Title

Subtitle

Master Thesis

Submitted in partial fulfillment of the requirements for the degree of:

**Master of Science in Engineering**

to the University of Applied Sciences FH Campus Wien

Master Degree Program: Software Design and Engineering

Author:

First name surname

Student Identification Number:

Number

Supervisor:

Degree firstname name

Date:

dd. mm. yyyy

Declaration of authorship:

I declare that this thesis is my own work and that I did not use any aids other than those indicated or any other unauthorized help (e.g., ChatGPT or similar artificial intelligence-based programs). I certify that this work does not contain any personal data, and that I have clarified any copyright, license or image-law issues pertaining to the electronic publication of this thesis. Otherwise, I will indemnify and hold harmless the FH Campus Wien from any claims for compensation by third parties. I certify that I have not submitted this thesis (to an assessor for review) in Austria or abroad in any form as an examination paper. I further certify that the (printed and electronic) copies I have submitted are identical.

Date: Signature:

Abstract

In contemporary software engineering, microservices architectures have become a favored alternative to monolithic systems, providing benefits in scalability, agility, and maintainability. Nevertheless, these advantages are accompanied by operational challenges, especially in configuration management, where maintaining consistency, security, and efficient updates across distributed services is essential. This thesis explores the effects of centralized configuration management, implemented via Spring Cloud Config Server, in contrast to traditional local configuration methods that embed configuration files directly within each service. The study emphasizes three primary evaluation dimensions: scalability, security, and maintainability. A controlled experimental methodology was employed, which involved the creation of a representative banking application comprising three essential services Accounts, Cards, and Loans accessible through an API Gateway. Initially, the system was implemented using local configuration for each microservice and subsequently restructured to utilize a centralized configuration server. Both configurations were deployed in Kubernetes-managed environments, ensuring uniform workloads, infrastructure specifications, and operational conditions. Performance and operational characteristics were evaluated through a series of experiments. Scalability was assessed by monitoring horizontal scaling responsiveness, saturation throughput, and cold start behavior under varying loads. Security was scrutinized through secret rotation drills, enforcement of access control, and auditability checks. Maintainability was examined by measuring change propagation time, rollback capabilities, and the operational effort needed for configuration updates. The experimental setup incorporated industry-standard tools such as Docker, Kubernetes, Helm, Prometheus, Grafana, and Keycloak to guarantee precise and reproducible measurements. The results indicate that centralized configuration management can substantially enhance maintainability.

This thesis contributes empirical evidence to guide architectural decision-making for organizations adopting or evolving microservices-based systems. Comparative insights on scalability, security, and maintainability can help software architects weigh the trade-offs between local and centralized configuration management, enabling informed choices that align with both technical requirements and organizational goals.

List of Abbreviations

SOA Service-Oriented Architecture

URIs Uniform Resource Identifier

DEV Development Environment

PROD Production Environment

API Application Programming Interface

RBAC Role-Based Access Control

HTTP Hypertext Transfer Protocol

GRPC Google Remote Procedure Calls

DDD Domain-Driven Design

ETCD Distributed key-value store

Json JavaScript Object Notation

AWS Amazon Web Services

SSM Soft Systems Methodology

WAF Web Application Firewall

TLS Transport Layer Security

MITM Man-in-the-Middle

JWTs JSON Web Tokens

JWKS JSON Web Key Sets

DDoS Distributed denial-of-service

RBAC Role-Based Access Control

SSO Single Sign-On

TSDB Time-Series Database

DVCS Distributed Version Control System

SQL Structured Query Language

NoSQL Not Only SQL

CI/CD Continuous Integration and Continuous Delivery

DVCS  Distributed Version Control System

MCP Model Context Protocol

JAR Compressed Archive Format

REST Representational State Transfer

DTOs Data Transfer Object

CRUD Create, Read, Update and Delete

ORM Object-Relational Mapping

IAM Identity and Access Management

CNCF Cloud Native Computing Foundation

AUT Application under test

K8s Kubernetes

Contents

Table of Contents

[Abstract iii](#_Toc205645847)

[List of Abbreviations v](#_Toc205645848)

[Contents vii](#_Toc205645849)

[1st CHAPTER – INTRODUCTION 1](#_Toc205645850)

[1.1 Background And Motivation 1](#_Toc205645851)

[1.1.1 Introduction to Microservices 1](#_Toc205645852)

[1.1.2 Why Configuration Management 3](#_Toc205645853)

[1.1.3 Personal Experience 5](#_Toc205645854)

