloading the others. It is also possible to implement timeouts to ensure that each component has a limited amount of time to accomplish its tasks, as well as to prevent one component from blocking the other.

Advantages

- The ability to preserve some functionality in the event of a service failure. The application's other features and services will continue to work.

- The advantages of isolating different parts of the system are that it is easier to design and implement failure-resistant systems. If one service or container fails, it will only affect a subset of the overall application, rather than the entire system.
- It is possible to create a fault-tolerant system by limiting the amount of resources consumed by a service. If a service consumes too many resources, it can be isolated and restarted, therefore reducing the possibility of the entire system going down.

Disadvantages

- When partitioning Service B and Services C into bulkheads, consider the level of isolation offered by the technology as well as the overhead in terms of cost, performance and manageability.
- As a result of the bulkhead pattern, additional overhead is introduced, such as the need for additional configuration, monitoring, and management.
- It may be expensive to set up and maintain the bulkhead pattern, depending on the specific requirements of the system.
- The added complexity and less efficient use of resources.

When to use this pattern

Combine bulkheads with retry, circuit breaker, and throttle patterns to provide sophisticated fault handling. As a best practice, when it comes to partitioning services into bulkheads, you should consider deploying them in separate virtual machines, containers, or processes. Containers offer the best balance between resource isolation and low overhead.

Sidecar pattern for service mesh

This pattern enables heterogeneous technologies and components to be used in applications by deploying the application components in separate processor containers to provide isolation and encapsulation. Additionally, it enables the application to use heterogeneous technologies and components. The Sidecar pattern is named after a motorcycle sidecar. The idea is to attach a sidecar to a parent application and provide support for the application. Both sidecars and parents share the same life cycle and are created and retired together. For example, the Ambassador sidecar pattern can also be used as a sidecar pattern. This is because it routes the calls directly to the Ambassador, which handles the request logging, routing, and circuit breaker functions for the application.

Advantages

- As a result of separating cross-cutting concerns from the main service, the main service can remain focused on its specific business logic and be more decoupled from the underlying infrastructure.
- A sidecar pattern provides flexibility in choosing the right technology or framework for a cross-cutting concern without modifying the main service.
- The sidecar pattern can make it easier to deploy new versions of the main service without deploying new versions of the sidecar.
- A sidecar is runtime and programming language independent of its primary application, so you do not have to develop one sidecar per language.
- Due to its proximity to the primary application, there is no significant latency in communication between them.

For example, as illustrated in *Figure 4.8: Sidecar Patterns:*

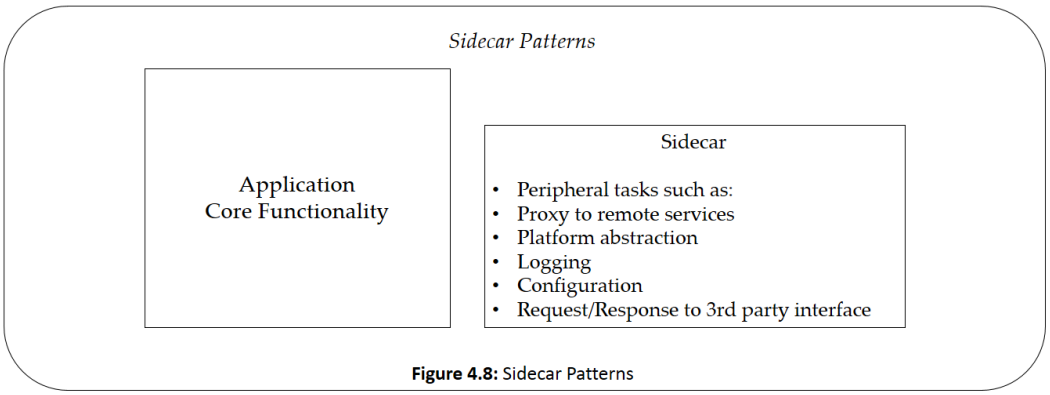


Figure 4.8: Sidecar Patterns

Disadvantages

- The sidecar service may introduce additional latency if it is deployed in a separate container or on a different machine from the main service. Latency can negatively impact the overall performance of the system.
- Consider building functionality into a sidecar as a separate service or as a more traditional daemon before adding it to a sidecar.
- When the cost of providing a sidecar service for each instance is not worth the benefit of isolation.

When to use this pattern

- There are a variety of languages and frameworks used in your primary application. The sidecar service can be used by applications written in a variety of languages and frameworks.
- Having a service that shares the full life cycle of your main application while being able to be updated independently is essential.

Strangler pattern for legacy systems

When you are migrating a legacy system, you should gradually replace certain parts of the system with new applications and services as the system evolves. As soon as all the features of the old system are replaced with the new system, it will eventually stall the old system and allow it to be decommissioned. It is expected that you will mostly be working with brownfield applications, which are large, monolithic applications (legacy codebases). As a result of the Strangler pattern, two separate applications are created in the same URI range, which helps provide a solution or rescue. Eventually, you will be able to turn off the monolithic application because the newly refactored application chokes or replaces the original application until the new application chokes or replaces it.

There are three steps involved in the strangler application process: transform, coexist, and eliminate.

- **Transform**: Create a parallel new site with modern approaches, based on the existing one.
- **Coexist**: Redirect the current site to the new one for a period of time so the new functionality can be implemented incrementally while leaving the existing site alone for a period of time.
- **Eliminate**: Get rid of the old features of the existing site and replace them with the new ones at new site.

Advantages

- This pattern enables an incremental migration from a monolithic application to a microservices-based architecture, reducing the risk of a big-bang migration and allowing for a more gradual transition.
- A phased roll-out of the new system is possible using the Strangler pattern, reducing the risk of introducing new bugs or interrupting existing functionality.

- Provides a way for system transformations to be done with a minimum amount of risk.
- During the refactoring process, old services will continue to work while the updated versions are being implemented.

For example, as illustrated in *Figure 4.9: Strangler Patterns:*

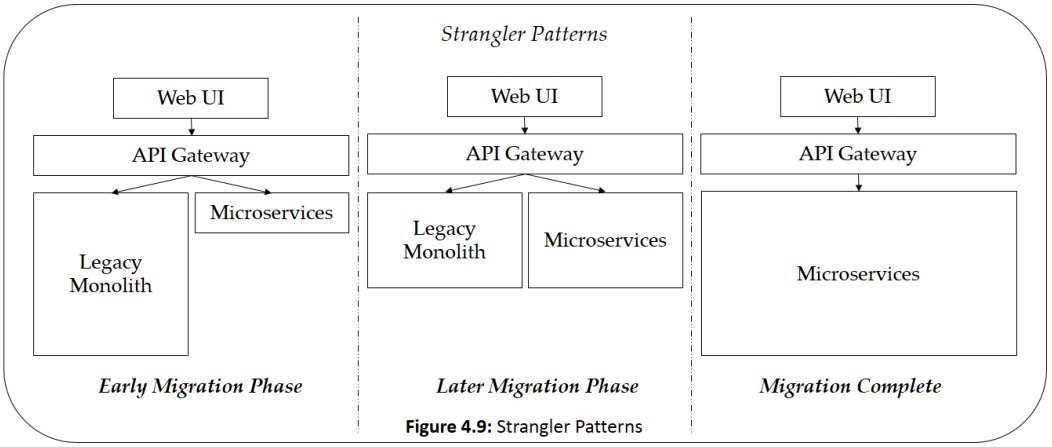


Figure 4.9: Strangler Patterns

Disadvantages

- As a result of implementing the Strangler pattern, the system can be more complex to manage and test, particularly in situations where there are many services or where the services are interdependent.
- A lot of attention needs to be paid to routing and network management on a regular basis.
- This requires you to ensure that you have a plan for reverting back to the old way of doing things when things go wrong, so that you can quickly and safely go back to the old way of doing things.

When to use this pattern

It is important to consider how you can handle both new and legacy systems that are potentially using the same services and data stores. Make sure both can access these resources simultaneously.

Integration pattern

Having learned how to decompose an application and how to decompose it, we will then move on to understanding integration patterns, which are the next important

task. This is essentially the idea of adding more services or structuring an application so that the services are able to communicate with each other and with the client in a very efficient manner.

API gateway pattern for API management

As its name suggests, API gateways are single entry points that aggregate calls to each microservice. While this may seem very similar to the aggregator pattern, it has some important differences. In addition to this, the new service does not store any data and instead is responsible for API assembly, request routing, and new authentication features. It is shown in *Figure 4.6* that the API gateway can serve multiple clients through a variety of communication channels. API Gateway is an API management system which exposes APIs based on the needs of the client. The gateway can then process requests through API composition, invoke multiple services and aggregate the results of the multiple services.

For example, as illustrated in *Figure 4.10: API gateway Patterns:*

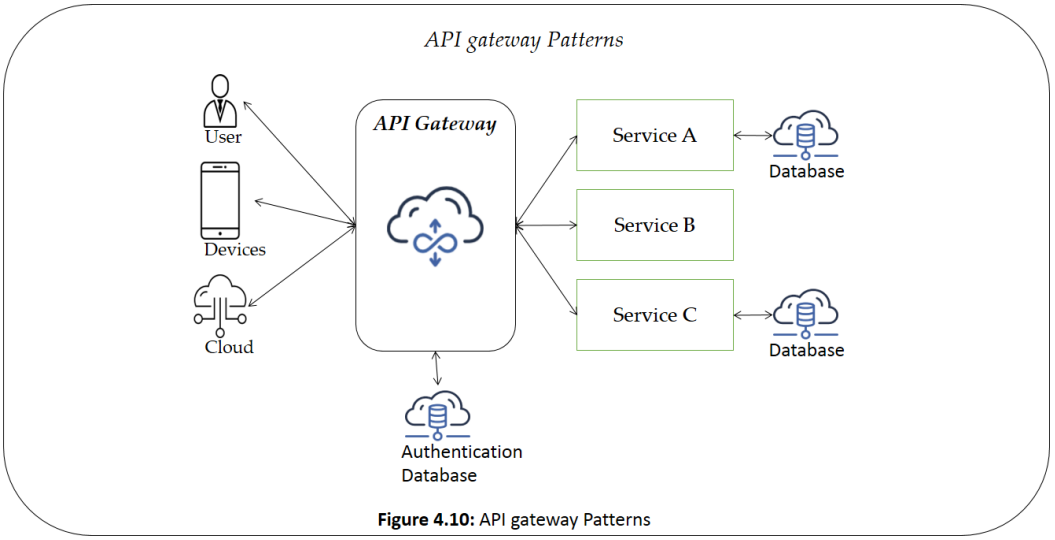


Figure 4.10: API gateway Patterns

Advantages

- Acting as a facade. The facade architectural pattern is not a new one - in essence, it implements a single interface in front of a complex system to improve its usability and provide loose coupling. It is possible to maintain and change the location of the backend components through the facade without affecting the client.

Disadvantages

- When a single point entry via API Gateway becomes inefficient, it can eventually become a monolithic process. This is where the **backend for frontend** (**BFF**) pattern is used: Applications can have multiple API gateways based on business tasks or client apps (such as separate gateways for web and mobile applications).
- An API gateway should help your API grow, which will increase traffic to your API, and your gateway should be prepared to handle these spikes and comfortably scale. In this article, Uber documents how they handled scaling challenges.

When to use this pattern

The API gateway provides clients with a cleaner interface through which to interact, which is another reason for its adoption. A pattern like this should be used for load balancing when high availability is required, but it can also be combined with rate limiting and throttling.

API aggregator pattern for composite services

A microservice architecture can consist of tens to hundreds of services. However, the problem with getting information about a specific product is that many additional calls are required to other services in order to get the necessary information. Aggregator patterns are designed to collect data from a variety of microservices and return it as an aggregate for processing. *Figure 4.5* shows how the new aggregator service will be able to maintain a common collection of information from various services in one database. This will enable it to store it. In this case, the various services can push events onto a messaging queue/bus, which are then collected by the aggregator, which records the events in the database.

- To reduce latency as much as possible, the gateway should be located as close as possible to the backend services.
- Gateways can cause a bottleneck in the process. Make sure that the gateway has sufficient performance to handle the load as well as the ability to scale to accommodate anticipated growth.
- Make sure that the design is robust by using techniques such as bulkheads, breaks, retries, and timeouts.
- A better approach would be to place an aggregation service behind the gateway rather than building an aggregation into the gateway itself.

- It is likely that demand aggregation will have different resource requirements than other services in the gateway, which can have an impact on the gateway's routing and offloading capabilities.

For example, as illustrated in *Figure 4.11: Gateway Aggregator Patterns:*

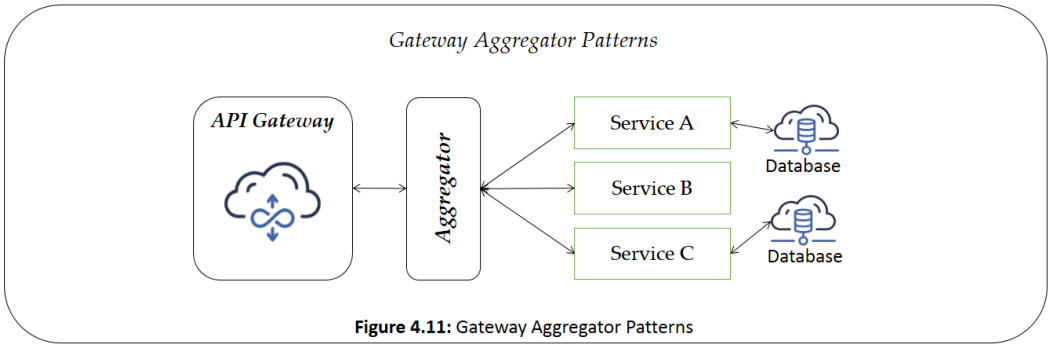


Figure 4.11: Gateway Aggregator Patterns

Advantages

- API Aggregator simplifies the overall system by hiding the complexity of interacting with multiple microservices from the client.
- The API Aggregator pattern centralizes the management of APIs, making it easier to address cross-cutting concerns such as security, monitoring, and logging.
- Using the API Aggregator pattern, caching can be implemented at the gateway level, reducing the number of requests to the underlying microservices and improving performance.
- A key component of the API Aggregator pattern is the ability to route requests to different microservices depending on the specific requirements, which allows for more flexible and decoupled service integrations.

Disadvantages

- You should ensure that the gateway service is properly designed to meet the requirements of your application with regard to availability as it may introduce a single point of failure.

When to use this pattern

A client can use this pattern if it has to communicate with multiple backend services in order to perform an operation. The client may be using a network with significant latency, such as a cellular network.

Gateway offloading pattern for performance

The offloading of shared or specialized service functions to the gateway proxy can simplify the development of applications by moving common service functionality, such as the use of SSL certificates, from other parts of the application to the gateway proxy. In order to handle complex security issues properly (token validation, encryption, SSL certificate management) and other complex tasks, team members with highly specialized skills may be necessary. The certificate required by an application needs to be configured and deployed to all instances of the application, for example, every time the application is deployed. In order to ensure that the certificate does not expire, it has to be managed with each new deployment. Every time an application is deployed, any generic certificate that expires must be updated, tested, and verified. It is a good idea to delegate some functionality to a gateway, especially cross-cutting functions like certificate management, authentication, SSL termination, auditing, protocol translation, and throttling.

For example, as illustrated in *Figure 4.12: Gateway Offloading pattern:*

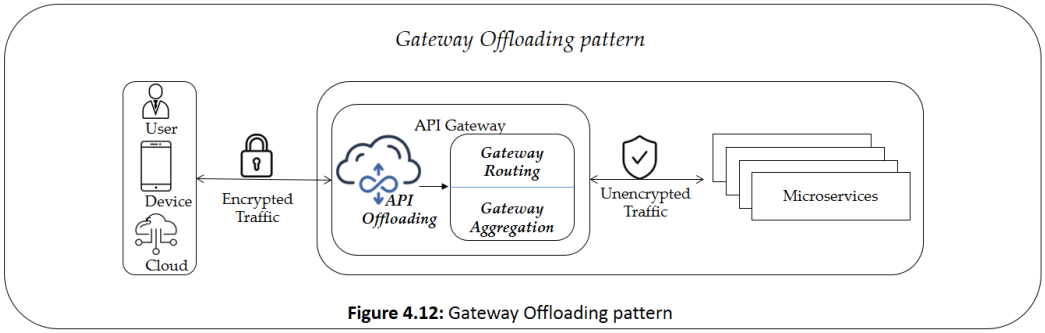


Figure 4.12: Gateway Offloading pattern

Advantages

- You can simplify the service development process by eliminating the need to distribute and maintain supporting resources such as web server certificates, secure web configurations, and website security codes.
- Simplifying the configuration process leads to easier management, scalability, and upgrading of the service.
- It is best to have dedicated teams implement functions requiring specific knowledge, such as Security. In this way, your core team will be able to focus on the application's functionality, while the relevant experts will take care of these specialized yet cross-functional concerns.
- You can configure the gateway to ensure some consistency in monitoring and logging requests and responses, even if the service is not properly

instrumented. This can be accomplished by configuring the gateway in such a way that a minimum level of monitoring and logging is maintained.

Disadvantages

- There should be a high level of availability and resilience for your gateway. Having multiple instances of the gateway will prevent the failure of a single point of failure.
- Ensure that the gateway is designed for the capacity and scale needs of your application and endpoints. Be sure that the gateway does not become a bottleneck for the application and is sufficiently scalable so that it does not become a bottleneck.

When to use this pattern

- In general, this is a feature that is common among application deployments with different resource requirements, including memory resources, storage capacity, and network connections, but may not be the same across all deployments.
- As part of this project, you would like to shift responsibility for network security, throttling, and other network boundary concerns to a more specialized team. Business logic should never be delegated to gateways.

Gateway routing pattern for traffic shaping

It helps with a single endpoint, route requests to multiple services or instances. Using this pattern, you can expose multiple services to a single endpoint and route requests to the appropriate service, expose multiple instances of the same service for load balancing or availability purposes, and expose different versions of the same service and route traffic across them.

For example, as illustrated in *Figure 4.13: Gateway Routing pattern:*

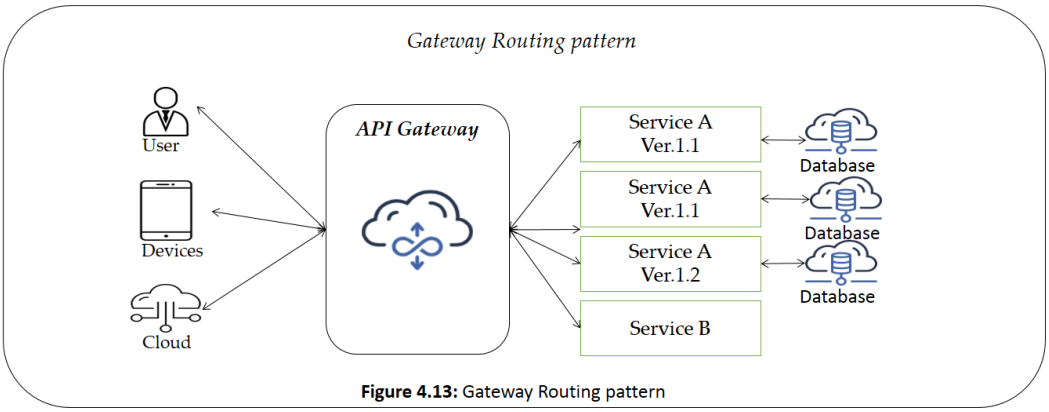


Figure 4.13: Gateway Routing pattern

Advantages

- The gateway routing level is the highest level of routing and is based on IPs, ports, headers, and URLs.
- There are several ways to limit public network access to the backend services, such as limiting them to only be accessed via the gateway or by using a private virtual network. The gateway serves as the public endpoint for the services.

Disadvantages

- Consider resiliency and fault tolerance capabilities when implementing the gateway service. It can introduce a single point of failure.
- There is a possibility that the gateway service will become a bottleneck for your business. Ensure the gateway service is scalable and capable of handling the load expected as your business grows.

When to use this pattern

When a client needs to access multiple services behind a gateway, you want to simplify the client application by using a single endpoint to access all the services that the client needs to consume. As part of your deployment strategy, you would like to implement a method in which clients can access multiple versions of the service at the same time.

Asynchronous messaging pattern for loose coupling

Asynchronous messaging is one of the most scalable patterns due to the asynchronous nature of microservices. This pattern uses asynchronous messaging between services for communication. Microservices communicate with one another via messaging channels.

Asynchronous communication can take a number of different forms, including:

- Service Request/Response - a service sends a request message to a recipient and expects to receive a response message within a reasonable period of time.
- The notification system is used when someone sends a message to a recipient, but the sender is not expecting a reply from the recipient. Neither is sent.
- A request/asynchronous response request is sent by a service to a receiver and is eventually expected to receive a response message from the receiver.

- Publish/subscribe service sends a message to zero or more recipients, one or more recipients, of whom some may reply to the message.

Technology examples that use asynchronous messaging are Apache Kafka, RabbitMQ.

The message passing pattern is another method that microservices use to communicate with each other. The services are connected by exchanging messages across a queue. This style of communication has a number of advantages, including the fact that it does not require service discovery and the services are not tightly coupled. As synchronous systems are tightly coupled, any problem that occurs in synchronous downstream dependencies can have an immediate impact on upstream callers. Depending on the specific requirements, such as protocols, Amazon Web Services offers a variety of services for implementing this pattern, including retries from upstream callers that can quickly fan out and amplify problems. In one possible implementation, **Amazon Simple Queue Service** (**Amazon SQS**) queueing and **Amazon Simple Notification Service** (**Amazon SNS**) are combined to implement the solution.

For example, as illustrated in *Figure 4.14: Asynchronous Messaging pattern:*

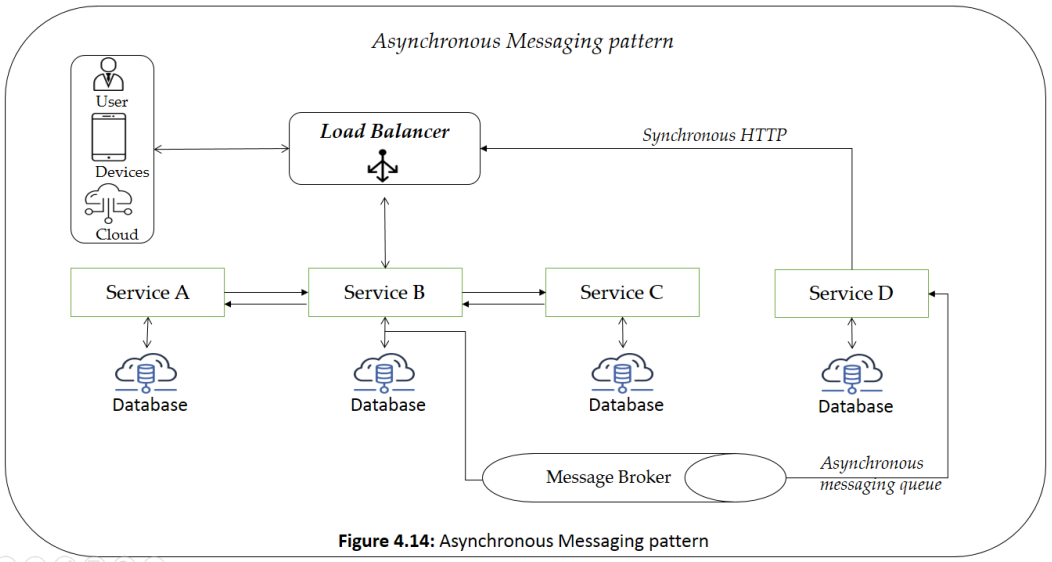


Figure 4.14: Asynchronous Messaging pattern

Advantages

- There is also the issue of reliability. In the event that one of our report generation services is down (and you know there will be one), or there is a network glitch (and you know that is going to happen), you will lose synchronous messages after a certain timeout, and for asynchronous messages, you will have retries and a dead letter queue.

- The other advantage of queues that they allow us to prioritize our requests. If the same service processes both product views and purchase requests, then we can make sure that the latter gets the first priority. If the service is under heavy load, we can serve the product views slowly, perhaps even dropping some. This will ensure that the purchase requests are not adversely affected by the load.

Disadvantages

- It is true that asynchronous messaging does add complexity to the process, since instead of sending a message and waiting for a reply, you must send a message, listen to a reply queue, and then route the message to the sending client based on a message ID.
- In addition, the use of asynchronous communication and the addition of a queue between components means that we must necessarily add some latency to the system. We are adding two hops to each communication instead of communicating directly, one from the sending service to the queue, and another from the queue to the receiving service, instead of communicating directly.

When to use this pattern

When it comes to a microservice architecture, asynchronous communication is the way to go. In asynchronous messaging, messages are exchanged between software systems in an asynchronous (non-blocking) manner. When the sender and the receiver of a message must remain independent of each other, and when the sender does not require a response from the receiver prior to continuing its work, this pattern is commonly used. An example of how asynchronous messaging is used is when system components are located on different machines or locations in a distributed environment. As a result, asynchronous messaging allows these components to communicate with each other without establishing direct connections, which can lead to improved system performance and scalability.

Branch pattern for parallel processing

As a result of the combination of the aggregator and chained design patterns, the branch pattern is the advanced version which aims to combine the positive aspects of both patterns to better serve the business layer of an application. This pattern is designed for simultaneous processing of requests and responses from two or more microservices at the same time. To put it simply, we can see in *Figure 4.9* that the developer has the option to dynamically configure service calls in this pattern. The calls in this pattern can also be made in a contemporary manner. Due to this, service A has the ability to call both service B and service C at the same time.

For example, as illustrated in *Figure 4.15: Branch pattern:*

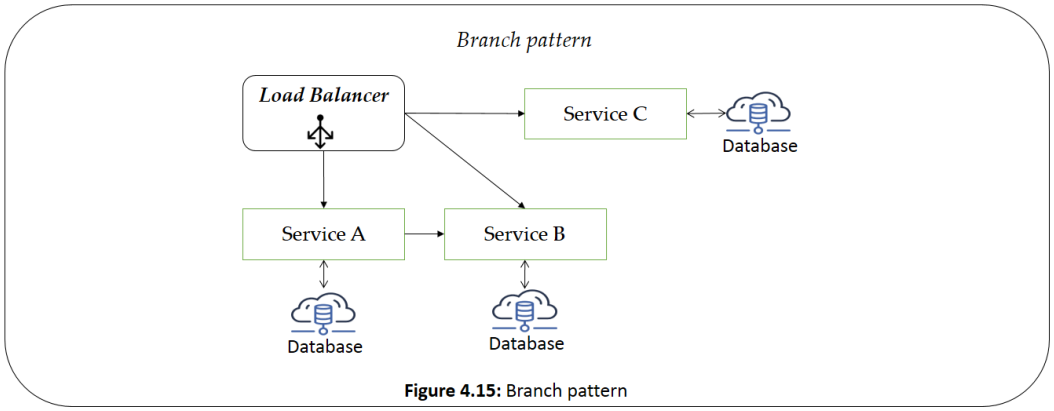


Figure 4.15: Branch pattern

Advantages

- Using a branch microservice pattern, it is possible for developers to configure service calls dynamically. All service calls will take place in a simultaneous manner, which means that service A will be able to call service B and service C at the same time.
- It is possible to support more than one version of a service simultaneously using the Branch pattern, which is useful in situations in which a new version of the service is being developed and tested, but the previous version remains in production.
- An incremental rollout is possible using the Branch pattern, which reduces the risk of introducing new bugs or disrupting existing functionality.

Disadvantages

- Add on complexity. The branch pattern can also cause code to be less efficient, as the microservice must evaluate each branch and determine which path to take.
- In some cases, implementing the Branch pattern may make the system more challenging to manage and test, particularly in situations where there are multiple versions of the service or where the services are interdependent.

When to use this pattern

A branch microservice design pattern allows the processing of responses and requests from multiple separate microservices at the same time. The chained design pattern

differs from the chained design pattern in that instead of being passed sequentially to two or more chains of mutually exclusive microservices, the request is sent to two or more chains of mutually exclusive microservices at the same time. Using this design pattern, users can generate responses using a single chain as well as a number of chains. It expands on the Aggregator design pattern.

Chained microservices pattern for sequencing

It is important to keep in mind that in a chained microservice design pattern, a single consolidated response to the request is produced. If a request is received by a client of service A, the client then communicates with service B, which in turn can communicate with service C. There is a high probability of synchronous HTTP request/response messaging being used by all services.

The key part to note is that the client will be blocked until the complete request/response chain, that is service <-> service B and service B <-> service C, is completed. Service B's request to Service C can be completely different than Service A's request to Service B. Likewise, Service B's response to Service A Form Service C to Service B can be completely different. And that is the whole point anyway, where different services increase their business value.

The other important aspect to keep in mind is to ensure that the chain is not made too long. It is important because the synchronous nature of the chain can appear on the client's side as a long wait, especially with regards to web pages that are waiting for the response to be displayed.

Advantages

- There is no doubt that the chained microservice pattern has its own advantages. The main advantage is the ease of implementation; due to the chain calls being synchronous, it is easier to understand the network calls. We do not prefer to use asynchronous communication methods. For example, as illustrated in *Figure 4.16: Chained Microservices pattern:*

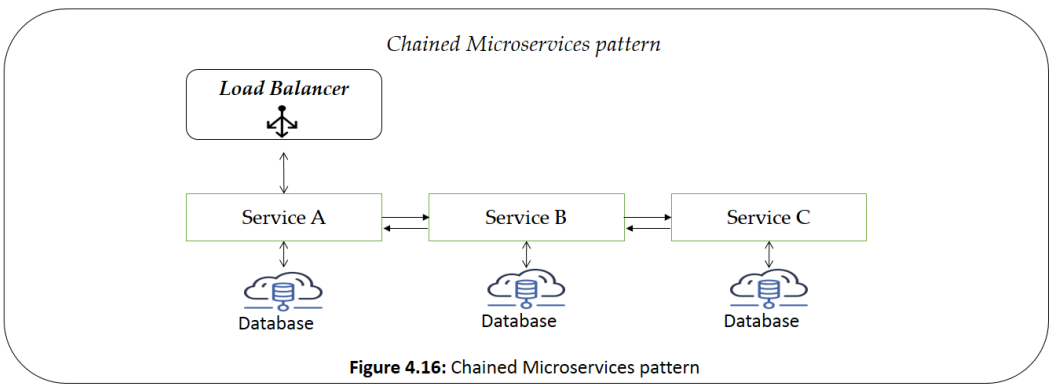


Figure 4.16: Chained Microservices pattern

Disadvantages

- Due to the fact that the client does not receive any output until the request has been processed by each service and the corresponding responses have been produced, it is always recommended to avoid making long chains since the client will wait until the chain is complete.
- One of the obvious problems with this pattern is its inability to handle asynchronous communication, which increases complexity and compromises its scalability.

When to use this pattern

If the current scope of the application is too large to be able to add in more microservices at the moment, then the chained microservice pattern can prove useful.

Database management pattern

Getting data into and out of a database is a critical component of any application. Therefore, this section will cover a wide variety of techniques for managing data using the following patterns: Command Query Separation of Responsibility, Event Acquisition, Database per Service, Shared Database per Service, and Saga Pattern.

Command Query Responsibility Segregator (CQRS) pattern for separation of concerns

The CQRS pattern is about separating the create, update, and delete operations from the retrieval operations. Essentially, the query side model keeps updated information by subscribing to the events that are published on the command side, so that it is always up to date. It is quite challenging to implement queries from multiple services using the database-per-service model, so the CQRS pattern has been designed to help implement queries from multiple services.

A common database for each service or a database for each facilitator is present in every microservice design. However, since there is only one database per service in the database for each service architecture, we are not able to implement a query in the database. This is why you can apply a pattern known as the CQRS pattern in this situation. Based on this design, the program will be split into two sections – command and query. It is the query section of the code that will take care of the materialized views, while the command section of the code will handle all requests that are completely related to the CREATE, UPDATE, and then the DELETE operations. To update the materialized views, a series of events utilizing the pattern mentioned above is used.

For example, as illustrated in *Figure 4.17: CQRS (Command Query Responsibility Segregator) pattern:*

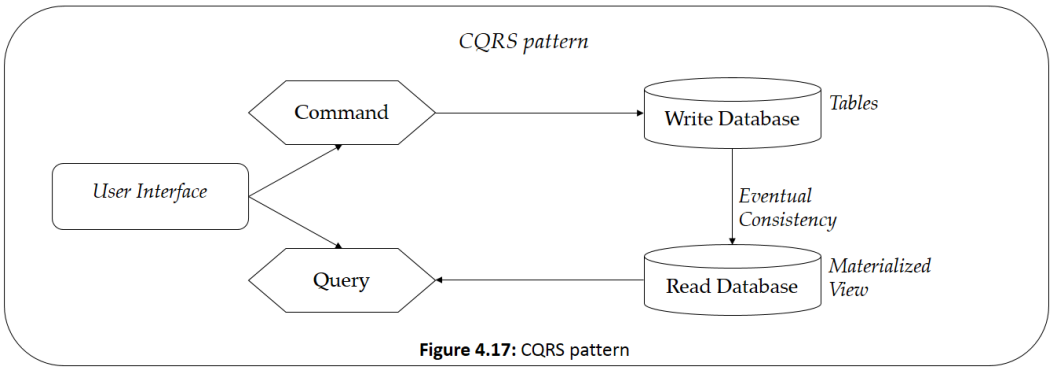


Figure 4.17: CQRS pattern

Advantages

- Using the CQRS pattern has the advantage of improving a system's scalability and performance. Using the CQRS pattern, the command and query components of a system can be optimized and scaled independently of one another by separating the responsibilities of the system into distinct components. As a result, the system will be able to handle large volumes of requests without degrading performance or availability.
- Consequently, the flexibility that is offered by this design pattern helps systems to stay more flexible over time, resulting in increased application performance, security, and scalability as they continue to evolve.
- The CQRS pattern can also facilitate the understanding and maintenance of the code. CQRS patterns can make a system more modular and easier to understand by dividing its responsibilities into separate components. As a result, developers can make changes to the system without affecting other parts of the code, making it easier for them to work with it.

Disadvantages

- Its partiality must be evaluated before implementation. CQRS also has the potential disadvantage of requiring additional resources and effort to implement than other patterns. In order to design and implement the CQRS pattern successfully, it may require more effort and resources due to the requirement of creating separate components for handling commands and queries.

When to use this pattern

The CQRS pattern is an excellent choice for cases where the number of data reads is much higher than the number of data writes. In cases like this, microservice design patterns like CQRS will help you to scale the read model independently rather than scaling the write model. It is recommended that the CQRS pattern be used along with event sourcing since the two patterns complement each other. CQRS can be used when systems have different loads on the read and write aspects of a database in order to separate those concerns.

Database per service pattern for decoupling

In a microservices architecture, loose coupling is a major component of the architecture, as each individual microservice is able to independently store and retrieve information from its own data store. When you use a database-per-service pattern, you can select the most appropriate data store (such as a relational or non-relational database) according to your business needs and application. There is no common data layer between microservices, changes to a single microservice database do not affect other microservices, individual data stores cannot be accessed directly by other microservices, and persistent data is only accessed through APIs. This means that microservices do not share a data layer. Additionally, decoupling your data stores improves the resiliency of your entire application and makes sure that one database cannot become a single point of failure for your application.

In the following illustration, you will see that the "Sales," "Customer," and "Compliance" microservices use different AWS databases. Using the Amazon API Gateway API, these microservices are accessed as Lambda functions of AWS and ensure that the data is kept private and not shared among the microservices. AWS **Identity and Access Management** (**IAM**) policies ensure that data is protected. It should be noted that each microservice uses a database type that meets the individual requirements, for example, "Sales" uses Amazon Aurora, "Customer" uses Amazon DynamoDB, and "Compliance" uses **Amazon Relational Database Service** (**Amazon RDS**) for SQL Server.

For example, as illustrated in *Figure 4.18: Database-per-service pattern:*

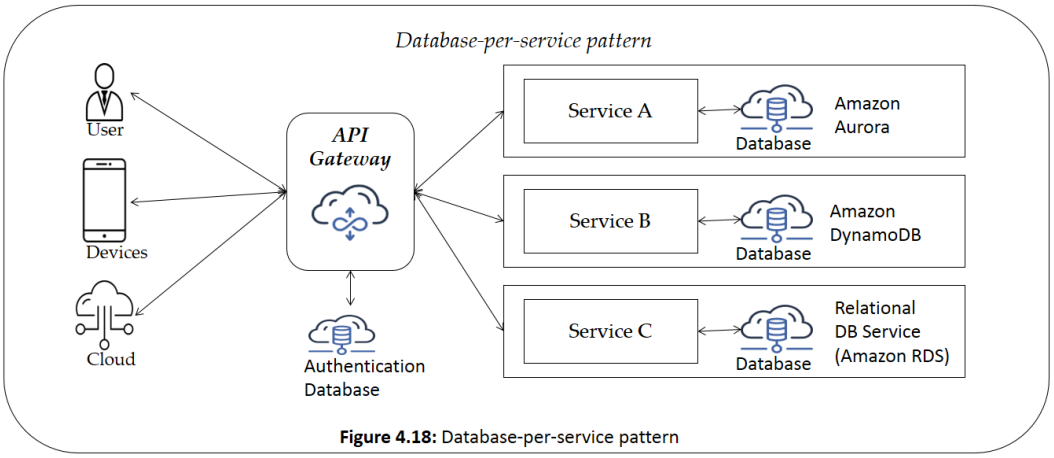


Figure 4.18: Database-per-service pattern

Advantages

- There needs to be loose coupling between your microservices. Microservices have different compliance or security requirements for their databases.
- A more precise control of scaling is required between your microservices.

Disadvantages

- In order to implement complex transactions and queries that span multiple microservices and data stores, you might have to deal with a lot of managing and maintaining multiple relational and non-relational databases.
- Your data stores need to meet two of the CAP theorem requirements: consistency, availability, or partition tolerance.

When to use this pattern

In cases where teams require complete ownership of their microservices for scaling and operational purposes, the database per service pattern should be used.

Shared database per service pattern for consistency

This pattern is characterized by the fact that a database is shared by many microservices at the same time. This pattern often leads to microservices becoming a distributed monolith, which can be a nightmare for the developer. Even though this pattern implies sharing a database, it does not imply that single tables should ever be shared among multiple microservices (which should never be done). Due to this tight coupling between the services, the system maintainability and performance are degraded as well. This leads to a very high degree of coupling between the services.

Advantages

When it comes to quick development time, a shared database pattern is very useful. Despite the fact that it is not the best practice, it does significantly reduce the development effort by a huge margin. For example, as illustrated in *Figure 4.19: Shared-database-per-service pattern:*

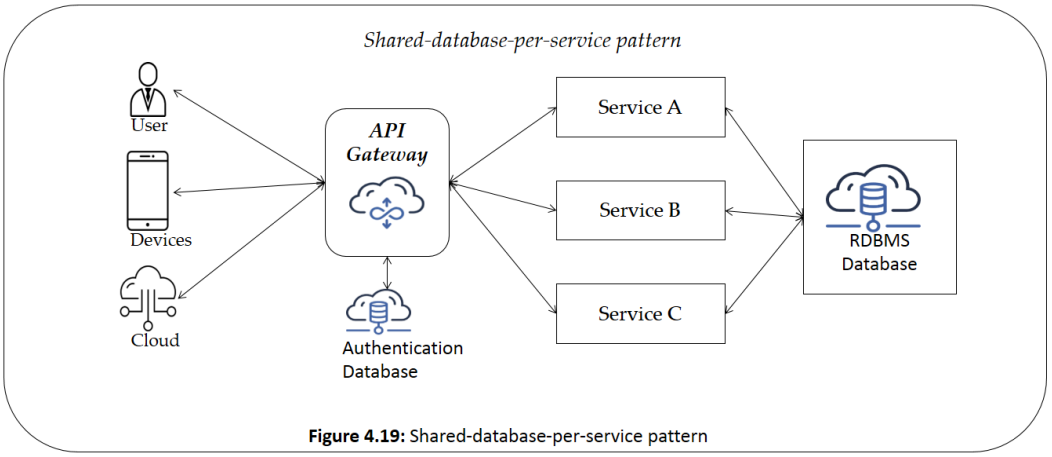


Figure 4.19: Shared-database-per-service pattern

Disadvantages

- Given that the database is shared between the two services, there is a risk that one of the services might corrupt or delete some data due to the fact that the database is shared.
- The performance of other services that are sharing the same database will be affected if one service fires expensive queries on the database.

When to use this pattern

The use of this pattern can serve as an encouragement when an organization prefers not to refactor an existing code base by many changes due to business requirements or other limitations.

Event sourcing pattern for auditing and reconciliation

The idea behind event sourcing is not storing state, but storing events in a system that can be replayed whenever the need arises. Since each event is irremovable and immutable, there is no need to update or delete them to guarantee good performance

at writing. These factors result in better performance. As opposed to updating data directly in data stores, events are stored in a series of events, instead of being updated directly in those data stores as they happen. In order to calculate the state of their own data stores, microservices replay events from an event store. Patterns like this provide visibility into the current state of an application as well as additional context regarding how it got to where it is at the moment. Although the command and query data stores may have different schemas, the Event Source pattern will still be able to reproduce the data for any given event even if the command data store and query data store have different schemas. In order to implement this pattern, either Amazon Kinesis Data Streams or Amazon EventBridge can be used.

An event sourcing system can be used to atomically update state and publish events. It is the traditional way to persist an entity that it is saved with the current state. Event sourcing takes a new approach to persistence that is event-centric. Business objects are persisted by storing a sequence of state change events. Each time an object's state changes, a new event is added to the sequence of events. Due to the fact that this is an operation, it is atomic in nature because it is an operation.

It is possible to reconstruct an entity's current state by replaying its events. To understand how event sourcing works, you can look at the order entity. Traditionally, each order is associated with a row in the **`ORDER`** table, along with a row in another table like the **`ORDER_LINE_ITEM`** table. An order is saved in the order service when it is saved by keeping its status changes: created, approved, shipped, and cancelled.

A number of events are recorded in an event log, which serves as both an event database and also as a message broker for the order. The event log contains enough information to reconstruct the order status. Services can subscribe to events through an API provided by the event store. Event stores are the backbone of an event-driven microservices architecture where all events stored in the event store are delivered from the event store to all interested subscribers.

Advantages

- This is the only audit logging solution that provides 100% accuracy for auditing, which is often an afterthought, leading to the inherent risk of incomplete auditing. With event sourcing, each state change corresponds to one or more events, ensuring that auditing is completely accurate.
- In this pattern, you will be able to identify and reconstruct the state of the application at any time in the past. This will create a permanent audit trail and make troubleshooting easier. However, the data will eventually become consistent, which may not be appropriate for some use cases.

For example, as illustrated in *Figure 4.20: Event Sourcing pattern:*

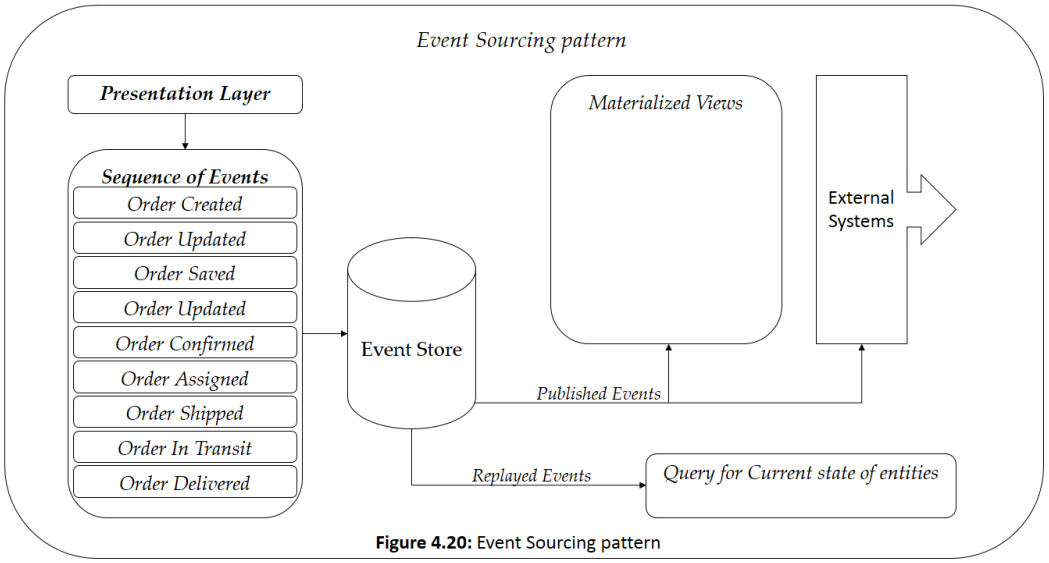


Figure 4.20: Event Sourcing pattern

Disadvantages

- As different types of events contain different payloads, there needs to be a single source of truth for defining and determining the correct format for each type of message.

When to use this pattern

Due to the fact that more applications are requiring real-time data in an asynchronous manner but in an organized manner, event sourcing is becoming increasingly popular.

Saga pattern for long-running transactions

There is a design pattern called Saga, which is one way to manage data consistency across microservices during distributed transactional situations. A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, then the saga performs compensating transactions that counteract the previous transactions.

It is essential that transactions are **atomic, consistent, isolated, and durable** (**ACID**). While transactions within a single service comply with ACID, cross-service data consistency requires a cross-service transaction management strategy in order to ensure data consistency.

- As far as Atomicity is concerned, it is an indivisible and irreducible set of operations that must all occur or none of them will.
- Consistency refers to the fact that the data only moves from one valid state to another valid state when a transaction is made.
- The isolation of concurrent transactions ensures that the data state of a concurrent transaction will be the same as the state of a sequentially executed transaction.
- The durable nature of the system means that committed transactions will remain committed even in the event of a system failure or power outage.

The two-phase commit protocol (2PC) is a distributed transaction protocol that requires all participants to commit or roll back a transaction before it can proceed. Although some of the participating implementations, including, are able to support this model, other systems, such as NoSQL databases and messaging, do not.

In the Saga pattern, a sequence of local transactions is used for transaction management. Each saga participant performs an atomic amount of work and updates the database and then publishes a message or event that triggers the next local transaction in the saga in response to the previous local transaction. A series of compensatory transactions are run by Saga when a local transaction fails, which restores the changes that were made by the previous local transaction.

There are two approaches to developing sagas:

- **Choreography based**: Participant transactions are orchestrated by an orchestrator.
- **Orchestration based**: Using a messaging queue, local transactions trigger local transactions in other services based on domain events.

Choreography saga pattern

With the application of publish-subscribe principles, the choreography provides the possibility of coordinating sagas. As part of the choreography, each microservice executes its own local transaction and publishes events to the message broker system, thereby triggering local transactions in other microservices. It can be very difficult and confusing to manage transactions between Saga microservices as the number of workflow steps increases. The choreography also decouples the direct dependency on microservices for the management of transactions from the choreography.

For example, as illustrated in *Figure 4.21: Choreography Saga Pattern:*

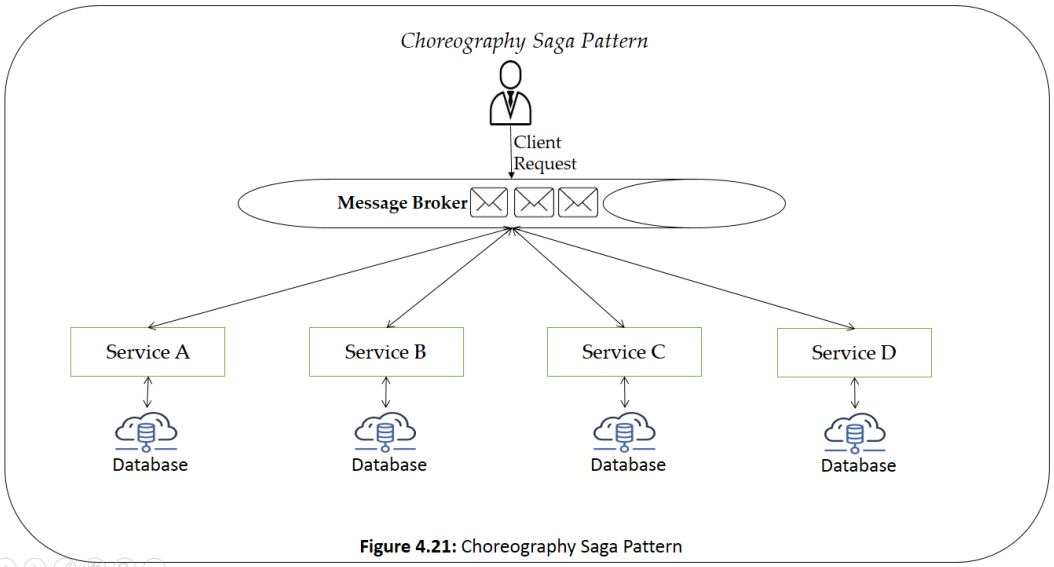


Figure 4.21: Choreography Saga Pattern

Advantages

- It is perfect for simple workflows that require a limited number of participants and do not require a coordination logic. It is not necessary to implement and maintain additional services. It does not introduce a single point of failure, since the responsibilities are distributed among the participants.

Disadvantages

- There is a risk of cyclic dependency between saga participants because they must consume each other's commands, which makes workflow confusing when adding new steps, as it is difficult to track which saga participants listen to which commands. Integration testing is difficult since all services must be running in order to simulate a transaction.

When to use this pattern

Choreography saga patterns are typically used for events-driven interactions between services, in which the interactions become a series of independent steps rather than a single, monolithic operation.

Orchestration saga pattern

An alternative to sagas is orchestration. It can be used to coordinate sagas by using a centralized controller microservice, allowing for the sequential execution of local

microservice transactions. This centralized controller microservice orchestrates the saga workflow and calls to sequentially execute local microservice transactions. As a result of orchestrating and managing saga transactions in a centralized manner, orchestrator microservices are able to roll back steps with compensating transactions in the event that any of them fail.

For example, as illustrated in *Figure 4.22: Orchestration Saga Pattern:*

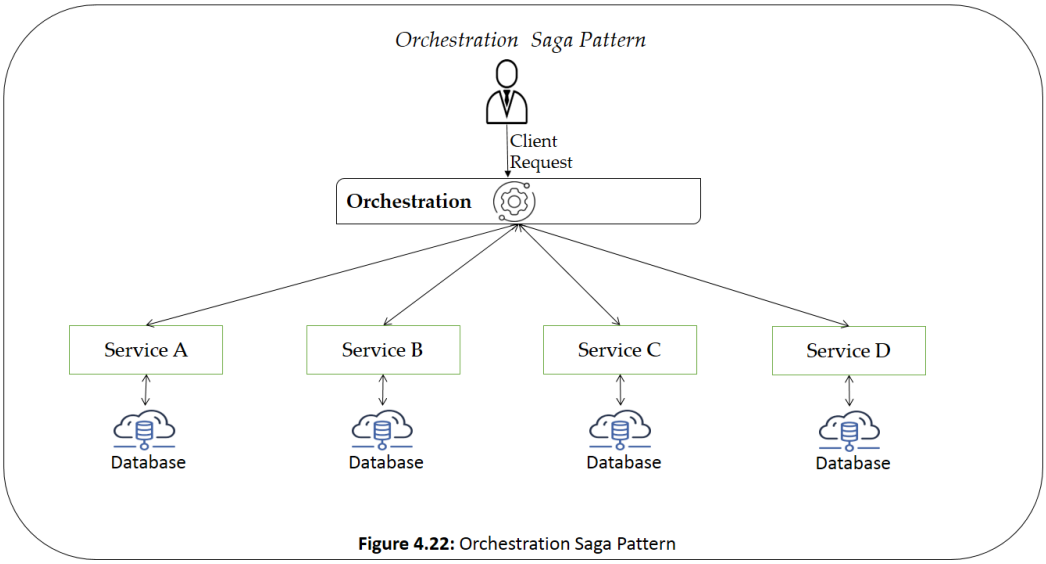


Figure 4.22: Orchestration Saga Pattern

Advantages

- When there is control over every participant in the process, as well as control over the flow of activities, this system will be ideal for complex workflows involving many participants or those that add new participants over time.
- As the orchestrator unilaterally depends on the participants of the saga, it does not introduce cyclical dependencies. There is no need for saga participants to be aware of other participants' commands. Separation of concerns simplifies the business logic.

Disadvantages

- Due to the additional complexity of the design, a coordination logic has to be implemented. There is an additional point of failure, since the orchestrator is responsible for managing the entire process.

When to use this pattern

- Data consistency in a distributed system without tight coupling must be ensured. If one of the operations in a sequence fails, it must be rolled back or compensated.
- The orchestration saga pattern is typically used when the interactions between services are more tightly coupled, and it makes it more natural to think of the interactions as a single, monolithic operation. Contrary to the choreography saga pattern which emphasizes independent steps and is more event-driven.

Observability pattern

In order to achieve observability, you must collect telemetry from endpoints and services in your multi-cloud computing environment, derived from instrumentation that originates from the endpoints and services. There are countless components of today's modern environment, including hardware, software, and cloud infrastructure, containers, open source tools, and microservices, that generate records of every activity that occurs. Observability is a way of understanding what is happening across all of these environments and between technologies so that you can identify and resolve problems so that your systems remain reliable and efficient, and your customers are happy. As modern cloud environments are dynamic and constantly changing in scope and complexity, most problems cannot be recognized or monitored due to the fact that they are dynamic. With observability, you are able to eliminate this common problem of unknown unknowns, allowing you to understand new types of problems as they arise continuously and automatically.

Distributed tracing pattern for root-cause analysis

It is possible for your applications to benefit from a number of benefits during the development and operation phases if you move from monolithic to microservices-centric designs. It is important to remember, however, that when your components are distributed across computers and physical locations, and subject to dynamic horizontal scaling across transient compute units, traditional tools for analyzing and gathering information become ineffective. Using distributed tracing, we are able to associate requirements with their execution in a distributed environment, giving us the means to pinpoint failures and performance issues in order to resolve them quickly. Thus, to reiterate, the challenge is to understand how an application behaves and find the cause of the problem.

Advantages

- Providing visibility into changes reduces the overhead required for rollbacks and deployments. Distributed tracing supports polyglot development because it is agnostic in nature. One trace can be propagated across multiple clients since it is agnostic.
- With distributed tracking, developers spend less time troubleshooting and debugging problems than without it, which leads to higher productivity.

Disadvantages

- One of the most prominent disadvantages is that aggregating and storing traces can require a significant amount of infrastructure, which can be costly in terms of human capital and monetary resources. For example, as illustrated in *Figure 4.23: Distributed Tracing Pattern:*

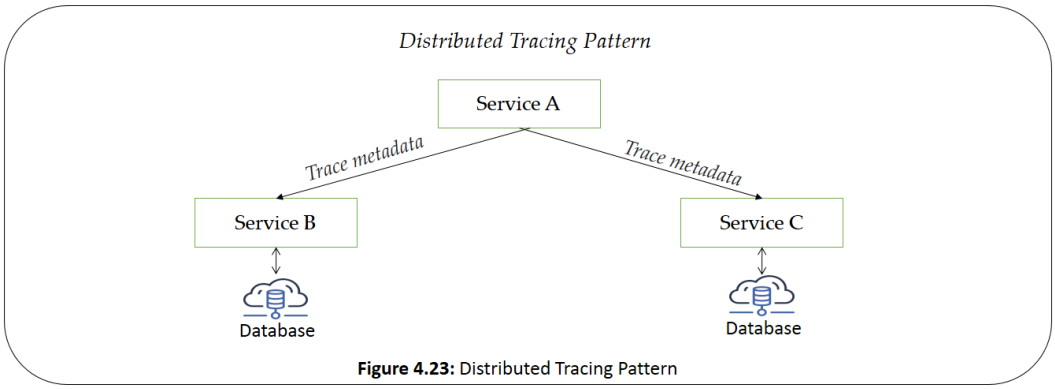


Figure 4.23: Distributed Tracing Pattern

When to use this pattern

It is possible to perform advanced latency analysis on your transactions with advanced latency analysis. With advanced latency analysis, you can obtain a detailed end-to-end analysis of the total time it takes to process a transaction. You can also examine the details to find the bottlenecks in your application.

Health check API pattern for self-healing

Monitoring containers and microservices in near real-time is critical to several aspects of microservices operations. Essentially, health monitoring can enable near real-time information about containers and microservices. Essentially, the main problem is how a live service should recognize that it is unable to respond to requests. The solution is implemented as an API endpoint that returns the health status of the microservice. This response code indicates the application's health status as well as

any components or services that it may utilize. As part of the monitoring process, the monitoring tool or framework will perform a check of latency or response time.

There are a number of typical checks that can be performed by monitoring tools, including:

It is important to validate the response code. For example, a response code of 200 (OK) indicates that the application has responded without error. In order to provide more comprehensive results, the monitoring system can also search for other response codes as well.

The health check API can be used by using any HTTP client, such as a web browser or a command-line tool like curl, to make an HTTP request to its endpoint. In response, the API would provide information regarding the current status of the service or application, which you could then use to monitor its performance and availability.

Check if your SSL certificate is not up-to-date and measure the response time, which is a combination of network latency and the time it took the application to complete the request.

In order to measure DNS latency, you may want to measure the response time of a DNS lookup for the application's URL.

For example, as illustrated in *Figure 4.24: Health Check API Pattern:*

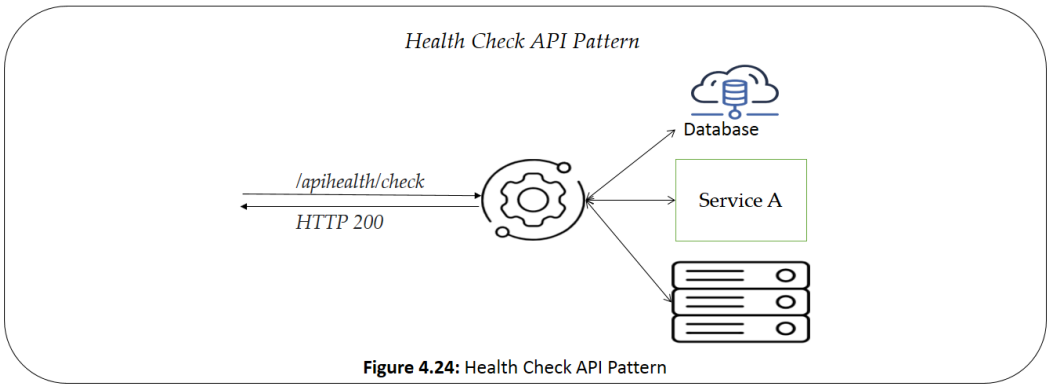


Figure 4.24: Health Check API Pattern

Advantages

- It is possible to test the availability levels and expected responses of an application using the API, by pinging it for memory usage as well as physical server resources. It is possible to check the status of an application's dependencies (external services and databases). The container orchestrators may also be able to use health checks to check the status of an app.

- Health Check API patterns provide a centralized and standardized method of monitoring and managing a system's health and status. In addition to ensuring that the system is functioning as expected, it can also assist with identifying and addressing potential issues or inefficiencies more quickly.
    - **Improved performance and user experience**: The Health Check API pattern can assist in improving overall system performance and user experience by providing real-time information about the system's health and status. In addition to helping to ensure that the system remains responsive and stable under heavy load, it can also be particularly useful for systems with high traffic and performance requirements.

Disadvantages

- It is important to consider the number of endpoints to expose for an application, as well as whether or not to use the same endpoint for monitoring as is used for general access, but to a specific pathway specially designed for health verification checks.

When to use this pattern

It is important to monitor the availability and correct operation of websites and web applications on a regular basis. Typically, this type of API is used in situations where it is necessary to monitor the availability and performance of a service or application, for example, in a production environment where downtime could have serious consequences.

Log aggregation pattern for centralized logging

When there is a microservices architecture and the need to troubleshoot a problem that spans multiple components, log aggregation is used. It attempts to solve the problem of how to view and search through a wide array of log files generated by individual distributed runtimes in a microservices environment in a way that is easy to understand and use. There is a solution to this problem by using a centralized logging service that aggregates logs from individual service instances and enables users to search, analyze and configure alerts based on specific logs that are produced by individual service instances.

It is a pattern of collecting log data from various components of a system and storing it in a central repository, such as a log file or database, via the use of dedicated tools and mechanisms. This can help to make the log data more accessible and easier to manage, as well as to prevent the log data from being lost or corrupted.

For example, as illustrated in *Figure 4.25: Log Aggregation Pattern:*

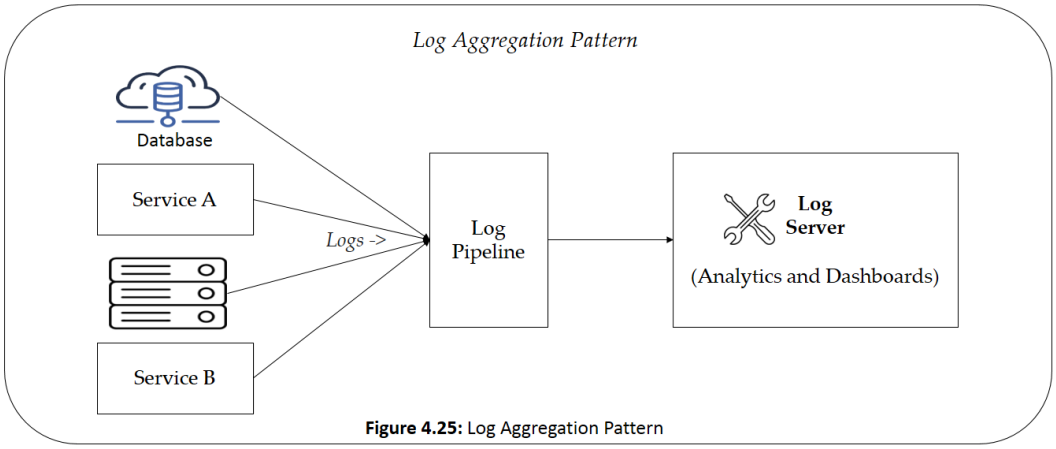


Figure 4.25: Log Aggregation Pattern

Advantages

- There are times when we need to debug problems across multiple components when we are using a microservice architecture, and logging aggregates are useful when we do so. The issue it attempts to solve is how to effectively view and search different log files resulting from individual distributed runtimes in a microservice environment.
- A log aggregation system saves engineers the time and effort of interpreting all the log files individually. This allows them to gain a better understanding of the systems they are responsible for. Additionally, log aggregation makes identifying correlations between events much easier.

Disadvantages

- There is no doubt that log management tools are resource-intensive. Not only do they require substantial storage for log data, but they also eat up a substantial amount of CPU time when collecting and managing log data.
- In addition, collecting and managing log data can result in a large amount of data that may be difficult to manage and interpret. Consequently, it can be difficult to identify and focus on the most relevant log data, as well as to identify trends or patterns within the data, making it difficult to identify the most important data.

When to use this pattern

There is a need to have some kind of machinery that can gather, store, aggregate, view, and analyse log data, which is why this is not an optional design pattern,

however some of the tools and third-party products can be evaluated in terms of how they can be implemented.

Application metrics pattern for performance monitoring

As a key component of the production environment, monitoring and alerting play a key role. Monitoring systems collect metrics from all parts of an application's technology stack that provide important information about the application's health. In addition to infrastructure-level metrics, there are also application-level metrics, such as service request latency and number of requests processed, that can be used to measure service request performance.

Service developers are responsible for collecting metrics about the behavior of their service in two ways. First, they have to instrument their service so they can collect data about it. The second step is to provide these metrics, along with those from the JVM and application framework, to the metrics server. It is possible to use an application metrics service such as AWS CloudWatch or Prometheus, which polls endpoints to obtain metrics, and Grafana, a data visualization tool, can be used to view metrics once they are in Prometheus.

Advantages

- The goal of metrics is to give developers an understanding of how a system is doing both in the past and in the present by giving them a numerical representation of data. Since metrics deal with numerical data, they offer the possibility of conducting statistical analysis and making predictions about how a system will behave in the future, and the purpose of metrics is to understand how an application behaves.

Disadvantages

- Among the potential disadvantages of using the Application Metrics pattern is that implementing and maintaining it can require significant resources and effort. In addition to the need for specialized tools and infrastructure, collecting and analyzing application metrics can be a complex process.

- Moreover, collecting and analyzing application metrics may generate a large amount of data, which can be difficult to manage and interpret.

When to use this pattern

There are a number of applications that can benefit from the Application Metrics pattern, including web applications, mobile applications, and distributed systems.

As well as improving the reliability and performance of an application, it can also provide valuable insights into how the application is being used and potential opportunities for optimization. In addition, the ability to monitor and track application metrics can be useful in meeting **service level agreements** (**SLAs**) and other performance requirements.

Audit logging pattern for compliance

The audit logging pattern is a design pattern that is used to capture and record information about the operations and events that occur in a software system. Among the many purposes of using audit logs, such as tracking and debugging errors, monitoring system usage and performance, and meeting compliance requirements, is tracking and debugging errors.

An audit logging pattern typically involves capturing and recording relevant information using dedicated logging mechanisms and tools. Among these details are the time and date of the event, the user or system responsible for initiating it, and the specific actions or operations that took place. In order to analyze and report the audit logs, they can be stored in a central repository, such as a database or log file.

There are a number of different types of systems that can benefit from the Audit Logging pattern, including web applications, distributed systems, and cloud-based applications. As well as improving a system's reliability and maintainability, it can also provide valuable insight into the system's behavior and usage. As part of compliance requirements, such as those relating to data privacy and security, the ability to track and record system events may be useful.

Exception tracking pattern for debugging

It is a design pattern for identifying, tracking, and managing errors and exceptions that occur in software systems. An error or exception is a situation in which the system is unable to perform a desired operation or function as expected, often as a result of invalid input, missing data, or other unexpected circumstances.

The exception tracking pattern involves the use of dedicated tools and mechanisms to capture and record information about errors and exceptions as they occur in the system. Input or data that was involved in the error can be included in this information, as well as the time and date of the error, the specific operation or function that caused the error, and the specific error. To identify and fix the underlying issues, the exception tracking system can be used to analyze and report on errors and exceptions.

A variety of different types of systems can benefit from the Exception Tracking pattern, including web applications, distributed systems, and cloud-based applications. Besides improving the reliability and maintainability of a system, it can also provide

valuable insights as to the types of errors and exceptions occurring within it. In addition, the ability to track and manage exceptions can assist in preventing errors from affecting the system's performance and availability.

Monitoring Vs microservices observability

The concepts of monitoring and microservices observability are related but distinct in software engineering. As part of monitoring, data about the performance and behavior of a software system is collected and analyzed with the aim of identifying and resolving potential issues or inefficiencies. On the other hand, microservice observability refers to the ability to understand the behavior and interactions of the individual microservices within a microservice-based system.

There is a significant difference between monitoring and microservices observability in terms of granularity. Monitoring typically involves aggregating data from across the entire system, providing an overview of the system's overall performance and behavior. The observability of microservices, on the other hand, involves analyzing the interactions and behavior of individual microservices and provides a more detailed, fine-grained view of the entire system.

For example, as illustrated in *Figure 4.26: Monitoring Vs Microservices Observability:*

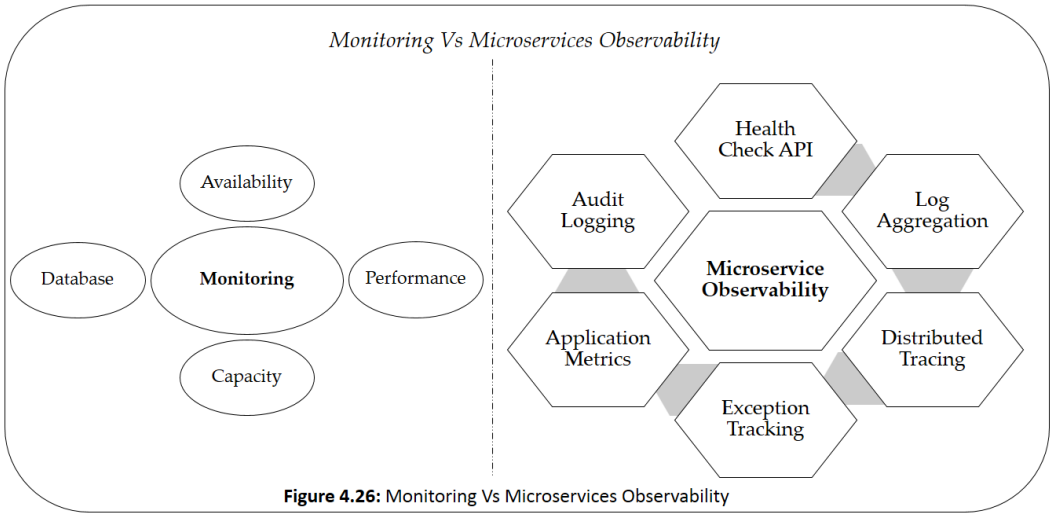


Figure 4.26: Monitoring Vs Microservices Observability

In addition, the focus of the two approaches is another important difference. Observability of microservices is aimed at understanding the behavior and interactions of the individual microservices that make up the system, as opposed to monitoring, which is typically focused on identifying potential issues or inefficiencies in the system. Observability of microservices can therefore be used to identify and debug interaction issues between microservices, while monitoring can be used to identify and address performance and availability issues related to the system as a whole.

A software system's performance and reliability can be managed and improved using both monitoring and microservices observability, both of which are important tools. The two tools have some overlap in terms of the data they gather and the goals they seek to achieve, but they serve different purposes and can be combined to provide a more comprehensive view of the system and its behavior.

By now we should be clear about the difference between monitoring vs observability. For today's workload it is not an optional feature rather it is a must have capability for a stable setup.

Cross-cutting concern pattern

The cross-cutting concern pattern is a design pattern that addresses concerns that are common across a number of modules or components in a software system. There are many aspects of the system that are associated with these concerns, such as logging, security, or performance, which are not specific to a particular module or component.

Blue-green deployment pattern for zero-downtime

A blue / green deployment is a deployment strategy in which you create two separate, but identical environments in which you run the current application version and the new application version, with the current environment running the current application version. By simplifying the rollback process if a deployment fails, a blue / green deployment strategy can greatly increase application availability and reduce deployment risk. Following the completion of testing on the green environment, live application traffic will be directed to the green environment and the blue environment will be deprecated once the green environment has been completed.

For example, as illustrated in *Figure 4.27: Blue-Green Deployment Pattern:*

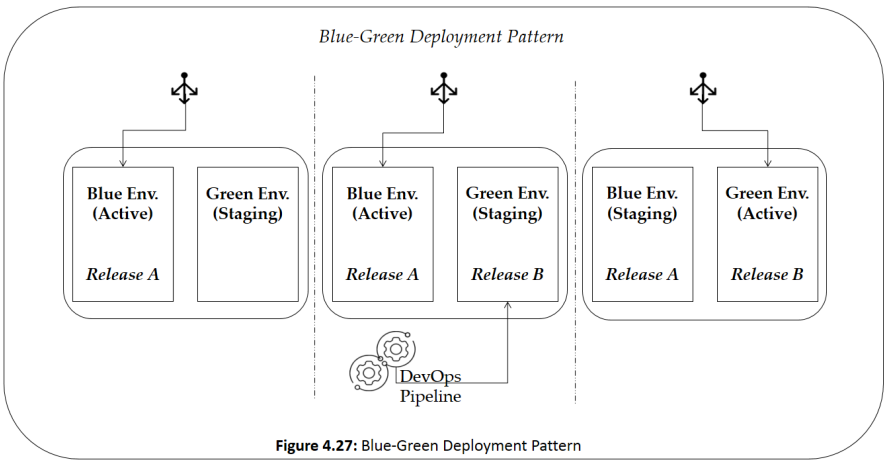


Figure 4.27: Blue-Green Deployment Pattern

Advantages

- In any deployment to production, there is always a risk that the deployment will fail, which could result in a system outage and adverse effects on users. The release can be tested and completed on a separate non-live environment before being deployed to production. Additionally, the original production environment can be maintained as a fall-back during the "burn-in" period for the new release as a fall-back.

Disadvantages

- In comparison with other deployment methods, blue-green deployments are more expensive, and provisioning infrastructure on-demand using IaC is an effective way to save money.
- There may be a period of slowness when users are suddenly switched to a new environment for the first time.

When to use this pattern

In particular, this pattern supports agile development practices with short release cycles using **Continuous Deployment** (**CD**) via an automated DevOps pipeline. This pattern is particularly well suited for production release schedules based on CD via an automated DevOps pipeline.

Canary pattern for incremental rollouts

Canary deployment refers to the process of introducing new updates in microservices in a safe and reliable manner. It is similar to blue-green deployment, but uses a slightly different approach to it. As opposed to another full environment waiting to be switched over once deployment has been completed, canary deployments cut over just a small subset of servers or nodes before the other servers or nodes have been completed. Generally, this deployment pattern has a lower risk of failure than the other deployment patterns (in contrast to the other deployment patterns). It can be set in the target environment and updated in small increments such as 5%, 25%, 50%, 75%, and 100%. Due to the incremental nature of the software, only a small number of users will be affected if an app-breaking bug becomes available in the production environment after a new release.

Advantages

- By using this pattern, organizations will be able to test in production with real users and compare the performance of both versions live, as well as define the threshold for triggering a rollback and accelerate the process since only a subset of users exposed to the updated version will have to be rerouted.

- Using the Canary pattern to release a new version of an application to a limited group of users reduces the risk associated with deploying a new version of an application. Especially useful for critical applications or applications that have a large user base, this can ensure that the application remains stable and available even if the new version encounters problems.
- Enhanced user experience as the Canary pattern is used to test a new application with real users before releasing it to the entire user base, it can assist in identifying and resolving potential issues that may negatively affect the user experience. This can help to improve the overall quality of the application, as well as to ensure that users have a positive experience when using the application.

Disadvantages

- A disadvantage of the canary pattern is that it can be challenging to implement, particularly for applications which have a large user base or complex architectures. The canary deployment process can require significant planning and effort, as well as the use of specialized tools and infrastructure.
- As a result, the canary pattern may require ongoing monitoring and management to ensure that the new version of the application performs as expected and to identify and resolve any potential issues. As a result, the implementation of the canary pattern can be more complex and costly.

For example, as illustrated in *Figure 4.28: Canary Pattern:*

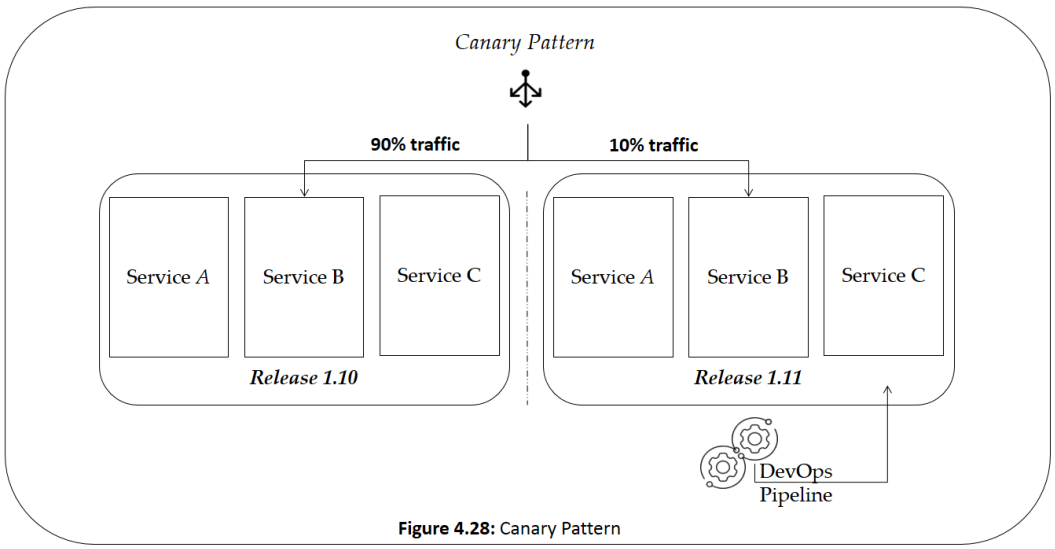


Figure 4.28: Canary Pattern

When to use this pattern

For applications that are of utmost importance to the business or have a large user base, the canary pattern can be particularly useful, since it can provide assurance that the application remains stable and available even when there are problems with the new version. As well as being useful for complex or multi-faceted applications, it can help ensure that the new version is stable and performs well before it is released to all users.

Canary Vs blue-green deployment pattern for deployment strategies

There are some key differences between the Canary pattern and Blue-Green deployment pattern when deploying software applications in a production environment.

- **Scale**: As part of the Canary Deployment pattern, a new version of an application is released to a small group of users, while the Blue-Green Deployment pattern involves running two identical versions of an application, one serving live traffic, the other testing and deployment.
- **User experience**: By using the Canary pattern, a new version of an application can be tested by real users before being released to the entire user base, allowing for the identification and resolution of potential issues before they affect the user experience. By contrast, the Blue-Green Deployment pattern does not require real user testing, since traffic is routed to the live application during the deployment and testing of the new version.
- **Rollback**: Since the Canary pattern is intended to test a new version of an application with a small subset of users before it is released to the entire user population, it does not typically include a rollback mechanism. By contrast, the Blue-Green Deployment pattern includes a rollback mechanism, since it involves running two identical versions of an application and switching between them as needed.

In general, the canary deployment pattern emphasizes testing and feedback, whereas the Blue-Green deployment pattern emphasizes availability and rollback. Both patterns can be useful in different situations, and which pattern will be the most suitable for a particular deployment will depend on the specific requirements and goals of the application.

Circuit breaker pattern for fault tolerance

It is a reliability pattern based on the concept of circuit breaker, which is a term that refers to the electrical switch that automatically protects an electrical circuit

from damage caused by an overload. In an application with two services invoked synchronously, it becomes relevant because one service may become unavailable, and requests will continue to pin it until it becomes available again. Until the services are able to pin unavailable services, this could exhaust network resources, resulting in low performance and poor user experience. Thus, the pattern prevents a failure of one component from cascading beyond its limit and affecting the entire system.

Circuit breakers are implemented as state machines with the following states:

- **Closed**: The proxy routes the request to the operation and keeps track of recent failures. If the operation is unsuccessful, the proxy increments the count. A proxy is placed in the Open state if the number of recent failures exceeds a specified threshold within a given period of time. The proxy starts a timeout timer at this point, and when it expires it becomes Half-Open.
- **Open**: The application's request fails immediately, and an exception is returned to the application as a result of the failure.
- **Half open**: Invoking the operation is permitted by a limited number of requests from the application. Upon successful completion of these requests, the circuit breaker switches to the Closed state (resets the failure counter) and assumes that the fault that caused the failure has been fixed.

For example, as illustrated in *Figure 4.29: Circuit Breaker Pattern:*

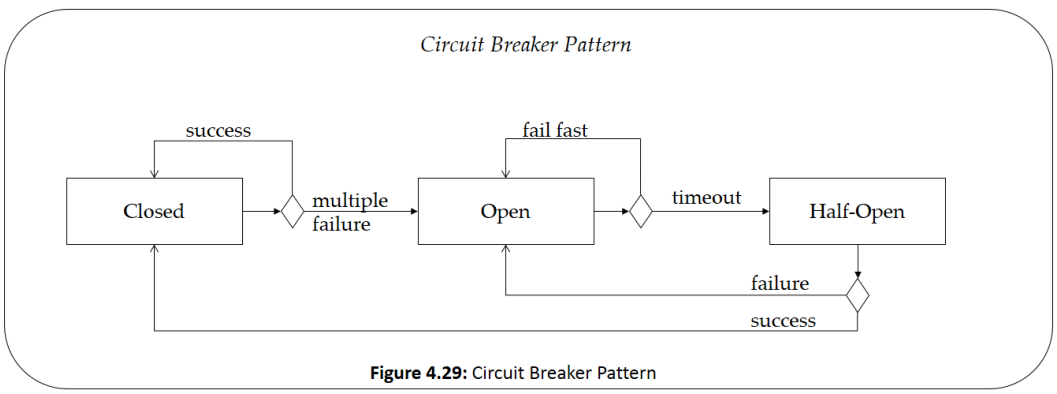


Figure 4.29: Circuit Breaker Pattern

Advantages

- A circuit breaker acts as a proxy for operations that might fail. It should monitor the number of recent failures that have occurred within the system, and use this information to determine whether to permit the operation to continue or simply return an exception immediately in the event of a failure.
- **Increased availability, and Improved resiliency**: The Circuit Breaker pattern prevents one service's failure from impacting the availability of the

entire system by breaking the circuit and stopping the flow of requests to the failing service. As a result, the system's overall availability can be improved.

Disadvantages

- There should be adequate logging even when the circuit is in half-open or open status, as it allows the administrator to further optimize the failure threshold or success threshold values even when the circuit is in half-open or open status.
- It is possible for a service to fail due to several reasons, for instance being unable to process requests due to overload. Circuit breakers must be designed in this situation to identify the cause and act accordingly.
- In some cases, the circuit breaker pattern can cause legitimate requests to be blocked, which can affect the availability and functionality of a system even when the target service is healthy and responsive.

When to use this pattern

Despite our best efforts, we cannot guarantee that each service will be fault-tolerant at all times. Networks can go down, systems can starve for resources, databases might go down, and so on. To handle such situations gracefully, we need to incorporate fault tolerance in our services and Circuit Breaker pattern addresses it very well.

When a system is unable to handle a large number of requests or has become unresponsive, the Circuit Breaker pattern prevents it from becoming overwhelmed. In addition to experiencing high levels of traffic, this can also occur when it is dependent on a service that is slow or unresponsive. By detecting these situations and automatically failing fast, the Circuit Breaker pattern prevents the system from trying to process requests that will fail in the future. The system can be made more reliable and available, and the load on the system can be reduced, allowing it to recover more quickly.

External configuration pattern for dynamic configuration

The External Configuration pattern is a design pattern for microservices that enables services to retrieve their configuration from external sources, including configuration servers or configuration databases. By using this pattern, microservice architectures can be made more flexible and maintainable. Many times, services are required to run in a variety of environments, and environment-specific configuration is needed, such as secret keys, database credentials, and so on. Changing a service for each environment comes with a number of disadvantages. In this case, we might be

wondering how we could make a service run in multiple environments without any modifications being made to it.

In a microservices architecture, each service may have its own unique configuration settings that control its behavior and functionality. Some of the configuration settings include the network address of the service, the maximum number of connections the service can handle, and the amount of memory or CPU allowed for it.

By using the External Configuration pattern, these configuration settings can be managed and maintained externally, rather than being hardcoded within the service itself. By doing so, it is much easier to modify and update configuration settings without rebuilding and redeploying the service. Additionally, it enables the system to be configured differently in different environments (for example: development, staging, and production), which can improve its flexibility and maintainability.

External Configuration is commonly implemented using configuration servers (such as Spring Cloud Config) and configuration databases (such as MongoDB or Apache ZooKeeper). Depending on the system's requirements and constraints, a specific technology will be used.

Service discovery pattern for service registration and discovery

This pattern is used to create microservice architectures that are more scalable, flexible, and resilient as it allows services to discover each other and communicate with each other across networks. It is also used to enable microservice architectures to become more scalable, flexible, and resilient.

In a microservices architecture, services are typically deployed in isolation of each other and run on different hosts or in different environments and may run on different hosts or in different environments. As a result, it can be challenging for services to find each other and communicate, as they may not have a fixed IP address or they may be deployed or scaled dynamically.

By providing a service registry that contains a list of all of the available services and their network addresses, the service discovery pattern addresses this problem. A service can query the service registry in order to find the network address of another service when it wishes to communicate with it. In this way, it is possible for services to discover one another dynamically, and the system can scale and evolve more easily as a result.

In order to implement service discovery, it is common to use DNS, load balancers, as well as dedicated service discovery tools such as consul and eureka. The specific technology used for the implementation will depend on the requirements and constraints of the system.

The types of service discovery are as follows: server-side and client-side.

Client-side service discovery pattern

It is the client-side microservice that searches for the service it needs, and uses a discovery server to locate that service, then makes a second request to the actual service. This is referred to as client-side service discovery. It is obvious that two calls will be made, one to a service registry and one to an actual service. This is called client-side service discovery.

For example, as illustrated in *Figure 4.30: Client-side Service Discovery Pattern*

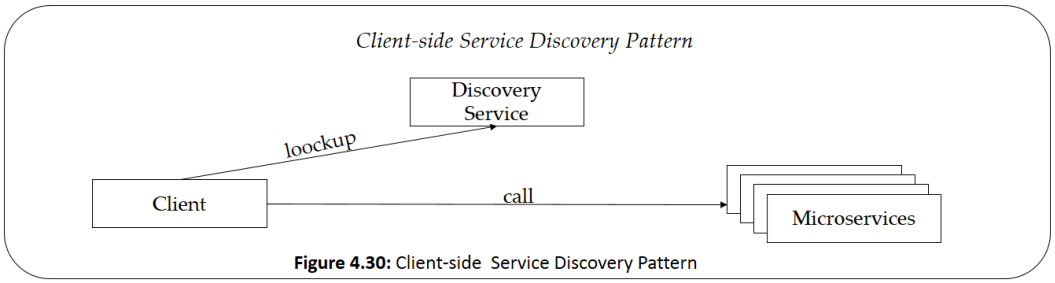


Figure 4.30: Client-side Service Discovery Pattern

Server-Side discovery pattern

This is called the Server-side service discovery process, where the client sends the information describing the service it wants to reach, and the load balancer will check with the service registry and transport the request to the right destination based on the information provided.

For example, as illustrated in *Figure 4.31: Server-Side Service Discovery Pattern:*

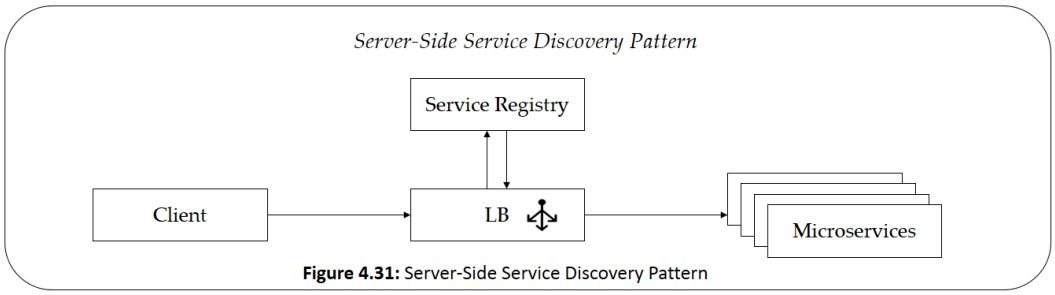


Figure 4.31: Server-Side Service Discovery Pattern

With microservice architectures, one of the primary challenges is allowing services to discover and interact with one another. In addition to making it difficult for services to communicate with each other due to the distributed nature of microservices architectures, they also pose additional challenges, such as monitoring the health of those systems and letting customers know when new applications are made available.

Service discovery methods

- **DNS based**: When it comes to the discovery of services using a DNS approach, standard DNS libraries are used as clients. In this model, each microservice receives an entry in a DNS zone file, and performs a DNS lookup to find the microservice.
- **Key/Value Store and sidecar based**: As the central service discovery mechanism, a key/value store and sidecar are used to connect a strongly consistent data store such as Consul or Zookeeper with a sidecar. Essentially, a sidecar is used to communicate with this mechanism. In this model, a microservice is configured to communicate with a local proxy. A separate process communicates with service discovery, and then uses that information to set up the proxy.
- **Specialized service discovery and library/sidecar based**: This model exposes functionality directly to the end developer using a library (and API) that is directly exposed to the developer, who uses the library to make communication with a specialized service discovery solution.

Advantages

- A service discovery system can locate a network automatically, meaning that a long configuration process will not be required in order to set it up. It consists of devices connecting via a common language over the network, allowing devices and services to connect without the need for manual intervention. (that is Kubernetes service discovery, Amazon Web Services service discovery)
- There is an advantage to server-side service discovery in that it makes the client application lighter since it is not required to deal with the lookup procedure and can simply provide a request for services to the router, as opposed to dealing with the lookup procedure.
- This type of service discovery has the advantage of avoiding the need for the client application to communicate with a router or a load balancer, because the client application does not have to travel through these devices.

Disadvantages

- **Having a dependency on the service registry**: The Service Discovery pattern relies on a central service registry for maintaining a list of available services, as well as the network addresses of those services. There is a possibility that the system will become unavailable or unstable if the service registry is not available or inaccessible, which can result in the services not being able to communicate with each other.

- The overhead and latency associated with querying a service registry and resolving the network address of a target service can significantly impact the performance of the communication process. For applications that require low-latency processing or high-throughput processing, this can have a major impact on the overall performance of the system.
- The service registry maintains a list of all the available services and their network addresses, which can also make it a valuable target for attackers. Therefore, proper security measures must be taken in order to protect the service registry and prevent unauthorized access from taking place.

When to use this pattern

Using the Service Discovery pattern is the ideal solution for microservice architectures in which services are deployed independently and can run on a variety of hosts and environments depending on their location. This pattern is particularly useful when it comes to situations in which the network addresses of services may change dynamically, for example, when services are deployed or scaled dynamically.

A service discovery pattern can also be useful in situations where the number of services or the size of the system is expected to increase over time, as well. As a result of having a central service registry, this pattern allows the system to scale and evolve more easily by eliminating the need to manually update the network addresses of the services.

Conclusion

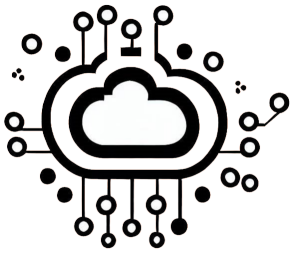
Building a Solid Foundation:

Navigating Microservice Architecture with Proven Design Patterns

In my opinion, a microservice architecture is one that allows functionalities to be flexible, testable, and scalable. However, the execution of a microservice architecture can present several challenges that a microservice design pattern can assist with overcoming. In order to achieve best practices when setting up microservices architecture, you need to use the right design patterns. In this chapter, we have discussed design patterns related to application decomposition into microservices, integration, data management, observability, as well as other cross-cutting concerns in order to achieve essential microservices design principles. With the level of details shared you will be able to select right combination of patterns for your microservices based application design.

Our next chapter will cover design patterns that are relevant to Cloud-Native architecture and show how to apply them to your applications. When you

implement the right design pattern based on your use case, you can increase the reusability of your components, which will reduce the time and effort it takes to develop your application. We will discuss each of these patterns with the help of use cases accompanied by diagrammatic representations of their components and interconnections. It is not surprising that *Chapters 4 (Design Patterns for Microservices) and 5 (Cloud-Powered Microservices)* together will cover all the key design patterns that will prove useful for building cloud-based microservices applications that are reliable, scalable, and secure.



Chapter 5
Cloud-Powered Microservices

Unleashing the Power of Cloud with Cloud-Powered Microservices

Introduction

In this chapter, we will cover the design patterns for Cloud-Native architecture and how to use them efficiently. Implementing the correct design pattern for your use case can help increase component reusability, thus reducing development time and effort. Each of these patterns will be reviewed using use cases with diagrammatic representations of their components and interconnections. Together, *Chapters 4 (Design Patterns for Microservices) and 5 (Cloud-Powered Microservices)* cover all the key design patterns that are useful for building cloud-based microservices applications that are reliable, scalable, and secure.

In cloud computing, computing services, such as storage, processing, networking, and others, are delivered over the Internet on a pay-as-you-go basis over the Internet. With cloud-based infrastructure and services, organizations can deploy and scale their applications quickly and easily without having to manage and maintain physical servers and other infrastructure.

As a result of combining microservices with cloud computing, organizations can reap many benefits such as increased scalability, reliability, and agility. The use of cloud-based infrastructure and services, for example, can allow organizations to automatically scale their applications up and down based on demand, and eliminate the need to employ complex and expensive scaling solutions. Further, by using

microservices, it becomes easier to decouple the different components of the system, which will allow you to update and modify individual services without affecting the rest of the system, making it easier to modify and update other components of the system as well. In our previous chapters, we discussed Modern Application Design Principles, Microservice Adoption Framework, and Microservice Design Patterns for Microservices. As we move forward, it is necessary to understand how Cloud, and in particular the key components of Cloud Native, such as cloud-based infrastructure and services, **Continuous Integration and Delivery** (**CI/CD**), some PaaS, and Serverless solutions can help us to take our solutions to the next level.

For example, as illustrated in *Figure 5.1: Cloud-Powered Microservices - Cloud Design Patterns* that enhance your microservices design.

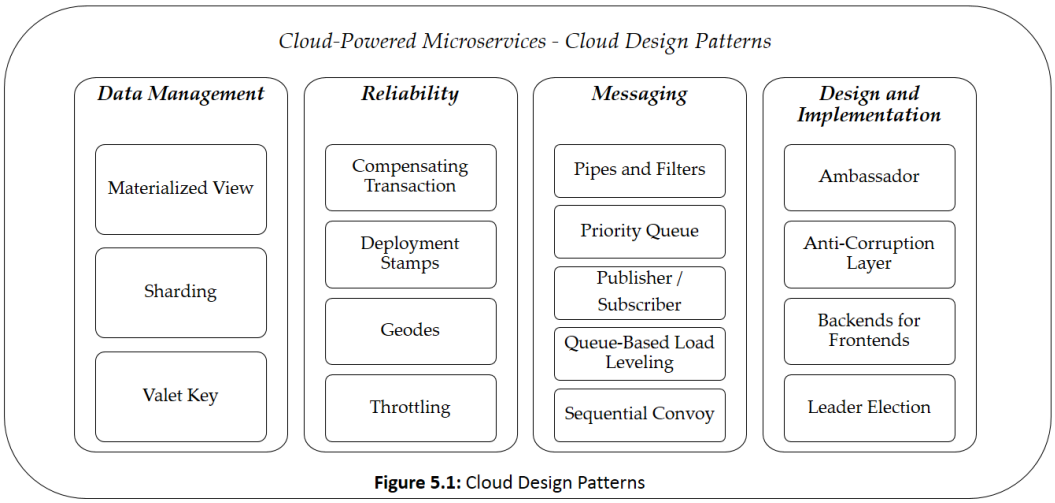


Figure 5.1: Cloud Design Patterns

In this chapter we will discuss Cloud Native Data Management, Reliability, Design and Implementation, and Messaging Patterns.

- **Data Management Design Patterns**: There are a number of proven design patterns for cloud native data management that have been developed specifically to solve common problems related to data management within the context of cloud computing. As a result, these patterns provide a structured approach to solving problems, and can be applied to a wide variety of scenarios and technologies associated with cloud-based computing. Some common cloud native data management design patterns include the Cache-Aside, CQRS, Event Sourcing, Index Table, Materialized View, Sharding, Static Content Hosting, and Valet Key.
- **Design and Implementation Design Patterns**: A set of proven solutions to common challenges associated with building, deploying, and managing cloud-based applications are known as cloud native design and implementation design patterns. There are a number of cloud-based

scenarios and technologies that can be applied to these patterns. There are several common design and implementation patterns for cloud native applications, including the Ambassador, Anti-Corruption Layer, Backends for Frontends, Compute Resource Consolidation, External Configuration Store, Leader Election, Pipes and Filters, and Static Content Hosting.

- **Messaging Design Patterns**: Using cloud native messaging design patterns, developers can solve common challenges associated with developing, deploying, and managing cloud-based messaging applications. Cloud-based messaging technologies and scenarios can be applied to these patterns, which provide a structured approach to solving problems. There are several common patterns associated with cloud-native messaging, including the Asynchronous Request-Reply, Claim Check, Choreography, Competing Consumers, Pipes and Filters, Priority Queue, Publisher-Subscriber, Queue-Based Load Levelling, Scheduler Agent Supervisor, and Sequential Convoy.
- **Reliability Design Patterns**: There are a number of practices and techniques that can be used to improve the reliability of a system, which are known as reliability design patterns. As a general rule, these patterns aim at identifying and addressing potential failure points within a system, as well as implementing mechanisms that can be used to detect and recover from failures that do occur. There are many patterns that will improve a system's reliability related to Redundancy, Failover, Monitoring, Recovery, Load Balancing. We will discuss some of the patterns related to Compensating Transactions, Deployment Stamps, Geodes, and Throttling, in this chapter.

Structure

In this chapter we will discuss following topics:

- Data management design patterns
    - Materialized view
    - Sharding
    - Valet Key
- Design and implementation patterns
    - Ambassador
    - Anti-corruption layer
    - Backends for frontends
    - Leader election
- Messaging design patterns

    - Pipes and filters
    - Priority queue
    - Publisher-subscriber
    - Queue-based load levelling
    - Sequential convoy
- Reliability
    - Compensating transaction
    - Deployment stamps
    - Geodes
    - Throttling
- Conclusion

Objectives

Microservices perform specific tasks and communicate with other microservices through well-defined APIs. Scalability and flexibility are among the key benefits of cloud-powered microservices. Because each microservice is independent, it can be scaled up or down as needed, without affecting the whole system. The objective of this chapter is to further enhance power of microservices by working closely with some of the cloud native patterns related to Data Management, Design and Implementation, Messaging, and Reliability. These pattern will help:

- **Improved scalability**: The ability to scale microservices independently allows applications to scale as required. This is particularly useful in the cloud, where resources can be easily added or removed as needed.
- **Enhanced flexibility**: Microservices are modular and can be developed and deployed independently, making it easier to update and modify individual components of the application. Some of the design and implementation pattern covered in this chapter will further enhance flexibility.
- **Improved reliability**: It is possible to design microservices to be highly resilient and fault-tolerant, which can enhance the overall reliability of the system. We will be discussing couple of cloud native pattern specialized to improve on the reliability.