[1.1.4 Problem Statement 6](#_Toc205645855)

[1.2 Research Objectives 7](#_Toc205645856)

[1.3 Research Question 8](#_Toc205645857)

[2. CHAPTER – LITERATURE REVIEW 9](#_Toc205645858)

[2.1 Microservices Outline 9](#_Toc205645859)

[2.1.1 Monolith to Microservices 10](#_Toc205645860)

[2.1.2 Monolith to Microservices: 5 Strategies 13](#_Toc205645861)

[2.1.3 Key Characteristics of Microservices 15](#_Toc205645862)

[2.1.4 Impacts of Migration to Microservice Architecture on Team 16](#_Toc205645863)

[2.2 Challenges In Microservices 17](#_Toc205645864)

[2.3 Centralized vs. Local Configuration 19](#_Toc205645865)

[2.4 Security and Observability Considerations 22](#_Toc205645866)

[2.4.1 Security 22](#_Toc205645867)

[2.4.2 Observability 23](#_Toc205645868)

[2.5 Identified Research Gaps in Microservices Literature 25](#_Toc205645869)

[3. CHAPTER – RESEARCH METHODOLOGY 27](#_Toc205645870)

[3.1 Research Design 27](#_Toc205645871)

[3.2 Tools and Technologies 27](#_Toc205645872)

[3.3 Experimental Setup 31](#_Toc205645873)

[3.4 Measurement 32](#_Toc205645874)

[4. Chapter - SYSTEM DESIGN 33](#_Toc205645875)

[4.1 Architecture Overview 33](#_Toc205645876)

[4.2 Implementation Approach 37](#_Toc205645877)

[4.2.1 microservice 37](#_Toc205645878)

[4.2.2 Maven Dependance 38](#_Toc205645879)

[4.2.3 Entity 39](#_Toc205645880)

[4.2.4 Data Transfer Object 40](#_Toc205645881)

[4.2.5 Repository 41](#_Toc205645882)

[4.2.6 Controller 42](#_Toc205645883)

[4.2.7 Exceptions Handler 43](#_Toc205645884)

[4.2.8 Application YML file 44](#_Toc205645885)

[4.3 Configuration Strategy 45](#_Toc205645886)

[4.3.1 API 45](#_Toc205645887)

[4.3.2 Docker 50](#_Toc205645888)

[4.3.3 Config Server 52](#_Toc205645889)

[4.3.4 Client-Side Service Discovery 55](#_Toc205645890)

[4.3.5 RESILIENCY 57](#_Toc205645891)

[4.3.6 Observability and Monitoring 59](#_Toc205645892)

[4.3.7 Distributed Tracing 64](#_Toc205645893)

[4.3.8 OpenID Connect, KeyCloak 66](#_Toc205645894)

[4.3.9 Event-Driven Micro Services 71](#_Toc205645895)

[4.3.10 Kubernetes 73](#_Toc205645896)

[4.3.11 Server-Side Service Discovery 76](#_Toc205645897)

[4.3.12 Postman Collection for Microservices Testing 78](#_Toc205645898)

[5. CHAPTER – EXPERIMENTATION & RESULTS 79](#_Toc205645899)

[5.1 Performance and Maintainability 79](#_Toc205645900)

[5.1.1 Local Configuration 79](#_Toc205645901)

[5.1.2 Centralized Configuration 85](#_Toc205645902)

[5.1.3 Docker Compose 88](#_Toc205645903)

[5.1.4 Spring Boot BOM and Shared Libraries 90](#_Toc205645904)

[5.1.5 Helm 93](#_Toc205645905)

[5.1.6 Result 95](#_Toc205645906)

[5.2 Security 97](#_Toc205645907)

[5.2.1 KeyCloak 97](#_Toc205645908)

[5.2.2 Docker Compose 100](#_Toc205645909)

[5.2.3 Result 101](#_Toc205645910)

[6. CONCLUSION 104](#_Toc205645911)

[List of Figures 108](#_Toc205645912)

[List of Tables 110](#_Toc205645913)

[Appendix and Bibliography 111](#_Toc205645914)

# CHAPTER – INTRODUCTION

## Background And Motivation

### Introduction to Microservices

A decade ago, applications were typically deployed as a cohesive unit, with all functionalities integrated and hosted on a single server. This architectural style, referred to as Monolithic, presents advantages such as simplified development and deployment for smaller teams, enhanced performance, and reduced cross-cutting concerns. Nevertheless, it also has drawbacks, including restricted agility, challenges in adopting new technologies, and reliance on a single code base. Various types of Monolithic architecture exist, such as Single-Process Monolith, Modular Monolith, and Distributed Monolith. To tackle the issues associated with large, monolithic applications, Service-Oriented Architecture (SOA) was developed, which organizes software systems into interoperable services. SOA provides advantages like reusability, improved maintainability, increased reliability, and the ability for parallel development; however, it also entails disadvantages such as complex management, significant investment costs, and additional overhead. In contrast, Microservices represent a model of independently deployable services centered around a business domain, facilitating the construction of more intricate systems. [1].

Considering the increasing intricacy of software systems and a relentless quest for agility, monolithic architectures have begun to reveal their limitations. Microservices architectures, which emphasize independent and self-sufficient services, offer a compelling alternative by providing enhanced scalability, faster deployment cycles, and improved maintainability. However, transitioning to this fragmented model presents numerous challenges. One of the most significant obstacles is the disintegration of existing monoliths into cohesive microservices. Precisely identifying microservice boundaries and functional responsibilities within a monolithic software system is a vital task, yet it frequently proves to be challenging. Various approaches have been devised to assist in this endeavor, examining features, dependencies, and execution patterns to potentially outline well-defined microservices. Despite these advancements, a comprehensive understanding of the strengths, weaknesses, and ongoing challenges related to current decomposition strategies remains elusive. This systematic literature review aims to fill this knowledge gap. Through a rigorous methodology, we systematically compile, analyze, and synthesize research contributions on monolith decomposition, with a specific focus on techniques for identifying microservices. Our investigation explores the research objectives, evaluation methods, and enduring challenges that characterize this field, with the goal of establishing a robust classification of decomposition approaches and identifying avenues for further refinement. [2].

The microservices architecture started to attract attention following the publication of several success stories from companies such as Netflix, Gilt.com, and Amazon. However, all these companies, along with many other successful microservices implementations, had one commonality — they originated from web-based companies that were either creating new applications or did not possess a significant legacy code base to overhaul. When a traditional corporation transitions to microservices, one challenge they encounter after selecting the initial green-field applications to explore microservices is that certain principles of the microservices architecture, especially the "Decentralized Data Management" and "Decentralized Governance" principles, are challenging to implement when it is necessary to refactor a large monolithic application. Fortunately, a solution to this challenge has existed for several years in the form of a pattern that Martin Fowler first documented in 2004, several years before his work on microservices. This concept is known as the "strangler application pattern," which aims to tackle the reality that one rarely operates in a green field. The applications that require microservices are often the largest and most complex on the web; however, leveraging the architecture of the web can offer a strategy for managing the necessary refactoring. The strangler application is a straightforward concept based on the analogy of a vine that constricts the tree it envelops. The premise is to utilize the structure of a web application — the fact that it is composed of individual URIs that correspond functionally to various aspects of a business domain — to divide an application into distinct functional domains and replace those domains with a new microservices-based implementation one domain at a time [3].

A group of trees with branches

AI-generated content may be incorrect.The Strangler Fig Pattern is a software migration pattern used to gradually replace or refactor a legacy system with a new system, piece by piece, without disrupting the existing functionality. This pattern gets its name from the way a strangler fig plant grows around an existing tree, slowly replacing it until the original tree is no longer needed. When to Use the Strangler Fig Pattern: When you need to modernize a large or complex legacy system. When you want to avoid the risk associated with a complete system rewrite or "big bang" migration. When the legacy system needs to remain operational during the transition to the new system as shown in Figure 1.

Figure 1 .Strangler Fig Pattern

### Why Configuration Management

Configuration management plays a vital role in the effective operation of microservices, encompassing elements such as environment variables, feature flags, service endpoints, and rate limits. Poorly managed configurations can result in downtime, erratic behavior, or security vulnerabilities. The difficulties associated with configuration management include decentralization, environment-specific settings, dynamic updates, and security issues. Decentralized storage may cause inconsistencies, while environment-specific configurations necessitate distinct environments for development, staging, and production. Safeguarding sensitive data, such as API keys or passwords, demands particular attention to prevent breaches. It facilitates dynamic updates without requiring redeployment or service restarts, thereby ensuring minimal disruption and a seamless rollout of settings across various microservice instances. Centralized configuration management systems, such as Spring Cloud Config, enable dynamic configuration updates to maintain service consistency, thereby preventing drift between environments and instances. Utilizing central repositories like Git or Consul in conjunction with Spring Cloud Config streamlines configuration management processes, making it easier to track changes and conduct audits. Environment-specific profiles (for instance, dev, staging, prod) are managed efficiently from a centralized location, and automatic configuration refresh (through Spring Cloud Bus, Kafka, or RabbitMQ) disseminates updates in real time to all clients, directly enhancing maintainability, operational efficiency, and system scalability. [4].