Data management design patterns

Almost all of the quality attributes of cloud applications are influenced by data management, which is a key component of cloud applications. As a result of

performance, scalability or availability requirements, data is typically hosted across multiple servers and in different locations, which can present a number of challenges. For example, data consistency must be maintained, and data will typically need to be synchronized across different locations. To maintain security assurances of confidentiality, integrity, and availability, data should be protected at rest, in transit, and by authorized access mechanisms.

We have already discussed CQRS, and Event Sourcing Data Management Design Patterns in previous chapter. Let us discuss some of the other key methods that will enhance our data management capabilities.

Materialized view

Materialized views are pre-computed tables that contain the results of a SELECT statement. They are sometimes called snapshot views since they contain the results of a SELECT statement in a table that can be queried like other tables.

In materialized views, the underlying data is not frequently updated and is frequently queried, but the data in the view is frequently queried. The results of the SELECT statement can be accessed quickly by materializing the view, rather than having to execute the SELECT statement each time the view is queried.

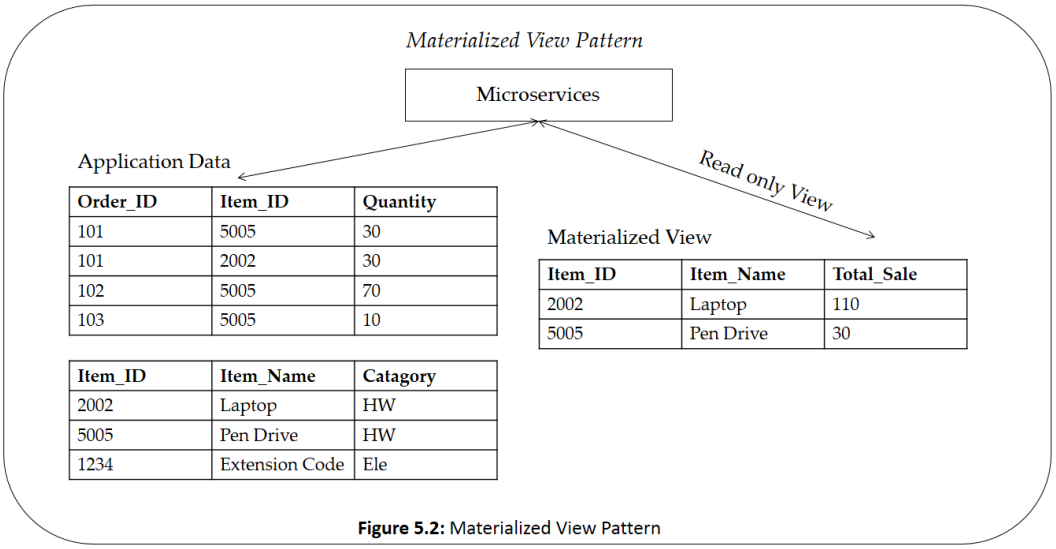
It is often a priority for developers and data administrators to consider how data will be stored rather than how it will be read when storing data. Usually, the storage format chosen depends on the data format, the requirements for managing data size and integrity, and the type of storage used. NoSQL document stores, for instance, often present data as aggregates containing all the information for each entity. It is important to note, however, that this may negatively impact queries. In order to obtain the required information, a query must extract all the data for the relevant entities when it needs only a subset of the data from some entities, for example a summary of orders for several customers without all of the order details.

Advantages

- Materialized views can improve the performance of queries that involve aggregating large amounts of data, since the results of the **`SELECT`** statement are stored and readily accessible. In applications with high concurrency or queries that are run frequently, this may prove particularly useful.
- Materialized views reduce database load by pre-compiling and storing the results of **`SELECT`** statements in a table in order to reduce the load on the database. As a result, the number of queries that are executed against the database can be reduced, which can contribute to improved system performance.

- Materialized views simplify queries by allowing you to retrieve precomputed results rather than having to write complex **SELECT** statements. This makes it easier to write and maintain queries, and also improves readability.
- In cases where the underlying data changes frequently and you wish to ensure that your queries return consistent results, materialized views can assist in ensuring data consistency by storing the results of **SELECT** statements in a table.
- A materialized view is often used in data warehousing applications to improve query performance against large datasets. The results of **SELECT** statements can be pre-computed and stored, allowing the data to be accessed more quickly and with fewer resources.

For example, as illustrated in *Figure 5.2: Materialized View Pattern:*



Order_ID	Item_ID	Quantity
101	5005	30
101	2002	30
102	5005	70
103	5005	10

Item_ID	Item_Name	Catagory
2002	Laptop	HW
5005	Pen Drive	HW
1234	Extension Code	Ele

Item_ID	Item_Name	Total_Sale
2002	Laptop	110
5005	Pen Drive	30

Figure 5.2: Materialized View Pattern

Disadvantages

- Materialized views have the disadvantage of containing stale data, which can be a major drawback. Materialized views are updated only when the REFRESH MATERIALIZED VIEW statement is executed, so if the underlying data changes frequently, the materialized view may not be reflected until it is refreshed.
- The materialized view requires additional maintenance as it must be refreshed to ensure that it contains up-to-date data. As a result, the process can be time-consuming and resource-consuming, particularly if the materialized view is based on a complex **SELECT** statement.

- This may be a problem if you work with large datasets or have limited storage capacity. Materialized views require additional storage space to store the results of the **`SELECT`** statement.
- Materialized views generally do not support DML (insert, update, delete) operations, which means that they cannot be used to modify data. In order to ensure that the materialized view reflects these changes, you must modify the underlying data in the base table.
- There may be compatibility issues with materialized views, and they may not be supported by all database systems due to differences in syntax. If you are using multiple database systems or need to migrate your data, this can make it more difficult to work with materialized views.

When to use this pattern

The underlying data of your **`SELECT`** statement does not change very frequently and you frequently use it. You wish to improve the performance of queries involving aggregation of large quantities of data. By pre-calculating and storing the results of **`SELECT`** statements in a table, you will reduce the load on the database. Accessing pre-computed results will simplify queries as opposed to writing complex **`SELECT`** statements.

Sharding

Data Sharding refers to the practice of dividing large datasets into smaller pieces, or "shards," and distributing them across multiple servers or databases. By allowing a system to handle more concurrent users and larger amounts of data, this can improve its performance and scalability.

Sharding patterns can be categorized into several types, including:

- Lookup strategy. In this strategy, the sharding logic creates a mapping that routes requests for data to the shard which contains the data using the shard key. The tenant ID may be used as the shard key in a multi-tenant application in which all the data for each tenant is stored together. There may be multiple tenants sharing the same shard, but data for a single tenant will not be spread across multiple shards.
    - **Advantages**: Using virtual shards offers greater control over shard configurations and usage. The use of virtual shards also minimizes the impact of rebalancing data because new physical partitions can be added to compensate for workload imbalances.
- Using range-based sharding, data can be divided according to a range of values. A user's data could be stored on one shard for users with last names

that begin with A-M, while data for users whose last names begin with N-Z would be stored on another shard.

    - **Advantages**: This approach is easy to implement and works well with range queries, since they are able to retrieve multiple items from a single shard at once.

- Using a hash function to map a key value to a specific shard, this type of sharding divides the data based on a hash function. Using a hash function, all data for users whose user IDs correspond to odd numbers could be stored on one shard, while data for users whose user IDs correspond to even numbers would be stored on another.

    - **Advantage**: In this way, there is a greater chance that data and load will be distributed more evenly.

For example, as illustrated in *Figure 5.3: Sharding Pattern*:

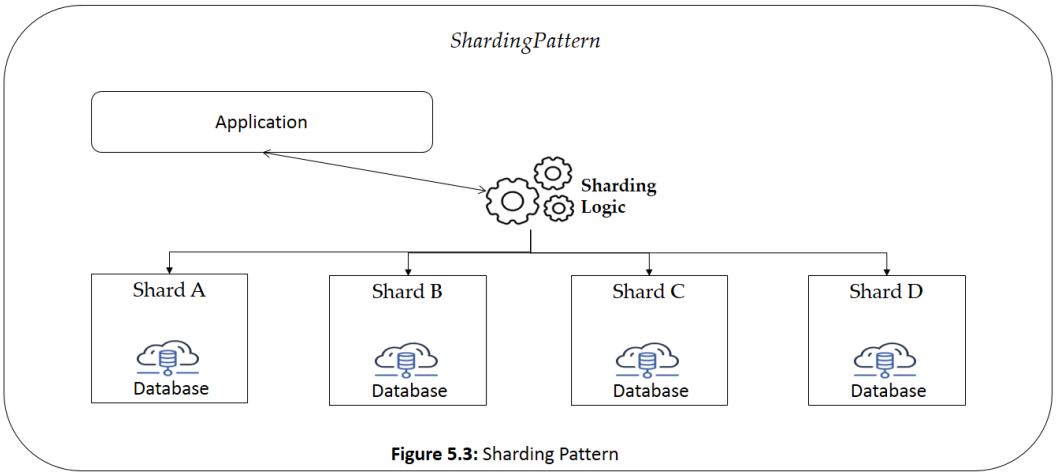


Figure 5.3: Sharding Pattern

Disadvantages

- There can be a lot of complexity involved when it comes to sharding, particularly when you are trying to use it to partition and distribute the data within an existing database. It takes a lot of planning and design, as well as custom development to implement sharding within an existing database.
- Due to additional complexity associated with routing requests to the appropriate shard and merging the results, there is an increased overhead to database operations when sharding is employed. This can result in slower performance in some cases as a result of the extra complexity.

- The difficulty of executing queries spanning multiple shards, since the data is distributed across multiple physical locations, can make it very difficult to perform certain types of data analysis or reporting for the data that is being distributed over multiple shards.

When to use this pattern

- It is possible to use Sharding when a database is growing too large for one server to handle efficiently: Sharding is a method by which a larger database is divided into smaller, more manageable pieces.
- Sharding can be used to distribute a database's workload across multiple servers to improve performance, in the case of a database experiencing performance problems because of a high volume of reads and writes.

Valet key

The valet key design pattern is used in situations where a user needs to give another user access to a system or resource, but does not wish to allow them full access to it. It is commonly used in situations where a user needs to grant access to a resource, but does not wish to give them full control over it.

This key or token is usually referred to as a valet key. The key allows time-limited access to specific resources as well as predefined operations such as reading and writing to storage or queues, or uploading and downloading in the web browser. In addition to enabling clients to perform the required operations without requiring the application to directly handle the data transfer, applications may create and issue valet keys to client devices or web browsers quickly and easily. As a result, the application and the server are not burdened with the overhead of processing and the impact on performance and scalability.

When implementing this pattern, you should consider the following factors:

- In the event that the key is leaked or compromised, it effectively unlocks the target item and is available for malicious use for the remainder of its validity period. To minimize the risk of unauthorized operations against the data store, specify a short validity period. The client may not be able to complete the operation before the key expires, however, if the validity period is too short.
- Generally, the key should allow the user to only perform the actions required to complete the operation, such as read-only access if the client is not permitted to upload data to the data store. In order to upload files, it is common to specify a write-only key, a location, and a validity period. It is crucial to specify correctly the resource or set of resources for which the key applies.

- Use HTTPS to deliver the key over a secure channel. The key can be embedded in a URL that users activate in a web page, or it can be used in a server redirection operation that triggers the download automatically.
- You should protect sensitive data while in transit. Sensitive data delivered through the application will usually be sent using **Transport Layer Security** (**TLS**), and this should be enforced for clients that directly access the data store.

For example, as illustrated in *Figure 5.4: Valet Key Pattern:*

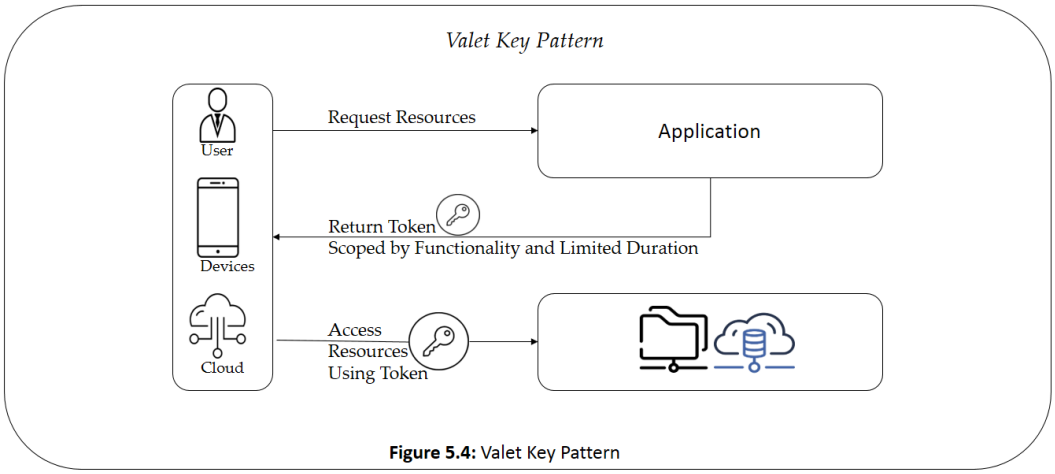


Figure 5.4: Valet Key Pattern

Advantages

- You can reduce the risk of unauthorized access to other resources by giving an application or service access only to the resources it requires.
- The performance of an application or service can be improved by only granting access to the resources it requires, thereby reducing the risk of overloading the system.

Disadvantages

- The implementation of the valet key pattern may require additional configuration and management efforts, which can lead to an increase in the complexity of your cloud environment.
- In terms of functionality, an application or service may be limited in terms of what functions it can perform, which could impact the overall value of the application or service.

When to use this pattern

A valet key minimizes resource loading and maximizes performance and scalability. By using a valet key, the resource doesn't need to be locked, no remote server call is required, the number of valet keys can be unlimited, and the data transfer is not impacted by a single point of failure.

Design and implementation patterns

Cloud-based applications can be built using a number of different design and implementation patterns. Among the factors considered in good design are consistency and coherence in component development and deployment, maintainability, so as to simplify administration and development, and reusability, allowing components and subsystems to be reused in other applications and situations. In the design and implementation phases, decisions are made that have a significant impact on the quality and overall cost of ownership of cloud-hosted applications and services. Some of the key design considerations would include Serverless computing, Microservices, Containerization, Security, Auto-scaling, and Reliability.

Ambassador

In the ambassador pattern, communication between microservices is handled by a separate service, known as an ambassador. By abstracting away, the underlying implementation details and providing a consistent interface for other services to interact with, the ambassador acts as a proxy for the microservices.

One of the major benefits of using the ambassador pattern is that it can help to decouple the microservices from one another. This allows it to be easier to update or change the implementation of a service without affecting the other services. Furthermore, ambassadors are capable of handling tasks such as authentication, and monitoring, thereby reducing the workload on microservices.

It is possible to implement the ambassador pattern using a wide variety of technologies, such as reverse proxies or load balancers. As a result of this pattern, common client connectivity tasks like monitoring, logging, routing, security (such as TLS), and resilience patterns can be offloaded in a language-independent manner. In order to extend the networking capabilities of legacy applications, or other applications that are difficult to modify, it is often used.

Advantages

- Offloaded features can be managed independently of the application. You are able to update and modify the ambassador without affecting the legacy functionality of the application. It also allows for separate, specialized teams

to implement and maintain security, networking, and authentication features that have been moved to the ambassador.

- The ambassador can handle authentication and rate limiting, which can help to reduce the burden on microservices themselves. This can allow the microservices to be scaled up and down more easily.
- The ambassador can handle tasks such as authentication and access control, which can help to enhance the application's overall security.
- The ambassador can perform tasks such as monitoring and error handling, which can contribute to the application's overall reliability.

For example, as illustrated in *Figure 5.5: Ambassador Pattern:*

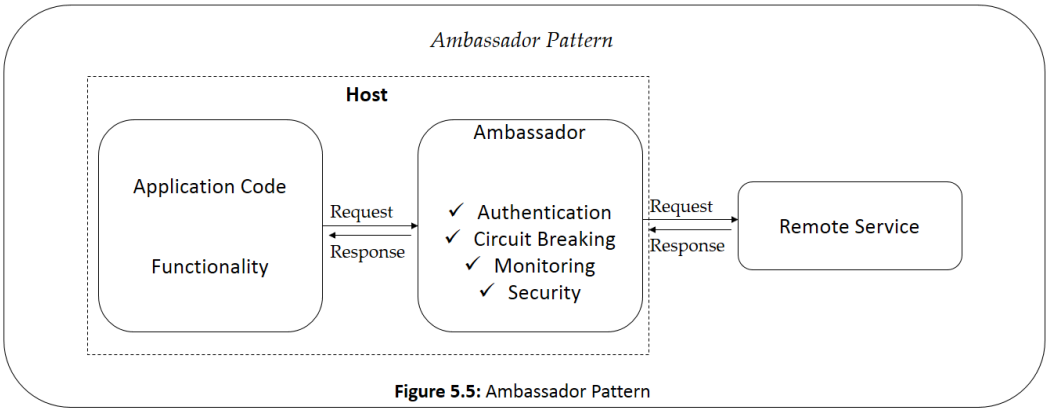


Figure 5.5: Ambassador Pattern

Disadvantages

- **An increase in complexity**: The ambassador pattern may add an additional layer of complexity to an application, which can make troubleshooting problems and ensuring that all components are functioning properly more challenging.

When to use this pattern

It is necessary to develop a common set of client connectivity features for multiple languages or frameworks. It is necessary to offload cross-cutting client connectivity concerns to infrastructure developers or other specialized teams. Legacy applications or applications that are difficult to modify must support cloud or cluster connectivity requirements.

Anti-corruption layer

An **anti-corruption layer** (**ACL**) is a design pattern that involves isolating an application from changes or differences in external systems or APIs by creating a separate layer in the application. The purpose of the ACL is to "shield" the rest of the application from any changes or inconsistencies in the external system, protecting it from what is known as "technical debt." For example, the data or functionality of most applications depends on other systems. As an example, when a legacy application is migrated to a modern system, it may still require existing legacy resources. New features must be able to access the legacy system. The transition to a modern system is particularly difficult when different features of a larger application are gradually moved.

Using the ACL pattern has the advantage of decoupling the internal and external systems, making it easier to make changes to either system without adversely affecting the other. The ACL can also serve as a tool for protecting the internal system from technical debt by isolating it from external problems and inconsistencies.

For example, as illustrated in *Figure 5.6: Anti-Corruption Layer Pattern:*

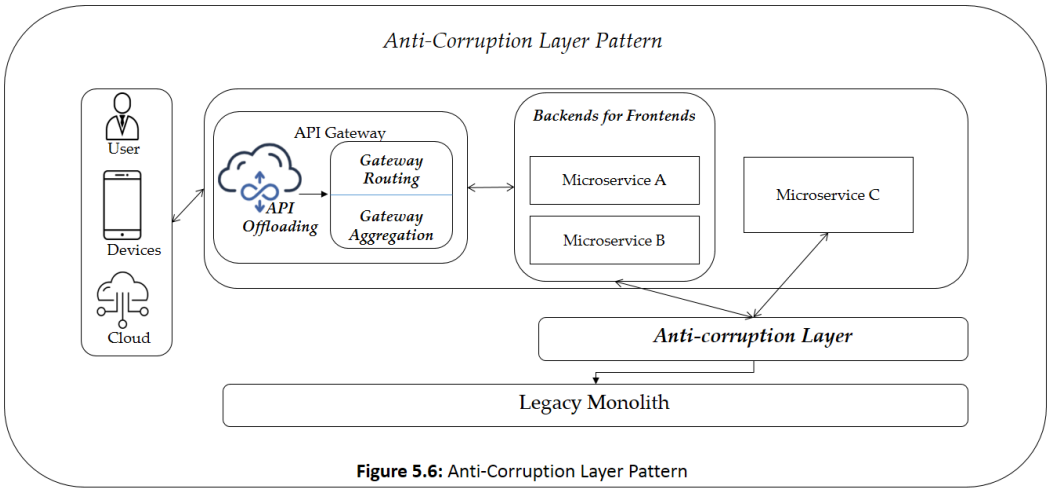


Figure 5.6: Anti-Corruption Layer Pattern

Advantages

- As a result of the ACL, the internal system is more resistant to changes or inconsistencies in the external system, making the internal system more resilient to changes or problems in the external system.
- An ACL provides enhanced flexibility by decoupling internal and external systems, making it easier to change or update either without affecting the other.

Disadvantages

- Anti-corruption layers may add latency to calls between the two systems, as well as adding an additional service that must be managed.
- ACL patterns can increase the complexity of an application, making it more difficult to troubleshoot problems and ensure that all components are working together properly.

When to use this pattern

This patterns works well when there is a plan to migrate to a new system in multiple stages, however integration between the legacy system and the new system must be maintained. In the case it's planned as an application migration strategy, consider whether the anti-corruption layer will be permanent or will be retired after all legacy functionality has been migrated.

Backends for Frontends

In the **Backends for Frontends** (**BFF**) pattern, backends are created separately for each frontend client instead of serving all clients from a single backend. In addition to web applications and mobile applications, this approach can be beneficial when building applications that have multiple frontend clients.

Usually, in the beginning, the application may be designed to serve a desktop web UI, and a backend service is developed to provide those features. Later, mobile applications were created interacting with the same backend. Backend services become general-purpose, serving both desktop and mobile users. Due to very different requirements on each device, this common backend becomes problematic.

Having a separate backend optimized for mobile clients can improve the overall user experience for mobile applications, which may have different performance and network connectivity requirements than a web application. It is also possible to decouple the frontend and backend with the BFF pattern, so that one can be changed without affecting the other. As each microservice is able to address a specific set of concerns, it is especially useful when building applications using a microservice architecture.

Advantages

- It is possible to improve the overall user experience by creating separate backends for each frontend client. By doing so, you are able to optimize the backend for the specific performance and network connectivity requirements of each client.

- As a result of the BFF pattern, each front-end client is able to have its own set of features and capabilities, rather than being constrained by a single backend for all clients. This allows applications with multiple front-end clients to be developed and maintained more easily.
- In applications that use microservices architectures, the BFF pattern can assist in decoupling the frontend and backend, making it easier to make changes to one without affecting the other.
- You can create more granular security controls by creating separate backends for each frontend client, improving the security of each client.

Disadvantages

- There is an increase in complexity when creating and maintaining separate backends for each frontend client as opposed to using a single backend for all clients.
- Implementing this pattern will likely result in code duplication across services.

For example, as illustrated in *Figure 5.7: Backends for Frontends Pattern:*

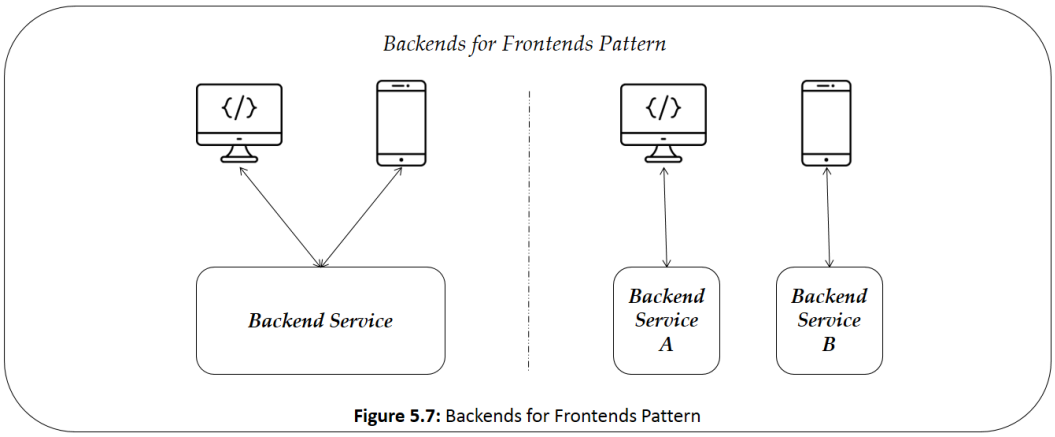


Figure 5.7: Backends for Frontends Pattern

When to use this pattern

Client-specific logic and behaviour should only be included in frontend-focused backend services. General business logic and other global features should be handled elsewhere in your application.

Leader election

There should be only one task instance elected to act as the leader, and this instance should coordinate the actions of other subordinate task instances. Each instance of the task instance can act as the leader if it is running the same code. Consequently, the election process must be carefully managed to prevent two or more instances from simultaneously taking on the leadership role.

A robust mechanism for selecting the leader is essential in the system. In many solutions, the subordinate task instances monitor the leader through a heartbeat method, or by polling. This method has to handle events such as network outages and process failures. It is necessary for the subordinate task instances to elect a new leader if the designated leader terminates unexpectedly, or if there is a network malfunction that renders the leader unavailable to them.

In a distributed system, there are several algorithms for electing a leader, such as:

- **Bully algorithm**: As part of this algorithm, each node announces its leadership claim to all of the other nodes. If a node receives a leadership claim from another node with a higher identifier, it backs down and allows the other node to assume leadership. When a node receives a leadership claim from a lower identifier node, it rejects that claim and asserts its own leadership. This continues until a leader is selected.
- **Ring algorithm**: It consists of a ring of nodes, each having its own unique identifier. Each node sends its identifier to its neighbors repeatedly, and the node with the highest identifier becomes the leader.
- **Paxos algorithm**: It involves multiple rounds of voting and message passing in order to ensure consensus among a group of nodes.

For example, as illustrated in *Figure 5.8: Leader Election Pattern*:

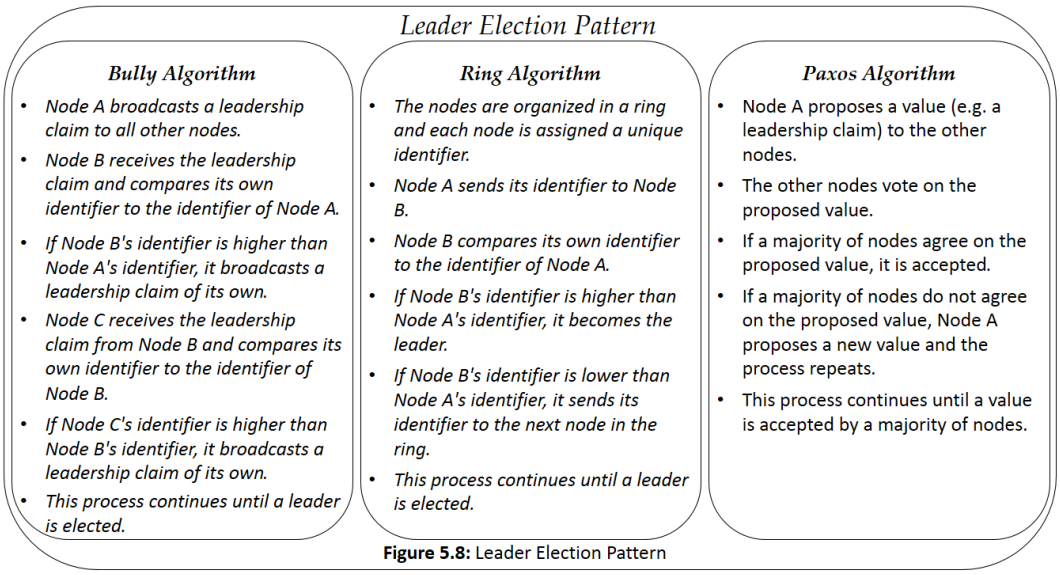


Figure 5.8: Leader Election Pattern

Advantages

- The leader provides a single point of coordination for the other nodes, making it easier to coordinate actions and make decisions. The leader can continue to function even if some nodes fail, as long as the majority of nodes remain online.

Disadvantages

- As the leader coordinates the actions of all the nodes in a large distributed system, it may become a bottleneck. It is possible for some leader election algorithms, such as Paxos, to be complicated to implement. This may require a considerable amount of message passing and voting, which may adversely affect performance.

When to use this pattern

By making decisions on behalf of the group and coordinating state updates, a leader can ensure that all nodes have a consistent view of the system state. Having decision-making centralized in a single node allows the system to potentially make decisions more quickly since it does not have to wait for input from all nodes.

As long as the majority of nodes are available, a leader can continue to function even if some nodes fail. This can improve the overall robustness and availability of the system.

Messaging design patterns

To maximize scalability, cloud applications require a loosely coupled messaging infrastructure connecting components and services. A messaging design pattern is a set of guidelines which can be used to help design and implement an effective messaging system. These patterns can be applied to a variety of messaging scenarios, including communication between microservices and communication between different systems. We have already discussed Asynchronous Messaging Pattern in last chapter (*refer Figure 4.14 for the flow*). Let us review other key messaging design patterns and how we can implement them for our scenario.

Pipes and filters

A message design pattern such as pipes and filters divides a larger process into smaller, independent steps that can be executed either simultaneously or sequentially. The data being processed flows through a series of pipes connecting the filters, and each step is represented by a filter. In the context of systems that need to process large

amounts of data in a complex manner, this pattern allows the data to be processed in smaller chunks, thus improving performance and scalability.

Pipes and filters can be implemented using a messaging system to pass data between the filters. Each filter can process the data and pass it to the next filter in the pipeline, or it can send the processed data to a separate location. One or more filters may be a bottleneck, especially if a large number of requests appear in a stream from one data source. The time it takes to process a single request is determined by the speed of the slowest filter in the pipeline. As a key advantage of the pipeline structure, parallel instances of slow filters can be run, thus increasing the throughput of the system by spreading the load.

Consider, for example, the processing of large volumes of log data. In order to analyse the log data, several filters could be used to parse it, filter out irrelevant entries, and aggregate it. By implementing each filter as its own microservice or process, the data can be passed between the filters using the messaging system. Filters in a pipeline can be located on various machines, allowing them to be scaled independently as well as utilizing the elasticity of many cloud environments. Filters with high computational requirements can be hosted on high-performance hardware, while filters that are less demanding can be hosted on commodity hardware that is less expensive.

For example, as illustrated in *Figure 5.9: Pipes and Filters Pattern:*

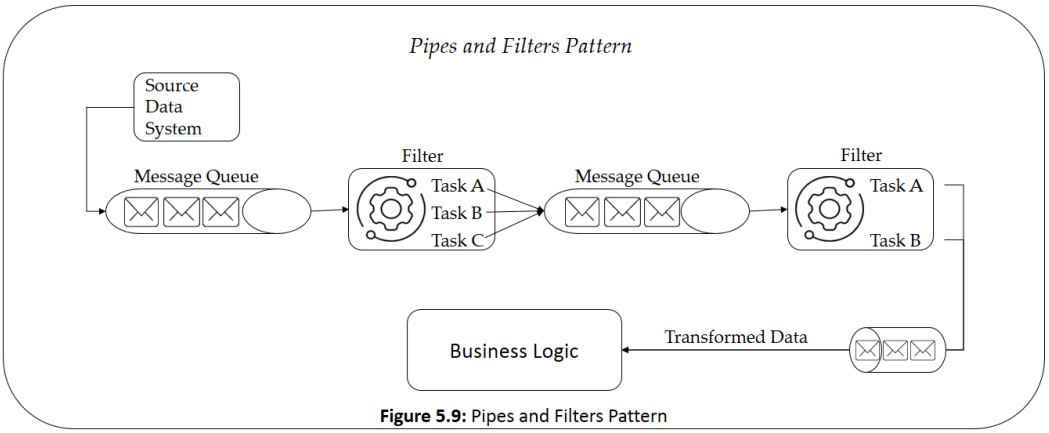


Figure 5.9: Pipes and Filters Pattern

Advantages

- The pipes and filters pattern provides a modular approach to dividing a larger process into smaller, independent steps. This helps to simplify the process of understanding and maintaining the system. Each filter can be implemented and tested separately, reducing complexity.

- Due to the fact that the filters can be executed in parallel, the pipes and filters pattern can improve the scalability of a system. By adding additional filters or increasing the number of instances of each filter, data can be processed more quickly and larger volumes can be handled.

Disadvantages

- This pattern provides increased flexibility, but it can also introduce complexity, especially if the filters are distributed among different servers.
- There will be repetition of messages if a filter in a pipeline fails after posting a message to the next stage. This will then run again, posting a duplicate message to the pipeline. In order to prevent this, the pipeline should detect and eliminate duplicate messages in order to avoid two instances of the same message being passed to the next filter.

When to use this pattern

An application's processing can easily be separated into a number of steps. Each of these steps needs to be scalable in a different way.

Priority queue

Queues usually follow a **first-in, first-out** (**FIFO**) structure, and consumers typically receive messages in the same order in which they are posted. However, priority queues are data structures that store a collection of items with a corresponding priority. This allows you to efficiently retrieve and remove items with the highest priority, as well as insert new items into the queue.

In the context of messaging design patterns, a priority queue can be used to prioritize the handling of messages. For example, a messaging system might have a priority queue for handling urgent messages, such as emergency alerts or critical system notifications. Priority queue messages are processed in advance of non-urgent messages, thus ensuring that critical information is delivered as soon as possible.

For example, as illustrated in *Figure 5.10: Priority Queue Pattern*:

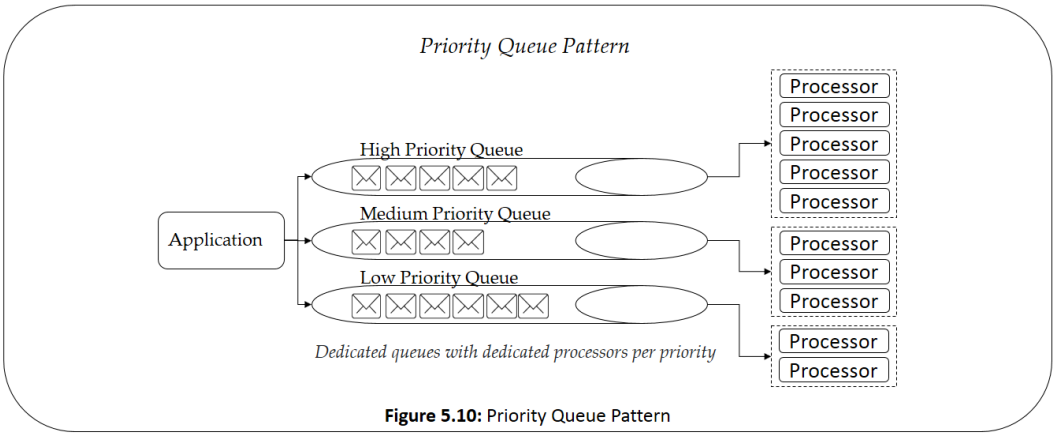


Figure 5.10: Priority Queue Pattern

Advantages

- By prioritizing certain messages, you can increase the efficiency of your system by ensuring that they are processed as quickly as possible.
- In addition, you will be able to ensure that resources are allocated more efficiently as a result of prioritizing certain messages, which will in turn reduce the overall operational costs of your system.

Disadvantages

- Identifying priorities can be challenging, and you may need to invest significant time and effort in this process.
- It may be possible for certain messages to be unfairly prioritized over others if the priority of a message is not properly set. This may lead to customer dissatisfaction.

When to use this pattern

There are many tasks that need to be handled by the system, each with a different priority. Different users or processes are required to be managed as per priority.

Publisher-subscriber

In this pattern, the sender packages events into messages, using a known message format, and sends these messages via the input channel. As a result, it increases scalability and improves sender responsiveness. It is the responsibility of the

messaging infrastructure to ensure messages are delivered to interested subscribers. Senders can send a single message to the input channel, then return to their core processing responsibilities.

- There is one output messaging channel per subscriber.
- Message brokers or event buses are mechanisms that copy messages from the input channel to the output channel for all subscribers interested in the message.

As the publisher and subscribers are not required to know about each other's implementation details, this pattern can help decouple components of an application. As new subscribers can easily be added or removed from the list without affecting the publisher or other subscribers, it provides flexibility and modularity.

For example, as illustrated in *Figure 5.11: Publisher-Subscriber Pattern:*

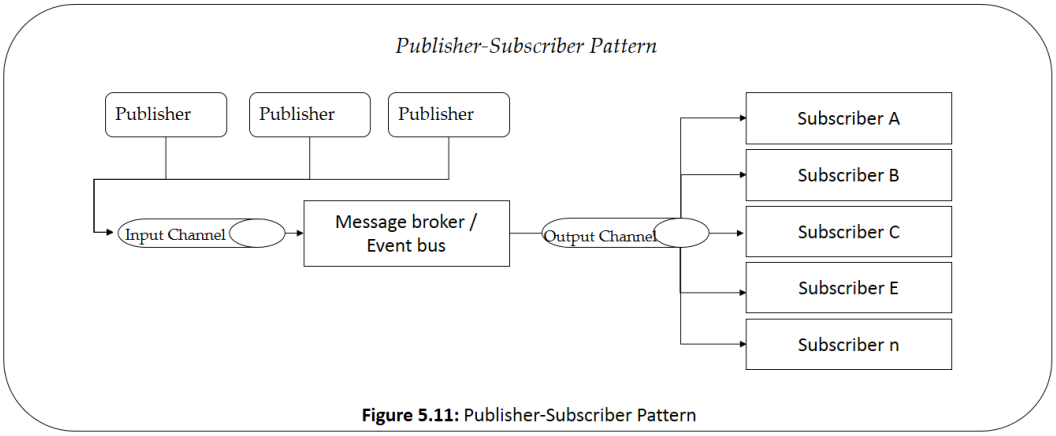


Figure 5.11: Publisher-Subscriber Pattern

Advantages

- Messages can be effectively managed even if one or more receivers are offline as it decouples subsystems that need to communicate.
- In addition to improving reliability, asynchronous messaging can assist applications in running smoothly even when there is an increase in load and can handle intermittent failures more effectively as well.
- Message routing provides separation of concerns for your applications. Each application can focus on its core capabilities, while the messaging infrastructure handles everything required to route messages reliably to multiple destinations.

Disadvantages

- Communication in a publish-subscribe system is considered unidirectional. If a specific subscriber needs to communicate status back to the publisher, use the Request/Reply Pattern. In this pattern, one channel is used to communicate with the subscriber and another channel is used to communicate with the publisher via a response channel.
- The order in which consumer instances receive messages is not guaranteed, and is not necessarily reflective of the order in which messages were created.
- As some solutions require that messages are processed in a certain order, the Priority Queue pattern can be used to ensure that certain messages are delivered before others.

When to use this pattern

You can use the publisher-subscriber pattern when you want to decouple components of an application in order to make them more modular and easier to maintain.

Queue-based load levelling

As the queue functions as a buffer that absorbs excess traffic and prevents the system from becoming overloaded, this pattern is useful for mitigating the impact of sudden spikes in workload. As well as improving overall system performance, it also provides a more predictable and controlled method for processing requests.

Due to the queue decoupling the tasks from the service, the service is able to handle the messages at its own pace, regardless of the volume of requests from concurrent tasks. Additionally, there will be no delay to the task if the service is unavailable at the time the message is posted.

Advantages

- By buffering incoming requests in a queue, the system can process them at a more consistent rate, which may improve overall performance.
- Load leveling reduces resource usage by smoothing out spikes in workload, thus reducing the impact on CPU and memory.

For example, as illustrated in *Figure 5.12: Queue-Based Load Levelling Pattern:*

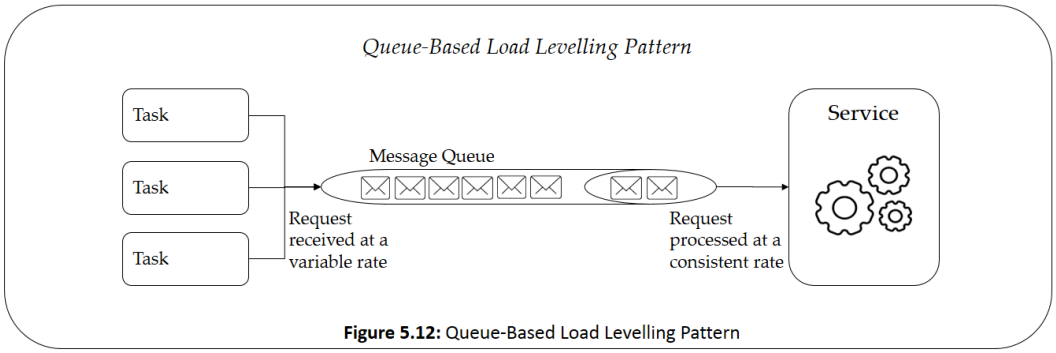


Figure 5.12: Queue-Based Load Levelling Pattern

Disadvantages

- A queue can add additional latency to the system, as requests may be delayed while waiting to be processed.

When to use this pattern

The pattern can be applied to any application that uses services that are subject to overloading.

Sequential convoy

Sequential convoys refer to a messaging pattern in which a series of messages are sent sequentially from one node to another in a network. It is used to ensure that a message is delivered to the intended recipient and that the message is received in the correct order. The application must be able to process messages in sequential order in order to handle increased load while also being able to scale out. This pattern is useful for ensuring that a set of related tasks are executed in the correct order and that the output of one task is available to the next task.

Advantages

- This pattern ensures that tasks are executed in the correct order: Sequential Convoy ensures that tasks are executed in the order in which they are defined, which can be useful if data integrity needs to be maintained or if certain tasks need to be completed ahead of others.
- By breaking the code down into smaller, more manageable tasks, the Sequential Convoy pattern can simplify code.

Disadvantages

- If the convoy contains computationally intensive tasks, it may be slower than other approaches that allow tasks to be executed simultaneously. Scalability is limited by the FIFO requirement for extreme throughput scenarios (millions of messages per minute or second).

For example, as illustrated in *Figure 5.13: Sequential Convoy Pattern*:

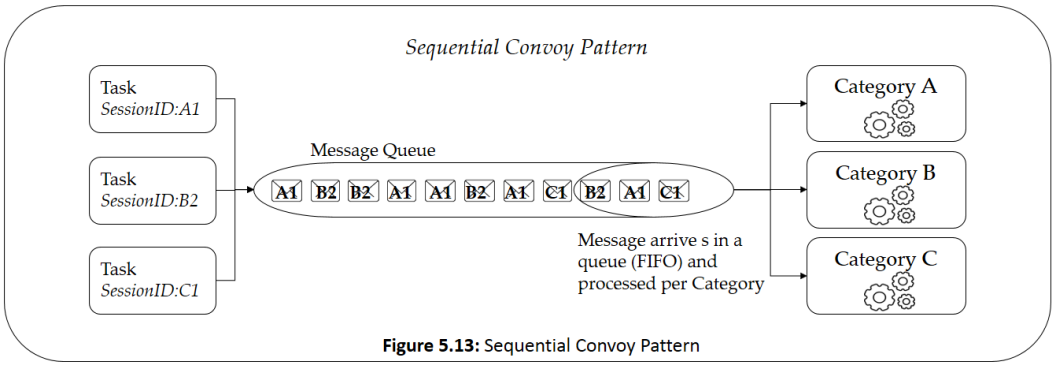


Figure 5.13: Sequential Convoy Pattern

When to use this pattern

If you have messages arriving in order, you should process them in the same order as well. Arriving messages may be "categorized" in such a way that the category becomes the unit of scale for the system as a whole.

Reliability

We have already discussed some of the reliability patterns that are specifically designed for Cloud Native Microservices architects in a previous chapter, such as Bulkheads and Circuit Breakers. Let us review some other key reliability design patterns in this section. Essentially, these patterns are all intended to improve the reliability and robustness of a system. They can be deployed individually or in combination to address a variety of types of failures or challenges that a system may encounter as it matures over time.

Compensating transaction

The compensating transaction is a transaction that undoes the effects of a previous transaction. It ensures that the overall system remains in a consistent state, despite an error in a previous transaction. Cloud-based applications modify data, which may be spread across a variety of data sources located in different geographical locations. Rather than attempting to provide strong transactional consistency to avoid contention and improve performance in a distributed environment, applications should implement eventual consistency instead.

Typically, a business operation consists of several separate steps as described in this model. It is possible that the overall view of the system state will be inconsistent while these steps are being performed. However, once the steps have been executed and the operation is complete, the system should become consistent again. It is common for compensating transactions to be used in conjunction with other reliability patterns, such as retries and timeouts, to handle temporary failures or to enhance the reliability of operations. As well as rolling back changes made by a transaction, they may also be used if the transaction fails or is cancelled by the user.

The system must be able to undo the effects of a compensating transaction in case it is necessary to track the system's state prior to and after each transaction in order to implement a compensating transaction. A database or other persistent storage mechanism can be used to store the system state.

Advantages

- **This ensures that the entire system stays in a consistent state**: Should a transaction fail, a compensating transaction can be used in order to restore the system to its previous state by undoing the effects of that transaction. As a result, data corruption or other problems resulting from incomplete or failed transactions can be prevented.

Disadvantages

- **It can be complex to implement**: Implementing a compensating transaction requires tracking the state of the system before and after each transaction, as well as the capability of undoing the effects of each transaction. It is not an easy process to accomplish, particularly if the system consists of many different transactions or is dependent on a large number of other transactions.

For example, as illustrated in *Figure 5.14: Compensating Transaction Pattern*:

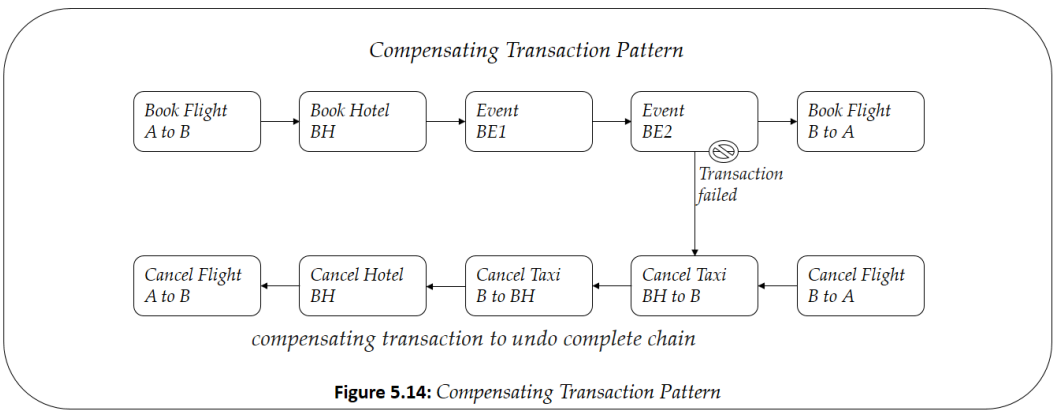


Figure 5.14: *Compensating Transaction Pattern*

When to use this pattern

This pattern should only be used if there is a need to undo operations if they fail. If possible, design solutions to avoid the complexity of requiring compensating operations wherever possible.

Deployment stamps

A Deployment Stamp pattern requires the creation of a stamp that is associated with a specific version of a component whenever that component is deployed. The stamp can then be used to track the progress of the deployment, as well as to identify which version of the component is currently running in the system.

You may have several stamps that represent a subset of your customers within a single geographical region, or you may have multiple stamps to support horizontal scaling out within that region. Stamps operate independently of one another and can be deployed and updated independently.

- In addition, it may be necessary for you to keep certain customers' data separate from that of other customers. Similarly, you may have some customers that require a greater amount of system resources than others, and you may wish to separate them on different sets of infrastructure.
- Depending on the customer, you may have some who are tolerant of frequent updates to your system, while others may be risk-averse and would prefer infrequent updates to the system that serves their needs. Deploying these customers in isolated environments may make sense.

For example, as illustrated in *Figure 5.15: Deployment Stamps Pattern*. Stamp 1, 2, and 3 can be on different version (release). Tenant C for example needs dedicated resources.