In my view, configuration management stands out as one of the most significant and contemporary subjects today. As evidenced, the software project is expanding continuously. If one examines the latest IT applications I have developed, they are not merely standard websites for restaurants, nor are they simply snapshots of data information. This will indeed necessitate substantial resources. This could imply that there will be numerous services involved. If you possess one or two micro services, or even five to six, I believe it is feasible to manage them individually. For instance, if you intend to modify the configuration, change the endpoint, or make similar adjustments, or if you plan to alter the code and conduct tests. However, as I mentioned, the scale is increasing, and the addition of more components indicates a rise in microservices and complexities. For example, if you have 30 or 40 microservices, managing them individually becomes impractical, and you may encounter unexpected challenges. This underscores the necessity for the community to advocate for configuration management rather than solely depending on sensitivity, which would configure each microservice individually. There are numerous configuration management solutions available. The community has initiated several, such as the Helm concept for Kubernetes, which we have implemented, focusing on how to configure Kubernetes and the entire microservice architecture so that once it is set up, it can be executed monthly without the need for individual configurations. I acknowledge that establishing a centralized configuration management system requires time, but once accomplished, the benefits of maintaining many microservices simultaneously become apparent, as does the ease of monitoring and troubleshooting, such as reviewing logs to identify bugs or issues, and the simplicity of scaling operations.

In this Master thesis I will try as much as possible to figure out the best way to configure our microservices as a center configuration and comparison to local configuration and see when its critic point that we say no we go local or centralized configuration.

### Personal Experience

After working at Alliance for three and half years on a substantial application related to Insurance, I have come to understand how a large community application can encounter issues from various angles.

One significant challenge is managing a large team, which can be difficult, and finding someone who possesses comprehensive knowledge of the entire application is equally challenging. If you consider a single line of code, it can be problematic; even if there are just one or two lines, updating them can impact every aspect of the application. Consequently, extensive testing is required, which can lead to significant delays.

At one point, we decided to transition to microservice architecture, and while we implemented it, we were unable to convert the entire application. The business layer, which interfaces with the database, remained monolithic, while we transformed other components into microservices. Despite this, we observed improvements in performance and latency, and we were able to organize our services into smaller teams. Each team included roles such as a Scrum Master, designer, product owner, and testers, allowing us to work concurrently in different programming languages. However, we still face the limitation of having a single database. For instance, adding a new column to the database required a lot of changes, highlighting the challenges of analytics applications.

Additionally, microservices come with their own set of issues. In my view, if a project can be divided into three or four microservices, it may be better to avoid microservices altogether. However, if the project necessitates more than five microservices,

I recommend pursuing that route. Another aspect I wish to highlight is that when utilizing microservices, you have the flexibility to select the programming language for each service. This is one of the significant advantages of microservices, a freedom that is not available in monolithic applications.

### Problem Statement

In a distributed system, especially one built on microservices architecture, managing configurations can become complex and challenging. Each service typically requires its own set of configurations, including database connections, API keys, feature flags, and environment-specific settings (e.g., development, staging, production). As the number of services grows, keeping track of all these configurations across multiple environments can quickly become unmanageable. That’s why we should use a centralized configuration [3].

Without centralization, changes needed to be replicated manually, increasing maintenance overhead and reducing flexibility. A dedicated configuration service improved consistency and simplified the deployment pipeline.

In a typical microservices system, each service has its own configuration file. While this seems simple at first, it causes big problems in large, distributed systems.

Challenges in Configuration Management

1. Decentralization: With multiple services, storing and managing configurations locally for each service can lead to inconsistencies.
2. Environment-Specific Configurations: Development, staging, and production environments require different configurations.
3. Dynamic Updates: Certain configurations, like feature toggles or throttling limits, may require runtime updates.
4. Security Concerns: Storing sensitive information like API keys or passwords needs special care to avoid breaches.

These issues lead to higher operational effort, more downtime, and potential vulnerabilities. Centralized configuration — where all services pull their settings from one place — promises to solve these challenges, but it’s still unclear how much of an impact it really makes in practice.

This thesis will explore that question through a hands-on comparison.

## Research Objectives

The main goal of this thesis is to assess the effects of centralized configuration management on microservices architecture through the implementation of Spring Cloud Config Server. Specifically, this research aims to investigate how the centralization of configuration data influences three critical operational factors: scalability, security, and maintainability.

In microservices environments, decentralized or per-service configuration can lead to redundancy, inconsistency, and operational overhead—particularly when dealing with large-scale distributed systems. This study aims to illustrate how externalizing configurations to a centralized service can enhance scalability by facilitating easier deployment of changes, mitigate security risks through centralized secret management and access control, and improve maintainability by streamlining configuration updates and auditing.

The research will establish two configurations for comparison: one utilizing local per-service configurations and the other employing centralized configuration via Spring Cloud Config Server. These configurations will subsequently be assessed based on established metrics, including deployment complexity, response time during configuration updates, RBAC enforcement, and the operational effort required to maintain configurations.

Furthermore, our objective extends beyond merely contrasting the centralized configuration with the local configuration of microservices; we also aim to develop a comprehensive microservice utilizing the most advanced technology available, specifically the latest spring shoes. Consequently, we are endeavoring to incorporate a wide array of technologies, such as the Joker orchestration, to enhance the security of our microservice to the greatest extent feasible. Therefore, we will strive to remain as current as possible, engaging in continuous learning and research to understand the latest developments in the field of microservices.

## Research Question

In what ways does centralize configuration management through Spring Cloud Config Server enhance scalability, security, and maintainability within microservices architectures when compared to local configuration management? The sub-questions are as follows: To what degree does centralized configuration alleviate the operational challenges associated with configuration changes? How does it facilitate the secure management of environment-specific variables and sensitive information? What effect does it have on the capacity to scale services independently and uniformly across different environments? By exploring these inquiries, the thesis seeks to offer practical insights and an empirical assessment of configuration strategies in distributed systems.

Alongside these fundamental inquiries, the thesis investigates the wider ecosystem necessary for facilitating effective configuration management. It analyzes the ways in which Docker enhances configuration portability, how Kubernetes facilitates dynamic updates to configurations and manages secure secret handling, and how tools such as Spring Cloud Vault and Config Maps play a role in safeguarding sensitive information. Furthermore, the research delves into how centralized configuration bolsters Develop

and operation ops automation, influence’s fault tolerance and system resilience, and aids in the monitoring, auditability, and governance of configuration modifications across distributed microservices.

We will also attempt to compare the merits of transitioning to microservice architecture versus maintaining our current system. Furthermore, if we decide to adopt a microservice approach, we must consider the critical factors involved. This includes deliberating whether to extend beyond local configuration or to adhere to centralized configuration. Additionally, we need to evaluate the impact of centralized configuration on the automation process—will it expedite or hinder our progress?

# CHAPTER – LITERATURE REVIEW

## Microservices Outline

Microservice architecture has increasingly emerged as the favored approach for creating distributed systems and large-scale applications in recent years. Unlike a monolithic structure, microservice architecture is defined by its loose coupling, with each service functioning independently of the others. Like a monolith, microservice includes business logic and local data storage, but it is specifically designed for single-purpose services only. This differentiation distinguishes microservices from monolithic architectures, which consolidate an entire application within a single service. As a result, a collection of microservices, each operating independently, functions as a suite of smaller services. The independent operation of these services facilitates maintenance due to their manageable size and organization, ensuring that if one microservice fails, the entire system remains operational As Show in Figure 2.

This design promotes high replaceability of components, allowing autonomous teams to perform maintenance and updates independently in most cases. Given the structural design of microservices and their independent operation, they depend significantly on internal communication among the services. A microservice is language-agnostic, meaning that components should not interact through language-specific functions or method calls (Microsoft, 2021). This poses a challenge when transitioning from a monolithic to a microservice architecture, as direct conversion via method calls is considered inefficient. Consequently, microservices employ inter-service communication protocols, such as HTTP or gRPC, for interaction. The architectural transformation is undergoing entails moving away from the legacy code base and modifying business areas to capitalize on the new opportunities offered by the updated digital architecture. Fowler observed that organizations with existing monolithic architecture would benefit from this transition before adopting microservice architecture, emphasizing that it would be challenging to develop applications from [6].

A diagram of a software company

AI-generated content may be incorrect.

Figure 2: Microservices Outline

### Monolith to Microservices

Monolithic, has pros such as simpler development and deployment for smaller teams, better performance due to no network latency, and a single code base. Cons include limited agility, difficulty in adopting new technologies, and a single code base.