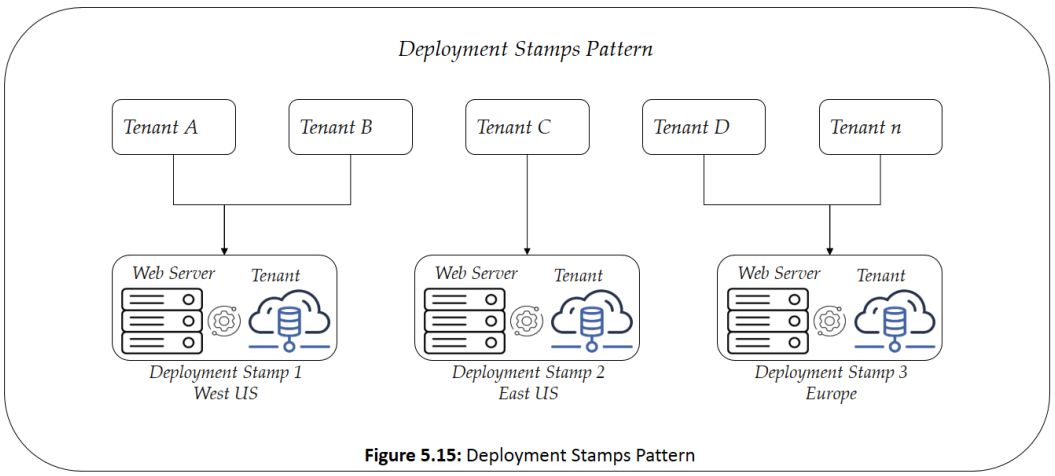


Figure 5.15: Deployment Stamps Pattern

Advantages

- A deployment stamp pattern enables the creation of unique identifiers that are associated with specific versions of software components, thus improving deployment tracking. In this way, deployment progress can be tracked easily and version versions of components can be identified in the system.
- It is easier to troubleshoot issues and maintain system reliability with stamps that track deployment progress and identify the version of a component that is currently being used.

Disadvantages

- It may be necessary to add additional infrastructure and complexity to the system as a result of implementing the deployment stamps pattern.
- To ensure that the deployment stamps accurately reflect the current state of the system, the deployment stamps pattern requires ongoing maintenance.

When to use this pattern

It is necessary to track the progress of software deployments and identify any issues that may arise during the deployment process when performing rolling updates in a cloud environment.

Geodes

We can enhance Deployment Stamp pattern to address multi-location near-edge compute using the Geode pattern. A Geodes pattern is a collection of backend services are deployed to a set of geographical nodes, and each of these nodes is capable of handling any request made by any client in any geographical region. By distributing request processing around the globe, this pattern enables serving requests in a style that is active-active, improving latency and increasing the availability of the service.

Getting data closer to the user is crucial for availability and performance. In today's modern cloud infrastructure, load balancing of front-end services and back-end services can be done geographically as well as replication of back-end services. When data is geo-distributed across a far-flung user base, the geo-distributed datastores should also be collocated with the compute resources that process the data. The geode pattern enables the compute to be near the data.

Advantages

- As a result of this design pattern, everything is implicitly decoupled, leading to a system that is highly distributed and decoupled.

Disadvantages

- For clustering in situations where there is no need for geographical distribution, consider availability zones and paired regions.

When to use this pattern

The objective is to implement a high-scale platform with users spread across a large geographical area.

Throttling

The throttling pattern is a design pattern that is used to manage the flow of requests to a system in order to prevent the system from being overloaded. Typically, the load on cloud applications varies based on the number of active users or the type of activity performed by users. During business hours, for example, the system may be required to perform computationally expensive analyses, or users may be more active. In addition, there may be sudden and unanticipated bursts of activity. This pattern is commonly used in situations where a high volume of requests is expected and we may not be able to process them all simultaneously.

Various strategies are available to handle varying loads in the cloud, depending on the business objectives of the application. Autoscaling can be used as a strategy for matching provisioned resources to user needs at any given time. Despite the fact that autoscaling can trigger the provisioning of additional resources, this provisioning does not occur immediately.

As part of the throttling pattern, mechanisms are implemented to limit the rate at which requests are processed. By setting limits on the number of requests that can be processed per unit of time (for example per second, per minute) or by setting limits on the number of requests that can be processed concurrently, this can be accomplished.

There are several throttling strategies that can be implemented by the system, including:

- Requests from an individual user who has already accessed system APIs more than n times per second over a specified period of time are rejected.
- By disabling or reducing the functionality of selected non-essential services, essential services can operate without interruption as long as sufficient resources are available.
- Refusing to perform operations on behalf of applications or tenants with a lower priority.

Advantages

- By limiting the rate at which requests are processed, the Throttling pattern can help to prevent the system from becoming overwhelmed and ensure that it remains stable and reliable.
- By limiting the rate at which requests are processed, the Throttling pattern can help to ensure that resources are used efficiently. This will ensure that the system is able to take advantage of the available resources to the fullest extent possible.

For example, as illustrated in *Figure 5.16: Throttling Pattern*:

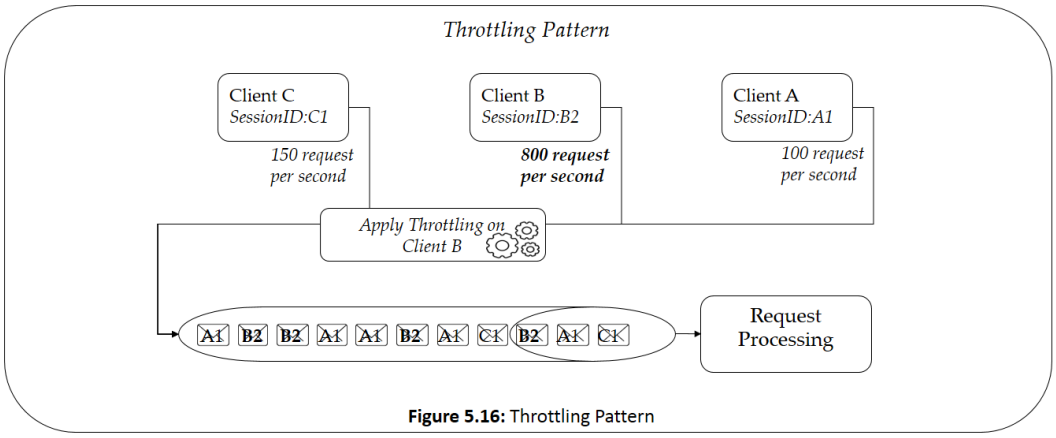


Figure 5.16: Throttling Pattern

Disadvantages

- Using the Throttling pattern, requests are processed at a slower rate, resulting in a reduction in overall system throughput. The Throttling pattern can increase latency, which can adversely affect the system's performance, by limiting the rate at which requests are processed.

When to use this pattern

In order to ensure that resources provided by an application are not monopolized by a single tenant. For handling bursts of activity.

Conclusion

Seamless Integration of Cloud and Microservices:

Accelerating Business Potential with Cloud Native Architectures

"Cloud-Powered Microservices" is a term that refers to microservices-based systems that are built and deployed on the cloud and that take advantage of Cloud Native functionality. As a software architecture style, microservices involve constructing one application as a set of small, independent services that communicate with one another via well-defined interfaces, often using lightweight messaging protocols. Each service is responsible for a specific business capability and can be developed, deployed, and scaled independently. We have discussed cloud native data management, reliability, design and implementation, and messaging patterns in detail.

In our next chapter, 'Monolith to Microservices Case Study' we will re-design an application from a legacy monolith to microservices based design using patterns and architect concepts described in the preceding five chapters.

CHAPTER 6
Monolith to Microservices Case Study

From Monolith to Microservices: A Practitioner's Journey

Introduction

In application modernization, a monolithic architecture is replaced with a microservice architecture. This is so that an existing application can be updated and improved to be used more effectively in a modern computing environment. In our previous chapters, we discussed the relationship between Microservices and cloud computing, since microservices are often deployed on the cloud using the Cloud native design pattern. By combining microservices and cloud native, enterprises are able to develop, deploy, and manage highly scalable, flexible, and hybrid cloud applications. Containers, service meshes, microservices, DevOps and **continuous integration/continuous deployment** (**CI/CD**) development practices, and declarative APIs are used to accomplish this. We have already discussed below listed the key steps, and now in this chapter we will examine the implementation approach from the practitioner's point of view using a case study.

Determine which monolithic components can be decomposed into microservices: When identifying components to be separated into microservices, consider the degree of independence and separation of concerns they exhibit. Components that are deployable independently, have clear boundaries, and are relatively self-contained are ideal candidates for microservices.

Break down the monolith into smaller components, extracting them into separate microservices: This step involves refactoring the code to extract the identified components into separate microservices, as well as making necessary changes to the build and deployment processes. Additionally, it require the design and implementation of APIs that will allow microservices to communicate with each other.

Develop and implement RESTful APIs for communication between microservices: Use a tool such as OpenAPI or Swagger to develop and implement RESTful APIs. When designing APIs, ensure that security, versioning, and documentation are taken into account.

Create continuous integration and delivery (CI/CD) pipelines for each microservice: CI/CD pipelines enable microservices to be built, tested, and deployed independently, improving speed and reliability.

The microservices should be deployed to a cloud platform or container orchestration system: Cloud platforms and container orchestration systems such as Kubernetes facilitate the deployment and management of microservices at scale.

Monitoring and optimizing the performance of the microservices: Use monitoring and logging tools to monitor the performance and reliability of the microservices, and make any necessary modifications to make them more scalable and reliable.

Continually iterate and improve the microservices: As the microservices evolve, you need to split larger ones into smaller ones, or merge smaller ones together as necessary. In order to ensure that your organization's architecture meets its needs, continually review and optimize it.

Structure

In this chapter we will discuss following topics:

- Transitioning from Monolith to Microservices Architecture
- Monolithic to Microservice Design Principle
- Challenges of Legacy Systems
- Strategies for Updating Legacy Systems to Microservices
- Migrating TravelGuru Application to Microservices: A Case Study
    - **Case Study:** Business Challenge
    - **Case Study:** Solution Delivered for Microservices Migration
    - **Case Study:** Technology Roadmap for Microservices Adoption
    - **Case Study:** Application Transition to Microservices Architecture

  - **Case Study:** Successful Database Migration to Microservices
  - **Case Study:** Business Outcome of Microservices Migration
  - **Case Study:** Best Practices Implemented in Microservices Migration
- Conclusion

Objectives

Our objective is to leverage Cloud Native Microservices to design or migrate a system that provides several key benefits. This includes faster time to market for new services and applications, thanks to an automated and comprehensive lifecycle management process. We aim to achieve a shorter time to market for new services and applications, and decouple them from the underlying infrastructure, making application development much easier. By leveraging microservices architecture and **continuous integration/continuous delivery** (**CI/CD**), we expect to see a significant increase in the cadence of small and regular updates to applications.

Using containers and microservices, we aim to give users the ability to deploy only the necessary components, rather than deploying entire monolithic network functions, which will help reduce the **total cost of ownership** (**TCO**). In our case study, we will evaluate a proposed technical stack and approach to modernize a travel website, with the goal of delivering a highly distributed and scalable solution that can adapt to changing customer needs and preferences.

Transitioning from monolith to microservices architecture

Remember how we started this book? In our first chapter, we discussed the importance of adopting cloud-native microservices for successful digital transformation, and the key drivers for cloud-native microservices adoption. In this chapter, we will apply these principles to our case study on modernizing the TravelGuru website/application.

Moving from a monolithic architecture to a microservices architecture involves breaking down a large, monolithic application into smaller, independently deployable components. By doing so, we can achieve faster time-to-market for new services and applications, improved lifecycle management, and greater application scalability and availability. The aim of transitioning from a monolithic architecture to a cloud microservices architecture is to modernize and improve the application, making it more suitable for modern computing environments. Depending on the organization's specific needs and constraints, there will be a different strategy for achieving this goal.

One of the key advantages of microservices is that they enable a greater degree of flexibility in application development and deployment. By decoupling individual services from the underlying infrastructure, we can develop and deploy applications that are highly distributed and easier to manage. This approach also allows us to use **continuous integration/continuous delivery** (**CI/CD**) to streamline the application update process. With the use of containers and microservices, users are able to deploy only what they need, rather than having to deploy entire monolithic network functions, thereby reducing the TCO.

In our case study, we will explore the steps involved in transitioning from a monolithic to a microservices architecture for the TravelGuru application. We will examine the challenges faced by the team and the approach taken to modernize the system. We will also discuss the benefits achieved and provide guidance on best practices for deploying microservices in a cloud environment.

Monolithic to microservice design principle

When transitioning from a monolithic architecture to a microservices architecture, there are several design principles that should be implemented to ensure a successful transition.

- **Single Responsibility Principle**: Each microservice should have a single, well-defined responsibility, and should be loosely coupled with other services.
- **Decentralized Data Management**: Each microservice should have its own data store, and data should be shared between services through APIs or message queues.
- **High Cohesion**: Each microservice should have high cohesion, meaning that all of its components and functionality should be related to its defined responsibility.
- **Loose Coupling**: Microservices should be loosely coupled, meaning that changes to one service should not affect the functionality of other services.
- **Automatic Scaling**: Microservices should be designed to automatically scale up or down based on demand.
- **Self-Containment**: Each microservice should be self-contained and should not rely on the state of other services.
- **Failure Isolation**: Microservices should be designed to fail independently, so that a failure in one service does not affect the functionality of other services.
- **Composability**: Microservices should be composable, meaning that they can be combined to create new functionality.

- **Stateless**: Microservices should be stateless, meaning that they do not maintain any client-specific state.
- **Asynchronous**: Microservices should be designed to communicate asynchronously, to minimize latency and improve scalability.
- **Automated Deployment**: Microservices should be designed for automated deployment, making it easy to deploy and update individual services.
- **Monitoring and logging**: Microservices should be designed with monitoring and logging capabilities to enable debugging and troubleshooting.
- **Service Discovery**: Microservices should be designed with service discovery capability to enable service-to-service communication.
- **API-driven**: Microservices should be designed with a well-defined API to enable easy integration with other services.
- **Event-driven architecture**: Microservices should be designed to use event-driven architecture to enable loosely-coupled communication between services.
- **Service observability**: Microservices should be designed for observability, meaning that the service should be able to provide metrics, tracing, and logging information to enable monitoring and troubleshooting.

Challenges of legacy systems

It is critical to understand a legacy system and challenges with the same for better planning it is modernisation journey. A legacy system is a computer system, application, or technology that has been in use for a long period of time. It is no longer actively being developed or supported by the vendor. They are often based on obsolete technologies and not be compatible with new systems or software.

As an example, a bank that still uses an old mainframe system to process transactions, that system is old and runs on proprietary software that no longer is maintained by the vendor, this system is expensive to maintain, it is difficult to integrate with new software and hardware, and it not be able to handle new data types or communications. Working with existing legacy code can be challenging, as it be difficult to understand the code and how it works, especially when the original developers are no longer available to assist. It also be difficult to understand the functionality of the system or how to make changes due to inadequate documentation of the code.

Moreover, existing legacy code be difficult to maintain, as it not be compatible with newer technologies and software, or it be vulnerable to security vulnerabilities. An organization need to weigh the costs and benefits of updating or maintaining legacy code before upgrading or maintaining it, as updating or modernizing legacy

code can be a time-consuming and costly process. The decision to rewrite legacy code be made by some organizations, but this can also be a very time-consuming and complex process, especially if the legacy code is large and complex. The key challenges of legacy systems are:

- **Technical debt:** The maintenance and updating of legacy systems can often result in significant technical debt, which refers to the expense of maintaining and updating an older system. Organizations find this to be a significant burden, making it difficult to implement new features or modify the system. In addition, as technology changes, the system becomes less and less compatible and more difficult to maintain and upgrade as it becomes older. As a result, the problem becomes more complex as the system becomes older.
- **Data migration:** It is necessary for organizations to consider how they will migrate data from an old system to a new one when updating or replacing a legacy system. During the migration process, organizations need to ensure data integrity and consistency, since this is a complex and time-consuming process. The data migration process can also be risky, as data be lost or corrupted during the migration, or the new system not be capable of handling the data from the old system.
- **System integration:** As legacy systems not be compatible with newer systems and software, it be difficult to integrate them into the existing technology landscape of an organization. For organizations requiring data or processes to be shared between systems, this can pose a significant challenge. Integrating a legacy system with new software and hardware requires custom development and testing, both of which can be time-consuming and costly.
- **User training:** Users also be resistant to change and be unwilling to learn new systems or processes when updating or replacing a legacy system. This can be a challenge for organizations with varying levels of technical expertise.
- **Security concerns:** It is possible that legacy systems lack security features that are present in newer systems, leaving organizations vulnerable to cyber-attacks. These systems not be able to encrypt data, authenticate users, or detect and prevent intrusion, making sensitive data vulnerable to cyber-attacks.
- **Vendor support:** It be difficult to obtain updates and technical support for legacy systems, since the vendor no longer support them. Organizations need to rely on third parties for support, which can be costly and not be reliable.
- **System upgrades and maintenance costs:** It is often necessary for organizations to weigh the costs and benefits of upgrading or maintaining

legacy systems because it can be a costly process. In addition to the cost of upgrading or replacing the system, organizations should also consider the benefits of a new system, including enhanced efficiency and security, as well as the cost of maintaining the existing system. It be more cost-effective to maintain the legacy system rather than upgrade it in some cases.

Strategies for updating legacy systems to microservices

Legacy systems can be modernized in several ways to improve efficiency, security, scalability, and efficiency. There are several approaches an organization can take when modernizing legacy systems:

- **Gradual Modernization:** One approach is to gradually modernize the legacy system by updating one component or module at a time. This approach allows organizations to keep the existing system running while gradually migrating to a new system. This approach can be time-consuming but reduces the risk of downtime and allows the organization to test each component thoroughly. For example, the travel website could update one component or module at a time. For example, they could start by migrating the hotel booking component to a microservices architecture. This would allow them to keep the existing system running while gradually migrating to a new system. Once the hotel booking component is migrated, they could move on to other components like flight booking or car rental.
- **Re-platforming:** This involves moving the legacy system to a new operating system or hardware platform, in order to improve performance and scalability. A cloud-based system can also be used to move the system, which offers additional benefits such as flexibility and cost savings in addition to increased flexibility. For systems that are running outdated hardware or operating systems, or that must be scaled up or down to meet changing demands, re-platforming is a good option. It is important to note, however, that this approach can be complex and time-consuming, since it requires significant changes to the system's infrastructure and require extensive testing and validation. As an example, the travel website could move their legacy system to a new operating system or hardware platform to improve performance and scalability. For example, they could move their system to a cloud-based platform, which would offer benefits like flexibility and cost savings in addition to increased scalability. This would help them meet changing demands and keep up with the growing traffic on their website.
- **Re-hosting:** During this process, the system is moved to a new hosting environment, such as a virtual machine or container, in order to improve

performance. The system can be scaled up or down to meet changing demands if it runs on outdated hardware or needs to be scaled up or down. However, this approach can also be complex, as it requires significant changes to the system's infrastructure and require extensive testing and validation. As an example, the travel website could move their legacy system to a new hosting environment, such as a virtual machine or container, to improve performance. For example, they could move their flight booking component to a container-based environment, which would allow them to scale the component up or down based on demand. This would help them meet changing demands and ensure that their website runs smoothly.

- **Re-architecting:** It involves making significant changes to the system's architecture, such as breaking up monolithic systems into microservices, in order to enhance scalability, reliability, and maintainability. A system that has become complex and difficult to maintain or scale is often subjected to this approach. It is, however, a complex and time-consuming process and can require significant changes to both the system's code and infrastructure. As an example, the travel website could break up their monolithic system into microservices to enhance scalability, reliability, and maintainability. For example, they could break up their hotel booking component into microservices like search, booking, and payment. This would help them manage the complexity of their system and ensure that each microservice is easily maintainable.

- **Re-coding:** As part of this process, the system's code is rewritten in order to improve performance, scalability, and security, as well as make the system easier to maintain and maintainable. The use of this approach is often recommended for complex systems that have become difficult to maintain or scale. However, this approach can be complex and time-consuming, as well as requiring a significant amount of expertise and resources. As an example, the travel website could rewrite their system's code to improve performance, scalability, and security. For example, they could rewrite their car rental component to make it more efficient and secure. This would help them keep up with the growing traffic on their website and ensure that their users' data is safe.

- **Wrapping:** In order to make the legacy system easier to use and integrate with other systems, it is necessary to create a new user interface or **application programming interface** (**API**). This approach can be a useful option for systems that have important functionality but are difficult to use or integrate with other systems. There is, however, the potential for this approach to be complex and time-consuming, as well as requiring significant changes to the system's infrastructure and code. As an example, the travel website could create a new user interface or API to make their legacy system easier to

use and integrate with other systems. For example, they could create a new API for their destination information component, which would allow other websites to easily integrate with their content. This would help them reach a wider audience and provide more value to their users.

- **Refactoring:** This involves making small changes to the system's code in order to improve performance, scalability, and security, as well as make the system easier to maintain. For systems that are difficult to maintain and scale, but that still have important functionality, this approach is often used. Although this approach is generally less complex and time-consuming than a complete re-write of the code, a significant amount of expertise and resources will still be required. As an example, the travel website could make small changes to their system's code to improve performance, scalability, and security. For example, they could refactor their travel guides component to make it more efficient and maintainable. This would help them manage the complexity of their system and ensure that each component is easily maintainable.

Migrating Travelguru application to microservices: A Case Study

The **Travelguru** company had a monolithic application that had been built over several years and had grown to include many features and functionalities. The monolithic architecture became a bottleneck for the company's growth as the company expanded, however. There was an issue with scaling, deploying, and maintaining the application. As a result, the company decided to move from a monolithic architecture to a microservices architecture in order to achieve better scalability, flexibility, and deployment reliability.

- **The Travelguru application** was a web-based application that provided several services to customers, for their travel needs including flight booking, hotel booking, car rental, vacation packages. In addition, there is support and interactive services like destination information, travel guides and reviews. As a result of partnering with various other platforms over the last year, this company has been able to enhance their user base as well as fine tune their business model in order to become more agile and user-friendly. Now the increased volume creating new positive challenges for business to scale at a much faster pace and to be able to rollout new services on monthly basis.

- **This application was built** using Java and was run on a single server. With the increasing number of customers and transactions, the application was struggling to keep up with the load, resulting in slow performance and frequent downtime. As a consequence of the company realizing that the monolithic architecture was not suitable for their current and future requirements, they changed the architecture.

- **Monolithic technology stack (Travelguru application)**: A monolithic technology stack refers to a software architecture where all the components of an application are tightly integrated and deployed as a single unit. In a monolithic stack, all the modules or services of the application share the same codebase, database, and runtime environment. In this case, below mention technologies were used with a monolithic architecture.
    - **Front-end**: JSP, ASP.NET, HTML and JavaScript (old web technologies) are used to create a dynamic user interface.
    - **Back-end**: The server-side logic was built in Java (older version 6).
    - **Database**: To store and manage customer data, booking information, and other website information, Oracle (version 11) is used.
    - **Search Engine**: The website uses open source search engine technologies to enable fast and efficient searching for flights, hotels, and other travel options.
    - **Web Server**: HTTP requests and responses are handled by Apache on the website.
    - **On-premises Infrastructure**: On-premises infrastructure, using virtual machines, provides scalability, reliability, and cost-effectiveness.
    - **Monolithic Architecture**: This website has a monolithic architecture, with all functionalities combined into a single codebase.

Case Study: Business Challenge

Running a monolithic architecture for a travel booking and planning website using old technologies like Java 6, Oracle 11g DB, Apache, and JSP, ASP.NET created several additional business challenges, such as:

- **Security vulnerabilities:** As these technologies are older, they have known security vulnerabilities that cannot be easily patched or resolved. This can put the website and user data at risk.
- **Lack of support:** As these technologies are no longer actively supported, it is difficult to find developers with expertise in these technologies, and there be limited resources available for troubleshooting and problem-solving.
- **Limited performance:** The older technologies are not able to handle the high demand and load of a travel booking and planning website, leading to poor performance and a poor user experience.
- **Limited ability to integrate with new technologies:** As the technologies are outdated, it be difficult to integrate them with newer technologies and tools that could improve the website's functionality and performance.

- Limited ability to scale: As the system is monolithic and uses outdated technologies, it is not able to handle the scalability required for high demand and experience performance issues.

Case Study: Solution Delivered for Microservices Migration

To overcome these challenges, the company adopted several strategies, including modernizing a monolithic architecture for a travel booking and planning website using cloud-based microservices:

Company '**Travelguru'** want to adopt a phase wise approach in order to minimize any impact on the running business and at the same time modernize to manage growing demand. Steps they followed:

- **Discovery and Planning:** As part of this phase, the existing monolithic application is analyzed to identify the functional components that can be broken down into smaller, independent services in order to facilitate the break down process. For example, the application functionality for flight booking, hotel booking, car rental, and package deals. Each of these functionalities can be broken down into separate microservices. A plan is also developed for re-architecting the application, replacing the old technologies with modern, cloud-native technologies, and selecting a cloud provider that is capable of providing the required services.
- **Proof of Concept (PoC):** A proof of concept is developed to demonstrate the feasibility of migrating to microservices and the new architecture by selecting a small subset of functionality of the existing application, for example flight booking, and using them in a Proof of Concept. For this phase, a small set of services is created and deployed in the cloud. The recommended technology stack for this phase would validate approach for complete migration/ modernization:
    - **Programming Language**: Java 11

      Java 11 is a widely used programming language that is well-suited for building microservices. It has a large developer community and a vast array of libraries and tools available, making it an ideal choice for building microservices.
    - **Framework**: Spring Boot

      Spring Boot is a popular and widely used framework for building microservices. It provides a comprehensive set of tools and features that make it easy to build, deploy, and manage microservices, making it a good choice for a PoC.

- **Database**: PostgreSQL

  PostgreSQL is a reliable and scalable open-source database that is well-suited for use in microservices architecture. It is a popular choice for building web applications, providing good performance and stability.

- **Service Mesh**: Istio

  Istio is a popular service mesh that provides a comprehensive set of tools for managing microservices. It offers features such as traffic management, service discovery, and security, making it a good choice for a PoC.

- **Cloud Provider**: Azure

  Azure is a cloud provider that offers a wide range of services for building and deploying microservices. It provides a scalable and reliable platform for hosting microservices, making it a good choice for a PoC.

- **Deployment and Scaling**: Kubernetes

  Kubernetes is a popular and widely used container orchestration platform that provides a scalable and reliable platform for deploying and managing microservices. It offers features such as automatic scaling, rolling updates, and self-healing, making it an ideal choice for a PoC.

Overall, the selected technology stack provides a reliable, scalable, and comprehensive set of tools for building, deploying, and managing microservices, making it an ideal choice for a PoC for migrating from a monolithic to a microservices architecture.

- **Incremental Migration:** In this phase, the existing application is incrementally migrated to the new microservices architecture. The migration is done in small chunks, and each chunk is tested thoroughly before being moved forward. The first service to be migrated, for instance, is the flight booking service, which is thoroughly tested and deployed to the cloud, and the second service is the hotel booking service. As a result of this approach, the migration is more controlled and less risky, as the system is still operational during this phase of the migration process.
- **Performance Optimization:** As part of this phase, the new microservices-based system is optimized for performance. For example, caching can be implemented to improve the performance of the system. As well as optimizing database queries, it is also possible to implement load balancing to distribute the load between multiple instances of the microservices in order to improve the system's performance.

- **Continuous Integration and Deployment (CI/CD):** In this phase, we will setup a CI/CD pipeline that will automate the testing, building, and deployment of the microservices. By doing so, it will ensure that the microservices are always up-to-date and stable. For example, Jenkins, Gitlab, has been used in order to automate the testing, building, and deployment.
- **Monitoring and Logging:** In the next phase of the development process, monitoring and logging tools will be set up for tracking the health and performance of the microservices as well as for identifying any problems and troubleshooting them as quickly as possible. Examples of these are Grafana, and Elasticsearch/Kibana. Elasticsearch open source used in this case.
- **Data Migration:** An important aspect of this phase involves migrating data from the old monolithic system to the new microservices-based system. This can involve migrating data from the old database to the new one, as well as ensuring that all data has been migrated properly and is accessible by the new system. Oracle to PostgreSQL DB migration done for this case.
- **Stabilization:** In this phase, the new microservices-based system is stabilized by fixing any issues that were encountered during the migration and thoroughly testing the system. Once all services have been migrated, the whole system is tested for its performance and scalability. Once all services are migrated, the system will be evaluated to ensure that it is stable, reliable, and can handle the expected load.
- **Deployment and maintenance:** As part of this phase, the new microservices-based system will be deployed and put into operation on the production environment. To ensure that the system remains stable and performs at its best, regular monitoring and maintenance is conducted. To ensure that the microservices and user data are secure, security measures are implemented as well.

It is recommended that the technology stack used in this phased approach be based on cloud-native technologies and a cloud-optimized architecture. As a result of this phased approach and the use of these technologies, businesses are able to minimize the risks associated with migration by testing and validating the new architecture and system incrementally, which allows for a less risky and more controlled transition. In addition, this approach allows the system to continue to function even during the migration process, thereby minimizing the disruption to the business during the migration process.

Target technology stack

- **Front-end:** Web technologies such as HTML, CSS, and JavaScript to create the user interface. It uses frameworks such as React, Angular to create dynamic and responsive UI.

- **Back-end:** Java, Python has been used to build the server-side logic.
- **Database:** DB Migrated to PostgreSQL to store and manage customer data, booking information, and other website data. Plan in place to have a separate **Data warehousing** (**DWH**) solution for better data insight and analytics.
- **Search Engine:** Elasticsearch customized for efficient searching of flights, hotels, and other travel options.
- **Web Server**: Apache to handle HTTP requests and responses.
- **Cloud Infrastructure**: The website hosted on a cloud infrastructure, using Microsoft Azure, to provide scalability, reliability, and cost-effectiveness. Infrastructure as a code methodology followed.
- **Containers and Orchestration**: The website used Docker containerization technologies and Kubernetes (Azure Kubernetes) for orchestration to manage and deploy the microservices.
- **Service Mesh**: Istio used for service mesh to handle service-to-service communication, traffic management, and security.
- **Monitoring and Logging**: The website used monitoring and logging tools such as Grafana, and Elasticsearch/Kibana to track performance, troubleshoot issues and analyze logs.

Case Study: Technology Roadmap for Microservices Adoption

Modernizing a monolithic architecture for this travel booking and planning website from (Java 6, Oracle 11g, JSP, JavaScript, Apache) to (Java 11, Spring Boot, PostgreSQL, Istio, Azure, and Kubernetes) offers the following advantages:

Technology that is up-to-date:

- **The Java 11 platform:** The new Java 11 release provides features such as improved performance, new language constructs, and support for new standards such as HTTP/2, TLS 1.3, and JDK Flight Recorder.
- **Spring Boot:** It provides a modern, easy-to-use framework for building microservices. Spring Boot provides an embedded servlet container, which simplifies the deployment process and eliminates the need to deploy WAR files. Moreover, it is equipped with several built-in features, including security, data access, and caching, which can help reduce the development time.

- **PostgreSQL:** The database is an open-source, robust relational database that is known for its performance, scalability, and high availability.
- **Istio:** Istio provides a service mesh that helps manage and secure microservices. It provides features such as traffic management, service discovery, load balancing, and monitoring.
- **Cloud-native services provided by Azure: Microsoft Azure** is a major cloud provider that provides scalability, security, and global reach. Azure provides a variety of services, including compute, storage, and networking, as well as advanced security features such as DDoS protection.
- Container orchestration systems such as **Kubernetes** can assist with the deployment, scaling, and management of containerized applications. A number of features are included with Kubernetes, including self-healing, automatic scaling, and rollbacks, which can improve its availability and scalability.
- **Performance improvement:** The improved performance and scalability of **PostgreSQL** can be achieved by migrating from Oracle 11g to PostgreSQL. By allowing for horizontal scaling, microservices can also help to improve the overall performance of the system. Using microservices architecture, new instances of a service can be added as a result of increased traffic, enabling horizontal scaling.
- **Enhanced security:** Service meshes such as Istio can provide security features such as authentication, authorization, and encryption. Furthermore, a cloud provider such as Azure can provide advanced security features such as DDoS protection and security groups.
- **Development and maintenance have been improved:** The use of microservices and Spring Boot can make development and maintenance of the system easier, by allowing for more granular updates and making it easier to test and deploy new features. The microservice architecture enables the monolithic application to be broken down into smaller, independent services, making it easier to make changes and updates to individual portions of the system without affecting the entire system as a whole.
- **Cost-saving improvements:** In order to reduce costs, businesses can use Azure's pay-as-you-go pricing, which allows them to pay only for the resources they use. Moreover, Azure offers a variety of pricing options, including reserved instances and spot instances, that can further reduce costs.

- **Using Kubernetes**: Businesses will be able to take advantage of automatic scaling, which can help to reduce costs by running only the number of instances required. Additionally, Kubernetes provides features such as automatic binpacking, which can help to optimize resource usage and reduce costs.
- **DevOps improvements:** The use of Kubernetes can facilitate the improvement of the availability and scalability of the system by providing features such as automatic scaling, self-healing, and rollbacks.
    - Using Istio, businesses can take advantage of features such as traffic management, service discovery, load balancing, and monitoring. These functions can improve the system's availability and scalability.
- **Productivity improvements for developers:** It is possible for businesses to reduce development time by utilizing Spring Boot's built-in features such as security, data access, and caching. As part of Spring Boot, an embedded servlet container is provided, which simplifies the deployment process and eliminates the need for WAR files.
- As a result of Java 11, businesses can take advantage of new language constructs such as enhanced type inference and improved exception handling, which can improve developer productivity.

Case Study: Application Transition to Microservices Architecture

The **Domain-Driven Design** (**DDD**) approach played a key role in the creation of the below microservices architecture. DDD helped in defining the boundaries of the different domains and subdomains of the travel booking and planning website, which helped in breaking down the monolithic application into smaller, more manageable microservices.

By focusing on the business domains, DDD helped to identify the different functional components of the application and how they interact with each other. This approach helped in defining the service boundaries and in ensuring that each microservice had a well-defined and focused responsibility.

DDD also helped in defining the language and terminology used within each domain, which helped in improving communication and collaboration between the development team and the business stakeholders. This led to a better understanding of the business requirements and ensured that the microservices were designed to meet those requirements.

In order to transform a monolithic travel booking and planning website into a microservice, the following functional components were converted as scalable microservices:

- **Search**: Elasticsearch would be used to index and search the data in real-time, providing fast, accurate search results. Elasticsearch would be configured to index data including flight, hotel and rental car availability, prices, and locations. Additionally, the search service would provide advanced search features such as full-text, geospatial, and faceted searches. There would be multiple types of searches that could be handled by the search service, such as flights, hotels, car rentals, vacation packages, and the like.
- **Booking**: The booking microservice would handle the business logic of the booking functionality, including validating availability, calculating prices, and processing payments. Spring Boot's built-in data access and caching capabilities can be utilized to improve performance and reduce development time by using the service. As part of the booking service, cancellations and modifications would also be handled, and real-time status updates would be provided.
- **Payment**: The payment microservice would handle the secure processing of credit card transactions using a payment gateway such as Stripe. Additionally, the service would handle the business logic associated with the payment functionality, such as handling refunds and generating invoices. Multiple payment methods, including credit cards, debit cards, and digital wallets, would also be supported.
- **User Management**: The user management microservice would handle the business logic of the user management functionality, including validating user credentials, managing user profiles and permissions. Additionally, Spring Boot's built-in security support would be utilized for the service, thus enhancing security and reducing development time. The user management service would also provide functionalities such as forgot password, account verification, and account deactivation.
- **Front-end**: A front-end microservice would handle all user interface requests and provide a responsive, user-friendly interface for all user interface requests. React.js, a JavaScript library for building user interfaces, would be used for the front-end of the service, and APIs would be used to integrate it with the other microservices.
- **Deployment and scaling**: The microservices would be deployed and scaled using Kubernetes and Istio. Kubernetes would be used for container orchestration, which would handle the deployment, scaling, and management of the microservices. It is expected that Istio will serve as an infrastructure mesh that will provide features such as traffic management, service discovery,

load balancing, and monitoring, thereby improving system availability and scalability.

- **Data migration**: In order to migrate data from the Oracle 11g database of the monolithic system to the PostgreSQL database, a controlled process would need to be utilized. In order to minimize downtime and ensure data integrity, the old database's data would need to be extracted, transformed if necessary, and loaded into the new PostgreSQL database. A plan for handling issues that arise during the migration, such as data loss or inconsistent data, is essential.
- **Personalization**: A personalization microservice would be responsible for providing users with personalized recommendations based on their search history, booking history, and browsing history. In addition to analyzing user data and offering personalized recommendations, this microservice would also use past searches, bookings, and browsing history to suggest options similar or related to the user's search, booking, or browsing history. In addition, it would provide personalized recommendations through collaborative filtering, content-based filtering, and hybrid methods.
- **Notifications**: Using the notifications microservice, users will receive notifications such as confirmation of bookings, flight status updates, and price drops. In order to communicate with users in real-time, this microservice would utilize technologies such as WebSockets and push notifications. It would also communicate with users via email, SMS and push notifications. As part of the notifications service, users would also be able to schedule notifications for future events, track delivery status, and manage their opt-in/opt-out preferences.

To ensure a smooth transition and minimize downtime, it is important to keep in mind that this is a complex process that requires careful planning, testing, and monitoring. Moreover, it is important to plan for handling problems that arise during the transition, such as data migration and service integration.

Case Study: Successful Database Migration to Microservices

Travelguru, a leading travel website, had been using a monolithic Oracle 11g database for many years. As the company grew, the database became increasingly complex and difficult to maintain. The database was also unable to scale to meet the company's growing needs. Travelguru recognized the need to migrate to a more scalable and flexible architecture.

Challenge: The primary challenge for Travelguru was to migrate the existing data from Oracle 11g to a new cloud-based PostgreSQL microservices architecture without any disruption to the website's availability. The migration had to be completed in a timely and efficient manner while ensuring the data's integrity and compliance with regulatory requirements.

Solution: Travelguru implemented a phased migration strategy to minimize downtime and ensure data consistency between the on-premises Oracle 11g database and cloud-based PostgreSQL microservices. The following steps were taken to complete the migration:

- **Detailed Analysis:** To ensure a successful migration, Travelguru analyzed the existing data schema and access patterns using ERwin and Oracle SQL Developer Data Modeler. This analysis helped identify the data entities required for the new microservices and their relationships.
- **Data Extraction**: Travelguru used tools like SQL Developer and Oracle Data Pump to extract data from the Oracle 11g database. The extracted data was then transformed to match the PostgreSQL schema. Large amounts of data were transferred securely and efficiently using Amazon Web Services DataSync.
- **Data Load:** Data was loaded into the new PostgreSQL microservice databases using tools like pgAdmin or pgLoader. Testing was conducted to ensure that the data was loaded correctly, and the microservices could access it properly. Database triggers or stored procedures were used to ensure data integrity and consistency.
- **Data Validation:** Data validation involved comparing data from the cloud-based PostgreSQL microservice databases to the Oracle 11g database and identifying any discrepancies. A tool like dbt was used to automate this process.
- **Data Synchronization:** Travelguru synchronized the on-premises Oracle 11g database and cloud-based PostgreSQL microservice databases using tools such as AWS DMS, Azure Database Migration Service, or Google Cloud SQL Replication. Data was replicated in real-time, ensuring consistency between on-premises and cloud-based databases.
- **Cutover:** The monolithic application was switched over to use cloud-based PostgreSQL microservice databases by updating the application code to use the new microservices and configuring the application accordingly. Travelguru implemented a phased migration strategy, which involved migrating data and services in smaller, incremental steps to minimize downtime.

- **Monitoring**: The new microservices and databases were monitored to ensure that they were functioning as expected. This included monitoring the performance and availability of the microservices and databases, as well as identifying and resolving any issues.
- **Security and Compliance:** Travelguru ensured that the data was protected and met regulatory and compliance requirements by setting up encryption, access control policies, and auditing mechanisms. The migration process adhered to compliance requirements, such as GDPR, HIPAA, or SOC 2.
- **Data Partitioning and Sharding**: Travelguru partitioned and sharded the data to improve performance and scalability based on the data access patterns and performance requirements.

Travelguru's migration from Oracle 11g to cloud-based PostgreSQL microservices was successful, ensuring the website's uninterrupted availability. The new architecture improved scalability, flexibility, and performance while adhering to regulatory and compliance requirements. By implementing a phased migration strategy, Travelguru minimized downtime and ensured data consistency.

Recommendations to minimize downtime

- **Test the migration process:** The migration process should be tested in a staging environment that is identical to the production environment before migrating the production database. This will allow you to identify and resolve any issues that arise during the migration process, thereby reducing the risk of downtime in the production environment. Create a replica of the production environment, configure the cloud-based PostgreSQL microservice databases, and run the migration process with sample data to test the migration process.
- **Use a phased approach:** Organize the migration process into smaller, manageable phases to minimize the risk of downtime and improve performance. This will allow you to migrate data and services incrementally. Phased migrations can be accomplished by migrating services one at a time, starting with non-critical services and progressing to critical services as necessary.
- **Use database replication:** Ensure that the data between the Oracle 11g database and the PostgreSQL microservice databases is replicated using a database replication solution, such as AWS DMS, Azure Database Migration Service, or Google Cloud SQL Replication. By configuring Golden Gate, Streams or Data Guard according to the use case, you will be able to switch over to the new databases gradually, minimizing downtime.
- **Optimize database performance:** Optimize the performance of cloud-based PostgreSQL microservice databases by configuring them correctly, indexing

tables, and implementing caching mechanisms. Having the right instance type, the right storage, and the right number of replicas configured can help to improve the performance of the microservices once the migration has been completed. The performance of microservices can also be improved through the use of caching mechanisms such as Redis, memcached, or Ehcache.

- **Implement a rollback plan:** A rollback plan should be developed in case of any issues during the migration process. This will enable you to quickly switch back to the on-premises Oracle 11g database, minimizing downtime.

Case Study: Business Outcome of Microservices Migration

There are several functional and business benefits gained from modernizing 'TravelGuru' a monolithic architecture based travel booking and planning website, including:

- **Improved customer experience**: Microservices architecture and newer technologies enable the system to offer faster and more responsive services, thereby improving the customer experience. Additionally, improved search functionality and personalization capabilities will make the system more appealing to users as well.
- **Increased flexibility**: This is achieved by breaking the monolithic application down into smaller, independently deployable services, thereby making the system more flexible, easier to maintain and more up-to-date.
- **Improved on demand scalability**: Cloud-based technologies enable the system to scale horizontally and vertically on demand, allowing it to handle increased traffic and demand.
- **Increased security**: As a result of utilizing Azure, the system took advantage of the built-in security features, such as Azure Security Center, to enhance its security.
- **Improved search functionality**: With Elasticsearch, the system can provide enhanced search functionality, such as full-text search and faceted search. This will provide a better user experience for customers and increase their chances of finding the appropriate travel option.
- **Improved business agility**: With microservices and cloud-based technologies, the system becomes more agile and responsive to the changing needs of the business, allowing it to quickly respond to new market trends and customer requirements, enabling the business to gain a competitive advantage.
- **Cost Savings**: The pay-as-you-go model of cloud providers will contribute to the reduction of costs associated with idle capacity by using cloud-based technologies.

Overall, the modernization of the architecture will result in a more robust, scalable, and efficient platform, which will benefit the long-term growth of the business, improve customer satisfaction, and increase revenue.

Case Study: Best Practices Implemented in Microservices Migration

Best practices that were implemented in the successful Monolith to Microservices Cloud-native migration for a travel related website:

- **Identifying the right set of microservices**: To identify the right set of microservices for the travel related website, the team could break down the monolithic application into smaller services such as flight booking, hotel reservation, car rental, and travel insurance. For example, the flight booking service could be responsible for handling flight search, availability, booking, and payment processing.
- **Designing for fault-tolerance and high availability**: To ensure that the microservices are highly available and fault-tolerant, the team could use load balancing, auto-scaling, and health checks. For example, the hotel reservation service could use auto-scaling to scale up or down based on demand and load balancing to distribute traffic to healthy instances.
- **Using containerization and orchestration tools**: To deploy and manage the microservices, the team could use containerization tools such as Docker and orchestration tools such as Kubernetes. For example, the flight booking service could be deployed as a Docker container and managed using Kubernetes.
- **Integrating with existing legacy systems**: To integrate the microservices with existing legacy systems, the team could use APIs and message queues. For example, the flight booking service could integrate with the airline reservation system using APIs and message queues.
- **Testing and monitoring**: To ensure that the microservices are working as expected, the team could perform extensive testing and set up monitoring tools to track the performance and availability of the microservices. For example, the car rental service could be tested using end-to-end testing to ensure that the booking, pickup, and drop-off process is working correctly, and monitored using metrics and logs to track the performance and availability.
- **Ensuring data consistency**: To maintain data consistency across different microservices, the team could use event-driven architecture and transaction management. For example, the hotel reservation service could publish an event when a booking is made, and the car rental service could subscribe to that event and update its availability accordingly. The team could also

use transaction management to ensure that all changes made by different microservices are committed or rolled back together.

- **Ensuring security and compliance**: To ensure the security and compliance of the microservices, the team could use encryption, authentication, and authorization mechanisms. For example, the travel insurance service could use encryption to protect sensitive customer data, and authentication and authorization to ensure that only authorized users can access the service.
- **Implementing DevOps practices**: To ensure faster and smoother deployment of microservices, the team could implement DevOps practices such as continuous integration, continuous delivery, and automated testing. For example, the flight booking service could be automatically tested and deployed to production when new code changes are committed to the repository.
- **Leveraging cloud-native technologies**: To take advantage of cloud-native benefits such as elasticity, scalability, and resilience, the team could use cloud-native technologies such as serverless computing, managed databases, and storage services. For example, the car rental service could use serverless functions to perform small tasks such as sending email notifications or updating availability.
- **Separating concerns and responsibilities**: To ensure that each microservice has a clear and well-defined responsibility, the team could separate concerns and responsibilities using domain-driven design principles. For example, the flight booking service could be responsible for handling flight-related operations such as search, booking, and payment processing, while the hotel reservation service could be responsible for handling hotel-related operations such as search, booking, and payment processing.
- **Implementing service discovery and routing**: To enable communication between microservices, the team could implement service discovery and routing mechanisms. For example, the flight booking service could discover and communicate with the hotel reservation service using a service registry and a routing mechanism such as API Gateway.
- **Monitoring and optimizing performance**: To ensure optimal performance of the microservices, the team could set up monitoring and optimization tools to identify performance bottlenecks and optimize the system accordingly. For example, the flight booking service could use a distributed tracing system to track the performance of each service call and identify performance issues.

Conclusion

Empowering the Travel Industry (case study):

Accelerating Growth through Microservices & Cloud Migration

In conclusion, investment in microservice adoption and cloud migration can bring significant benefits for a monolithic travel booking and planning website. The microservice architecture provides increased scalability and flexibility, while cloud-based technologies provide increased security, improved scalability, and cost savings. The system will be able to provide faster and more responsive services, improved data management, improved search functionality, and better customer experience as a result of using newer technologies such as Java 11, Spring Boot, PostgreSQL, Istio, Azure, Elasticsearch, and Kubernetes. Additionally, it can improve the system's overall performance, increase developer productivity, and enhance its ability to adapt to changing business requirements. We believe that investing in microservice adoption and cloud migration can translate into long-term benefits for the business and help it to remain competitive in the marketplace. Our top three learnings from this case study include:

- A microservices architecture offers increased scalability and flexibility: A monolithic application is broken down into smaller, independently deployable services that enable faster and more frequent updates, resulting in a more current and feature-rich application.
- By leveraging cloud-based technologies, like Azure, the system can benefit from the built-in security features, while reducing the costs associated with maintaining on-premises infrastructure and software. Cloud-based technologies provide cost savings, improved security, and improved scalability: In addition, cloud-based technologies can enhance scalability, enabling the system to cope with increased traffic and demand.
- Incorporating newer technologies like Java 11, Spring Boot, PostgreSQL, Elasticsearch, Istio, and Kubernetes into the system enables the system to deliver faster and more responsive services, better data management, improved search functionality, and enhanced customer service. As a result, the business will be more satisfied with its customers, earn more revenue, and gain a competitive advantage.

Next, in *Chapter 7 'Inter-Service Communication'*, we will examine inter-service communication, service meshes, and message brokers as methods for improving the overall performance and reliability of distributed systems by enabling communication and collaboration between different services.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7
Inter-Service Communication

Seamless Inter-Service Communication for Microservices

Introduction

In a microservice architecture, inter-service communication refers to communication between microservices. Microservice architecture involves running a distributed system on different machines, with each service being a component or process of the enterprise application. All of these services work together to handle requests from clients of this enterprise application, so they interact using an inter-service communication method. Thus, these services must work together at all times. There are two main approaches to inter-service communication in microservice architecture: synchronous communication and asynchronous communication.

Synchronous communication involves the use of REST APIs, **Remote Procedure Calls** (**RPCs**), or gRPC. These mechanisms allow services to interact with each other in real-time, making direct requests and awaiting responses. This approach is suitable for scenarios where immediate response and tight coordination among services are required.

On the other hand, asynchronous communication involves the use of message broker software such as Apache Kafka, RabbitMQ, or other similar solutions. With asynchronous communication, services communicate by exchanging messages through a message broker, enabling decoupling and asynchronous processing. This approach is beneficial for scenarios where loose coupling, scalability, and fault tolerance are priorities.

There are a few key differences between asynchronous and synchronous communication in microservices related to Performance, Reliability, and Design Complexity. Which communication pattern to use in a microservices-based architecture depends on the specific application requirements and constraints of the system. During this chapter, we will examine both asynchronous and synchronous communication styles as well as how to determine which pattern is most appropriate for a given use case.

Increasingly, distributed systems use service meshes to improve reliability, scalability, and security. Service meshes are dedicated infrastructure layers that connect services in a distributed environment. The system's performance and reliability can be improved with features like load balancing, circuit breaking, and observability. In addition to simplifying management of the system and improving its performance, service meshes provide a dedicated infrastructure layer that handles communication between services.

Structure

In this chapter we will discuss following topics:

- Inter-Service communication
    - Challenges of distributed systems
    - Communication models
- Synchronous Inter-Service communication
    - RESTful APIs
    - Remote Procedure Calls (RPCs)
    - gRPC
- Asynchronous Inter-Service communication
    - Message brokers
    - Message broker models
    - Message broker software
        - RabbitMQ
        - Apache Kafka
        - IBM MQ
        - Azure service bus
        - Amazon Simple Queue Service (SQS)
- Event-driven communication

    - Publish-Subscribe Architecture
    - Event-Driven Architecture
    - Event Sourcing
- Serialization
    - Serialization Formats (JSON, Protobuf, Thrift)
    - Serialization Libraries (Jackson, Gson, Protobuf, et)
    - Best Practices for Serialization
- Service Mesh
    - Features of Service Mesh
    - Tools/third-party Products for Service Mesh
    - Istio Service Mesh
- Idempotent Operations
    - Implementing Idempotency
- Conclusion

Objectives

We will discuss the purpose of inter-service communication, service meshes, and message brokers in this chapter to improve the overall performance and reliability of distributed systems by enabling communication and collaboration between different services. It is important to understand the differences between asynchronous and synchronous communication in microservices-based architectures because the choice of communication pattern can have significant impacts on the performance, reliability, and complexity of the system.

Throughout this chatter, we will provide a clear and concise understanding of message brokers and service meshes as approaches to inter-service communication within a microservices architecture. Through this, you should gain a solid understanding of the key differences, types, and advantages of these two key approaches. This will enable you to make an informed decision regarding which approach is most appropriate for your application.

Inter-Service communication

Inter-service communication is the process of exchanging data and messages between microservices. It is a critical aspect of microservices architecture because microservices are typically designed to be loosely coupled and independently

deployable. Therefore, they need to communicate with each other to perform complex operations and deliver value to end-users.

There are several reasons why inter-service communication is important in microservices architecture:

- Microservices typically represent a specific domain or business capability. Therefore, they need to communicate with each other to provide a complete business solution.
- Microservices are designed to be independently deployable, which means they can be scaled, updated, or replaced without affecting other microservices. Therefore, they need to communicate with each other to perform complex operations that span multiple microservices.
- Microservices are designed to be loosely coupled, which means they can use different programming languages, frameworks, or platforms. Therefore, they need to communicate with each other using well-defined interfaces and standard protocols.

Challenges of distributed systems

It is important to note that distributed systems are made up of multiple components that are connected via a network, working together to achieve a common goal. These systems may be challenging to design and maintain for the following reasons:

- **Communication latency**: The components of a distributed system may be located at different locations, and communication between them may be slow, causing processing delays.
- **Network failures**: In distributed systems, communication between components can be delayed or interrupted due to network failures or other issues.
- **Consistency:** There can be challenges in maintaining consistency between the different components in a distributed system, particularly if they are updating shared data. This can result in inconsistencies or conflicts in the data.
- **Concurrency:** Multiple requests may be handled concurrently by distributed systems, which can lead to race conditions and deadlocks.
- **Partial failure:** It is possible for individual components in a distributed system to fail or become unavailable. This can result in a partial failure of the system as a whole, in which some components continue to operate while others do not.

- **Integration complexity:** Distributed systems can present challenges in terms of integrating different components and ensuring that they work seamlessly as a result of differences in technology, protocols, and data formats.
- **Monitoring and debugging**: Monitoring and debugging issues in a distributed system may be difficult due to the involvement of multiple components and the difficulty in reproducing them.
- **Security**: It may be difficult to ensure that data is secure and that only authorized users have access to it on distributed systems due to security threats such as hacking or data breaches.

For example, in an airline industry distributed systems is being used for various components such as ticketing, reservation, and baggage handling, etc. This geographical distribution can introduce delays in communication between these components, impacting the overall processing time. For example, when a customer makes a reservation, the request needs to be transmitted to the appropriate system for processing, which can experience latency if the systems are not well-connected or communicating. Handling and minimizing communication latency is crucial for providing efficient and responsive services to the airline customers. All these challenges can be handled with a proper inter-service communication design. It is crucial to ensure that communication channels are reliable, efficient, and secure for the smooth operation of a microservices architecture. We can design inter-service communication in microservice architecture using two approaches that is synchronous communication or Asynchronous communication. We might use both Synchronous and Asynchronous communication within an application for managing communication between different components.

Communication models

There are two primary communication models in microservices architecture: synchronous and asynchronous.

- **Synchronous communication** means that a client sends a request to a microservice and waits for a response before proceeding. Synchronous communication is useful when a client needs an immediate response from a microservice to continue its work. Examples of synchronous communication include RESTful APIs, **remote procedure calls** (**RPCs**), and gRPC. For example, consider an e-commerce website that allows users to purchase products online. When a user adds a product to their cart and clicks "Checkout," the website sends a synchronous request to the Payment microservice to process the payment. The website waits for a response from the Payment microservice before displaying the confirmation page to the user.

- **Asynchronous communication** means that a client sends a message to a microservice without waiting for a response. Asynchronous communication is useful when a client does not need an immediate response from a microservice or when a microservice needs to perform long-running or background tasks. Examples of asynchronous communication include message brokers and event-driven architecture. For example, consider a shipping microservice that needs to notify the Inventory microservice when a product is shipped. The shipping microservice can send an asynchronous message to the Inventory microservice using a message broker. The Inventory microservice can consume the message and update its database without the shipping microservice waiting for a response.

In summary, inter-service communication is essential in microservices architecture because microservices are designed to be loosely coupled and independently deployable. Synchronous communication is useful when a client needs an immediate response from a microservice, while asynchronous communication is useful when a client does not need an immediate response or when a microservice needs to perform long-running or background tasks.

Synchronous inter-service communication

A synchronous inter-service communication is a method of communicating in real-time between two or more services. The sender sends a request and waits for the response before continuing. Although this provides immediate feedback and error handling, it may also result in slower performance if one of the services takes a long time to respond. In order to facilitate synchronous inter-service communication, RESTful APIs, RPCs, and gRPC can be used.

RESTful APIs

It is a software architectural style that defines a set of constraints that can be applied to the development of web services using **Representational State Transfer (REST**). Providing interoperability between computer systems on the Internet is possible with web services that adhere to the REST architectural style, called RESTful web services. Through the use of a uniform and predefined set of stateless operations, RESTful web services allow requesting systems to access and manipulate textual representations of web resources.

A RESTful web service is based on the HTTP protocol and operates on resources identified by a **Uniform Resource Identifier** (**URI**) through HTTP verbs (GET, POST, PUT, DELETE, and the like.). In addition to defining the set of operations and resources on which they operate, the **Application Programming Interface** (**API**) also specifies the format of the response and request messages. RESTful APIs can be used with **Hypertext Transfer Protocol Secure** (**HTTPS**) to provide secure communication over the internet. Through **Secure Sockets Layer/Transport Layer**

Security (**SSL/TLS**), HTTPS establishes an encrypted connection between a client (such as a web browser) and a server, encrypting the data being transmitted. By doing so, we can avoid man-in-the-middle attacks and other types of cybercrime.

Advantages

- The RESTful API is well established and widely used and is supported by a large developer community. It uses HTTP, which is a well-known and popular protocol.
- As stateless services, they are scalable and easily cacheable since the server does not store any client context between requests.

Disadvantages

- They can be less efficient than other types of APIs, such as SOAP, which include additional features such as error handling and security.
- They do not have built-in support for complex data types, such as attachments or multi-part messages. They do not have a standard way to specify the API structure, which can lead to inconsistency in the design of RESTful APIs.

Remote Procedure Calls (RPCs)

It is common practice to implement microservices using RPCs. This allows the creation of distributed systems in which different parts of a program can be executed on different machines, potentially improving efficiency and scalability. RPC operates using a client-server model, in which the client sends a request to the server to execute a particular function, along with any necessary arguments. A network protocol, such as HTTP or TCP/IP, is used to communicate between the client and the server. The server then executes the function and returns the result to the client.

Advantages

- It is easy to create distributed systems using RPCs without the need for complex programming, making the creation of distributed systems and adding new functionality to existing systems easy.
- RPCs enable greater interoperability between applications and platforms by enabling communication between different systems and languages.

Disadvantages

- Using RPCs can add some overhead, as the client and server must communicate using a network protocol. This can potentially lower performance when compared to a system that executes all functions locally.

- **Risks associated with RPCs**: If RPCs are not implemented correctly, they can potentially create security vulnerabilities. For instance, if the server does not authenticate client requests properly, it could be susceptible to attacks.

gRPC Remote Procedure Calls

A **remote procedure calls** (**RPC**) system developed by Google is known as gRPC (gRPC Remote Procedure Calls). In addition to supporting many languages and platforms, gRPC is highly efficient, has low network overhead, and supports many languages and platforms as well as HTTP/2 as the transport medium. Protocol Buffers are used for interface description.

Advantages

- gRPC supports bi-directional streaming, which allows for asynchronous communication between the client and server. This is useful for real-time applications and for the transmission of large amounts of information.
- The gRPC protocol includes built-in load balancing, which can distribute traffic among multiple servers and improve a system's scalability.
- In order to ensure the security of communication between microservices, gRPC supports multiple authentication mechanisms, including TLS and OAuth2.

Disadvantages

- In a microservices architecture, gRPC can introduce some complexity, as communication between services can become more complex as a result.
- There is limited browser support for gRPC, making it an unsuitable choice for applications that require browser support.
- gRPC uses Protocol Buffers for the description of interfaces, which may require additional configuration.

Asynchronous Inter-Service communication

The term asynchronous inter-service communication refers to the use of asynchronous communication between different services within a distributed system. An asynchronous communication model does not require the sender and receiver to communicate simultaneously. Instead, the sender sends a message and does not wait for a response before proceeding with other tasks. The receiver can then process

the message at their own pace, without blocking the sender. In a microservices architecture, asynchronous communication can be useful for communicating between services when they are performing different tasks. Asynchronous communication can facilitate greater scalability by allowing each service to process requests independently and according to the availability of resources. As a result, the system can continue functioning even if one service becomes unavailable, which can also increase reliability.

It is possible to communicate asynchronously between services using a variety of technologies and protocols, including message queues, event buses, and remote procedure calls.

Message brokers

Message brokers are software applications that allow applications, systems, and services to communicate and exchange information with one another. They accomplish this by translating messages between formal messaging protocols. As a result, interdependent services can communicate directly, regardless of whether they are written in different languages or implemented on a different platform. A message broker is used to facilitate communication between different components of distributed systems and microservices architectures.

Advantages

- It is possible to decouple different systems and services by using message brokers. This decoupling can make it easier to update and maintain the system, since changes to one component do not necessarily affect others.
- As resources become available, message brokers can facilitate the scalability of a system by allowing different components to process messages simultaneously.
- In the event that one of the systems is unavailable, message brokers can provide reliability by storing and forwarding messages.
- A message broker can provide security by encrypting messages and authenticating senders and receivers.

Disadvantages

- **Complexity**: The implementation and maintenance of a message broker can be challenging, as it requires setting up additional infrastructure and adding logic for sending and receiving messages. This can increase the overall complexity of the system.

- **Latency**: The use of a message broker can result in additional latency in the communication between microservices, depending on the system's size and complexity. In particular, this can apply to scenarios where the message broker is located on a separate machine or in a different network from senders and recipients.
- **Reliability**: As another component that must be available and functioning correctly for communication to take place, message brokers can introduce additional points of failure into a system. This increases its overall complexity and risk.

Message broker models

To facilitate communication between different systems and services, message brokers can utilize a variety of different models. Some common message broker models include:

- **Publish-subscribe model**: Messages are published to a topic by the producer, and multiple message consumers subscribe to topics from which they wish to receive messages in this message distribution pattern, often called "pub/sub." All messages published to a topic are distributed to all applications subscribed to it. This method is known as broadcast distribution, where the message's publisher and consumers are in a one-to-many relationship.
- **Point-to-point model**: It is the distribution pattern used in message queues in which the sender and receiver have a one-to-one relationship. Point-to-point messaging is used when messages must be acted upon only once and must be sent to only one recipient and consumed only once. Each message in the queue is sent to only one recipient and consumed only once.
- **Request-response model**: As part of the request-response model of message brokers, a sender (client) submits a request to the message broker, which forwards it to the specific receiver (server). Once the request has been processed by the server, it will return a response to the message broker, and the message broker will forward the response to the client. In situations where the sender expects a response to the request, this model is often used to facilitate synchronous communication between different systems and services. In the request-response model, the sender sends a request and waits for the response before proceeding with other tasks. This ensures that the response is received before any further processing is undertaken.
- **Event-driven model**: The event-driven model of message brokers involves a sender sending an event message to a message broker, which then forwards the message to the appropriate receivers, who then process the message and take appropriate action. It is commonly employed to facilitate asynchronous

communication between different systems and services, in which the sender does not need to wait for a response before continuing with another task. Using an event-driven model, the sender can send an event and proceed with other tasks, while the receivers process the event as resources become available. An event-driven model is often used when various systems or services must be notified of specific events or changes, or when an event does not need to be processed in a specific order. Applications that require rapid scaling or that require large amounts of data processing may find it useful.

Message broker software

Message broker software is software that acts as an intermediary between the sender and receiver of a message, receiving messages from the sender and forwarding them to the intended recipient. As a result, it facilitates communication between different systems, applications, and services within a distributed environment.

A variety of message broker software platforms are available, including Apache Kafka, RabbitMQ, IBM MQ, Azure Service Bus, and Amazon **Simple Queue Service** (**SQS**).

RabbitMQ

A message broker software that implements the **Advanced Message Queuing Protocol** (**AMQP**), is an open-source software. A distributed system can benefit from RabbitMQ when it facilitates communication between various systems and services. It supports a variety of messaging patterns, including publish-subscribe, point-to-point, request-response, and event-driven messaging.

Advantages

- It is scalable horizontally by adding more servers to the cluster, enabling it to handle large volumes of data.
- It provides features such as message acknowledgment, message persistence, and automatic recovery to ensure that messages are delivered in a reliable manner.
- It is available in a wide variety of languages and platforms, enabling it to be used in a variety of applications.

Apache Kafka

Kafka is an open-source platform for developing real-time data pipelines and streaming applications based on distributed event streaming. The Kafka cluster works by allowing producers to send data to topics, and consumers to read data

from those topics. Messages published to Kafka are stored for a configurable period of time, allowing consumers to review data at their own pace.

Advantages

- It is scalable horizontally by adding more brokers to the cluster, which allows it to handle large volumes of data.
- Kafka ensures that messages are delivered reliably by storing all published messages for a configurable period of time. Additionally, it offers features such as message acknowledgments and automatic recovery to ensure that messages are delivered reliably.
- Kafka is designed to handle high levels of data throughput, making it well-suited for applications that require real-time processing of large amounts of data.

IBM MQ

It is widely used in a variety of industries and applications, and it is designed to be highly scalable and reliable. IBM MQ (formerly IBM WebSphere MQ) is a messaging middleware platform that makes data exchange between applications, systems, and services possible.

In IBM MQ, producers can send messages to queues within the system, and consumers can receive those messages from those queues. It provides features such as message persistence, message acknowledgment, and automatic recovery to ensure that messages are delivered reliably. A number of different messaging patterns are supported by IBM MQ, including point-to-point, publish-subscribe, request-response, and event-driven messaging.

Advantages

- It is designed to be highly scalable, so that it can handle large volumes of messages. The cluster can be scaled horizontally by adding more servers.
- A security feature of IBM MQ is encryption and authentication, which is used to protect messages and maintain communication integrity.

Azure service bus

As a messaging platform provided by Microsoft Azure, Azure Service Bus enables applications, systems, and services to exchange data. There are many industries and applications that use it because it is highly scalable and reliable. There are a number of messaging patterns supported by Azure Service Bus, including point-to-point, publish-subscribe, request-response, and event-driven messaging. In addition, it

supports a variety of languages and platforms, making it easy to use in a variety of environments.

Advantages

- It is designed to be highly scalable, which enables it to handle large volumes of messages, and can be scaled horizontally by adding more servers to the cluster.
- To ensure that messages are delivered reliably, Azure Service Bus provides features such as message persistence, message acknowledgement, automatic recovery, and message expiration and discarding.
- To protect messages and ensure the integrity of communication, Azure Service Bus includes security features such as encryption and authentication. For Azure resources, Service Bus supports security protocols such as **Shared Access Signatures** (**SAS**), **Role Based Access Control** (**RBAC**), and managed identities.

Amazon Simple Queue Service (SQS)

It is designed to be highly scalable and reliable, and it has been widely utilized in a variety of industries and applications. Amazon **Simple Queue Service** (**SQS**) allows you to send, store, and receive messages at any volume between software components without losing messages or requiring other services to be available. A variety of messaging patterns can be supported by SQS, including point-to-point, publish-subscribe, request-response, and event-driven messaging.

Advantages

- It can scale horizontally by adding more servers to the cluster, allowing it to handle large volumes of messages.
- SQS provides features such as message persistence, message acknowledgement, and automatic recovery, which ensure the delivery of messages reliably. You also have the ability to deduplicate messages while processing messages at high scale.
- You can send sensitive data securely between applications using AWS Key Management and Centrally manage your keys with SQS security features, including encryption and authentication.

Event-driven communication

Event-driven communication is a form of asynchronous communication where microservices communicate with each other by producing and consuming events.

An event is a notification that something has happened in a microservice. Event-driven communication is useful in microservices architecture because it allows microservices to be loosely coupled, independent, and scalable.

There are several best practices for implementing event-driven communication in microservices architecture:

- **Use a message broker**: A message broker is a middleware that facilitates the exchange of events between microservices. A message broker provides several features such as message queuing, message filtering, and message routing that simplify event-driven communication.
- **Use a publish-subscribe architecture**: A publish-subscribe architecture is a pattern where a publisher sends events to a message broker, and multiple subscribers receive the events. A publish-subscribe architecture allows microservices to communicate without knowing about each other and decouples the producer from the consumer.
- **Use an event-driven architecture**: An event-driven architecture is a pattern where microservices are designed to be event-driven. An event-driven architecture promotes loose coupling and scalability by allowing microservices to respond to events instead of calls.
- **Use event sourcing**: Event sourcing is a pattern where the state of a microservice is derived from a series of events. Event sourcing allows microservices to be auditable, scalable, and recoverable.

Publish-subscribe architecture

A publish-subscribe architecture is a pattern where a producer sends events to a message broker, and multiple subscribers receive the events. A publish-subscribe architecture is useful when a producer needs to notify multiple subscribers of an event without knowing who the subscribers are. For example, consider an e-commerce website that sells products to customers. When a customer places an order, the Order microservice can send an event to the Payment and Shipping microservices. The Payment and Shipping microservices can subscribe to the Order event and process the payment and shipping of the order.

One commonly used message broker is Apache Kafka, which is a distributed streaming platform that provides features such as fault-tolerance, scalability, and high throughput. Kafka uses a publish-subscribe model where producers write events to topics, and consumers subscribe to topics to receive events. Kafka also supports features such as retention policies, partitioning, and replication that simplify the management of events.

Event-driven architecture

An event-driven architecture is a pattern where microservices are designed to be event-driven. In an event-driven architecture, microservices communicate with each other by producing and consuming events. An event-driven architecture is useful when microservices need to communicate asynchronously and be loosely coupled. For example, consider a video streaming website that allows users to upload and view videos. When a user uploads a video, the Video microservice can send an event to the Encoding microservice to encode the video. The Encoding microservice can send an event to the Notification microservice to notify the user when the video is ready to view.

One commonly used protocol for event-driven communication is Apache Avro, which is a data serialization system that supports schema evolution and efficient binary encoding. Avro provides a compact binary format that reduces the size of events and improves performance. Avro also provides features such as schema validation, data compression, and RPC integration that simplify event-driven communication.

Event sourcing

Event sourcing is a pattern where the state of a microservice is derived from a series of events. In event sourcing, events represent changes to the state of a microservice, and the current state of the microservice is derived by replaying the events. For example, consider a banking application that allows users to transfer money between accounts. When a user transfers money, the Transfer microservice can produce an event that represents the transfer. The Account microservice can consume the event and update the balance of the source and destination accounts. The balance of the accounts can be derived by replaying the events that represent the transfers. Event sourcing is useful when a microservice needs to be auditable, scalable, and recoverable. One commonly used database for event sourcing is Apache Cassandra, which is a distributed NoSQL database that provides features such as scalability, fault-tolerance, and tunable consistency. Cassandra stores events as rows in a table and provides features such as partitioning, replication, and indexing that simplify the management of events. Cassandra also integrates with Apache Kafka to provide a complete event-driven architecture.

Overall, event-driven communication is a powerful pattern for microservices architecture that promotes loose coupling, scalability, and resilience. By using message brokers, publish-subscribe architectures, event-driven architectures, and event sourcing, microservices can communicate asynchronously and react to events instead of calls, which simplifies the design and implementation of distributed systems.

Serialization

Serialization is the process of converting data structures or objects into a format that can be transmitted or stored, such as binary or text. In the context of microservices, serialization is an important aspect of inter-service communication since it allows services to exchange data in a standardized way. Serialization can help microservice implementation, especially when different services are based on different technology stacks, by providing a common format for communication between these services. Let's consider an example to understand this in more detail:

Suppose we have a microservices-based e-commerce application that consists of several services such as Order Service, Product Service, and Payment Service. The Order Service is written in Java, the Product Service is written in Python, and the Payment Service is written in Go. To communicate between these services, we need a common data format that all services can understand. This is where serialization comes in. For example, let's say the Order Service needs to retrieve information about a product from the Product Service, and the Payment Service needs to process a payment for an order created by the Order Service. To transmit this data between the services, we can define a schema using a serialization format such as JSON or Protobuf.

We can use a serialization library that is compatible with all the languages used in our microservices-based application. For instance, we can use Protobuf for serialization, which has libraries available for Java, Python, and Go.

Serialization formats

There are many serialization formats available, but some of the most commonly used ones for microservices are JSON, Protobuf, and Thrift.

- **JSON** is a lightweight text-based format that is easy to read and write, making it a popular choice for web-based applications. JSON is also human-readable and can be used with almost any programming language.
- **Protobuf and Thrift** are binary serialization formats that are designed for efficient and compact data transfer. They provide features such as schema evolution, field-level data validation, and data compression, making them suitable for high-performance and large-scale systems. Protobuf and Thrift also provide code generation tools that simplify the process of working with serialized data.

Serialization libraries

There are many serialization libraries available for different programming languages, such as Jackson and Gson for Java, Protobuf for Java, C++, and Python, and Thrift for multiple languages.

These libraries provide APIs for encoding and decoding data in different formats and handle serialization details such as schema evolution, field mappings, and type conversion. They also provide features such as streaming, compression, and security that can improve the efficiency and reliability of inter-service communication.

Best practices for serialization

Here are some best practices to consider when working with serialization:

- Use a compact serialization format such as Protobuf or Thrift for large data transfers to reduce network bandwidth and improve performance.
- Use a text-based serialization format such as JSON for small or human-readable data transfers to improve ease of use and debugging.
- Define a schema or contract for your data using a language-agnostic format such as Avro or Swagger, to ensure compatibility between different services.
- Consider using code generation tools to create serialization and deserialization code for your data models to reduce the potential for errors.
- Use appropriate security measures such as encryption and authentication to protect your data during transfer and storage.

Service mesh

It is recommended that you review *Chapter 4, Service Discovery Pattern* before diving into Service Mesh concepts. Service Discovery Pattern covered in detail in *Chapter 4* includes the types of service discovery (client-side, server-side), as well as the types of service discovery methods such as DNS based, Key/Value Store based and sidecar based, and specialized service discovery (library/sidecar based). Both the service discovery pattern and service mesh facilitate communication between services in distributed systems. The service discovery pattern is a design pattern that is appropriate for small systems, whereas the service mesh is a dedicated infrastructure layer that is more appropriate for large, complex distributed systems.

Service Mesh: Containers and container orchestrators, like Kubernetes, have resolved numerous challenges by packaging services with their own runtimes and efficiently mapping them to machines. However, a crucial aspect was still missing - effective management of inter-service communication. This is where the concept of a service mesh comes into play. The service mesh serves as a powerful design pattern that abstracts the underlying network infrastructure, providing a standardized solution by deploying sidecar proxies alongside your services. These proxies, often leveraging technologies like the Envoy proxy, handle critical networking tasks, security enforcement, and observability.

By incorporating a service mesh, you can streamline your operational processes significantly. Service developers no longer need to invest excessive time and effort in manual connection and networking configurations, retries, or timeout setups. Instead, the burden is shifted to the dedicated sidecar proxies, which effectively handle these tasks on behalf of the services. For example, as illustrated in *Figure 7.1, Istio Service Mesh Architect i*n an Istio service mesh, several components work together to provide advanced networking, security, and observability features.

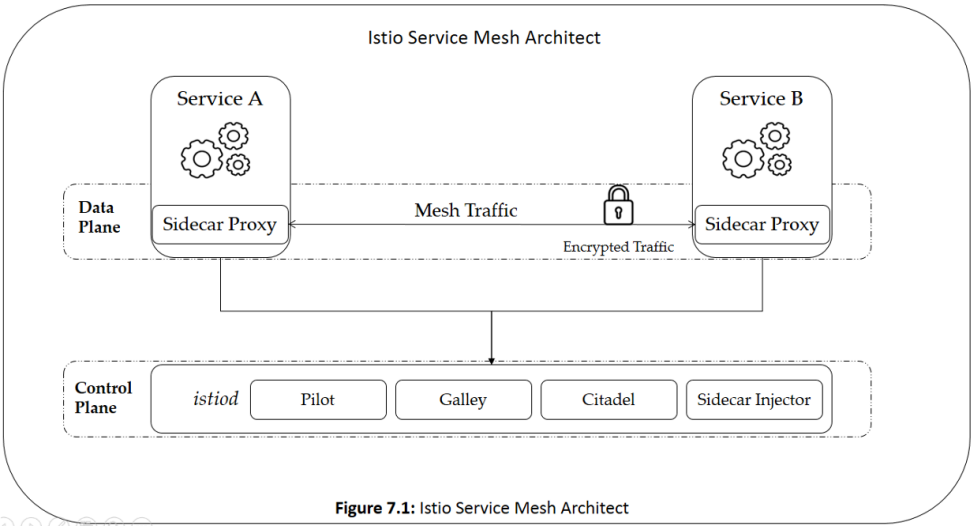


Figure 7.1: Istio Service Mesh Architect

Here's an explanation of the key components from above *Figure 7.1*:

- **Data Plane**: The Data Plane consists of sidecar proxies deployed alongside each service within the mesh. These proxies intercept and control network communication between services. In Istio, the Envoy proxy is commonly used as the sidecar proxy. The sidecar proxies handle traffic routing, load balancing, service discovery, and enforcing policies defined in the Control Plane.
- **Control Plane**: The Control Plane manages and configures the behaviour of the service mesh. It consists of several components:
    - **Pilot**: Pilot is responsible for service discovery, traffic management, and configuration distribution to the sidecar proxies. It translates high-level routing rules into configurations understood by the sidecar proxies, enabling features like load balancing, circuit breaking, and fault injection.
    - **Galley**: Galley is responsible for validating and distributing configuration changes within the Control Plane. It ensures that the configuration changes made through various mechanisms, such as Kubernetes ConfigMaps or Istio's **Custom Resource Definitions**

(**CRDs**), are validated, converted into the appropriate format, and distributed to the relevant components.

- **Citadel**: Citadel is responsible for providing secure service-to-service communication within the mesh. It manages and issues **Transport Layer Security** (**TLS**) certificates for authentication and encryption between services. Citadel integrates with identity providers, such as Kubernetes Service Account Tokens or external certificate authorities, to establish trust and secure communication channels.
- **Sidecar Injector**: The Sidecar Injector is a web-hook component that automatically injects the sidecar proxies into the pods of services that are part of the mesh. It intercepts pod creation requests and modifies them to include the necessary sidecar containers.

- **Mesh Traffic**: Mesh Traffic refers to the actual network traffic flowing between services within the service mesh. The sidecar proxies control and route this traffic based on the policies and configurations set in the Control Plane.

These components work together to provide advanced networking capabilities, secure communication, and observability within the service mesh. On top of the infrastructure layer, a service mesh is a platform layer that enables the communication of individual services in a managed, observable, and secure manner. Using this platform layer, we can create robust enterprise applications based on chosen infrastructure that utilize various microservices. Service meshes utilize consistent tools to factor out all the common concerns associated with running a service, such as monitoring, networking, and security. As a result, service developers and operators can focus on developing and managing applications for their users rather than implementing measures to address individual service challenges. In addition to improving the reliability and performance of the system, it provides a wide range of features to help manage communication between services.

Features of service mesh

A service mesh is a dedicated infrastructure layer that handles service-to-service communication within a microservices architecture. Some of the features of a service mesh include:

- **Load balancing:** Service meshes typically include load balancing capabilities to ensure that incoming requests are distributed equally across multiple instances of a service. This can improve the performance and reliability of the system by ensuring that no single instance of a service is overloaded. As a result of the observed latencies or the number of outstanding requests, load balancing at the session level can significantly improve performance over Kubernetes' layer-4 load balancing.

- **Observability:** To facilitate monitoring and understanding of the behavior of the system, service meshes offer observability features such as metrics, logging, and tracing. A variety of metrics can be used to assess the system's performance and health, including request rates, errors, and latencies. In addition to providing information about events and activities within the system, logging also provides information on errors and requests. Observability features such as Tracing can assist in identifying issues and improving the overall reliability and performance of the system by providing information regarding the flow of requests through the system, including the path that requests take and the time it takes for a request to complete.
- **Security:** Security features such as encryption and authentication are provided by service meshes in order to protect communication between services and to ensure system integrity. By encoding data during transit, encryption helps protect it so that it can only be accessed by authorized parties. By verifying the identity of users and services, authentication ensures that only authorized parties can access the system. By implementing a service mesh, achieving zero trust becomes easier. Service meshes provide these authentication and authorization identities through a central certificate authority that certifies each service.
- **Service discovery:** A service mesh often includes a service discovery capability to identify and communicate with other services. These services are routed based on URL path, host header, API version, or other application-level criteria. Service meshes provide a dedicated infrastructure layer that handles communication between services, which can simplify the management of the system.
- **Retry of failed requests:** You can configure the maximum number of retries, along with a timeout period in order to limit the maximum latency of a service mesh, by understanding HTTP error codes.
- **Circuit breaking:** An instance that consistently fails requests will be temporarily marked as unavailable by the service mesh. After a backoff period, the instance will be re-tried. Circuit breakers can be configured based on a number of criteria, including the number of consecutive failures. In the event that one service fails, circuit breaking capabilities can often be included in service meshes to prevent cascading failures.
- **Multi-tenancy:** In order to maximize efficiency, it is more efficient to share infrastructure among tenants and use service mesh settings and policies to separate them. A multi-tenancy service mesh is divided into tenants with their own logical isolation, which means that the services within each tenant cannot interact with services from other tenants in the mesh. As a result, each tenant can have a separate environment within the mesh, which can

be useful for separating different organizations or applications. To allow administrators to control access to resources within the multi-tenancy service mesh, **role-based access control** (**RBAC**) is often included in multi-tenancy service meshes. To ensure that tenants do not consume excessive amounts of resources, they may also include features such as quotas and rate limits.

- **Optimized Communication:** As new services are added to an application, or new instances of existing services are added environment becomes more complicated. Service meshes capture all aspects of service-to-service communication as performance metrics as well. It is possible, for example, to collect data about how long it took for a retry to succeed in the case of a service failure by using a service mesh. As data on failure times for a given service aggregates, rules can be written to determine the optimal wait time before retrying that service, ensuring that the system does not become overburdened by unnecessary retries.
- **Improved reliability:** A service mesh can improve the overall reliability of the system by providing features such as load balancing and circuit breaking. By distributing incoming requests equally across multiple instances of a service, load balancing can help improve the system's performance and reliability by ensuring that no single instance of a service becomes overloaded.

Tools/third-party products for service mesh

There are several popular tools and third-party products available for implementing a service mesh. Some of the most popular ones are as follows:

- **Istio**: As an open-source service mesh platform with load balancing, circuit breaking, and observability capabilities, Istio is designed to be scalable and reliable, as well as supporting a variety of programming languages and platforms. The Istio service runs alongside the service as a sidecar proxy, which means that it communicates with other services in the mesh.
- **Linkerd**: Open-source Linkerd is an easy-to-use and lightweight service mesh platform. Similar to Istio, Linkerd is implemented as a sidecar proxy and provides features such as load balancing, circuit breaking, and observability. It also supports various programming languages and platforms.
- **Consul Connect**: In addition to load balancing, circuit breaking, and observability, Consul Connect is a service mesh platform from HashiCorp. Designed as a sidecar proxy, Consul Connect integrates with other HashiCorp tools, such as Consul and Nomad, as well as supporting a variety of programming languages and platforms. Consul Connect is scalable and reliable.

- **AWS App Mesh**: As a service mesh platform provided by **Amazon Web Services** (**AWS**), Amazon App Mesh offers load balancing, circuit breaking, and observability features. AWS App Mesh is a sidecar proxy implemented by Amazon Web Services and is designed to be scalable and reliable. It integrates with other AWS services, including Amazon ECS and Amazon EKS.
- **Azure Service Fabric Mesh**: In addition to load balancing and circuit breaking, Azure Service Fabric Mesh provides observability, load balancing, and circuit breaking features as part of its service mesh platform. In addition to being scalable and reliable, Azure Service Fabric Mesh is integrated with other Azure services including Azure Functions and **Azure Kubernetes Service** (**AKS**). It is implemented as a sidecar proxy.
- **Google Cloud Anthos**: As part of Google Cloud Anthos, Istio is an open-source service mesh platform that includes features such as load balancing, circuit breaking, and observability. Istio is an open-source service mesh platform. As well as supporting a wide range of programming languages and platforms, it is designed to be reliable and scalable.
- **IBM Cloud App Mesh**: It is also based on Istio, an open-source service mesh platform. IBM Cloud App Mesh is scalable and reliable and provides features such as load balancing, circuit breaking, and observability. In addition to supporting a wide range of programming languages and platforms, it integrates well with other IBM Cloud services, including IBM Cloud Kubernetes Service.
- **Google Cloud Network Service Tiers**: An integrated network architecture based on Google Cloud Network Service Tiers includes a service mesh referred to as **Google Front End** (**GFE**). GFE is a reverse proxy service that provides features including load balancing, circuit breaking, and observability as a reverse proxy service. As well as being scalable and reliable, it integrates with other Google Cloud services such as Google Kubernetes Engine.
- **Google Cloud Endpoints**: In addition to providing a service mesh for APIs, Google Cloud Endpoints also offers features such as load balancing, circuit breaking, and observability. It is designed to be highly reliable and scalable. As part of Google Cloud Endpoints, other Google Cloud services such as **Google Kubernetes Engine** (**GKE**) and Google Cloud Functions are integrated.

Istio service mesh

Let us review Istio in more detail as it is one of the most popular open-source extensible service mesh that is built on Envoy and provides teams with the ability to connect, secure, control, and observe their services using an extensible open-source

service mesh. Since Istio was open-sourced in 2017, it has been a collaboration with IBM, Google, and Lyft, who contributed to its original components as well as the Cloud Native Computing Foundation, which donated Envoy in 2017.

In this way, Istio has had time to mature and improve its feature set and now offers a wide range of features, including load balancing, traffic routing, policy creation, metrics, and service-to-service authentication. Istio consists of a set of components that work together to form a service mesh, including Envoy, Mixer, and Pilot.

- **Envoy**: The Envoy proxy is a high-performance, modern proxy which serves as a data plane for the service mesh. It routes traffic between microservices, load balances, and handles other network-related functions. Envoy is deployed as a sidecar alongside each microservice in the system, allowing it to intercept and control all network traffic to and from the service.
- **Mixer**: It is a component that provides policy enforcement and telemetry collection for the service mesh. Observability features of Mixer are also provided, including the collection of metrics and logs from the service mesh as part of observability features. It can be used to enforce access control policies, rate limits, and other types of policies on traffic flowing through the service mesh.
- **Pilot**: It is responsible for configuring Envoy proxies and providing them with routing and traffic management rules in order to provide service discovery and traffic management capabilities for the service mesh. Pilot also provides features such as canary releases and A/B testing, allowing developers to roll out new versions of their microservices with confidence.

Features of Istio

Istio is an open-source service mesh platform that provides several features to manage, secure, and observe microservices in a distributed environment. Some of the key features of Istio are:

- Secure communication between services in a cluster using TLS encryption and strong authentication and authorization based on identity
- Load balancing for HTTP, gRPC, WebSockets, and TCP traffic
- With rich routing rules, retries, failovers, and fault injection, you can control the behavior of traffic in a fine-grained manner
- Providing access controls, rate limits, and quotas through a pluggable policy layer and configuration API
- Metrics, logs, and traces are automatically generated for all traffic within a cluster, including ingress and egress

Idempotent operations

In a microservices architecture, idempotent operations can be particularly important for maintaining data consistency and preventing duplicate requests. For instance, if a client sends a request to create a new order, but the request fails due to a network error, the client may retry the request. If the order creation operation is not idempotent, this can result in duplicate orders being created in the system.

Implementing idempotency

To implement idempotency in a microservices-based system, we can assign a unique identifier to each request that the client sends. This identifier can be generated by the client or by the service handling the request. When the service receives a request with a unique identifier, it checks to see if the same request has already been processed. If the request has already been processed, the service returns the same response as the previous request. If the request has not been processed, the service processes the request and returns a response.

For example, suppose we have an e-commerce application that uses a microservices architecture. The Order Service is responsible for creating new orders. When a client sends a request to create a new order, the Order Service generates a unique identifier for the request and saves it in a database. The Order Service then processes the request and sends a response to the client. If the client retries the request with the same unique identifier, the Order Service checks the database to see if the request has already been processed. If the request has already been processed, the Order Service returns the same response as before. If the request has not been processed, the Order Service processes the request and returns a new response.

By implementing idempotency in this way, we can ensure that duplicate orders are not created in the system and that data consistency is maintained. Additionally, clients can safely retry requests without worrying about creating duplicate data in the system.

An idempotent operation is one that can be repeated multiple times without changing the result. In other words, applying it more than once will always produce the same result as applying it once.

- In distributed systems, idempotent operations can be useful, as they ensure that operations are performed only once, even if the same operation is repeated multiple times. This can ensure the integrity of the system and prevent unintended effects.
- A distributed environment often requires the use of idempotent operations in conjunction with other techniques such as pessimistic locking, versioning, and transaction processing.

- The use of idempotent operations within a microservices architecture can help to ensure that requests to microservices are handled reliably, even if they are retried due to failures or delays.
- In order to avoid unintended consequences when applying non-idempotent operations more than once, it is important to carefully design operations so that they are idempotent when appropriate. Inconsistencies may occur in data if a non-idempotent operation updates a database record multiple times if the operation is repeated.
- The ability to make an operation idempotent may not always be possible, in which case it may be necessary to use other techniques, such as transactions or versioning, to ensure consistency.

For better understanding let us see some of the examples:

- Reading data from a database or other storage system is typically idempotent, since it does not modify the data in any way.
- Data deletion: Data deletion from a database or other storage system is typically idempotent, which means that the data cannot be retrieved or modified once it has been deleted.
- The updating of data in a database or other storage system can be made idempotent by identifying each update with a unique identifier and only implementing the update if the identifier does not already exist.
- The GET, HEAD, and DELETE HTTP methods are idempotent, whereas the POST and PUT methods are not. This means that HTTP requests utilizing GET, HEAD, or DELETE methods are safe to retry, but not those using POST or PUT methods.

Conclusion

Unlocking the Power of Microservice Communication: Navigating Synchronous, Asynchronous, and Event-Driven Paradigms

In conclusion, this chapter has covered a variety of topics related to inter-service communication in microservice design. We have discussed the challenges of distributed systems, the importance of choosing the appropriate communication pattern, and the benefits of both synchronous and asynchronous communication models.

We have explored the most common synchronous communication patterns, including RESTful APIs, **remote procedure calls** (**RPCs**), and gRPC. Additionally, we

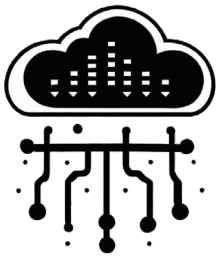
have examined the advantages and disadvantages of asynchronous communication patterns, including the use of message brokers such as RabbitMQ, Apache Kafka, IBM MQ, Azure service bus, and Amazon **Simple Queue Service** (**SQS**).

Furthermore, we have delved into the topic of event-driven communication and how it can be used to build publish-subscribe and event-driven architectures, as well as event sourcing. We have also discussed the importance of serialization and provided an overview of serialization formats, libraries, and best practices.

Finally, we have introduced the concept of a service mesh and its features, as well as some of the popular third-party tools used to implement it. We have also highlighted the benefits of Istio, an open-source service mesh platform that provides advanced traffic management, security, and observability for microservices.

Overall, by understanding the concepts and techniques discussed in this chapter, developers can design reliable, scalable, and secure inter-service communication systems that meet the specific requirements of their microservice architecture.

In the next chapter, we will explore "Event-driven data management", a pattern that allows microservices to communicate with one another and coordinate data management. Event-driven architecture is common in modern applications built with microservices and allows for communication between decoupled services. Events can be used to implement business transactions across multiple services. Each step in a transaction involves updating a business entity and publishing an event that triggers the next step.



CHAPTER 8
Event-Driven Data Management

Event-Driven Data Management for Agile Microservices

Introduction

Microservices are becoming increasingly popular as a means of constructing scalable and flexible systems, but managing data in a microservices environment can be quite challenging. By allowing each microservice to store its own data and communicate changes to that data through publishing and subscribing to events, event-driven data management can assist in addressing these challenges. Since each microservice stores its own data, each microservice can evolve independently of the others, which results in increased flexibility and scalability, as a result of event-driven data management. In addition, by using events to communicate data changes, it allows for asynchronous communication between microservices, which improves the system's performance.

It is also crucial to note that **Command Query Responsibility Segregation** (**CQRS**) is another important concept related to event-driven data management (we discussed it in Chapter 4). CQRS is a pattern for separating read and write operations into multiple microservices. Therefore, scalability and performance can be further enhanced by allowing different microservices to specialize in different types of data access.

Data governance challenges are also created by the implementation of event-driven data management. It is a must to implement data quality checks, data lineage

tracking, and data security controls across multiple microservices in order to ensure accurate, complete, and consistent data. To comply with data privacy laws and regulations, it is also necessary to implement data encryption, secure data transfer protocols, and user access controls for microservices.

In a microservices architecture, data lifecycle management is also essential to maintaining and managing data throughout its lifecycle, from creation to deletion. It is necessary to implement policies regarding data retention, data archiving, and data deletion in order to ensure that data is not retained indefinitely and is disposed of securely. In order to ensure that data is accurate, complete, and consistent, these design patterns and practices must be implemented. This will enhance the flexibility and scalability of the system.

Structure

In this chapter we will discuss following topics:

- Event-driven data management and data governance
- Technologies for event-driven data management
    - AWS Kinesis
    - Google Cloud Pub/Sub
    - Azure Event Grid
    - Apache Kafka on Kubernetes
- Event sourcing and CQRS
- Event-based data replication
- Event-driven data validation
- Event-driven data integration
- Event-based data access control
- Event-based data lineage
- Data governance in microservices
- Data privacy and compliance
- Data Lifecycle Management
- Conclusion

Objectives

By utilizing event sourcing and CQRS, Event-Driven Data Management, Event-Driven Data Replication, Event-Driven Data Validation, Event-Driven Data

Integration, Event-Driven Data Access Control, and Event-Driven Data Lineage, microservice projects can be more efficiently and effectively managed as a result.

- A microservice architecture can benefit from event sourcing and CQRS because they enable better separation of concerns and simplify data management.
- Event-based data replication ensures that all microservices are accessing the most up-to-date data in real-time.
- Event-driven data validation ensures that only valid data is processed and stored within a microservice, reducing the risk of data inconsistency.
- A data ecosystem can be created more easily with event-driven data integration, allowing seamless integration of data from multiple microservices.
- An event-based data access control system enhances security by preventing unauthorized users from accessing specific data within a microservice.
- Using event-based data lineage, it is possible to track data changes over time in a microservice, making it easier to troubleshoot and audit the data.

Ultimately, event-driven data management enables better data management, real-time data replication, data validation, data integration, data access control, and data lineage tracking, which are all essential components of microservice architecture.

Event-driven data management and data governance

Event-driven data management and data governance are essential concepts for developing and managing microservices architectures. Microservices are becoming increasingly popular for building scalable and flexible systems by breaking down monolithic architecture into small, independently deployable services. The management of data in a microservices architecture, however, can be challenging. Using the event-driven data management design pattern, microservices can store their own data and communicate changes by publishing and subscribing to events.

- **Event sourcing and CQRS**: By using a log of events instead of storing the current state of the system directly, event sourcing allows for a more flexible and scalable data management system. CQRS provides more scalable and flexible data management by separating read and write operations into separate microservices. As an example, in an e-commerce system, a microservice handles the write operations (such as updating or adding products) whereas another microservice handles the read operations (such as displaying products).

- **Event-based data replication**: Using events, data can be replicated across microservices. For instance, when a user updates their profile information in one microservice, an event is sent to the other microservices that require the same update.
- **Event-driven data validation**: A microservice managing a customer's billing information may send an event when a new credit card is added to ensure data consistency. An event handler within another microservice can validate credit card information prior to processing any new orders. For example, a microservice that handles a customer's billing information may send an event when a new credit card is added.
- **Event-driven data integration**: Data can be integrated by utilizing events from different microservices. For example, a microservice that manages a customer's profile information can send an event whenever a new address is added. When orders have not yet been shipped, an event handler in another microservice that manages the customer's orders could update the shipping address.
- **Event-based data access control**: By using events, microservices can be used to control data access and enforce data security. For instance, when a user logs in or out, the microservice that manages user authentication and authorization may send an event. It may be possible to grant or revoke access to sensitive data based on the event handler of another microservice that manages sensitive data.
- **Event-based data lineage**: In a system, data can be tracked by events, such as when data is received from external sources and an event is sent when new data is received. An event handler in another microservice that maintains a data lineage could record the origin and flow of that data through the system.
- **Data governance in microservices**: Data governance practices can be implemented as part of a microservices architecture in order to ensure the accuracy, completeness, and consistency of data. Examples include data quality checks, data lineage tracking, and data security controls.
- **Data privacy and compliance**: Data privacy laws and regulations may apply to microservices and event-driven data management. In order to ensure that sensitive data is protected and is only accessible to authorized individuals, it is necessary to implement data encryption, secure data transfer protocols, and user access controls.
- **Data lifecycle management**: Data can be managed throughout the entire lifecycle of a microservices architecture, from creation to deletion. As a means of ensuring that data is not retained longer than necessary and is securely disposed of, it may be necessary to implement policies regarding data retention, archiving, and deletion.

Technologies for event-driven data management

Event-Driven Data Management is a paradigm that allows systems to respond to events in real-time and is widely used in applications that require continuous processing of high volumes of data, such as financial trading, social media analytics, and IoT. To manage these large amounts of data, a distributed and scalable architecture is needed, which can efficiently process and store data streams from various sources.

Technologies for Event-Driven Data Management provide a range of tools and services that enable real-time processing of data streams. These technologies include messaging systems, stream processing engines, event hubs, and other related services. These technologies allow data to be ingested, processed, and stored in real-time, enabling applications to respond to events as they occur.

These technologies provide a scalable and distributed architecture for managing the flow of events, allowing for seamless communication and coordination between different components in a system. This architecture enables the efficient and reliable processing of data streams, even at high volumes, and ensures that data is processed and stored securely and reliably.

Some of the cloud native technologies used for the same are AWS Kinesis, Google Cloud Pub/Sub, Azure Event Grid, and Apache Kafka on Kubernetes.

AWS Kinesis

As a fully managed service provided by **Amazon Web Services** (**AWS**), Kinesis allows easy ingress and egress of streaming data. It can also be used for data integration, data replication, and real-time streaming. A microservice running in one region can be replicated to another microservice running in another region using Kinesis in order to ensure disaster recovery and high availability. There are three key components to the kinesis platform: kinesis data streams, kinesis data firehose, and kinesis data analytics.

- Using kinesis data streams, you can collect, process, and analyze data streams in real-time. It can handle millions of events per second, which makes it suitable for processing high-volume data streams.
- With kinesis data firehose, you can easily load streaming data into data lakes, data stores, and analytics tools.
- Kinesis data analytics provides SQL-based processing and analysis of streaming data.

Advantages

- **Scalability**: Kinesis can handle millions of events per second, making it suitable for handling high-volume data streams.
- **Reliability**: Kinesis is designed to automatically scale and replicate data across multiple availability zones, providing high availability and fault tolerance.
- **Cost-effective**: Kinesis is a pay-as-you-go service and can be cost-effective for large scale event-driven data management.

Disadvantages

- **Latency**: Kinesis has a small latency of a few seconds, which may not be suitable for some real-time streaming use cases that require sub-second latencies.
- **Limited storage**: Kinesis has a default retention period of 24 hours for data streams, and older data is automatically deleted. This can be extended to 7 days but it may not be sufficient for some use cases that require longer retention periods.
- **Limited event size**: Kinesis has a maximum event size of 1MB which may not be sufficient for some use cases that require larger event sizes.

Google Cloud Pub/Sub

A messaging service provided by **Google Cloud Platform** (**GCP**) that enables the exchange of messages between microservices in a reliable, scalable, and asynchronous manner. Real-time data replication, integration, and streaming can be performed using it. As an example, Pub/Sub can be used to enable a multi-cloud event-driven architecture between microservices running on Google Cloud Platform and those running on Amazon Web Services. Pub/Sub systems send messages to topics and receive messages from subscribers. Topics and subscribers can be managed through the GCP Console, the cloud command-line tool, or the Cloud Pub/Sub API.

Advantages

- **Scalability**: Pub/Sub can handle millions of messages per second, making it suitable for handling high-volume data streams.
- **Reliability**: Pub/Sub provides at-least-once delivery semantics and automatic retries for failed deliveries, providing high availability and fault tolerance.

- **Asynchronous**: Pub/Sub allows for asynchronous communication between microservices, which can improve system performance and scalability.
- **Multi-cloud**: Pub/Sub can be used to send messages between microservices running on GCP as well as on other cloud providers, enabling a multi-cloud event-driven architecture.
- **Flexibility**: Pub/Sub supports various message formats such as JSON, Avro, and Protocol Buffers, and allows for custom attributes on messages.

Disadvantages

- **Limited storage**: Pub/Sub has a default retention period of 7 days for messages, and older messages are automatically deleted. This retention period can be increased but it may not be sufficient for some use cases that require longer retention periods.
- **Complexity**: Setting up and managing Pub/Sub can be complex, especially for large scale, high volume deployments.

Azure event grid

Microsoft Azure's Event Grid is a fully managed service that allows users to publish and subscribe to events from various Microsoft Azure services and third-party services. This service can be used for data replication, integration, and real-time streaming based on events. A multi-cloud event-driven architecture is enabled by the ability to send events from a microservice running on Azure to another microservice running on a different cloud provider, using Event Grid. With Event Grid, you can create custom events, subscribe to built-in Azure events, and route events to various endpoints such as Azure Functions, Azure Logic Apps, or webhooks. Events are published to topics and received by subscribers following a publish-subscribe model.

Advantages

- **Reliability**: Event Grid provides at-least-once delivery semantics and automatic retries for failed deliveries, providing high availability and fault tolerance.
- **Easy to use**: Event Grid provides a simple API for sending and receiving events, and also has pre-built connectors for popular Azure services and third-party services.
- **Multi-cloud**: Event Grid allows for events to be sent and received between Azure services and services running on other cloud providers, enabling a multi-cloud event-driven architecture.

Disadvantages

- **Limited storage**: Event Grid has a default retention period of 24 hours for events, and older events are automatically deleted. This retention period can be increased but it may not be sufficient for some use cases that require longer retention periods.
- **Latency**: Event Grid has a small latency of a few seconds, which may not be suitable for some real-time streaming use cases that require sub-second latencies.
- **Limited event size**: Event Grid has a maximum event size of 64KB which may not be sufficient for some use cases that require larger event sizes.

Apache Kafka on Kubernetes

Event streaming platforms such as Apache Kafka are distributed, high-performance, and fault-tolerant. They can be used for real-time streaming, data integration, event replication, and load balancing. As a result of its easy scaling, load balancing and automatic failover, Kafka is a popular choice for event-driven data management on Kubernetes. Additionally, Kubernetes facilitates better management and monitoring of the Kafka cluster, as well as providing a platform for deploying other microservices.

Advantages

- **Scalability**: Kafka can handle millions of events per second, making it suitable for handling high-volume data streams.
- **Reliability**: Kafka provides fault-tolerance and automatic data replication, providing high availability and fault tolerance.
- **High Throughput**: Kafka can handle high-throughput and low-latency data streams.
- **Easier management and monitoring**: when deployed on Kubernetes, it allows for easier management and monitoring of the Kafka cluster.

Disadvantages

- **Limited storage**: Kafka has a default retention period for events, and older events are automatically deleted. This retention period can be increased but it may not be sufficient for some use cases that require longer retention periods.
- **Latency**: Kafka has a small latency of a few seconds, which may not be suitable for some real-time streaming use cases that require sub-second latencies.

Event sourcing and CQRS

By using event sourcing, the state of an application can be stored as a sequence of events. Each event represents a change in the state of the application, such as the creation, modification, or deletion of an object. An application log contains the events. This sequence of events can be replayed to reconstruct the application's state at any given moment.

An event-sourced system has the following characteristics:

- The state of the system is stored as a sequence of events.
- The events are stored in an append-only log, which guarantees that the events are immutable and can be replayed in the same order.
- The current state of the system is derived by replaying the events.
- The events can be used to reconstruct the state of the system at any point in time, providing a clear audit trail and historical analysis of the system state.

In a CQRS application, commands (write operations) are used to change the state of the system, while queries (read operations) are used to retrieve information.

A CQRS system has the following characteristics:

- The read and write operations are handled separately, allowing for better scalability.
- The read model is optimized for querying and retrieving data, while the write model is optimized for handling changes to the system.
- The read and write models can be implemented using different technologies and patterns, depending on the requirements of the system.
- The read and write models can be scaled independently, allowing for better resource utilization.

As a result of combining event sourcing and CQRS, an event-driven system architecture can be developed. While event sourcing provides a clear audit trail of changes to the system state, CQRS provides a method for handling reads and writes separately. In this manner, event-driven architectures can handle high-scale read and write loads while providing a clear audit trail and historical analysis of the current state of the system.

An example of how Event Sourcing and CQRS can be used together in a banking system is as follows:

The write-side of the system generates an "AmountDeposited" event when a customer deposits money into their account and stores the event in the event store.

The event contains information such as the customer's account number, the date and time of the deposit, and the deposit amount.

The read-side of the system can then use the events in the event store to build a view of the current balance of each account. This is accomplished by replaying all of the events for a particular account and updating its view of the account balance accordingly. In the case of three "AmountDeposited" events for an account, the read-side is able to add up the amounts from those events to determine the account's current balance.

In the event store, the write-side creates an "AmountWithdrawn" event when a customer withdraws money from their account. The read-side updates its view of the account balance by replaying the events for that account, including the most recent "AmountWithdrawn" event.

This example allows for complete separation of the write and read sides, which has several advantages. Rather than worrying about updating views of the data overhead, the write-side can focus on storing events in the event store quickly and efficiently. Without having to worry about the overhead of generating events, the read-side can focus on providing fast and efficient access to the current state of the data.

The combination of Event Sourcing and CQRS can lead to a robust, scalable, and maintainable banking system capable of handling large amounts of transactions, providing efficient access to account information, and providing a complete audit trail of all transactions.

Event-based data replication

Using event-based data replication, data is replicated between different systems or services by propagating events that indicate changes to the data. When data needs to be made available in multiple locations or systems, this approach is useful since it allows for real-time or near-real-time data replication. In event-based data replication, changes are captured by capturing events, such as the creation, modification, or deletion of an object, that represent changes to the data. The events are propagated to other systems and services, which consume them and update their local copies of the data accordingly.

Event-based data replication can be used in a variety of situations, including:

- **Distributed Systems**: Data needs to be replicated across multiple systems, such as a cluster of servers, to ensure high availability and fault tolerance.
- **Multi-Region**: Data needs to be replicated across multiple regions, such as different data centers or cloud providers, to ensure low latency and high availability.

- **Cross-System**: Data needs to be replicated across different systems, such as a database and a cache, to ensure consistency and high performance.
- **Microservices**: Data needs to be replicated across multiple microservices, to ensure consistency and low latency.

Advantages

- **Low Latency**: Data is replicated in near real-time, which can be useful in situations where low latency is important.
- **High Availability**: Data is replicated to multiple systems, which can increase the availability of the data.
- **Consistency**: By using events, it is possible to ensure that all systems or services are updated with the same information.
- **Flexibility**: Event-based data replication can be used with different types of systems and services, such as databases, caches, and message queues.

There are a variety of **technologies** that can be used to implement event-based data replication, depending on the requirements of the system. Some of the most common technologies include:

- **Message Queues**: It is possible to capture and propagate events using technologies such as Apache Kafka, RabbitMQ, and AWS Kinesis. These technologies provide a durable, scalable, and fault-tolerant way to handle events, and can be used to replicate data in real-time and batch in both instances.
- **Data Replication Platforms**: It is possible to capture and propagate database events using technologies such as Debezium, MySQL replication, and MongoDB replication. Data replication technologies such as these allow for the replication of data from one or more source databases to one or more target databases, and can be utilized for both real-time and batch replication applications.
- **Event-Driven Architecture Frameworks**: A variety of technologies can be used to capture and propagate events between different microservices, including Apache Camel and Spring Cloud Stream. It is possible to implement both real-time and batch data replication using these frameworks, which provide a means of implementing event-based data replication between microservices.
- **Cloud-Based Services**: Cloud providers such as Amazon Web Services, Microsoft Azure, and Google Cloud offer services for capturing and propagating events, including Amazon Kinesis, Azure Event Grid, and

Cloud Pub/Sub. It is possible to implement both real-time and batch data replication using these services. They provide both a scalable and fault-tolerant method of handling events.

It's worth noting that the choice of technology for implementing event-based data replication will depend on the requirements of the system, such as scalability, fault-tolerance, and durability.

Event-driven data validation

The event-driven data validation technique involves validating data as it is being captured, usually as an event, prior to storing or analysing it. In situations where data needs to be validated as soon as it is captured, to ensure that it meets certain constraints or standards, this approach can be used, as it enables real-time or near-real-time data validation. As a result of event-driven data validation, events that represent changes to data, for example, the creation, modification, or deletion of objects, are recorded. These events are then passed through a series of validation checks, which are designed to ensure that the data meets certain constraints or standards. If the data passes all validation checks, it will be stored or processed. If the data fails one or more validation checks, an error message will be generated or the data is rejected. Event-driven data validation can be used in a variety of situations, including:

- **Data Quality**: Ensuring that data meets certain standards of quality, such as completeness, accuracy, and consistency.
- **Business Rules**: Ensuring that data meets certain business rules, such as constraints on data values or relationships between data elements.
- **Compliance**: Ensuring that data meets certain compliance requirements, such as data privacy and security regulations.

Advantages

- **Low Latency**: Data is validated in near real-time, which can be useful in situations where low latency is important.
- **High Data Quality**: By validating data as soon as it is captured, it is possible to ensure that the data meets certain standards of quality.
- **Compliance**: By validating data as soon as it is captured, it is possible to ensure that the data meets certain compliance requirements.
- **Flexibility**: Event-driven data validation can be used with different types of systems and services, such as databases, message queues, and microservices.

The validation process can be implemented with a variety of technologies such as rule engines, data quality tools or simple regex validation checks.

Event-driven data integration

As part of event-driven data integration, changes to the data are propagated between different systems and services through the propagation of events. In situations where data has to be made available in multiple locations or systems, or where data needs to be transformed before it can be stored or processed, this approach can be useful for real-time or near-real-time data integration. By capturing events that represent changes in the data, such as the creation, modification, or deletion of an object, event-driven data integration works. An integration step is then used to transform, validate, or route this data to the appropriate target system or service based on a series of integration steps.

Event-driven data integration can be used in a variety of situations, including:

- **Data Synchronization**: Synchronizing data between different systems, such as a CRM system and an ERP system, to ensure consistency and completeness.
- **Data Transformation**: Transforming data between different formats, such as JSON and XML, to ensure compatibility and efficiency.
- **Data Consolidation**: Consolidating data from multiple sources, such as different databases or microservices, to provide a single view of the data.

Advantages

- **Low Latency**: Data is integrated in near real-time, which can be useful in situations where low latency is important.
- **High Data Quality**: By integrating data as soon as it is captured, it is possible to ensure that the data meets certain standards of quality, such as consistency and completeness.
- **Flexibility**: Event-driven data integration can be used with different types of systems and services, such as databases, message queues, and microservices.
- **Real-time integration**: It allows to integrate data in real-time as soon as the event is captured.

The integration process can be implemented with a variety of technologies such as **enterprise service bus** (**ESB**), data integration platforms or event-driven architecture frameworks.

Event-based data access control

Using event-based data access control, the system controls access to data by capturing events that occur during the processing of data. In situations where unauthorized access to data or the restriction of access is required depending on the circumstances of an event, this approach is suited to providing real-time or near-real-time access control. An event-based data access control system captures events that represent data changes, such as the creation, modification, or deletion of objects, which are used to control access to data. In order to ensure that the user or system requesting access to data is authorized to do so, these events are then passed through a series of access control checks. Data access is granted if the user or system is authorized. Data access is denied if the user or system is not authorized.

Event-based data access control can be used in a variety of situations, including:

- **Data Security**: Ensuring that data is protected from unauthorized access, such as by encrypting the data or by restricting access to the data to certain users or systems.
- **Data Privacy**: Ensuring that data is protected from unauthorized access, such as by masking the data or by restricting access to the data to certain users or systems.
- **Compliance**: Ensuring that data access is compliant with certain regulations, such as data privacy and security regulations.

Advantages

- **Low Latency**: Access control is applied in near real-time, which can be useful in situations where low latency is important.
- **High Data Security**: By controlling access to data as soon as it is captured, it is possible to ensure that the data is protected from unauthorized access.
- **Flexibility**: Event-based data access control can be used with different types of systems and services, such as databases, message queues, and microservices.
- **Real-time control**: It allows to control access to data in real-time as soon as the event is captured.

Event-based data access control can be implemented with a variety of technologies such as **identity and access management** (**IAM**) systems, access.

Event-based data lineage

Using event-based data lineage, data's lineage or history can be tracked as events are generated by the system in real-time or near-real-time. In situations where data needs to be traced back to its origins, or where data must be audited for compliance or quality assurance purposes, this approach provides real-time or near-real-time lineage tracking. As part of event-based data lineage, data changes, including the creation, modification, or deletion of a data object, are recorded. Based on these events, a lineage graph is constructed. This graph illustrates the relationships between the different versions of the data and the systems or processes that were responsible for creating or transforming the data. The lineage graph can be used to trace the origin of data, to understand how data has been transformed over time, or to identify the systems or processes that have been involved in the data's lifecycle. Event-based data lineage can be used in a variety of situations, including:

- **Data Governance**: Ensuring that data is properly managed and controlled, such as by tracking the origin of data and understanding how it has been transformed over time.
- **Compliance**: Ensuring that data is compliant with certain regulations, such as data privacy and security regulations, by tracking the origin of data and understanding how it has been transformed over time.
- **Data Quality**: Ensuring that data is of high quality, such as by tracking the origin of data and understanding how it has been transformed over time.

Advantages

- **Low Latency**: Lineage is captured in near real-time, which can be useful in situations where low latency is important.
- **High Data Traceability**: By capturing lineage as soon as it is generated, it is possible to ensure that data is traceable back to its origin.
- **Flexibility**: Event-based data lineage can be used with different types of systems and services, such as databases, message queues, and microservices.
- **Real-time tracking**: It allows to track the data lineage in real-time as soon as the event is captured.

Event-based data lineage can be implemented with a variety of technologies such as data governance platforms, data catalogs, data lineage tools, and event-driven architecture frameworks.

Data governance in microservices

In microservices, data governance is defined as a set of policies, procedures, and standards that ensure that data is properly managed and controlled within a microservice architecture. Data quality, data security, data privacy, data compliance, and data lineage are examples of data quality issues. Data governance in microservices is difficult because each microservice operates independently and has its own storage and processing capabilities. In this situation, it can be challenging to ensure that data is consistent across all microservices and that data is being used in a secure and compliant manner.

To address this challenge, organizations may implement a number of different strategies, such as:

- **Data Catalogs**: A centralized data catalog can be used to document the data that is being used by each microservice, including information such as data schema, data lineage, and data usage policies.
- **Data Governance Platforms**: Platforms such as Collibra and Informatica can be used to manage data governance policies, procedures, and standards across all microservices.
- **Data Quality and Data Governance Microservices**: These microservices can be used to ensure that data is of high quality and that data is being used in a compliant and secure manner.
- **Event-Driven Data Governance**: Event-driven data governance can be used to track the lineage of data and to ensure that data is being used in a compliant and secure manner.
- **Identity and Access Management (IAM)**: IAM systems can be used to ensure that only authorized users and systems have access to data, regardless of which microservice they are interacting with.

Additionally, organizations may also need to implement appropriate data security and data privacy measures in order to ensure that their data is protected from unauthorized access and that it is used in a compliant manner.

Overall, the goal of data governance in microservices is to ensure that data is properly managed and controlled across all microservices, regardless of where the data is stored or how it is utilized. With the implementation of appropriate data governance strategies and technologies, organizations may be able to ensure that their data is of the highest quality, that they are using the data in a compliant and secure manner, and that their data can be traced easily throughout their lifecycle.

Data privacy and compliance

Compliance with data privacy laws, regulations, and guidelines refers to the set of laws, regulations, and guidelines that organizations must adhere to in order to protect personal information of individuals and to ensure that the data is used in an ethical and lawful manner.

Some of the key data privacy and compliance regulations include:

- **General Data Protection Regulation** (**GDPR**): This regulation applies to organizations operating in the European Union and governs the collection, storage, and use of personal data.
- **California Consumer Privacy Act** (**CCPA**): This regulation applies to organizations operating in California and governs the collection, storage, and use of personal data.
- **Health Insurance Portability and Accountability Act** (**HIPAA**): This regulation applies to organizations handling personal health information and governs the collection, storage, and use of personal data.
- The **Payment Card Industry Data Security Standard** (**PCI DSS**): This regulation applies to organizations handling credit card information and governs the collection, storage, and use of personal data.

To ensure compliance with these regulations, organizations must implement appropriate data privacy and security measures, such as:

- **Data Encryption**: Data should be encrypted both at rest and in transit to protect against unauthorized access.
- **Data Access Control**: Access to personal data should be restricted to only those who need it to perform their job.
- **Data Governance**: Organizations should implement data governance policies, procedures, and standards to ensure that data is being used in a compliant and secure manner.
- **Data Privacy Microservices**: Microservices can be used to ensure that data is being used in a compliant and secure manner.
- **Data Privacy Compliance Tools**: Tools such as OneTrust and TrustArc can be used to automate compliance with data privacy regulations.
- **Data Privacy and Compliance Training**: Organizations should ensure that all employees are trained on data privacy and compliance regulations and best practices.

The organization should also implement a data breach response plan in addition to these strategies, as well as perform regular audits to ensure compliance with data privacy and security regulations. It is imperative that organizations take the necessary steps to ensure they are complying with all laws and regulations governing the collection, storage, and use of personal data, since data privacy and compliance are critical issues for organizations. It is possible for organizations to protect the personal information of individuals and use data in an ethical and lawful manner by implementing appropriate data privacy and security measures.

Data Lifecycle Management

Organizations use **data lifecycle management** (**DLM**) to manage the entire lifecycle of their data, from creation to deletion, through the use of policies, procedures, and technologies. The goal of DLM is to ensure that data is effectively managed, protected, and used in a manner that supports the organization's business objectives.

The key stages of the data lifecycle include:

- **Creation**: Data is created through various means such as data entry, data collection, data import, and data generation.
- **Processing**: Data is processed, transformed, and analyzed to extract insights and value.
- **Storage**: Data is stored in various forms such as databases, data lakes, and data warehouses.
- **Backup and Recovery**: Data is backed up to ensure that it can be recovered in the event of a disaster or data loss.
- **Archiving**: Data that is no longer required for day-to-day operations is archived, either for long-term retention or for compliance purposes.
- **Deletion**: Data that is no longer needed is deleted in order to save storage space and to protect against data breaches.

DLM involves implementing policies and procedures to govern how data is created, stored, processed, and deleted, as well as technologies that support data management, such as data catalogues, data governance platforms, data archiving tools, and data deletion tools.

Advantages

- **Data Quality**: DLM ensures that data is of high quality and that it is accurate, complete, and consistent.

- **Data Governance**: DLM ensures that data is being used in a compliant and secure manner.
- **Data Privacy**: DLM ensures that personal data is protected and that it is being used in an ethical and lawful manner.
- **Data Efficiency**: DLM helps organizations to use data more efficiently by reducing storage costs, improving performance and reducing data duplication.
- **Data Compliance**: DLM ensures that organizations are in compliance with data privacy and security regulations.

As part of DLM implementation, many teams and stakeholders are involved, including IT, data science, data governance, legal, and compliance. A comprehensive understanding of the organization's data management objectives and the technical, legal, and compliance requirements is required. It is imperative that organizations utilize DLM to ensure that their data is of high quality, compliant, secure, and efficient throughout its entire lifecycle, from creation to deletion. As a result, organizations are able to maximize the value of their data and satisfy internal and external stakeholders' needs.

Conclusion

Unleashing the Potential of Event-Driven Data:

Driving Efficiency, Real-Time Insights, and Seamless Integration

Overall, learning about Event-Driven Data Management will assist you in developing a better understanding of data management techniques such as Event-driven data sourcing, CQRS, event-based data replication, event-based data validation, event-driven data integration, event-based data access control, and event-based data lineage, among others. As a result of these techniques, data can be managed more efficiently and effectively in a microservice architecture. Several technologies, including AWS Kinesis, Google Cloud Pub/Sub, and Azure Event Grid, can be used to implement Event-Driven Data Management, providing event-driven data streaming, event-driven data integration, and event routing. With these technologies, you can manage event-driven data, handle data streams in real-time, handle data integration, control data access, and track data lineage, all of which are vital aspects of developing modern applications.

In the next chapter, 'Microservices without Servers: The Serverless Approach', we will discuss the serverless approach to building microservices. This approach utilizes cloud-based platforms such as AWS Lambda, Google Cloud Functions, or

Azure Functions to build and deploy microservices. By using this approach, cost savings are possible, automatic scaling is possible, and integration with other cloud-based services is easy. However, it is also subject to limitations such as cold starts and limitations on memory and compute resources, which can negatively impact performance and restrict the size and complexity of microservices.

Chapter 9
The Serverless Approach

Revolutionizing Microservices with Serverless Architecture

Introduction

The adoption and growth of cloud computing will certainly continue to accelerate. As companies move beyond their initial forays into cloud computing and have established its benefits, they are increasingly seeking additional applications. The market for serverless platforms is growing rapidly, including AWS Lambda, Azure Functions, IBM Cloud Functions, and Google Cloud Functions.

The term "serverless" refers to a cloud execution model that facilitates the development and implementation of cloud-native applications more efficiently and cost-effectively.

There is a true pay-as-you-go service offered by functions-as-a-service, which means the infrastructure scales transparently in response to the needs of the application. In serverless computing, the service provider takes care of all the infrastructure (server-side IT), so all you need to do is write code. Technically, servers remain involved, however. As opposed to traditional enterprise architectures, serverless computing does not rely on servers, but rather on how they are implemented and managed.

Top benefits of serverless computing:

- **No infrastructure management**: Serverless platforms allow you to deploy your code and have it run in high availability.

- **Dynamic Scalability**: Serverless computing allows for dynamic scaling up and down of the infrastructure to match the demands of any workload within seconds.
- **Faster time to market**: The use of serverless applications reduces the operations dependencies on each development cycle, allowing teams to deliver more functionality in less time.
- **More efficient use of resources**: Organizations can reduce their total cost of ownership by implementing serverless technologies.
- **Flexibility**: With serverless computing, developers can easily create and deploy microservices without having to worry about infrastructure management.
- **Security**: Security measures such as IAM, encryption, network security, etc., can be implemented by the cloud provider to ensure the security of sensitive data and systems.

With the rise of cloud computing and the need to build and deploy applications in a more cost-effective and scalable manner, serverless computing has gained tremendous popularity in recent years. Cloud computing in which the cloud provider manages the infrastructure and dynamically allocates resources as required for the execution of code is known as serverless computing. This model involves the developer writing code and uploading it to the cloud, which then executes the code as soon as a specific event or trigger occurs. By doing so, the developer does not need to manage or allocate server resources, providing a cost-effective and highly scalable solution.

Structure

In this chapter we will discuss following topics:

- Understanding the serverless architecture
- Serverless framework
- Function-as-a-Service (FaaS) platforms
    - AWS Lambda
    - Azure functions
    - Google cloud functions
- Serverless approach and edge computing
- Serverless monitoring and logging
- Serverless security

- Best Practices for serverless microservices development
- Serverless microservices case studies
- Conclusion

Objectives

The objective of this chapter is to build modern, scalable, and cost-effective cloud applications. Architects and Developers must understand the serverless computing paradigm, including the Serverless Architecture, the Serverless Framework, and FaaS platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions. In addition to Serverless Monitoring and Logging, Serverless Security, and Best Practices for Serverless Microservices Development, these topics are essential. With serverless computing, developers can focus on developing business logic rather than worrying about the underlying infrastructure, resulting in reduced operational overhead, increased scalability, and reduced cost. Increasing application performance and reducing latency are also significant benefits of the Serverless approach and Edge Computing. Insights into how these concepts can be effectively applied in practice can be gained from real-world Serverless microservices case studies. Ultimately, understanding these topics will allow developers to create efficient, secure, and highly available cloud applications that can adapt to the changing demands and workloads of the cloud.

Understanding the serverless architecture

Using the serverless architecture, the cloud provider manages the underlying infrastructure and dynamically allocates resources according to code execution requirements. As part of this architecture, the developer writes code and uploads it to the cloud, which executes it in response to specific triggers or events. Consequently, the developer does not need to manage or allocate server resources, resulting in an extremely cost-effective and highly scalable solution.

The key components of a serverless architecture are:

- **Functions**: Serverless architecture is based upon functions, small, independent units of code that are executed when specific events or triggers occur. Functions can be written in a variety of programming languages and deployed and executed in the cloud.
- **Event triggers**: Cloud providers execute functions in response to specific events or triggers, such as HTTP requests, messages sent to queues, timers, and the like. When an event occurs, the associated function processes the event and returns its results.

- **API Gateway**: An API gateway is used to expose the functions to the internet, allowing external clients to invoke them. The API gateway functions as a front-end for the functions, handling authentication, authorization, and traffic management. In addition to caching, logging, and monitoring, it also provides other features.
- **Cloud infrastructure**: Cloud providers manage the underlying infrastructure, which includes servers, storage, databases, and the like. They also dynamically allocate resources so that end-users experience high availability and low latency.
- **Database Services**: In serverless architectures, data is stored and retrieved using a variety of database services, including NoSQL databases, SQL databases, and key-value stores. As a result, developers can focus on writing code rather than managing databases since these services are typically fully managed by the cloud provider.
- **Message queues**: The use of message queues in a serverless architecture allows functions to process events asynchronously. Functions can receive events from a message queue, process them, and send the results to other functions or databases.
- **Monitoring and Logging**: This architecture provides developers with a variety of tools for monitoring and logging, including performance metrics, error reporting, and debugging. These tools assist them in identifying and resolving issues with their functions.
- **Security**: Cloud providers are responsible for the security of the infrastructure. Security measures such as IAM, encryption, network security, etc., can be implemented to ensure that sensitive data and systems are protected.

Use cases for Function-as-a-Service (FaaS)

- **Dynamic Web Applications**: The Serverless Framework can be used to build dynamic web applications that require the ability to handle varying levels of user traffic. By deploying and executing functions based on user requests, infrastructure management and scaling are not necessary. In order to reduce the costs and complexity associated with managing infrastructure, serverless functions could be deployed to handle authentication and authorization, for example.
- **Event-driven applications**: As an event-driven framework, the Serverless Framework is ideally suited to the development of applications that process data from multiple sources, such as chatbots or IoT applications. Functions can be triggered by events, process the data, and return the results, allowing

them to scale dynamically based on demand. Serverless functions, for example, could be deployed to process data from a chatbot, providing users with real-time insights and recommendations.

- **Microservices**: The Serverless Framework provides a natural fit for microservices-based applications, where functions can be deployed and executed as independent units of code. By managing and deploying microservices in this manner, developers are able to concentrate on providing value to their customers rather than managing and deploying them. An image processing function, for example, might be handled by a serverless function, reducing infrastructure management costs and complexity.
- **Backend services**: By using the Serverless Framework, backend services such as authentication, authorization, image processing, and data processing can be built without the need for server management. Functions can be triggered by events, such as a user request, and return the results to the client. This reduces the cost and complexity of managing infrastructure, while allowing for dynamic scalability as demand changes.
- **Batch processing**: In order to reduce the cost and complexity of managing infrastructure, the Serverless Framework can be utilized for batch processing tasks such as data extraction and analysis. Functions may be triggered by a timer, process data, and return the results, reducing infrastructure management costs. Organizations are able to process large amounts of data efficiently and cost-effectively using the Serverless Framework because batch processing can be scaled dynamically.

Therefore, serverless architecture and Function-as-a-Service offer an affordable and scalable means of building and deploying applications. They are well suited for dynamic web applications, event-driven applications, microservice-based applications, backend services, and batch processing tasks. Organizations can reduce costs, improve scalability, and deliver value to their customers by leveraging the benefits of serverless computing.

Serverless framework

Using the Serverless Framework, developers can create, deploy, and manage microservice-based applications using cloud platforms, including Amazon Web Services, Microsoft Azure, and Google Cloud, which are all open-source. As a result, developers can easily build and deploy serverless applications using this tool set. It provides tools and abstractions for building and deploying microservices.

A Serverless Framework allows developers to define microservices as individual functions, which are then automatically deployed to the cloud. This allows developers to focus on writing code while the framework takes care of the underlying

infrastructure. As part of the Serverless Framework, tools for managing cloud resources, including Lambda functions and DynamoDB tables, can also be provided, allowing resources to be managed centralized and automated. By providing detailed logs and metrics, the framework allows for real-time monitoring of performance and status of microservices, as well as identifying and resolving problems.

Key features

Some of the key features of the serverless framework:

- **Microservice support**: Serverless framework provides native support for creating, deploying, and managing microservices. It makes it easier for developers to create, deploy, and manage microservices as individual functions.
- **Event-driven design**: The serverless framework supports event-driven design, which is well suited to microservices architecture. In order to build dynamic, event-driven applications, each microservice can be triggered by an event, such as a user request or data update, and executed in response.
- **Decentralized deployment**: Developers can deploy and manage microservices independently using the Serverless Framework, allowing them to update and scale microservices based on their needs.
- **Built-in templates**: A number of common use cases can be handled with the Serverless Framework, including building REST APIs, processing data, and facilitating authentication and authorization. As a result, developers can easily begin building microservices without having to invest time and effort in writing complex code.
- **Customizable plugins**: The serverless framework provides a range of customizable plugins that can be used to extend its functionality and support custom use cases. By doing so, it is possible to extend the capabilities of the framework to meet specific requirements by adding new functionality, such as security and monitoring, to microservices.
- **Debugging and testing**: By providing tools for debugging and testing microservices, the Serverless Framework simplifies the identification and resolution of problems. With this solution, microservices can be tested locally before being deployed to the cloud, and they can be debugged in real time with the built-in debugger.
- **Cost optimization**: A serverless framework enables organizations to reduce infrastructure costs and improve the performance and reliability of microservices by scaling microservices dynamically based on demand.

- **Cross-platform support**: Using the Serverless Framework, it is possible to build and deploy microservices to a wide variety of cloud platforms, including Amazon Web Services, Microsoft Azure, and Google Cloud.
- **Integrations**: With the Serverless Framework, a variety of popular tools and services can be integrated, including AWS Lambda, AWS S3, and AWS DynamoDB. By leveraging the full capabilities of the cloud platform, microservices can be easily integrated with existing tools and services.
- **Automated deployment**: Using the Serverless Framework, developers can easily deploy microservices to the cloud using automated deployment. Using the framework, microservices can be deployed via a simple command-line interface, thereby automating the deployment process and reducing manual effort.
- **Resource management**: Cloud resources, such as Lambda functions and DynamoDB tables, can be managed using the Serverless Framework. As a result, resources can be managed in an automated and centralized manner, reducing the need for manual effort and improving consistency.
- **Monitoring and logging**: It is possible to monitor and log the performance and status of microservices in real time with the serverless framework, which offers monitoring and logging capabilities. The framework provides detailed logs and metrics, which can be used to identify and resolve issues, as well as to improve the performance and reliability of microservices.
- **Support for multiple languages**: This framework supports multiple programming languages, including JavaScript, Python, TypeScript, and more, making it possible to build microservices based on the language of your choice and take advantage of all the cloud platform's features.

Function-as-a-Service platforms

The **Function-as-a-Service** (**FaaS**) platform is a cloud-based computing platform that enables developers to create and deploy microservices in a serverless environment through the use of cloud computing. They provide a way to execute code in response to events, such as HTTP requests, changes in database records, or the arrival of messages in a queue.

For example, let's outline the process of architecting a **Function-as-a-Service** (**FaaS**) platform for a travel booking website. This example will focus on the key steps involved in designing the platform architecture, irrespective of the specific cloud provider (AWS, Azure, or GCP).

- **Identify Functionality**: Determine the key functions or microservices required for the travel booking website, such as user registration, flight search, hotel booking, payment processing, and email notifications.

- **Function Design**: Define the logic and functionality for each function. For example, the flight search function would handle search queries, interact with the flight database, and return relevant results.
- **Function Packaging**: Package each function along with its dependencies and configurations into deployable units. This may involve using containerization technologies (for example Docker) or function packaging tools provided by the FaaS platform.
- **Event Triggers**: Identify the events that will trigger function execution. In the case of the travel booking website, event sources can include HTTP requests from users, database updates for new bookings, or scheduled tasks for sending email notifications.
- **Function Invocation**: Configure the event sources to trigger the corresponding functions. For example, an HTTP request to the flight search endpoint should invoke the flight search function.
- **Scaling and Resource Allocation**: Configure the FaaS platform to automatically scale the functions based on incoming workload. This ensures that the platform can handle a high volume of requests during peak booking periods.
- **Execution Environment**: Specify the runtime environment for each function, ensuring that it includes the necessary resources (CPU, memory, and so on) and supports the programming language used for function development.
- **Function Execution and Monitoring**: Implement the function execution logic within the FaaS platform. The platform will execute the functions, handle input data (for example user search criteria), process the logic, and capture outputs (for example search results). Implement monitoring and logging mechanisms to track function execution and performance.
- **Integration with External Services**: Integrate the functions with external services required for the travel booking website, such as databases for storing bookings, payment gateways for processing transactions, and email services for sending notifications. Utilize the FaaS platform's integration capabilities or APIs to interact with these services.
- **Billing and Cost Management**: Set up the FaaS platform's billing and cost management features to track the resources consumed by the functions. This ensures efficient resource allocation and helps manage costs associated with function execution.
- **Developer Tools and SDKs**: Leverage the developer tools, SDKs, and CLIs provided by the FaaS platform to facilitate local testing, debugging, and

deployment automation. These tools streamline the development process and enhance developer productivity.

By following this process, you can architect a FaaS platform for a travel booking website, enabling functions to handle various tasks and integrating them seamlessly with external services.

AWS Lambda

Amazon Web Services (**AWS**) Lambda is a serverless computing service that enables developers to run code based on events, such as database changes, messages arriving in a queue, or HTTP requests. Lambda automatically manages the underlying infrastructure, so developers need not worry about servers, virtual machines, or operating systems.

As demand for a service increases, AWS Lambda can automatically allocate additional resources to handle the increased load. This eliminates the need for developers to manage capacity or provision new servers. It is also possible to build and deploy complex applications using AWS Lambda, which is integrated with other AWS services, such as Amazon S3, Amazon API Gateway, and Amazon DynamoDB.

Features of AWS Lambda

- **Event-driven computing**: By triggering a function in response to the upload of a new image to an Amazon S3 bucket, AWS Lambda can be used as an example of event-driven computing. After performing image processing, the function can store the processed image back in the S3 bucket, including resizing and watermarking.
- **Microservices architecture**: Microservice architectures can be built using AWS Lambda by creating separate functions for each service, for example. For instance, a function can be created for handling user authentication, another for handling image processing, and another for handling payment processing. This allows for greater scalability and resiliency, as well as making it easier to deploy and manage individual functions.
- **High availability**: It is possible to achieve high availability using AWS Lambda by deploying multiple functions across multiple Availability Zones. AWS Lambda automatically routes traffic in the event of a failure in a particular Availability Zone to healthy functions in other zones, ensuring that the application continues to operate.
- **Monitoring and logging**: Using Amazon CloudWatch Logs to monitor the logs generated by Lambda functions is an example of how AWS Lambda can be used for monitoring and logging. In addition to providing insight into

how the application is functioning, this can assist in identifying performance issues and debugging problems.

- **Scalability**: The use of automatic scaling policies with AWS Lambda can be used to ensure that the appropriate amount of resources are always available as an example of how AWS Lambda can be used for scalability. When the number of function invocations increases, Amazon Web Services Lambda can automatically provide additional resources to meet the increased demand.
- **Cost optimization**: As an example of how AWS Lambda can be utilized for cost optimization, factors such as frequency and duration of function invocations, as well as the cost of storing and transferring data can be considered. For instance, if a function is only invoked infrequently, it can be configured to use less resources, reducing costs.

Advantages of AWS Lambda

- **Cost-effective**: AWS Lambda only charges per second of compute time consumed by an application, making it a cost-effective way to run infrequently used applications.
- **Easy integration**: Using AWS Lambda, you can build complex applications without the need to manage multiple services.
- **Focus on code**: The AWS Lambda service eliminates the need for developers to manage servers, virtual machines, or operating systems, enabling them to concentrate on writing code instead.

Disadvantages of AWS Lambda

- **Cold starts**: A cold start can occur when an application has not been used for an extended period of time. This can increase latency.
- **Limited resources**: AWS Lambda provides limited resources such as memory and CPU, which may affect the performance of an application.
- **Debugging challenges**: Due to the dynamic nature of the underlying infrastructure, debugging serverless applications can be more challenging than debugging traditional applications.

The AWS Lambda service provides a convenient method of building and running applications through a cost-effective, scalable, and flexible serverless computing service. In spite of some challenges, including cold starts and limited resources, AWS Lambda is an extremely popular choice for organizations seeking to build and deploy serverless applications.

Azure functions

In Azure functions, developers can build and run event-driven applications and microservices without having to manage infrastructure, enabling them to focus on writing code while Azure handles the underlying infrastructure, scaling, and maintenance. It is a Serverless compute service provided by Microsoft Azure. The Azure Functions programming language supports a wide range of programming languages, including C#, Java, JavaScript, and Python. As a result, it is accessible to a wide range of developers. During periods of high traffic, the service automatically scales up and down to handle the load, so organizations do not have to worry about managing infrastructure during peak periods.

With Azure functions, customers only pay for the resources they use, making it a cost-effective option for organizations of all sizes. Deployment is easy and can be performed using a variety of tools and processes, including Azure DevOps and Git. In addition, Azure Functions can also be integrated into other Azure services, such as Event Grid, Azure Cosmos DB, and Azure Storage, enabling complex, multi-tier applications to be developed. In terms of monitoring and logging, Azure functions provides a rich set of logging and monitoring tools for monitoring and troubleshooting applications. Additionally, the service provides automatic diagnostics and logging, capturing performance metrics and system events.

Features of Azure functions

- **Event-driven computing**: The Azure functions can be triggered by a variety of events, such as data changes, messages, and API calls, making it an excellent choice for event-driven applications.
- **Language support**: As Azure functions supports a wide range of programming languages, including C#, Java, JavaScript, and Python, it is accessible to a wide variety of developers.
- **Automatic scaling**: During periods of high traffic, Azure functions automatically scales up or down to handle the load, so organizations do not have to worry about managing infrastructure.
- **Pay-per-use pricing**: In addition, Azure Functions provides a cost-effective solution for organizations of all sizes, since customers are only charged for the resources they consume.
- **Easy deployment**: It is easy to deploy and can be completed using a variety of tools and processes, such as Azure DevOps and Git.
- **Integration with Azure services**: With Azure functions, you can build complex, multi-tier applications that integrate with other Azure services such as Event Grid, Azure Cosmos DB, and Azure Storage.

- **Logging and monitoring**: The Azure functions platform offers a rich set of logging and monitoring tools that enable you to track the health of your applications. These tools enable you to troubleshoot any problems that may arise.
- **Security**: The Azure functions platform provides a secure environment, with features such as role-based access control, security certificates, and encryption.
- **Serverless computing**: Azure functions is a great choice for organizations seeking to reduce operational overhead and improve productivity by building and running event-driven applications and microservices without managing infrastructure.
- **Increased productivity**: In addition to eliminating the need to manage infrastructure, Azure functions allows developers to focus on developing their applications instead of managing infrastructure.

Disadvantages of AWS Lambda

- **Limitations on customizing the underlying infrastructure**: Due to the fact that Azure Functions uses a shared infrastructure, customers cannot customize the underlying infrastructure.
- **Cold starts**: It is possible for Azure Functions to undergo a cold start, where the function must be started from a state of dormancy. This can result in a delay in processing requests.
- **Performance**: An infrastructure's performance can be affected by the number of functions running on it and the size of the resource allocation.

The Azure functions service allows organizations to develop and run event-driven applications and microservices easily using Serverless compute services. For organizations seeking to reduce operational overhead and improve productivity, it is an excellent choice due to its cost-effective pricing, ease of deployment, and scalability.

Google cloud functions

Cloud functions, offered by Google Cloud, is a serverless computing platform. Without managing the underlying infrastructure, developers can execute their code upon specific events, such as the update of a database or the creation of a new file. As soon as developers upload their code to Google Cloud functions, the platform automatically provisioned the necessary resources to run the code, scaling up or down as required. By eliminating the need to manage servers, developers are able to focus their attention on developing their applications rather than managing servers.

As Google Cloud functions supports a variety of programming languages, including JavaScript, Python, and Go, it can be used to perform a variety of tasks, including back-end processing and microservice development. Additionally, the platform integrates with other Google Cloud services, such as Google Cloud Storage, Google BigQuery, and Google Cloud Pub/Sub, enabling developers to develop more powerful applications. Google Cloud Functions is an effective solution for developing and running serverless applications in the cloud that is flexible, scalable, and cost-effective.

Serverless approach and edge computing

The landscape of cloud computing is constantly evolving, and two notable advancements have emerged: serverless computing and edge computing. Serverless architecture allows developers to focus solely on writing code without worrying about infrastructure management, while edge computing brings computation and data storage closer to the network edge for improved performance. Combining these two paradigms brings forth a logical progression that enables even more efficient and responsive application architectures.

A Perfect Match: Microservices and Serverless at the Edge:

Microservices, an architectural pattern consisting of small, independent, and loosely coupled services, align perfectly with serverless computing at the edge. Breaking down applications into smaller, modular services simplifies deployment and management. Each microservice can be deployed as a serverless function, ensuring efficient resource utilization and scalability. This approach enables organizations to distribute workloads across multiple edge nodes, enhancing performance and responsiveness.

Serverless approaches can benefit edge computing:

- **Resource Optimization**: Due to the limitations of the edge devices, edge computing resources such as processing power, memory, and storage are limited. By allocating only the resources necessary for a specific task and releasing them once the task has been completed, a serverless approach can help optimize these resources. The result is a reduction in waste and improved resource utilization, making edge computing applications easier to deploy and manage.
- **Event-Driven Processing**: Designed for event-driven processing, serverless computing is an ideal solution for edge computing scenarios requiring real-time processing of data. By deploying edge computing in **Internet of Things** (**IoT**) deployments, data collected from sensors can be processed and acted upon in real time without having to be transmitted to a central data center.

- **Easy Deployment**: Serverless computing facilitates the deployment and management of edge computing applications. Developers are able to deploy their code to the serverless platform and the platform will handle the scaling, resource allocation, and security for them. By doing so, organizations can deploy and manage edge computing solutions more easily, even for non-technical users.
- **Cost Effective**: In contrast to charging for reserved resources, serverless computing is designed to be cost-effective as it only charges for the resources that are used during specific tasks. By doing so, small-scale edge computing deployments that do not require a large amount of resources all the time can result in significant cost savings.
- **Improved Resilience**: Considering that edge computing devices can often be deployed in remote or difficult-to-access locations, it is imperative that they are deployed in a resilient manner. By providing automatic failover and disaster recovery capabilities, serverless approaches can enhance edge computing deployment resilience, ensuring that edge computing deployments remain operational even in the event of a failure.

For example, how serverless edge computing can benefit traffic management systems:

This scenario involves a manufacturing company deploying a system that can provide real-time monitoring of its machines and predictive maintenance alerts. As part of its performance monitoring systems, the company has installed a network of sensors and devices on its machines. Edge devices process and analyze this data to provide predictive maintenance alerts and determine if any maintenance is necessary. This system can be deployed using a serverless approach, which reduces infrastructure costs and maintenance requirements for the company. The sensor data can be processed in real-time using a serverless computing platform that allocates resources only when necessary and releases them once the processing has been completed. As a result, deployment and management costs are reduced, and the processing occurs in real-time, without any delays. Furthermore, a serverless computing platform ensures that the system is always available and secure, even if the company does not possess in-house technical expertise.

In this case, the serverless approach has helped the manufacturing company deploy a cost-effective and scalable industrial IoT solution that allows it to monitor the performance of its machines in real-time and to provide predictive maintenance alerts. As a result, the company has been able to improve its maintenance processes, reduce downtime, and improve its overall operating efficiency.

Serverless monitoring and logging

Some of the most popular options for monitoring and logging serverless applications include Amazon CloudWatch, Microsoft Azure Monitor, and Google Cloud Monitoring. We briefly discussed some of the other options as well.

- **Amazon CloudWatch**: Specifically designed to monitor and log serverless applications and resources, Amazon CloudWatch is a monitoring and logging service provided by **Amazon Web Services** (**AWS**). It is possible to monitor the performance and health of your serverless applications, including AWS Lambda functions, Amazon API Gateway APIs, and AWS App Runner applications, in real-time with Amazon CloudWatch. For future analysis and troubleshooting, you can also log and store your application logs, metrics, and traces in CloudWatch.

Monitor and log serverless events using CloudWatch:

- **Real-time Monitoring**: Your serverless applications will be monitored in real-time by CloudWatch, allowing you to quickly identify and respond to problems as they arise.
- **Log Management**: You can store and analyze all your serverless application logs in one place with CloudWatch, a centralized log management solution.
- **Alarm Management**: By setting up alerts and notifications based on predefined metrics and conditions, CloudWatch provides alarm management capabilities.
- **Metrics Collection**: By collecting and storing metrics from your serverless applications and resources, CloudWatch allows you to monitor the performance and health of your applications over time.
- **Trace Collection**: You can easily debug and optimize your serverless applications using CloudWatch by collecting and storing traces.

In addition to being cost-effective and easy-to-use, CloudWatch is a popular solution for organizations deploying serverless solutions on Amazon Web Services.

Microsoft Azure Monitor: As a monitoring and logging service provided by Microsoft Azure, Microsoft Azure Monitor facilitates the monitoring and troubleshooting of cloud-based applications and services by organizations. Performance and health indicators of Azure resources, including Azure functions, a serverless computing platform, are provided by Azure Monitor.

Serverless monitoring and logging is provided by Azure monitor

- **Real-time Monitoring**: Real-time monitoring of your serverless applications allows you to quickly identify and resolve issues.
- **Log Management**: Serverless application logs can be analyzed and stored in Azure Monitor centralized log management solution.
- **Alarm Management**: Using Azure Monitor, you can set up alerts and notifications based on predefined metrics and conditions.
- **Metrics Collection**: Monitor the performance and health of your serverless applications with Azure Monitor, which collects and stores metrics.
- **Trace Collection**: By collecting and storing traces from your serverless applications, Azure Monitor makes it easy to debug and optimize your applications.
- **Integration with Azure Monitor for Containers**: Using Azure Monitor for Containers, you can monitor and troubleshoot containerized workloads, including serverless functions. By integrating Azure Monitor with Azure Monitor, you can monitor and troubleshoot your serverless applications and containers in one place.
- **Google Cloud Monitoring**: Monitor and troubleshoot cloud-based applications and services with Google Cloud Monitoring, a monitoring and logging service offered by **Google Cloud Platform** (**GCP**). Using Google Cloud Monitoring, you can analyze the performance and health of GCP resources, including Cloud Functions.

Serverless monitoring and logging is provided by Google Cloud Monitoring:

- **Real-time Monitoring**: Monitoring your serverless applications in real time allows you to identify and resolve issues quickly.
- **Log Management**: Serverless application logs can be stored and analyzed in one place using Google Cloud Monitoring.
- **Alarm Management**: With Google Cloud Monitoring, you can set up alerts and notifications based on predefined metrics.
- **Metrics Collection**: With Google Cloud Monitoring, you can monitor the performance and health of your serverless applications and resources.
- **Trace Collection**: You can easily debug and optimize your serverless applications with Google Cloud Monitoring, which collects and stores traces.

Other popular options

- **New Relic**: Serverless applications and resources are monitored, logged, and alerted in real time with New Relic. For organizations deploying serverless solutions across multiple cloud platforms, New Relic supports a wide range of services and platforms.
- **Thundra**: Serverless observability platform Thundra monitors, logs, and debugs serverless applications in real time. By providing a detailed view of serverless functions and underlying infrastructure, Thundra makes troubleshooting and optimizing serverless applications easier.
- **Serverless Framework Dashboard**: Serverless Framework Dashboard provides monitoring and logging for serverless applications. Organizations deploying serverless solutions with the Serverless Framework prefer Dashboard because it provides real-time monitoring, logging, and alert management for serverless functions.

Serverless security

The advent of serverless computing has introduced new security challenges that must be addressed in order to ensure the protection of applications and data. Here are some best practices and key considerations for securing serverless applications:

- **Identity and Access Management (IAM):** In the context of serverless computing, IAM is the process of controlling access to serverless resources, such as functions and APIs. AWS IAM, for example, provides numerous features that enable users to manage their access to AWS services, including Lambda. The Lambda function can only be accessible to specific IAM users or roles by creating an IAM policy.
- **Encryption**: Ensure the confidentiality and privacy of sensitive data by encrypting it. As an example, when using **Amazon Web Services** (**AWS**) S3 for data storage, you can enable server-side encryption for all objects stored in the bucket. In this manner, all data stored in S3 is encrypted while at rest.
- **Network Security**: Implementing network security measures such as firewalls and virtual private networks is necessary to protect serverless applications from network-based threats. With Amazon **Virtual Private Cloud** (**VPC**), you can create a secure network environment for your serverless applications that prevents unauthorized access to your data and resources.
- **Input Validation**: The input validation process involves checking the data received by a serverless function before processing it. This prevents attacks such as SQL injection and cross-cutting. For example, when creating a REST

API using AWS API Gateway, input validation can be accomplished using request validation rules, which are defined in the API Gateway configuration.

- **Resource Access Control**: In order to prevent unauthorized access to other resources, such as databases and APIs, it is crucial to implement appropriate access control policies and network segmentation for serverless functions. You can create a security group for Amazon RDS for database storage that only allows access to the database from specific IP addresses, such as those used by serverless functions, when using Amazon RDS for database storage.
- **Monitoring and Logging**: Monitoring and logging are critical components of a serverless application security strategy. It is also possible to use CloudWatch to store logs from serverless functions so that real-time detection and response to security incidents can be accomplished. For example, using Amazon CloudWatch, you can set up alerts and notifications to monitor the security and health of your serverless applications.
- **Third-Party Components**: The functionality of serverless applications is often extended by third-party components, such as libraries and plugins. As a result, it is necessary to assess the security of these components and implement appropriate controls to mitigate potential security risks. For example, when using a third-party library in your serverless function, you should verify the security of the library and monitor for updates and vulnerabilities.

Ultimately, serverless applications must be secured with a combination of appropriate security controls and best practices, such as IAM, encryption, network security, input validation, resource access control, monitoring and logging, and third-party components. By adhering to these best practices, you can ensure the security of your serverless applications and prevent potential security threats from occurring. The next chapter will cover cloud-related security in detail.

Best practices for serverless microservices development

In recent years, serverless computing has become increasingly popular, providing organizations with a flexible and scalable platform for developing and deploying applications. Serverless computing has the advantage of enabling the development and deployment of microservices, which are independent, modular components of an application that can be developed independently. We will discuss some of the best practices for developing serverless microservices in this section.

- **Start Small**: As a first step toward serverless microservices implementation, it is recommended that you begin small and gradually scale up as needed.

This allows you to experiment with different configurations and technologies, and to make any necessary changes without having to make major changes to your entire application.

- **Focus on Functionality**: In order for serverless microservices to be successful, they should be focused on delivering specific functionalities. Each microservice should have a well-defined purpose and should be designed to address a specific problem.
- **Use Event-Driven Architecture**: The serverless computing architecture is based on an event-driven architecture, where functions are triggered by events, such as a request for a REST API or a message in a message queue. It is important to keep this in mind when designing serverless microservices, and to design each microservice to respond to specific conditions.
- **Implement Monitoring and Logging**: Monitoring and logging are critical components of any application, including serverless microservices. You can monitor the health and performance of your microservices in real time using tools such as Amazon CloudWatch, Microsoft Azure Monitor, or Google Cloud Monitoring.
- **Use a Configuration Management Tool**: It can be challenging to manage the configuration of multiple microservices when your application grows. Using a configuration management tool, such as AWS Systems Manager or HashiCorp Terraform, can enable you to manage and automate the configuration of your microservices.
- **Implement Security Best Practices**: As with any other application, serverless microservices are susceptible to security risks. In order to protect against potential security threats, it is imperative to implement appropriate security controls, including IAM, encryption, network security, and input validation.
- **Use a CI/CD Pipeline**: A continuous integration and continuous deployment pipeline, such as AWS CodePipeline or Jenkins, can be used to automate the process of building, testing, and deploying your serverless microservices.
- **Design for Resilience**: Serverless computing is based on a pay-per-execution model, which means each function is charged based on how often it is executed. For this reason, microservices must be designed to be resilient to failures, such as by using retry logic.
- **Optimize Cold Starts**: Serverless microservice performance can be significantly impacted by cold starts, which occur when a function is invoked for the first time after a period of inactivity. It is possible to minimize cold start times by provisioning more memory, prewarming instances, and optimizing your code to minimize cold start times.

- **Use Managed Services**: The serverless computing platforms, including AWS, Azure, and Google Cloud, provide a variety of managed services that can be used to construct serverless microservices. As an example, you can reduce the operational overhead of your microservices by using managed databases, such as AWS DynamoDB or Azure Cosmos DB, or managed message queues, such as AWS SQS or Azure Service Bus.
- **Test and Debug Efficiently**: Debugging serverless microservices can be challenging, as it requires you to recreate the execution environment for each function in order to resolve the issue. As an alternative to using AWS SAM Local or Azure Functions Core Tools to simulate the execution environment locally, you can use these tools. Additionally, it is vital that you implement robust testing practices, including unit testing, integration testing, and end-to-end testing, to ensure that your microservices function as they should.
- **Consider Cost Optimization**: It is essential to consider cost optimization when developing serverless microservices because serverless computing operates on a pay-per-execution basis. By reducing memory sizes, implementing autoscaling, and monitoring usage patterns, you can identify and eliminate unnecessary usage in order to reduce costs.

Serverless microservices case studies

These case studies showcase the benefits of using serverless microservices, including improved scalability, reduced infrastructure costs, and faster time to market.

- **Netflix**: The Netflix service processes and distributes video content in real-time using serverless microservices. AWS Lambda functions are used to process and transcode video content in real-time, triggered by events such as the start of a video by a subscriber. As a result of the platform scaling automatically, video content is delivered smoothly and without interruptions, as the functions are also automatically scaled according to demand. Even during periods of high demand, Netflix is able to provide subscribers with a high-quality streaming experience.
- **Capital One**: In the Capital One credit decision engine, machine learning algorithms are used to make credit decisions triggered by events, such as a customer applying for a loan. These algorithms are constructed using serverless technologies such as AWS Lambda and API Gateway. During periods of high demand, the engine can process billions of requests per month, as the functions are automatically scaled by the platform to meet demand. Capital One is able to provide real-time credit decisions to its customers, improving the customer experience and reducing processing times for loan applications.

- **The New York Times**: AWS Lambda is used to develop the content management system for the New York Times, which processes incoming news articles and images, stores the content in a database, and makes it available for publication. In addition to its scalability, the CMS is highly secure, with data encrypted at rest and in transit, and it is highly secure, with the platform automatically managing the scaling of functions based on demand. As a result, The New York Times is able to manage a large volume of content and ensure that the content is protected and secure, while reducing operational overhead and costs associated with managing the CMS.
- **Amazon Alexa**: Using serverless microservices, Amazon Alexa processes voice commands and provides relevant information to users via voice commands. The system is designed to respond to voice commands in real-time using AWS Lambda and other serverless technologies. As soon as a user makes a voice request, such as asking for the weather or setting a reminder, a Lambda function processes the request and provides the user with the pertinent information. Also, the platform scales the functions to meet demand, ensuring that Alexa can handle a large number of requests even during periods of high demand.

It is evident from these examples that organizations can create and deploy cost-effective, scalable, and secure applications using serverless microservices. By breaking down complex systems into smaller, independent functions, serverless microservices provide organizations with the opportunity to focus on delivering business value, while the platform manages and scales the underlying infrastructure.

Conclusion

From Infrastructure Management to Business Innovation: Driving Success with Serverless Microservices

In conclusion, the serverless approach has emerged as a popular way to build and deploy microservices. This is because of its numerous benefits, including reduced operational overhead, improved scalability, and cost savings. As a result of serverless microservices, organizations are free to focus on delivering business value without having to manage the underlying infrastructure. In the event of high demand, the platform is capable of scaling and managing the infrastructure to ensure that the microservices remain available and performant.

Additionally, serverless microservices give organizations the option of quickly and easily modifying and scaling their services as needed without the need for manual intervention. By responding quickly to changing business requirements,

organizations can ensure that their applications remain relevant and valuable to their customers while maintaining relevance and value.

'Security by Design for Microservices', our next chapter, will address the construction and deployment of microservices. As microservices and the cloud become increasingly popular, it is important to implement security measures that protect against potential vulnerabilities and threats. Security in the cloud is the primary concern of top IT executives. Among the most rapidly growing technologies, cloud computing offers a number of advantages, including flexibility, cost savings, and accessibility. Cloud computing, however, is associated with a number of security concerns, and this is the main reason why enterprises hesitate to move to the cloud.

Chapter 10
Cloud Microservices - Security by Design

Security-First Approach to Cloud-Native Microservices

Introduction

Microservices have become an increasingly popular method of developing and deploying applications over the past few years. Microservices provide greater flexibility and scalability, making them an attractive option for organizations looking to streamline their IT infrastructures.

The benefits of microservices, however, come with new security challenges. Microservices are designed as distributed systems, so a vulnerability in one microservice could compromise the entire system as a whole. Additionally, since microservices are typically deployed in cloud environments, they may be vulnerable to a variety of external threats.

In order to address these challenges, it is imperative to adopt a "security by design" approach to building and deploying microservices. This means incorporating security into the design of the system from the beginning, rather than adding it as an afterthought.

Key principles of security by design for cloud microservices:

- **Authentication and Authorization**: The purpose of authentication is to verify that a user or system is who they claim to be. The purpose of authorization is to determine what actions the user or system is authorized to perform. To ensure that only authorized users and systems can access

your microservices, you should use strong authentication mechanisms such as multi-factor authentication, OAuth, or JWT tokens. As part of this process, access control policies are implemented in order to determine what actions each user or system is authorized to perform, such as reading, writing, or deleting data.

- **Encryption**: The process of encryption involves transforming data into a secure code in order to protect it from unauthorized access. For microservices, encryption is important to protect data in transit and at rest. To ensure that all data transmitted between microservices is encrypted, use HTTPS for communication between them. Furthermore, ensure that sensitive data is encrypted before it is stored in a database, and that encryption keys are securely stored.
- **Monitoring and Logging**: Implement monitoring tools to track system performance, resource usage, and security events. Monitoring and logging are crucial for detecting potential security breaches and identifying the root cause of any problems that may arise. These tools can include real-time monitoring of log files, as well as automated alerts for suspicious activity. Ensure that logs are securely stored, as well as implementing a retention policy to ensure that logs are retained for a period that is appropriate.
- **Patching and Updating**: It is critical that you maintain the security of your microservices architecture by keeping all software and systems up-to-date with the latest security patches and updates. Implement a patch management process to ensure that all updates are applied in a timely manner, not just the microservices themselves, but the underlying infrastructure and operating system as well.
- **Separation of Concerns:** By separating your microservices into multiple layers, each of which serves a specific purpose and function, you can minimize the impact of a security breach on the rest of the system. This involves breaking down the functionality of your system into smaller, more manageable pieces, followed by the implementation of appropriate security controls at each layer. Furthermore, this prevents unauthorized access to critical parts of the system.
- **Testing:** It is crucial to conduct rigorous security testing on your microservices in order to identify potential weaknesses and vulnerabilities. Automated testing can be performed using tools such as penetration testing, vulnerability scanning, and fuzz testing; manual testing may include code reviews, threat modelling, and red team/blue team exercises. To identify any potential weaknesses in how the microservices interact with each other, it is important to test not only individual microservices, but also the entire system.

It is possible to ensure that your microservices architecture is secure and resistant to potential attacks by incorporating these principles into the architecture. Remember, security is not a one-time effort, but is a continuous process that requires constant attention and adaptation to changing threats.

Structure

In this chapter we will discuss following topics:

- Cloud Microservices - Security by Design
- Authentication and Access Control
    - Authentication and authorization mechanisms in cloud microservices
    - Role-based access control (RBAC)
    - Multi-factor authentication (MFA)
    - Access control lists (ACLs)
- Communication Security
- Data Security
    - Data security and encryption techniques for microservices
    - Security of data in transit and at rest
    - Immutable infrastructure
- Container Security
- Monitoring and Incident Response
- Compliance and Risk Management
    - Compliance and regulatory considerations
    - Threat modelling
    - Penetration testing
- Infrastructure Security
- Threat Detection and Response
- Continuous Security Monitoring
- Conclusion

Objectives

The objective of learning **"Cloud Microservices - Security by Design"** covering authentication and access control, communication security, data security, container

security, monitoring and incident response, compliance and risk management is to develop an in-depth understanding of security principles and practices, which are essential in designing and implementing secure microservice architectures.

The specific objectives of this chapter is to equip you with and understanding of Security aspect including:

- Knowledge about different types of security threats and risks associated with cloud microservices.
- Learning about the best practices for implementing authentication and access control mechanisms for microservices.
- Understanding the importance of communication security and how to implement secure communication channels between microservices.
- Gaining knowledge about different data security techniques and how to protect sensitive data in microservices.
- Learning about the best practices for monitoring and incident response in microservices to detect and respond to security incidents in real-time.
- Understanding the regulatory compliance requirements and risk management strategies for cloud microservices.

As a result of learning "Cloud Microservices - Security by Design", learners will acquire the knowledge and skills necessary to design and implement secure microservice architectures in the cloud while ensuring compliance with regulatory standards and managing security risks effectively.

Cloud Microservices - Security by Design

In today's fast-paced and dynamic business environment, microservices-based cloud applications have become increasingly popular due to their scalability, agility, and cost-effectiveness. However, the distributed nature of microservices architecture and the use of cloud services introduce new security challenges that need to be addressed to ensure the confidentiality, integrity, and availability of the system. In this context, authentication and access control, communication security, data security, container security, monitoring and incident response, compliance and risk management, infrastructure security, threat detection and response, and continuous security monitoring are crucial components of a comprehensive security strategy for microservices-based cloud applications. In this chapter, we will discuss each of these components in detail and highlight the security mechanisms and best practices that can be used to secure a microservices-based cloud application. For example, as illustrated in *Figure 10.1*, *Cloud Microservices - Security by Design*.

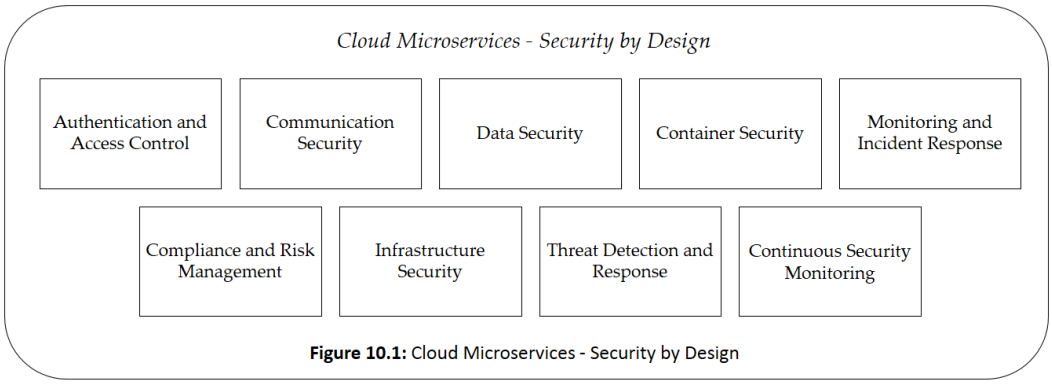


Figure 10.1: Cloud Microservices - Security by Design

Authentication and access control

An important aspect of cloud microservices security is authentication and access control. Microservices architecture involves dividing a large application into smaller, independent services that can communicate with each other. A microservice can have its own authentication and access control system.

Authentication and authorization mechanisms in cloud microservices

Cloud microservices require authentication and authorization mechanisms to verify the identity of users and systems and to control access to resources. Here are some common mechanisms used for authentication and authorization:

- **OAuth**: For cloud microservices, OAuth is a widely used authentication and authorization protocol. By providing access tokens to authorized users, OAuth allows users to grant access to their resources to third-party applications without sharing their credentials. As part of OAuth, users are provided with access tokens that can be used to access protected resources.
- **OpenID Connect**: This authentication protocol is based on the OpenID protocol, which was developed on top of OAuth. Through OpenID Connect, users can log in to applications using their existing social media or email accounts. Applications can use OpenID Connect to verify users' identities and obtain basic information about their profiles.
- **JSON Web Tokens (JWT)**: A JWT is a secure means of transferring claims between parties. JWTs are self-contained tokens that contain information about the user or system and are digitally signed to ensure their authenticity. Frequently, cloud microservices use JWTs to implement authorization and access control policies.

- **SAML:** Security Assertion Markup Language (SAML) is an XML-based standard for exchanging authentication and authorization information between parties. By using SAML, users are able to access multiple applications without having to log in to each application separately. SAML is commonly used in enterprise environments to provide **single sign-on** (**SSO**) functionality.

Authentication and authorization mechanisms should be selected based on the specific requirements of your application as well as the resources that you want to protect. To ensure the effectiveness of your authentication and authorization mechanisms over time, it is important to select one that is secure, intuitive, and interoperable. Additionally, it is important to properly configure and maintain your authentication and authorization mechanisms.

Role-based access control (RBAC)

Among the most widely used methods of access control in cloud microservices is RBAC, which is a method for controlling access to resources based on the roles assigned to users within an organization or system. In RBAC, users are assigned roles, and each role is assigned a set of permissions which determine what actions they are permitted to perform.

Each role in an RBAC system is defined by a set of permissions that allow users to access specific resources and perform specific actions. As an example, a developer can be assigned the role of "developer," which allows them to access the development environment and modify code, while a tester can be assigned the role of "tester," which enables them to access the testing environment and execute test cases.

RBAC has several benefits for cloud microservices:

- **Simplifies Access Control Management**: By defining permissions based on roles rather than individual users, RBAC simplifies access control management. As a result, access control policies are less administratively burdensome, and users are able to ensure that they have access to the resources they need to do their jobs more easily.
- **Enforces Least Privilege**: By ensuring that users have only the permissions they need to perform their job responsibilities, RBAC enforces the principle of least privilege. As a result, accidental or intentional data breaches caused by users accessing resources they are not permitted to access are minimized.
- **Increases Security**: It prevents unauthorized access and protects against insider threats by ensuring that only authorized users have access to resources.

The implementation of RBAC requires careful planning and design to ensure that the roles and permissions are appropriate for the specific use case you are implementing. To ensure that your RBAC policies remain effective in preventing unauthorized access to your resources, it is important to review and update them regularly.

For example, imagine that you have a microservices application that provides access to customer data. The application has the following roles:

- An administrator has full access to all customer data, has the ability to create and modify user accounts, and can change access control policies.
- Managers have access to customer data for the customers they manage, and they can create and modify user accounts for the employees they supervise.
- A salesperson has access to customer data for the customers they are assigned to, and can create and modify their own user accounts.
- Staff members can access customer data in order to provide customer support and can create and modify their own user accounts.

The permissions associated with each role describe what actions the user is permitted to perform. For example, the Salesperson role may have the following permissions:

- View customer data for customers they are assigned to.
- Edit customer data for customers they are assigned to.
- Create and view their own user account.

As part of RBAC implementation, each user is assigned to one or more roles. For example, a salesperson might be assigned to the salesperson role, while an administrator might be assigned to the administrator role. Depending on the roles and permissions associated with those roles, access control policies can be defined. For example, a policy may permit salespeople to view and edit customer data for the customers with which they are assigned, while denying them access to data for all other customers.

The role-based access control system simplifies management and reduces error risk by allowing administrators to manage access control policies based on roles rather than individual users. Additionally, RBAC ensures that users have only the permissions they need to perform their job responsibilities, reducing the risk of accidental or intentional data breaches.

Multi-factor authentication (MFA)

MFA is a security mechanism used in cloud microservices to prevent unauthorized access to resources. It requires users to provide two or more forms of authentication before granting access to one system, application, or data.

The most common types of factors used in MFA are:

- Something the user knows (such as a password, PIN, or answer to a security question)
- Something the user has (such as a smart card, token, or mobile device)
- Something the user is (such as a biometric, such as a fingerprint or facial recognition)

There are several types of attacks that can occur against a password, so MFA provides an additional layer of security. By requiring multiple factors, MFA makes it more difficult for an attacker to gain access to a system or application, even if one factor has been compromised.

Here are some of the benefits of using MFA in cloud microservices:

- **Improved Security**: Multiple forms of authentication improve security by making it harder for attackers to gain unauthorized access to resources.
- **Reduced Risk of Credential Theft**: Even if an attacker obtains a user's password, they would still need the other authentication factor to gain access, so MFA can reduce the risk of credential theft.
- **Compliance:** To ensure sensitive data is protected, many compliance frameworks and regulations require the use of **multi-factor authentication** (**MFA**).

Cloud microservices can be provided with a variety of MFA solutions, including **time-based one-time passwords** (**TOTPs**), SMS codes, and push notifications. When choosing an MFA solution, you need to ensure it meets your security requirements and integrates well with your cloud microservices platform.

In addition, it is important to note that MFA is not a silver bullet and should be used in conjunction with other security measures, such as RBAC and regular security audits, in order to provide a comprehensive security program.

Access control lists (ACLs)

The use of ACLs in cloud microservices allows users or groups to control access to resources based on their identities or the groups they belong to. The ACL is used to specify which users or groups are permitted to perform specific actions on a resource, such as reading, writing, or executing.

Each entry in an ACL specifies the type of access (read, write, execute, and the like), the user or group, and whether the permission is granted or denied.

In contrast to traditional file permissions, which allow only read, write, and execute permissions on files and directories, ACLs provide a more fine-grained approach to access control. With ACLs, you can specify permissions for specific users or groups, allowing for more granular control over access to resources.

Here are some of the benefits of using ACLs in cloud microservices:

- **Improved Security**: By providing a finer-grained access control mechanism, ACLs increase security by reducing the possibility of unauthorized access to data.
- **Flexibility**: Access control policies can be more easily defined with ACLs, since you can specify permissions for specific users or groups.
- **Simplified Management**: With ACLs, you can delegate control over specific resources to other users or groups rather than relying on a single administrator to manage everything.

Communication security

Security protocols should be implemented in cloud microservices to ensure communication security and protect sensitive information from unauthorized access. Here are some best practices for implementing security protocols in cloud microservices:

- **Use Transport Layer Security (TLS) for encryption**: TLSS is a protocol for encrypting network communications. Using TLS can help prevent man-in-the-middle attacks, eavesdropping, and data tampering by encrypting data transmitted between a client and a server. TLS can be implemented at an application level in a microservice architecture using tools such as OpenSSL or Let's Encrypt. As an example, you can build a secure server using TLS using the https module in a microservices-based application built with Node.js.
- **Implement authentication and authorization**: The authentication and authorization of microservices are crucial components of microservices security. Authentication verifies the identity of users and microservices while authorization determines which actions can be performed by a user or microservice. For example, a popular approach for implementing authentication and authorization in microservices is to use OAuth 2.0 with **JSON Web Tokens** (**JWT**). JWT is a compact, URL-safe way to represent claims to be transferred between two parties. Spring Security and Spring Cloud Security can be used to implement OAuth 2.0 with JWT in a microservices-based application built using Spring Boot.

- **Use mutual SSL authentication**: As a security technique, mutual SSL authentication, also known as two-way SSL authentication, requires that both the server and client authenticate each other through SSL/TLS certificates. It provides an additional level of security, ensuring that only authorized clients can access the server and only authorized servers can access the client. Mutual SSL authentication can be implemented using certificates. The `Microsoft.AspNetCore.Authentication.` Certificate package can be used to implement mutual SSL authentication in a microservices-based application built using ASP.NET Core.
- **Implement rate limiting**: The purpose of rate limiting is to limit the number of requests a client may make to a server within a given period of time. In addition to preventing denial-of-service attacks, rate limiting can also improve the application's performance and reliability. In a microservices-based application built on Kubernetes, you may use `Istio` to implement rate limiting at different levels, such as at the application level or the API gateway level.
- **Use virtual private networks (VPNs)**: It is a secure and private network connection that enables remote users or microservices to connect to a private network as if they were directly connected to it. VPNs are a method of establishing a secure, private network connection over a public network using a number of tools, including OpenVPN and WireGuard. OpenVPN can be used to create a VPN between microservices in a Docker-based application. It contributes to the security of communication between microservices and prevents eavesdropping and data tampering.
- **Secure Shell (SSH)**: SSH provides encryption, data integrity, and authentication for establishing secure shell connections between a client and a server. In a microservices-based application, SSH can be used for establishing secure communication channels between microservices.
- **Service Mesh**: Service mesh is a dedicated infrastructure layer for managing service-to-service communication in a microservices-based application. As well as load balancing, service discovery, and traffic management, it also offers security features such as mutual TLS authentication, rate limiting, and access controls. Istio, Linkerd, and Consul are popular implementations of service mesh.
- **JSON Web Tokens (JWT)**: A popular approach to authentication and authorization within microservices is the use of JWT. JWT is a compact, URL-safe method of expressing claims that can be transferred between parties. In a microservices-based application, JWT can be used to establish secure communication channels between microservices, without the need for session state. JWT can be used to securely transmit authentication and authorization information between microservices.

- **Network Segmentation**: Using network segmentation, microservices can be isolated and protected from potential attacks by dividing a network into smaller subnetworks. As part of network segmentation, **virtual LANs** (**VLANs**), **virtual private networks** (**VPNs**), or **software-defined networking** (**SDN**) can be used.

Data security

Here are some key data security measures to consider when working with cloud-based microservices:

Data security and encryption techniques for microservices

As part of the implementation of security protocols in cloud microservices, data security is an essential component. A microservice-based application usually exchanges and processes sensitive data, which must be protected from unauthorized access and data breaches. The following techniques are commonly used to secure data in microservices-based applications:

- **Encryption**: As part of encryption, data is encoded in a way that only authorized parties can access it. Encryption can be used to secure data both at rest (when it is stored) and during transit (while it is being transmitted). In microservices-based applications, there are several encryption techniques that are used, including symmetric key encryption, asymmetric key encryption, and hashing. Several encryption tools, including OpenSSL and Bcrypt, can be used to implement encryption in microservices.
- **Digital Signatures**: In order to ensure data integrity and authenticity, digital signatures are used. A digital signature is a mathematical method for verifying the authenticity of digital messages or documents. By implementing digital signatures in microservices, you can ensure that data has not been tampered with during transmission or storage. Tools such as GnuPG and OpenSSL can be used to implement digital signatures.
- **Access Control:** Using access control, data and services are restricted based on the identity or role of the user. You can implement access control at the application level by using tools such as OAuth or OpenID Connect, or you can implement it at the infrastructure level by using tools such as **Amazon Web Services** (**AWS**) **Identity and Access Management** (**IAM**).
- **Tokenization**: As part of tokenization, sensitive data is replaced with non-sensitive data called tokens, which can then be used in place of the sensitive data, reducing the risk of data breach. The tokenization process can be used for

protecting sensitive information such as credit card numbers, social security numbers, and other personally identifiable information. Microservices can be implemented using tools such as Stripe and CyberSource.

- **Masking**: As part of masking, sensitive data is replaced with non-sensitive data of the same type, but without any relationship to the original data. In storage or transmission, masking can protect sensitive data. In order to implement masking in microservices, tools such as Oracle Data Masking and IBM InfoSphere Optim can be utilized. Masking can be used to protect data such as credit card numbers, social security numbers, and other personally identifiable information.

Security of data in transit and at rest

Securing data at rest (in storage) and data in transit (during transmission) are both critical aspects of securing data in microservices-based applications. Here are some common techniques used to secure data at rest and data in transit:

- **Security of data Data at Rest**: Data at rest refers to data that is stored on a physical device, such as a hard drive, solid-state drive, or USB drive. This data is vulnerable to unauthorized access, theft, and damage, so it is essential to ensure its security using below methods:

- **Encryption**: By encoding the data in such a manner that only authorized parties can access it, encryption is one of the most commonly used techniques for protecting data at rest. There are two types of encryption: symmetric key encryption and asymmetric key encryption. During symmetric key encryption, the same key is used to encrypt and decrypt the data, whereas for asymmetric key encryption, different keys are used. A microservices-based application might also utilize a hashing technique, which is a one-way process that converts data into a fixed-size string of characters. A hashing technique is commonly used to store passwords and other sensitive information. AWS **Key Management Service** (**KMS**), for example, provides an easy-to-use and secure method for managing encryption keys used to secure data at rest. As well as supporting symmetric and asymmetric key encryption, it provides a range of key management functions, including rotation and versioning of keys.

- **Access Control:** By using tools such as OAuth or OpenID Connect, access control can be implemented at the application level to restrict access to data and services based on the user's identity or role. In order to implement access control at the infrastructure level, tools such as **Amazon Web Services** (**AWS**) **Identity and Access Management** (**IAM**) or Google Cloud Identity and Access Management can be used. It is possible to implement access control in a microservices-based application that stores sensitive data using

OAuth, which allows users to grant third-party applications access to their data without sharing their login credentials, as an example. A user could, for example, use OAuth to allow a financial management application to access their bank account information without sharing their banking login credentials.

- **Redundancy**: A redundancy system ensures that data will not be lost in the event of a system failure through the replication of data across multiple servers or storage devices. AWS offers a number of services that can be used to implement data redundancy in microservices-based applications, including Amazon S3 for storing and retrieving files, Amazon EBS for storing blocks, and Amazon RDS for storing databases.
- **Security of data Data in Transit**: Data in transit refers to data that is being transmitted from one location to another over a network, such as the internet or a local network. This type of data is vulnerable to interception, modification, and theft, so it is important to take steps to secure it using below methods:
- **Transport Layer Security (TLS)**: This protocol is used to secure communication over the internet by encrypting data during transmission. As an example, Let's Encrypt provides TLS certificates for websites and applications. Let's Encrypt can be used to encrypt data during transmission to secure communication between microservices.
- **Virtual Private Network (VPN)**: Using a VPN, users can securely connect to a private network over the internet. VPNs are also useful for protecting data during transmission by encrypting it. For instance, OpenVPN is a free, open-source VPN that can be used to establish a secure connection between microservices. This VPN uses TLS encryption to protect communication between microservices.
- **Secure Sockets Layer (SSL)**: SSL was a deprecated protocol for securing communication over the internet. SSL encrypts data during transmission in order to protect data in transit. As an example, OpenSSL is a free, open-source toolkit that can be used to implement SSL encryption in microservice-based applications. It offers a range of cryptographic features, including encryption, decryption, digital signatures, and hashes.

Immutable infrastructure

By making the infrastructure immutable, which means that it cannot be altered after deployment, immutable infrastructure improves the security of microservices-based applications. By doing so, unauthorized changes to the system are prevented, the attack surface is reduced, and any changes to the infrastructure are tracked and intentional.

Whenever a new version of a microservice is deployed, a new server or container is created to host that version in an immutable infrastructure model. Once the old server or container has been destroyed, any changes made to the old service version are then discarded. This ensures that the system remains in a known and secure state, and reduces the risk of vulnerabilities being introduced through changes made to the infrastructure.

The following are examples of techniques and tools used to implement immutable infrastructure in microservice-based applications:

- **Containerization**: The container is a lightweight, portable, and self-contained environment that may be used to host microservices. By deploying a new container for each new version of a microservice, containers can be used to implement immutable infrastructure, which ensures that the infrastructure remains unchanged after deployment. Containerization platforms such as Docker are popular for implementing immutable infrastructure in microservices-based applications.
- **Configuration Management**: In addition to automating the deployment of infrastructure components, such as servers and databases, configuration management tools can be used to ensure the integrity and consistency of the infrastructure. By automating the deployment of new servers or containers for each new version of the microservice, configuration management tools can be utilized to implement immutable infrastructure. As an example, Ansible can be used to implement immutable infrastructure in microservice-based applications using configuration management.
- **Infrastructure-as-Code (IaC)**: Managing infrastructure components using code is known as Infrastructure-as-Code. Immutable infrastructure can be implemented using IaC by defining the desired state of infrastructure as code and automatically creating new infrastructure components when changes are made. For example, Terraform is a popular tool for implementing immutable infrastructure in microservices-based applications.

Container security

Microservices-based applications require container security, which is a crucial aspect of securing them. Containers provide isolation and portability, which makes them an ideal choice for hosting microservices. However, containers can also introduce security challenges, including vulnerabilities in container images, insecure image registry systems, and runtime security issues. Some examples of techniques and tools for securing containers in microservices-based applications are presented following:

- **Secure Image Creation**: To create secure container images, it is important to start with a minimal and secure base image, remove all unnecessary

components, and verify that all necessary software components are up-to-date and do not have known vulnerabilities. The container image must also be constructed in a secure environment to prevent tampering. For example, Docker Content Trust can be used to guarantee that container images are signed and verified in order to prevent tampering.

- **Image Registry Security**: The purpose of image registries is to store container images and make them available to the container runtime environment. It is important to make sure that the image registry is protected from unauthorized access, tampering, and distribution of malicious images. One such open-source image registry is Harbor, which provides features such as role-based access control, vulnerability scanning, and image signing and verification.
- **Container Runtime Security:** In order to ensure container runtime security, the container runtime environment must be secured and the containers must be isolated from each other and the host system. It is important to ensure that the container runtime environment is up-to-date, as well as that the containers are configured to run in an isolated and secure manner. As an example, Kubernetes provides several security features, including Pod Security Policies, that can be utilized to ensure containers are running in a secure and isolated environment.
- **Container Orchestration Security**: As part of a microservices-based application, container orchestration is the process of managing and scaling containers. It is essential that the container orchestration system be secure, as well as the containers being deployed and managed securely. A popular service mesh such as Istio can be used to secure container communication within a microservices-based application by providing features such as mutual TLS authentication, access control, and traffic management.

Microservices-based applications can be secured by utilizing these techniques and tools against container vulnerabilities, image tampering, and other security challenges associated with containerization.

Monitoring and incident response

Monitoring and incident response are essential components of microservices-based applications. However, as microservices can be distributed across multiple servers and containers, it may be difficult to detect and respond to security threats. In microservices-based applications, the following techniques and tools are used for monitoring and responding to incidents:

- **Centralized logging and monitoring**: With centralized logging and monitoring, a unified view of the microservices environment is provided,

allowing proactive detection of security incidents. It is crucial to collect logs from all the microservices and centralize them for analysis and monitoring. Monitoring tools can be used to analyze logs in real time and to detect security incidents. Using Elasticsearch and Kibana, microservice logs can be collected, analyzed, and visualized.

- **Security monitoring and incident response in microservices environments**: An important component of security monitoring in microservices is analyzing the behavior of the microservices and detecting suspicious behavior. Detecting and responding to security incidents, such as attacks or data breaches, involves monitoring network traffic, system events, and user activity. The Elastic Stack can be used in microservices environments to monitor security events and alerts and to provide automated incident response. Log collection, analysis, visualization, and real-time alerts are provided by Elasticsearch, Logstash, Kibana, and Beats.
- **Distributed Tracing**: With distributed tracing, it is possible to identify performance issues and detect security incidents by providing insight into the flow of requests between microservices. It is important to trace requests across multiple microservices and to correlate logs and metrics from different microservices. As an example, OpenTelemetry provides a framework for distributed tracing of microservices environments as an open-source project. With OpenTelemetry, developers can instrument their microservices to capture trace data and send it to a centralized system for analysis.

These techniques and tools enable microservice-based applications to be monitored for security incidents and threats, and incidents can be detected and responded to quickly and efficiently.

Compliance and risk management

The security of microservices must include compliance and risk management. The storage and processing of sensitive data by microservices-based applications requires adherence to relevant regulations and standards, as well as protection against threats and vulnerabilities, in addition to ensuring compliance with relevant regulations and standards.

Compliance and regulatory considerations

In microservices-based applications, compliance and risk management techniques and tools can be found here. Security for microservices relies heavily on compliance and regulatory considerations. In order to ensure the protection of sensitive data

stored and processed by microservices-based applications, they must adhere to relevant regulations and standards. Security of microservices requires consideration of the following compliance and regulatory issues:

- **General Data Protection Regulation (GDPR)**: In the European Union, the GDPR regulates the privacy and protection of data of individuals. Organizations must comply with GDPR when storing or processing personal data of EU citizens. According to GDPR, organizations must implement appropriate organizational and technical measures to protect personal data.
- **Health Insurance Portability and Accountability Act (HIPAA)**: A US law called HIPAA regulates the use and disclosure of **protected health information** (**PHI**). Microservices-based applications that store or process PHI are required to comply with HIPAA. In order to protect PHI, organizations must implement appropriate technical and organizational measures.
- **Payment Card Industry Data Security Standard (PCI DSS)**: Microservices-based applications that process credit card payments must comply with PCI DSS, a standard that outlines security requirements for companies that process credit card payments. The PCI DSS requires organizations to implement appropriate technical and organizational measures to protect cardholder data.
- **ISO 27001 (Information Security Management System):** The ISO 27001 standard specifies the requirements for establishing, implementing, maintaining, and continually improving an **information security management system** (**ISMS**). For the protection of sensitive data stored or processed in microservices-based applications, an ISMS based on ISO 27001 can be implemented.

Microservice-based applications can ensure that the data is protected and minimize the risk of data breaches and other security incidents by complying with relevant regulations and standards.

Threat modeling

Identifying, evaluating, and prioritizing potential threats to microservices-based applications are the objectives of threat modeling. It involves identifying the assets that need to be protected, potential attackers, and the methods that attackers can use to exploit vulnerabilities in the application. Threat modeling can help minimize the risk of security incidents by identifying and addressing vulnerabilities before they can be exploited by attackers.

The following are some key steps involved in threat modeling for microservice-based applications:

- **Identify the assets**: A threat model begins with identifying the assets that need to be protected, such as sensitive data, servers, databases, and applications.
- **Create an architecture diagram**: A microservices-based application architecture diagram should be created once the assets have been identified, and this diagram should include all components of the application, such as servers, databases, APIs, and external dependencies.
- **Identify potential threats**: As a next step, it is important to identify potential threats to the application. This can be accomplished through brainstorming with the team or through the use of tools that assist in this process.
- **Analyze the potential impact**: Following the identification of potential threats, the next step is to assess the likelihood of the threats occurring, as well as the potential impact of the threats on the application.
- **Prioritize threats**: Threats can be prioritized based on their potential impact, allowing us to determine which threats require immediate attention and which can be addressed later on.
- **Develop mitigation strategies**: Following the prioritization of threats, mitigation strategies to address the vulnerabilities can be developed. These strategies may include implementing security controls, amending the application architecture, or updating the code.

Microservices-based applications can identify potential vulnerabilities and prioritize efforts to address them by performing threat modeling. By doing so, data breaches and other security incidents are minimized, and compliance with relevant regulations and standards is ensured.

Penetration testing

It is a method of testing the security of microservices-based applications by simulating an attack on the application, also known as pen testing. Penetration testing aims to identify vulnerabilities that may be exploited by attackers and provides recommendations for improving application security. Microservices-based applications require penetration testing in order to comply with compliance regulations and manage risks.

The key steps involved in conducting a microservices penetration test are as follows:

- **Scoping**: A penetration test begins with the identification of the systems, applications, and data that will be tested, as well as the specific security controls that will be examined.

- **Reconnaissance**: After obtaining information about the application that will be tested, the next step is to gather information about the network architecture, the operating system, and the application itself.
- **Vulnerability scanning**: An automated vulnerability scan identifies potential vulnerabilities within an application by using automated tools.
- **Exploitation**: Following the identification of potential vulnerabilities, the next step is to attempt to exploit them. This involves utilizing various tools and techniques to gain unauthorized access to the application.
- **Reporting**: Upon completion of the penetration test, a report is generated containing a summary of the vulnerabilities identified, the impact of the vulnerabilities, and recommendations for remedying the vulnerabilities.

Microservices-based applications are able to identify vulnerabilities and prioritize efforts to resolve them by conducting regular penetration testing. In addition to reducing the risk of data breaches and other security incidents, this ensures compliance with relevant regulations and standards.

Infrastructure security

Microservices-based applications deployed in the cloud require a high level of infrastructure security. Following are some key steps involved in ensuring infrastructure security:

- **Ensuring secure configuration**: This involves configuring firewalls, access controls, and logging in accordance with best practices for securing the cloud platform. In AWS, for example, organizations can enforce security policies and monitor activity by using services such as AWS Config, AWS CloudTrail, and AWS **Identity and Access Management** (**IAM**).
- **Apply security patches and updates regularly**: In order to ensure that the infrastructure remains protected against known vulnerabilities, organizations should apply security patches and updates on a regular basis. For example, in AWS, organizations may use services such as AWS Systems Manager to automate the management of patches across multiple instances.
- **Enforcing strong password policies**: By enforcing password complexity and rotation policies and limiting the number of administrative users, strong password policies can help to prevent unauthorized access to the infrastructure. It is possible for organizations to enforce strong password policies and limit administrative access in AWS using AWS IAM.
- **Limiting network exposure of microservices**: In order to limit network exposure, network security groups, firewalls, and other security controls can be used. Only microservices requiring access to the internet or external networks should be exposed to the network. It is possible for organizations

to limit network exposure by using AWS VPCs and security groups, for instance.

- **Implementing secure backups and disaster recovery plans**: To ensure business continuity in case of a security incident or other disruption, it is necessary to have a disaster recovery plan in place. Organizations can use services such as AWS Backup to automate backup and recovery of data by securely storing and testing backups regularly. For example, organizations can utilize AWS Backup services to ensure backups and recovery of data are performed efficiently.

By following these steps, organizations can ensure that the underlying cloud infrastructure for their microservices-based applications is secure and can minimize the risk of data breaches and other security incidents.

Threat detection and response

A critical part of cybersecurity is threat detection and response, as it assists organizations in identifying and responding to potential security threats. In addition to improving threat detection, incident response plans help organizations respond quickly to incidents by using machine learning and artificial intelligence.

As examples of how machine learning and artificial intelligence can be used in threat detection and response, we provide the following:

- **Machine learning and AI for identifying security threats and vulnerabilities**: To detect security threats and vulnerabilities in cloud infrastructure, cloud service providers can analyze the vast amount of data generated by their cloud infrastructure using machine learning and artificial intelligence algorithms. For example, an artificial intelligence system could monitor cloud instances and storage for anomalous behavior, such as unusual traffic patterns, unauthorized access attempts, or unexpected changes to the configuration or data. Additionally, machine learning models may be used to identify known attack patterns and malware signatures and identify potential threats before they cause damage.
- **Incident response plans for responding quickly to security incidents**: As part of their incident response plans, cloud service providers should specify how they will handle various security incidents. It is important that these plans include procedures for detecting the nature and scope of an incident, isolating affected systems, and restoring operations as soon as possible. A cloud service provider, for instance, should have a plan in place to investigate a data breach quickly, contain the damage, and notify affected parties as soon as possible if a cloud customer reports a data breach.
- **Monitoring and responding to suspicious activity, such as DDoS attacks**: By combining automated and manual monitoring tools, cloud service providers can detect and respond to suspicious activity in their infrastructure

in a timely manner. In order to identify traffic patterns associated with DDoS attacks and block traffic from the offending IP addresses, they can use network monitoring tools. To identify and block attacks targeting cloud instances, storage, and other resources, they can also use **intrusion detection and prevention systems** (**IDS/IPS**).

Cloud security relies heavily on threat detection and response, and cloud service providers can ensure the safety of their cloud infrastructure and protect the data of their customers by using machine learning and artificial intelligence to identify security threats and vulnerabilities, incident response plans, and monitoring for suspicious activity such as DDoS attacks.

Continuous security monitoring

As part of continuous security monitoring, automated tools and processes are implemented that scan, detect, and respond to potential security threats within the cloud environment continuously. Using these tools and processes will ensure that the cloud environment is secure and free of potential vulnerabilities and threats.

- **Implementing automated security testing, such as security scanning and vulnerability assessments**: As a result of automated security testing tools offered by cloud providers, organizations can identify potential vulnerabilities in their cloud environments. Amazon Inspector, for example, automatically evaluates the security and compliance of AWS applications. Its goal is to provide actionable recommendations for remediation of vulnerabilities in the cloud environment. Another example is Azure Security Center, which provides continuous security assessment and monitoring of Azure resources.
- **Implementing continuous security monitoring**: Cloud providers also provide tools and services that enable continuous security monitoring. For example, AWS CloudWatch enables organizations to monitor and collect operational data and logs from AWS resources in real-time. Alarms can be set up so that organizations are made aware of any unusual activity, allowing them to detect potential security risks. Azure Security Center, for example, provides continuous security assessment and monitoring of Azure resources. By using the service, potential security threats can be detected and mitigation recommendations can be provided.
- **Proactively monitoring for threats and vulnerabilities**: In order to detect and respond to potential security threats, cloud providers provide real-time alerts and notifications. AWS, for instance, offers Amazon GuardDuty, a threat detection service that continuously monitors for malicious activity and unauthorized behavior within AWS accounts. In order to detect potential security threats, the service analyzes CloudTrail event logs and VPC flow logs using machine learning. Another example is Azure Security Center,

which provides alerts for potential security threats, such as network attacks, malware infections, and suspicious user activity.

- **Implementing automated security incident response procedures**: Cloud providers provide automated incident response procedures that help organizations respond quickly to potential security incidents. For example, Amazon Web Services offers AWS Config, which offers automated remediation for security incidents. AWS Lambda functions can be automatically triggered to remediate any security violations within the AWS environment using the service.

Conclusion

Building Fortresses in the Cloud:

Mastering Security in Microservices for Resilience and Compliance

In conclusion, having an understanding of the concepts, principles, and best practices of security in microservices architecture allows learners to develop a comprehensive understanding of the various security threats and risks associated with cloud microservices and how to mitigate them effectively.

During the chapter, you have gained an in-depth understanding of different security aspects such as authentication and access control, communication security, data security, container security, monitoring, and incident response, as well as compliance and risk management.

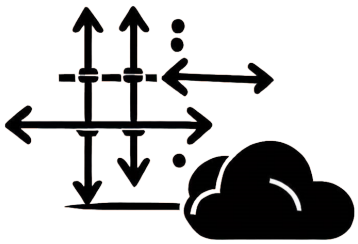
Overall, this understanding will help you to develop and implement a secure microservices architecture that meets regulatory standards, meets business requirements, and is resilient to security threats and risks. Learning through a security-by-design approach allows you to create scalable, reliable, and secure cloud infrastructure by ensuring security is integrated into the microservice architecture from the beginning.

In our next chapter, we will discuss the step wise process of migrating on-premises application to Cloud: A Practitioner's Journey.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 11
Cloud Migration Strategy

Smooth Cloud Migration with a Comprehensive Strategy

Introduction

The purpose of this chapter is to review the concepts we have covered in the last ten chapters and see how they can be utilized to plan and implement a cloud migration strategy. As a result of our learnings from cloud-native microservices, modern application design principles, microservice adoption frameworks, design patterns for microservices, cloud-powered microservices, monolith to microservices case studies, inter-service communication for microservices, event-driven data management for microservices, microservices without servers, and security by design for microservices.

Planning and executing a cloud migration strategy

Several benefits can be realized from migrating to the cloud for organizations, including scalability, flexibility, and cost savings. However, it is a complex process that requires careful planning and execution. Several key considerations should be taken into account by organizations when implementing a cloud migration strategy. In this chapter we will discuss the approach and principles for planning and executing a Cloud Migration Strategy.

First step is to define migration goals and then build a Migration strategy to achieve the same.

To begin, it is necessary to assess the current environment, which includes identifying the applications and systems that are currently running on-premises and their dependencies. As a result of this assessment, organizations will be able to determine which applications and systems are suitable for migration to the cloud.

Following this, it is crucial to define the cloud strategy, which includes identifying which applications and systems should be migrated to the cloud, selecting the cloud provider, and also determining the cloud deployment model.

Migrating to the cloud requires consideration of application modernization. By utilizing containerization, microservices architecture, or serverless computing, organizations can modernize existing applications to make them more cloud-friendly. It is also beneficial to refactor applications to incorporate cloud-native services, including databases, storage, and messaging.

When migrating to the cloud, organizations should ensure that their cloud provider provides appropriate security measures, including encryption, firewalls, and identity and access management. Security and compliance are critical considerations. There should also be consideration of compliance requirements, such as HIPAA or GDPR, and the cloud provider should be capable of meeting these requirements.

An effective migration plan can be created once the cloud strategy has been defined and the applications and systems have been assessed and modernized. The migration plan should outline the steps required to migrate applications and systems to the cloud, as well as how to test and validate the migrated applications to ensure they are working correctly.

To ensure their cloud environment is performing as expected after migration, organizations must monitor and optimize it. This includes monitoring applications and systems to ensure they are performing correctly and optimizing the cloud environment to ensure the organization is getting the most out of its cloud investment.

In conclusion, planning and executing a cloud migration strategy is a complex process that requires careful consideration of a range of factors. It is possible for organizations to successfully migrate to the cloud and benefit from the benefits associated with cloud computing by following a systematic approach and implementing best practices and learnings from the areas discussed in this chapter.

Structure

In this chapter we will discuss following topics:

- Cloud Migration Goals
    - Capex and Opex Cost Optimization
    - Optimize resource consumption & dynamic elasticity

  - Vendor and application consolidation
  - Agility & Innovation via DevOps, Multi-cloud, PaaS, AI/ML, IoT
  - Scalability, Flexibility, and Global reach
  - Reliability, Availability and Security
  - Customer Experience and Insights
  - IT Modernization and Integration
  - Reduce, consolidate, retire the physical data center footprint

- Cloud Migration Principles
  - Security First: Secure the Network, Protect the Data, and Control Access
  - Monitor and Optimize Workloads for Cost
  - Deploy infrastructure as code
  - Make allocations match Demand
  - Automate and implement DevOps practices
  - Training the staff for future mode of operations
  - Leverage cloud-native services

- Cloud migration strategy
  - Business Goals and Objectives
  - Cloud Service Provider Selection
  - Data Security and Compliance
  - Cost Optimization
  - Scalability and Flexibility
  - Legacy Systems
  - Change Management
  - Performance and Reliability
  - Governance
  - Continuous Improvement
  - Migration strategy
  - Migration Plan
  - Skills and training

    - Performance and Optimization
    - Data Management
    - Vendor Management
    - Stakeholder communication
    - Organizational Change Management
- Cloud migration life cycle strategy:
    - Assessment Stage
    - Planning Stage
    - Design Stage
    - Execution Stage
    - Testing Stage
    - Cutover Stage
    - Post Cutover Stage
- Conclusion

Objectives

In this chapter, you will gain an understanding of the Cloud Migration process, as well as the goals, principles, strategies, and stages of the life cycle. As a result of this knowledge, individuals will be able to effectively plan, design, execute, and manage cloud migration projects while optimizing costs, resources, security, reliability, scalability, flexibility, and customer satisfaction. Furthermore, this knowledge will facilitate the integration of IT systems, modernization of legacy systems, and compliance with regulations related to data security and environmental sustainability.

Cloud migration goals

From on-premises infrastructure to the cloud, cloud migration is the process by which an organization's IT systems, applications, and data are moved. Several goals are achieved through cloud migration, including cost optimization, resource optimization, consolidation of vendors and applications, agility and innovation, scalability and flexibility, global reach, reliability, availability, security, customer experience and insights, IT modernization and integration, and reducing the footprint of physical data centers. For example, as illustrated in *Figure 11.1: Cloud Migration Goals*.

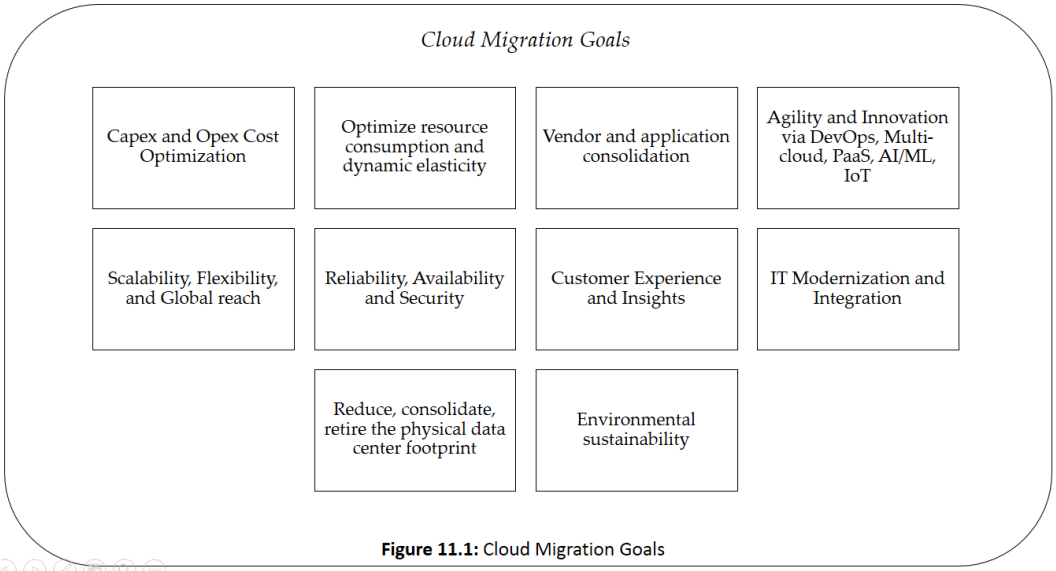


Figure 11.1: Cloud Migration Goals

Capex and Opex cost optimization

As part of cloud migration, it is essential to optimize capex and Opex costs. In IT, Capex refers to the capital expenditures associated with purchasing and maintaining infrastructure and equipment, whereas Opex refers to the operational expenditures associated with running and maintaining systems and applications.

As a result of cloud migration, organizations can reduce Capex and Opex costs by eliminating the need to maintain costly hardware, software, and infrastructure. An organization can eliminate the costs associated with running an email server on-premises by migrating it to the cloud, for example, eliminating the need for costly hardware, software licenses, and maintenance.

It is also possible to optimize Capex and Opex costs through the adoption of cloud-based storage solutions through cloud migration. Organizations can eliminate the need for expensive on-premises storage equipment, reduce backup and disaster recovery costs, and only pay for the storage capacity they require with cloud storage.

Optimize resource consumption and dynamic elasticity

A key objective of cloud migration is to optimize resource consumption and achieve dynamic elasticity. The amount of computing resources required to run applications and services is called resource consumption, which refers to how much processing power, storage, and bandwidth are required. In computing, dynamic elasticity refers to the ability to scale computing resources according to changing demands.

By using cloud-based services such as **Infrastructure-as-a-Service** (**IaaS**) and **Platform-as-a-Service** (**PaaS**), organizations can optimize resource consumption and achieve dynamic elasticity. As an example, organizations can use elastic scaling capabilities of cloud-based IaaS services if they migrate their website to the cloud. It is possible for an organization to automatically scale up its computing resources during periods of high traffic, and then scale down its resources when traffic levels return to normal during periods of low traffic.

Utilizing cloud-based load balancers is another example of optimizing resource consumption and achieving dynamic elasticity as a result of cloud migration. By distributing traffic between multiple servers, load balancers ensure that no single server becomes overloaded. Organizations can optimize resource consumption by migrating to the cloud and using cloud-based load balancers to handle traffic fluctuations by scaling resources up and down as needed.

Vendor and application consolidation

Cloud migration is aimed at consolidating vendors and applications. Consolidation reduces costs, streamlines operations, and improves efficiency, and helps organizations reduce costs. Migration to cloud-based services provided by a single vendor can assist organizations in consolidating vendors and applications. In addition to simplifying procurement, reducing vendor management costs, and ensuring better interoperability, this can result in reduced vendor management costs.

When an organization utilizes multiple vendors for email, CRM, and human resources applications, it can consolidate these applications by using a single cloud-based platform such as Microsoft 365 or Salesforce to consolidate these applications. By providing a single platform for all applications, this can reduce the complexity of managing multiple vendors and improve efficiency.

Agility and innovation via DevOps, Multi-cloud, PaaS, AI/ML, IoT

By leveraging cloud-based services such as DevOps, multi-cloud, PaaS, AI/ML, and IoT, organizations can achieve agility and innovation.

- **DevOps**: To improve efficiency and speed up the delivery of software products, DevOps is a software development methodology that emphasizes collaboration between development and operations teams. The implementation of cloud-based development tools and services can assist organizations in adopting DevOps practices.
- **Multi-cloud**: By using multiple cloud computing services from different providers, organizations can achieve higher levels of resilience, reduce vendor lock-in, and optimize costs. Multi-cloud computing is the use of

multiple cloud computing services from different providers. Migration to cloud-based services from multiple providers can assist organizations in adopting multi-cloud strategies.

- **PaaS: Platform-as-a-Service** (**PaaS**) is a cloud computing model that provides a platform for developing, testing, and deploying software applications. It is possible to reduce the development and deployment costs of software applications by using PaaS. Migration to cloud-based PaaS platforms such as Google App Engine or Microsoft Azure can assist organizations in adopting PaaS.
- **AI/ML: Artificial Intelligence/Machine Learning (AI/ML**) refers to a set of technologies that enable computers to learn and perform tasks that are typically performed by humans. By allowing organizations access to cloud-based AI/ML platforms such as Amazon SageMaker or Google Cloud AI Platform, cloud migration can assist organizations in adopting AI/ML.
- **IoT:** The **Internet of Things** (**IoT**) is a network of sensors, software, and connectivity embedded in physical objects, vehicles, buildings, and other objects. By providing access to cloud-based IoT platforms such as Amazon Web Services IoT and Microsoft Azure IoT Hub, cloud migration can assist organizations in adopting IoT.

Scalability, flexibility, and global reach

Migration to the cloud requires scalability, flexibility, and global reach as fundamental goals. By utilizing cloud-based services that can dynamically allocate resources to meet changing demands and provide access to a global network of data centers, cloud migration can assist organizations in achieving scalability, flexibility, and global reach. For example, an organization that experiences periodic spikes in traffic can use cloud-based services such as auto-scaling to dynamically allocate resources to meet changing needs by using cloud-based services such as auto-scaling. A business can also use cloud-based **content delivery networks** (**CDNs**) to distribute content across a global network of data centers in order to reach a global audience.

Reliability, availability, and security

The objectives of cloud migration are to improve reliability, availability, and security. Cloud migration provides organizations with access to cloud-based services that are designed to be highly available, fault-tolerant, and secure. In order to ensure that a company's mission-critical applications are always available, cloud-based load balancers can be utilized to distribute traffic across multiple instances. As an example, an organization requiring high availability for its mission-critical applications can utilize cloud-based security services such as encryption and access control to ensure data security.

Additionally, cloud service providers provide **service-level agreements** (**SLAs**) which guarantee a certain level of uptime and availability. These agreements can help organizations ensure their applications and data are always available.

Customer experience and insights

By providing access to cloud-based analytics, machine learning, and artificial intelligence services, cloud migration can help organizations improve customer experience and gain insights into customer behavior.

Through the implementation of cloud-based services, organizations will be able to process and analyze data in real-time, improving customer experience. Organizations can use this information to better understand their customers' behavior and preferences, thereby tailoring their products and services accordingly. Retail organizations, for example, can analyze customer behavior on their website through cloud-based analytics services and identify patterns in customer preferences by using cloud-based analytics. Based on this information, they can then develop personalized promotions and offers tailored to the needs and preferences of individual customers.

In addition to providing access to cloud-based services enabling machine learning and artificial intelligence, cloud migration can assist organizations in gaining insight into customer behavior. Organizations can utilize these services to identify patterns and trends in customer behavior that may otherwise be difficult or impossible to identify using traditional analytics approaches. As an example, a financial services company can use cloud-based machine learning services to analyze customer behavior and identify potential fraud patterns. This helps the organization prevent and detect fraud in advance, thereby increasing customer trust and loyalty.

By analyzing data from social media and other sources, organizations can also gain insights into customer sentiment and feedback using cloud-based data analytics services. The organization may be able to identify opportunities to improve its products and services based on a better understanding of customer needs and preferences.

IT modernization and integration

The use of cloud-based services that enable automation, scalability, and flexibility can assist organizations in achieving the goal of IT modernization and integration.

As a result, organizations may be able to streamline their IT operations, reduce costs, and improve overall efficiency as a result. To take advantage of cloud-based automation tools and services, organizations that use legacy on-premises systems can migrate to a cloud-based system. By automating routine tasks, such as backups and software updates, the organization can reduce the workload of its IT staff and improve efficiency.

On-premises systems can also be integrated with cloud-based systems through cloud-based integration services. Using cloud-based integration services, an organization can streamline its business processes by connecting its on-premises ERP system with a cloud-based CRM system.

Reduce, consolidate, and retire the physical data center footprint

By providing access to cloud-based services that eliminate the need for physical hardware and infrastructure, cloud migration can assist organizations in reducing, consolidating, and retiring the physical footprint of their data centers. Consequently, organizations can decrease their data center footprint, reduce costs, and improve overall efficiency. For example, an organization that has several on-premises data centers can consolidate them into a single cloud-based data center to reduce costs and improve efficiency. By utilizing cloud-based services such as virtual machines, storage, and databases, the organization can eliminate the need for physical hardware.

Organizations can also reduce their data center footprint by eliminating the need for a secondary physical data center for disaster recovery purposes by utilizing cloud-based disaster recovery services. As a substitute, the organization can use cloud-based disaster recovery services to replicate its data and applications to a cloud-based data center that can be activated in the event of a disaster.

Cloud migration principles

As a result of cloud migration, an organization's data, applications, and other IT resources are moved from on-premises infrastructure to cloud-based infrastructure. A successful cloud migration requires careful planning and adherence to cloud migration principles in order to make maximum use of the scalability, flexibility,

and cost savings that the cloud platform offers. For example, as illustrated in *Figure 11.2: Cloud Migration Principles:*

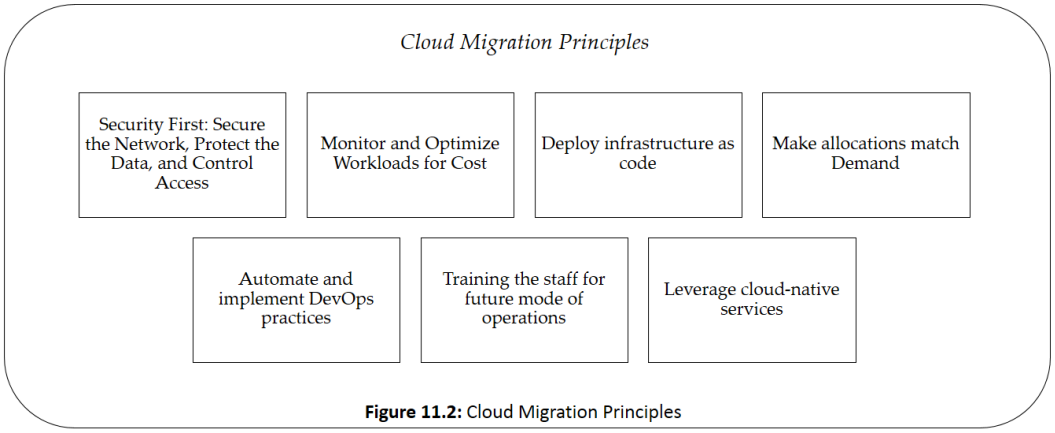


Figure 11.2: Cloud Migration Principles

Security first: Secure the network, protect the data, and control access

When planning for cloud migration, it is essential to prioritize security as a critical principle. In the cloud, there are a number of security risks that organizations need to address. When securing cloud environments, organizations should focus on the following three areas:

- **Secure the Network**: Any cloud environment relies heavily on the network, which must be secure to prevent unauthorized access and attacks. During the transition to the cloud, organizations should ensure that their cloud provider has implemented robust network security controls such as firewalls, intrusion detection and prevention systems, and security gateways. Additionally, organizations should implement their own network security controls, such as using a **virtual private network** (**VPN**) to establish secure communication between their network and that of their cloud provider. In addition, organizations should monitor their network traffic regularly in order to detect any suspicious activities and respond as quickly as possible to any possible security incidents.
- **Protect the Data**: When moving to the cloud, organizations should implement robust data security controls to prevent theft, loss, and damage, including encryption, access controls, and backups. Organizations can ensure their data is protected even if it falls into the wrong hands by encrypting data at rest and in transit. Access controls may be implemented to ensure that only authorized personnel have access to sensitive information. Having a robust backup and recovery strategy in place is critical in the event of data loss or damage. It is also important for organizations to ensure that any regulations

applicable to their industry or location regarding data privacy and security are being followed.

- **Control Access**: To ensure that only authorized personnel can access the organization's resources in a cloud environment, access control is critical. Multi-factor authentication, role-based access control, and **identity and access management** (**IAM**) should be implemented by organizations as robust access control mechanisms.
- The use of **multi-factor authentication** (**MFA**) requires the user to provide two or more forms of identification, including a password and a biometric indicator, such as a fingerprint or facial recognition. Organizations can assign different roles to different personnel and grant access to resources based on those roles using **role-based access control** (**RBAC**). A centralized platform for managing user identities and access across multiple cloud services is provided by IAM solutions.

Monitor and optimize workloads for cost

The organization must ensure that it is not overpaying for cloud resources when moving to the cloud by monitoring and optimizing workloads for cost. It involves continuously monitoring the utilization of cloud resources, identifying underutilized resources, and adjusting resources accordingly. A company may, for example, have a development environment that is only used during working hours. By using automation tools, such as AWS Lambda, the organization can automatically spin up and down the environment at set times, thereby saving costs when the environment is not in use.

It is also possible to track resource usage and costs by project, department, or application using cloud resource tags, which allow organizations to identify areas where cost saving can be made and allocate costs more accurately.

A continuous monitoring and optimization of workloads for cost can help organizations reduce their cloud spending and utilize cloud resources in the most efficient manner possible.

Deploy infrastructure as code

In order to automate provisioning and management of cloud resources, infrastructure as code uses configuration files. As a result, organizations can manage their cloud infrastructure more efficiently, consistently, and scalable, while reducing the risk of human error. As an example, an organization may create templates that contain the infrastructure and services that are required for a particular application using a tool such as AWS CloudFormation. The templates can then be managed, tested, and deployed in a repeatable and consistent manner, thereby reducing errors and saving time.

The use of infrastructure automation tools such as Ansible, Chef, or Puppet can also provide organizations with the ability to manage infrastructure and application configurations at scale, ensuring consistency and reducing the risk of configuration drift.

Using infrastructure as code, organizations can automate the deployment, scaling, and management of cloud resources, reduce human error, and ensure that their cloud environment remains consistent, scalable, and maintainable over time.

Make allocations match demand

An important aspect of cloud migration involves matching the allocation of resources with the actual demand for those resources. This means scaling up resources in times of high demand and scaling down in times of low demand. During the holiday season, for example, an e-commerce website may experience an increase in traffic. By using cloud resources such as AWS EC2 Autoscaling, the website can automatically increase the number of servers during peak periods and reduce them when demand subsides. By doing so, the website will be able to handle the increased traffic without incurring unnecessary costs or overprovisioning resources.

Additionally, organizations can use cloud storage solutions such as AWS S3 and Azure Blob Storage to store and retrieve data as needed without having to use physical storage devices. By doing so, storage resources can be allocated according to demand, rather than being overprovisioned.

Organizations can ensure that cloud resources are used efficiently, reduce costs, and increase overall performance by making allocations match demand.

Automate and implement DevOps practices

In order to automate and implement DevOps practices, automation tools are employed to streamline the deployment and management of cloud resources. In order to ensure that applications and services are delivered quickly and reliably, DevOps practices ensure development and operations teams work seamlessly together. In order to ensure that applications are delivered quickly and reliably, an organization may use tools such as Jenkins to automate the development and testing process before deploying them to the cloud.

The use of containerization technologies, such as Docker and Kubernetes, is another example. These tools allow organizations to package applications into containers. This makes them more portable, scalable, and easier to deploy and manage.

Automation and DevOps practices can reduce the time and effort needed to deploy and manage cloud resources, while improving the reliability and quality of applications and services.

Training the staff for future mode of operations

In order to prepare the employees for future mode of operations, it is necessary to educate them regarding the cloud platform and its capabilities, as well as provide them with the necessary skills to effectively operate and maintain cloud resources. As an example, an organization may offer training to its employees on cloud computing concepts such as infrastructure as code, containerization, and microservices. The training may also include a description of the specific cloud platform and services that the organization plans to use.

It is also possible to establish a **cloud center of excellence** (**CCoE**). A CCoE is a group of experts within the organization who provide guidance and support to other teams throughout the cloud migration process. Furthermore, the CCoE can provide training and workshops in order to ensure employees are familiar with the cloud platform as well as the best practices for maintaining and operating cloud resources. An organization can reduce the risk of errors and downtime by training its staff for the future mode of operations.

Leverage cloud-native services

Instead of migrating legacy applications and services as-is to the cloud, cloud-native services utilize services that are specifically designed for cloud platforms. It is typically the case that cloud-native services are highly scalable, fault-tolerant, and easy to deploy and manage.

Similarly, rather than trying to migrate a legacy database to the cloud, an organization can use cloud-native databases such as Amazon Aurora or Google Cloud Spanner. Moreover, cloud-native databases provide features such as automatic backups and point-in-time recovery, in addition to being highly scalable and able to scale up and down automatically to meet demand.

Serverless computing platforms, such as AWS Lambda and Azure Functions, can also be used as examples. These platforms enable organizations to run code without having to manage the underlying infrastructure. As demand increases or decreases, the platforms automatically scale up and down. Users only pay for the resources they consume.

Cloud-native services enable organizations to maximize the scalability, availability, and performance of cloud platforms.

Cloud migration strategy

Developing a cloud migration strategy requires careful consideration and involvement from leadership to ensure that the strategy aligns with the organization's goals, budget, and compliance requirements. For example, as illustrated in *Figure*

11.3: Cloud migration strategy during the planning of a cloud migration strategy, leadership should consider the following key points:

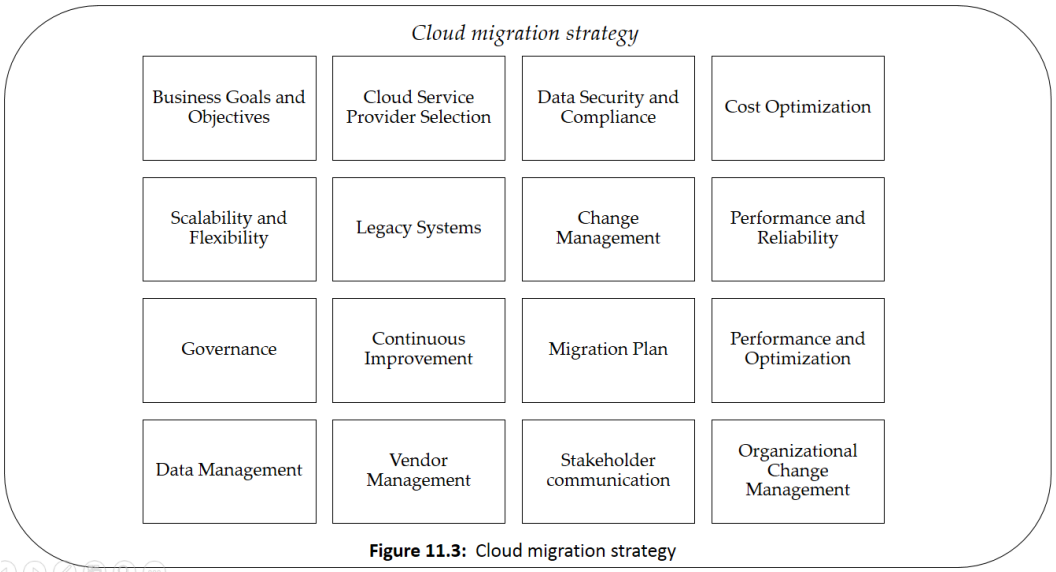


Figure 11.3: Cloud migration strategy

Business goals and objectives

Business goals and objectives should be identified as part of the cloud migration strategy. The cloud migration strategy should be aligned with the organization's overall business goals.

- Clarify the business objectives and goals that the cloud migration strategy will support.
- Ensure alignment with business objectives by involving key stakeholders in the planning process.

Cloud service provider selection

To determine which cloud service provider is most suitable for your organization, you should evaluate the provider's capabilities, pricing, security, compliance, and support, among other factors.

- A review of multiple cloud service providers should be conducted based on key criteria such as cost, security, compliance, and customer support.
- Validate the capabilities of shortlisted providers by performing a **proof of concept** (**POC**).

Data security and compliance

Any cloud migration strategy should place a high priority on data security and compliance. Ensure that the selected cloud provider complies with all relevant regulations and industry standards, such as GDPR and HIPAA.

- Make sure you are familiar with your organization's data security and compliance requirements.
- It is important to choose a cloud service provider that meets these requirements and provides transparent security and compliance controls.

Cost optimization

Cloud migration is primarily motivated by cost savings. Therefore, it is crucial to consider the **total cost of ownership** (**TCO**) when planning a cloud migration strategy. Assess the costs associated with moving to the cloud, including infrastructure, license fees, and support costs.

- Identify the costs associated with moving to the cloud and identify areas where cost savings can be achieved by conducting a TCO analysis.
- Using cloud cost optimization tools can assist in managing costs.

Scalability and flexibility

Scalability and flexibility of cloud services allow businesses to quickly adapt to changing market conditions. In order to accommodate future growth, ensure that the cloud migration strategy allows for scalability and flexibility.

- Consider future market conditions and potential growth opportunities.
- A flexible and scalable cloud service provider is the best choice.

Legacy systems

Consider refactoring or re-architecting any legacy systems that need to be migrated to the cloud.

- Determine which legacy systems need to be migrated to the cloud.
- Evaluate the existing architecture and plan to refactor or rearchitect the system to accommodate the cloud environment.

Change management

The success of any cloud migration strategy depends on the success of change management. To ensure a smooth transition, leaders should plan for the impact of the change on employees, including training and support.

- Engage key stakeholders in the planning process to ensure their support and buy-in.
- Communicate, train, and provide support as part of an effective change management plan.

Performance and reliability

In order to ensure business continuity and maintain customer confidence, ensure that the chosen cloud provider offers a high level of performance and reliability.

- It is important to select a cloud service provider that provides high-performance and reliable services.
- Perform a performance and reliability test before and after migrating to the cloud.

Governance

Establish clear governance structures and processes for the cloud migration project to ensure that the strategy aligns with the organization's policies and standards.

- The governance structures and processes for the cloud migration project should be clearly defined.
- Ensure compliance with policies and standards for the use of cloud services.

Continuous improvement

To assess the effectiveness of the cloud migration strategy and identify opportunities for further optimization and improvement, leadership should establish continuous improvement processes. The process may include regular performance reviews, user feedback, and benchmarking against industry standards.

- Plan to review and improve the cloud migration strategy and plan on a regular basis, such as quarterly or bi-annually.
- Analyze and prioritize areas for improvement based on data and metrics.
- Identify areas for improvement by soliciting feedback from stakeholders.

Migration strategy

This may include a lift-and-shift approach, re-architecting, or hybrid approach, depending on the organization's business requirements and IT infrastructure.

- Ensure that the cloud migration strategy is aligned with the overall business strategy and objectives of the organization.
- Take into account the impact on stakeholders, such as customers, employees, and partners.
- To determine the best cloud deployment model for the organization, use a data-driven approach.

Migration plan

There should be a detailed migration plan developed by the leadership that outlines the scope, timeline, and budget of the migration. In order to keep stakeholders informed of the migration progress, this plan should consider the order in which applications and services will be migrated, potential risks, and communication plans.

- To reduce risk and improve transparency, break down the migration plan into manageable phases or stages.
- Each phase of the migration plan should have clear roles and responsibilities.
- Make sure that the migration plan includes contingency plans for potential risks or problems.

Skills and training

The leadership team should assess the skills and training required to manage the cloud infrastructure and ensure that the IT team is trained in the required technologies and tools to support the cloud environment.

- Develop and maintain the necessary skills for cloud computing by providing training and development opportunities for IT staff.
- To supplement the skills of the IT team, consider partnering with a third-party vendor or consultant.
- Ensure that the organization has a culture of learning and development.

Performance and optimization

To ensure that the cloud infrastructure meets the business requirements and delivers the expected benefits, leadership should establish performance metrics and ongoing

monitoring and optimization processes.

- Track and measure the effectiveness of the cloud environment by establishing performance metrics and **key performance indicators** (**KPIs**).
- Identify opportunities for resource optimization by regularly monitoring resource usage.
- Enhance the efficiency of the cloud environment by utilizing automation and other tools.

Data management

Data management and storage in the cloud environment should be considered by leadership, including data backup and recovery processes, disaster recovery plans, and data lifecycle management.

- Ensure compliance with regulations and industry standards by developing clear data management policies and procedures.
- Protect sensitive data by using encryption and other security measures.
- To ensure data recovery in the event of a disaster, regularly back up data and test disaster recovery plans.

Integration

It is important for leaders to evaluate how cloud-based applications and services will integrate with existing IT systems and processes, including legacy systems, on-premises applications, and third-party applications.

- Identify requirements and test plans for the integration strategy.
- If you wish to streamline the integration process, you may wish to consider using API management tools or other integration platforms.
- To ensure that integration issues are resolved quickly, IT teams and stakeholders should establish clear communication channels.

Vendor management

Cloud providers and other vendors should be managed in accordance with service level agreements, security requirements, and compliance obligations. This may include auditing vendors, monitoring performance, and maintaining ongoing relationships.

- **Service level agreements** (**SLAs**) should establish clear expectations and requirements for cloud providers.

- Ensure that cloud providers' performance is regularly reviewed and assessed in relation to their service level agreements.
- Ensure that clear communication channels are maintained with cloud providers to ensure that issues are addressed as soon as possible.

Stakeholder communication

In order to keep stakeholders informed of the migration progress and address any concerns or questions they may have, leadership should develop a communication plan, which may include regular updates, training and education programs, and support services.

- Establish clear communication channels and provide regular updates to stakeholders regarding the cloud migration process.
- To reach all stakeholders, use a variety of communication channels, such as email, video conferencing, and town hall meetings.
- Ensure that feedback is provided and that concerns are addressed in a timely and transparent manner.

Organizational change management

The leadership of an organization should consider the organizational change management aspects of cloud migration, including how to manage resistance to change and ensure that employees have the necessary skills and resources to be successful in the new environment.

- In order to manage cultural and organizational changes, use a change management framework such as ADKAR or Kotter's 8-Step Model.
- Inform stakeholders of the benefits of the cloud environment and address any concerns or resistance they may have.
- Ensure that employees are trained and supported to work in a cloud environment.

A successful cloud migration strategy requires careful planning, evaluation, and execution. Business leaders should consider these key points and work closely with their IT team and cloud service provider to ensure a successful transition.

Cloud migration life cycle strategy

The process of cloud migration involves the transfer of data, applications, and services from an organization's on-premises infrastructure to a cloud-based environment. A brief overview of the cloud migration lifecycle and the strategic decisions required at each stage is provided below. For example, as illustrated in *Figure 11.4: Cloud*

migration life cycle strategy this process involves several stages requiring strategic decisions.

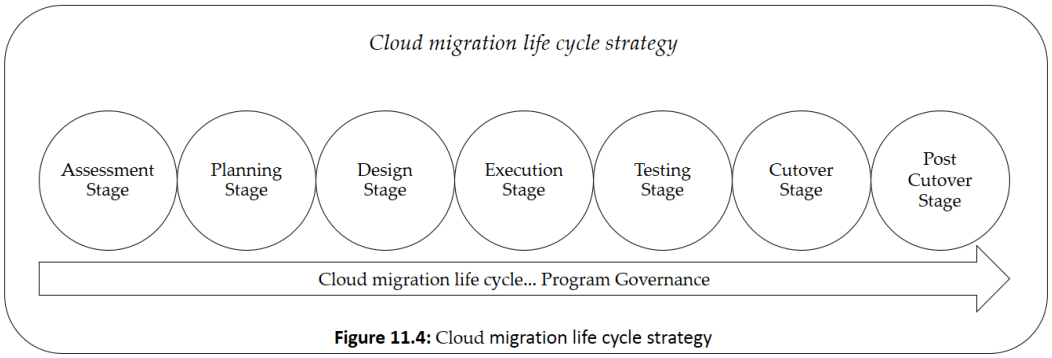


Figure 11.4: Cloud migration life cycle strategy

Assessment stage

An assessment of the organization's existing IT infrastructure is conducted at this point in order to determine which applications, data, and services will be migrated to the cloud. As a strategic decision, it is necessary to identify potential roadblocks to the migration, determine the business goals, and prioritize the migration at this point. Establish business objectives for cloud migration, such as cost savings, scalability, and agility, for example. Determine which applications and services are the most critical and should be prioritized for migration. Determine what legacy systems can be retired or replaced after an analysis of the existing IT infrastructure.

- **Define Business Objectives**: The best practice is to clearly define and document the business objectives that the cloud migration is intended to achieve. For example, a company may define its business objective as reducing IT infrastructure costs by 30% over the next three years by migrating to the cloud.
- **Identify Workloads for Migration**: Identify workloads which are best suited for cloud migration based on factors such as business criticality, complexity, and resource usage. As an example, a company may choose to migrate its non-critical web applications first to the cloud, followed by its more complex ERP system.
- **Assess Workload Dependencies**: Ensure that all dependencies are taken into account before migration and understand the dependencies among workloads. Suppose a company has an application that depends on a database server, and both of these components must be moved to the cloud in order to ensure the application functions properly.
- **Analyze the Current Infrastructure**: An analysis of the existing IT infrastructure to identify hardware, software, and network configurations is the best practice. For example, a company may conduct an analysis of

its current IT infrastructure to determine the number and types of servers, storage devices, and network devices that will need to be migrated.

- **Identify Security and Compliance Requirements**: The best practice is to identify security and compliance requirements and make sure the cloud service provider is capable of meeting them. A company may be required to comply with HIPAA compliance standards and may require the cloud service provider to implement the appropriate security controls.
- **Evaluate Cloud Service Providers**: A best practice is to evaluate and compare the capabilities, pricing, and support of multiple cloud service providers. As an example, a company may evaluate multiple cloud service providers, including Amazon Web Services, Microsoft Azure, and Google Cloud Platform, and determine which one best meets its needs.
- **Determine Cloud Deployment Models**: Analyze the business needs and workloads to determine the most appropriate cloud deployment model (public, private, hybrid). For example, a company may choose to deploy non-critical workloads in a public cloud and critical workloads in a private cloud in a hybrid cloud deployment model.
- **Develop a Proof-of-Concept**: Test the cloud deployment model and migration process by developing a proof-of-concept. In order to test the migration process and evaluate the performance and scalability of the cloud deployment model, a company may migrate a non-critical workload to the cloud for the purposes of developing a proof-of-concept.

Per application assessment stage

Note: An application assessment is a separate activity required for each application. Sometimes it is referred to as an application discovery process.

- **Identify the applications**: Prior to migration, identify the applications that need to be migrated, determine their dependencies, and ensure all dependencies have been accounted for. In order to avoid performance issues due to on-premises vs. cloud hosting, there will be some applications that need to come together for the cutover.
- **Assess Application Architecture**: Determine if any changes need to be made to the architecture of each application before migration. Suppose a company is migrating its monolithic web application from on-premises to the cloud. As part of the application assessment process, the monolithic architecture of the application is determined to not be suitable for a cloud environment and needs to be refactored into microservices. This is due to the fact that the monolithic application will be less scalable, less resilient, and less adaptable to cloud environments.

  - The company will need to redesign the application into smaller, modular components, each containing its own functionality and communication protocols. Consequently, the application is more flexible and adaptable to the cloud environment as these components can be deployed and managed independently. Additionally, this architecture allows for the efficient allocation of resources, as resources can be assigned to specific components in accordance with their needs.
  - As part of the microservices architecture, the company will also have to consider how the application will be stored, secured, and integrated with other systems. In order to ensure that the refactored application functions correctly and securely in the cloud environment, these considerations will be considered when developing the migration strategy and test plan.

- **Assess Application Interactions**: Identify any potential issues that may arise from the interaction between each application and other systems and applications. It is crucial to identify and assess application interactions in modern IT environments, since applications typically interact and integrate with other applications, systems, and databases. These interactions, which are complex, can have a significant impact on cloud migration success.
- **Identify Potential Issues**: There can be compatibility issues, integration problems, or data migration challenges during migration due to interactions between applications. By identifying these potential issues early in the assessment phase, they can be addressed proactively and avoided later.
- **Prioritize Migration Order**: Migration orders can be prioritized based on application interactions. In order to avoid disrupting other systems or to ensure dependencies are migrated first, applications with critical dependencies on other systems may need to be migrated later in the process.
- **Evaluate Cloud Readiness**: Determine if any modifications or refactoring are necessary to make each application cloud-ready. In order to develop a migration plan that addresses the unique needs of each application, assessing cloud readiness helps identify the hardware, software, and networking requirements.
  - **Determine Feasibility**: It is possible to decide whether an application can be migrated to the cloud by evaluating its cloud readiness. Some applications may not be compatible with cloud environments or may need significant modifications. A cloud readiness assessment can identify potential challenges and determine whether it is feasible to migrate each application.
  - **Estimate Costs**: An application's cloud readiness is essential to estimating migration costs. To make sure the application or data is compatible with the cloud environment, identify any hardware or software requirements.

Planning stage

As part of this phase, the organization develops a comprehensive migration plan describing the scope, timeline, and budget of the migration. It is important to make strategic decisions regarding the cloud provider, the migration strategy, and the appropriate tools and technologies for the migration. For example:

- A cloud provider should be selected based on factors such as security, compliance, and cost.
- Choose a migration strategy, such as lift-and-shift, re-architecture, or a hybrid approach.
- Plan the migration in a realistic timeframe and within a reasonable budget.

Design stage

During this stage, the organization designs the cloud infrastructure that will be used to host its applications and services. The strategic decision is to determine the optimal cloud architecture, data management, and security policies. For example:

- A multi-cloud architecture, a hybrid cloud architecture, or a serverless architecture is the optimal cloud architecture.
- Identify and implement data management policies, including data storage, backup, and recovery.
- Ensure compliance with regulations by establishing security policies and procedures.

Execution stage

During this stage, the organization migrates its applications, data, and services to the cloud infrastructure. The strategic decision is to manage the migration process, prioritize workloads, and minimize disruptions to its business operations. For example:

- Set a priority for workloads and determine the order in which applications and services will be migrated.
- Select the appropriate migration tools and technologies, such as the AWS Migration Hub or the Azure Site Recovery service.
- Keep stakeholders informed of the migration progress by developing a communication plan.

Testing stage

As part of this phase, the organization tests the migrated applications and services to ensure their smooth operation on the new cloud infrastructure. In order to ensure that the applications are functioning correctly, it is necessary to determine the appropriate testing methodology and tools.

For example:

- Identify the most appropriate testing methodology, such as unit testing, integration testing, or performance testing.
- In order to ensure that the migrated applications and services are functioning properly, use testing tools and automation.
- Make sure that a rollback plan is in place in case issues are discovered during testing.

Cutover stage

The cutover stage is the final phase of the cloud migration process, which involves moving the organization from the old on-premises infrastructure to the new cloud-based infrastructure. The types of cutover decision-making strategies that organizations can use vary based on their needs and objectives.

Big Bang Cutover

It is a rapid approach, but it can also be risky, as any issues or errors can affect the entire organization. The organization moves all of its applications and data to the cloud simultaneously. In organizations with limited downtime windows, this approach is ideal for migrations that need to take place quickly.

An e-commerce start-up wishes to move all of its infrastructure to the cloud before the holiday season in order to handle increased traffic. As they have a limited timeframe for migrating and want to avoid any downtime during the holiday season, they choose the Big Bang cutover strategy.

Advantages

- In a short period of time, all applications and data are migrated to the cloud.
- Managing the old infrastructure is minimal once the migration has been completed.
- Coordination between the two environments is minimal.

Disadvantages

- **High risk:** Any problems or errors encountered during the migration could have a significant impact on the entire organization.
- A limited amount of testing has been conducted before the migration, which may lead to issues in the new environment.
- There are limited options for fallback if the migration fails.

Phased Cutover

In the phased cutover approach, the organization moves applications and data in a phased approach, ensuring that each phase is tested and validated before progressing to the next. In contrast to the Big Bang cutover strategy, this approach is less risky, as any issues can be identified and resolved before moving on to the next phase. However, this approach requires careful planning and coordination and can take longer.

A healthcare organization wishes to move its electronic health records system into the cloud. The organization chose a phased cutover strategy, moving one department at a time, beginning with the least critical department. Following the validation of each system, they move on to the next department. This ensures that the system is validated before moving forward.

Advantages

- A phased approach minimizes risk by allowing for testing and validation of applications and data before moving onto the next phase.
- **Flexibility**: The migration plan can be adjusted based on the results of each phase.
- **A better fallback option**: If a phase fails, the organization can revert to its previous configuration.

Disadvantages

- Taking a phased approach to migration requires a longer period of time than switching over in one go.
- A higher level of coordination is required during the migration process: The organization must coordinate between two environments.
- During the migration process, the organization must manage both the old and new environments simultaneously.

Parallel Cutover

As part of the parallel cutover approach, the organization runs both the old on-premises infrastructure and the new cloud-based infrastructure simultaneously for a period of time, in order to test and validate the new environment while minimizing risks and downtime.

For a financial services company that wishes to move its trading application to the cloud, a parallel cutover strategy is chosen, which involves running both on-premises infrastructure and cloud-based infrastructure simultaneously for one week. By doing so, the new system can be thoroughly tested and validated before it is completely transitioned to the cloud.

Advantages

- It is low risk to run both environments simultaneously to allow for testing and validation prior to switching completely to the new environment.
- **A better fallback option**: If the new environment fails, the organization can easily switch back to the old environment.
- With minimal downtime, the organization can switch to the new environment.

Disadvantages

- The migration process requires the organization to manage both environments simultaneously.
- During the migration process, a greater degree of coordination is required between two environments.
- The cost of running two environments simultaneously may be greater than the cost of a phased cutover or a Big Bang.

Post cutover stage

It is crucial for the organization to optimize and manage the cloud environment effectively during the post-cutover stage in the cloud migration life cycle following the cutover stage. Here are some strategic planning considerations for FinOps and AIOps in the post-cutover stage:

FinOps: A **Financial Operations** (**FinOps**) framework is designed to help organizations optimize their cloud spending. Here are some strategic planning considerations for FinOps post-cutover:

Monitor cloud costs: Monitor cloud costs on a regular basis in order to identify any overruns or inefficiencies. Use cost management tools provided by the cloud provider or third-party providers in order to gain visibility into cloud costs.

Optimize cloud spend: Use cost optimization techniques such as resizing, rightsizing, and instance scheduling to reduce cloud spend without compromising performance. Analyze cloud usage data to identify areas of inefficiency or waste.

Implement governance and controls: Manage cloud spending in accordance with organizational goals and budgets. Establish budgets and alerts to ensure that spending does not exceed allocated amounts.

Conduct regular cost reviews: Maintain regular cost reviews to track progress, identify new opportunities for optimization, and ensure that cloud costs remain within reasonable limits.

- **AIOps**: In the post-cutover stage, there are some strategic planning considerations for **Artificial Intelligence for IT Operations** (**AIOps**).
- **Implement monitoring and alerting**: Monitor cloud infrastructure and applications using monitoring and alerting tools. Utilize machine learning to identify anomalies and potential issues before they become problematic.
- **Automate incident management**: Automate the incident triage and resolution process using incident management tools that employ machine learning. This can help reduce downtime and improve service levels.
- **Use predictive analytics**: It can be beneficial to use predictive analytics to identify potential issues and to proactively address them before they become critical. This will improve the availability of the systems and reduce downtime.

Optimization stage

This stage involves optimizing the organization's cloud infrastructure in order to increase performance, scalability, and cost-effectiveness. The strategic decision involves analyzing performance metrics and adjusting the cloud infrastructure as needed to meet business objectives. For example:

- Analyze performance metrics and determine areas for optimization, such as scaling up or down in response to demand.
- Identify and eliminate unnecessary resources or optimize usage in order to reduce costs.
- The cloud infrastructure should be monitored and maintained on an ongoing basis to ensure that it meets the business needs.

Overall, successful cloud migration requires careful planning, execution, and ongoing optimization to ensure that the business objectives are met and the cloud infrastructure is delivering the expected results.

Conclusion

Seamless Transition to the Cloud:
Unlocking Efficiency, Scalability, and Sustainability for Growth

In conclusion, the migration to the cloud has become an increasingly popular strategy for organizations seeking to optimize their operations, reduce costs, and improve their overall performance. The goals of cloud migration include Capex & Opex cost optimization, resource optimization, vendor and application consolidation, agility & innovation, scalability, flexibility, IT modernization and integration, and environmental sustainability.

These goals can only be achieved by following certain principles during the cloud migration process, including following security first approach, optimizing workloads, automating and implementing DevOps practices, training staff for future modes of operations, and utilizing cloud-native services.

Furthermore, organizations need to develop a cloud migration strategy that aligns with their business goals and objectives, selects the right cloud service provider, prioritizes data security and compliance, implements change management, and establishes governance process enabling continuous improvement.

Overall, cloud migration is a complex and multifaceted process that requires careful planning, execution, and monitoring. As outlined above, organizations can successfully migrate to the cloud and achieve their desired results by following the goals, principles, strategies, and life cycle stages discussed in this chapter.

Index

B

C

D

E

S

Mastering Cloud-Native Microservices

DESCRIPTION

Microservices-based cloud-native applications are software applications that combine the architectural principles of microservices with the advantages of cloud-native infrastructure and services. If you want to build scalable, resilient, and agile software solutions that can adapt to the dynamic needs of the modern digital landscape, then this book is for you.

This comprehensive guide explores the world of cloud-native microservices and their impact on modern application design. The book covers fundamental principles, adoption frameworks, design patterns, and communication strategies specific to microservices. It then emphasizes on the benefits of scalability, fault tolerance, and resource utilization. Furthermore, the book also addresses event-driven data management, serverless approaches, and security by design. All in all, this book is an essential resource that will help you to leverage the power of microservices in your cloud-native applications.

By the end of the book, you will gain valuable insights into building scalable, resilient, and future-proof applications in the era of digital transformation.

KEY FEATURES

- Gain a comprehensive understanding of the key concepts and strategies involved in building successful cloud-native microservices applications.
- Discover the practical techniques and methodologies for implementing cloud-native microservices.
- Get insights and best practices for implementing cloud-native microservices.

WHAT YOU WILL LEARN

- Gain insight into the fundamental principles and frameworks that form the foundation of modern application design.
- Explore a comprehensive collection of design patterns tailored specifically for microservices architecture.
- Discover a variety of strategies and patterns to effectively facilitate communication between microservices, ensuring efficient collaboration within the system.
- Learn about event-driven data management techniques that enable real-time processing and efficient handling of data in a distributed microservices environment.
- Understand the significance of security-by-design principles and acquire strategies for ensuring the security of microservices architectures.

WHO THIS BOOK IS FOR

This book is suitable for cloud architects, developers, and practitioners who are interested in learning about design patterns and strategies for building, testing, and deploying cloud-native microservices. It is also valuable for techno-functional roles, solution experts, pre-sales professionals, and anyone else seeking practical knowledge of cloud-native microservices.

BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-869-9