SOA emerged as an approach to address the challenges of large, monolithic applications by organizing software systems as a collection of interoperable services. This approach offers benefits such as reusability, better maintainability, higher reliability, parallel development, and complex management due to communication protocols. However, it also has cons such as high investment costs and extra overload.

A diagram of a computer service

AI-generated content may be incorrect.Microservices, independently releasable services modelled around a business domain, are easier to develop, test, and deploy, increase agility, and scale horizontally. They can represent various services, such as Accounts, Cards, and Loans, but together they could form an entire bank system As Show in Figure 3.

Figure 3 : Monolithic vs. SOA vs. Microservices

The shift from monolithic architectures to microservices is frequently motivated by the necessity for quicker delivery cycles, enhanced scalability, and increased resilience.

In contrast to monolithic systems, microservices permit independent deployments, thereby minimizing the risk of causing system-wide failures during updates.

They promote technological diversity, enabling each service to be developed with the most appropriate technology stack for its specific function.

Communication among services is generally managed through lightweight protocols such

Nevertheless, the transition to microservices brings about operational complexity, necessitating strong DevOps practices, service discovery, monitoring, and distributed data management.

Table 1: Monolithic vs. SOA vs. Microservices Comparison

|  |  |  |  |
| --- | --- | --- | --- |
| Aspect | Monolithic Architecture | Service-Oriented Architecture (SOA) | Microservices Architecture |
| Scope | Single, unified application | Broad architectural style with reusable enterprise services | Focused on independently deployable services |
| Size of Services | One large application or module | Larger, domain-specific services (e.g., business process layers) | Small, focused, single-purpose services |
| Data Management | Shared, centralized database | Often shared databases across services | Each service has its own database (Database-per-Service) |
| Communication | Internal method calls | Standard protocols (e.g., SOAP over ESB) | Lightweight RESTful APIs, messaging (Kafka, RabbitMQ) |
| Technology Diversity | Limited to one tech stack | Standardized stack with limited flexibility | Freedom to use different tech stacks per service |
| Deployment | Entire app is deployed together | Services deployed independently, but often tightly integrated | Fully independent deployments per service |
| Scalability | Entire application scales as a whole | Can scale services, but often at coarse level | Each microservice scales independently |
| Development Speed | Slower due to tight coupling | Moderate, depending on service size and dependencies | Faster due to small, independent components |
| Flexibility | Limited by monolithic nature | More flexible, but changes can affect multiple services | Highly flexible; services evolve independently |
| Team Structure | Large, centralized development team | Multiple teams, usually around domains or layers | Small, cross-functional teams per service |
| Maintenance | Complex and error-prone with tight coupling | Easier than monoliths but still involves coordination | Easier, isolated maintenance of smaller codebases |

Table 2: Critical Decision Points – Monolith vs. SOA vs. Microservices

|  |  |  |  |
| --- | --- | --- | --- |
| Decision Point | Monolith wins when… | Microservices win when… | SOA wins when… |
| Team size & autonomy | 1–8 developer single team | Multiple autonomous teams mapped to bounded contexts | Many systems/teams across departments |
| Domain boundaries | Still evolving/unclear | Clear bounded contexts with minimal crosstalk | Cross-domain integration & orchestration needed |
| Deployment frequency | Few releases per month | Continuous deploys per service | Central release governance with staged gates |
| Scaling pattern | Scale all-or-nothing is fine | Need hot paths to scale independently | Need to mediate capacity across many systems |
| Data consistency | Mostly ACID with cross-entity transactions | Eventually consistency acceptable; sagas OK | Mix of both; enterprise data contracts |
| Latency & chattiness | In-process calls preferred | Network hops acceptable; APIs well designed | Heavy mediation, translation, routing required |
| Operational maturity | Minimal DevOps/observability | Strong CI/CD, tracing, metrics in place | Enterprise service bus, central monitoring |

### Monolith to Microservices: 5 Strategies

1. Incremental Refactoring

Incremental refactoring denotes the gradual conversion of a monolithic system into a microservices architecture. This methodology facilitates the stepwise breakdown of a monolith into microservices, thus reducing the likelihood of business interruption. In the process of incremental refactoring, the first step is to pinpoint the elements of the monolith that are most suitable for transformation into independent microservices. These elements may consist of functionalities that are relatively detached from the rest of the system or those that would benefit significantly from the advantages offered by microservices, including improved scalability and faster deployment. [6].

2. Strangler Pattern

The strangler pattern represents a strategy that entails the gradual replacement of segments of a monolithic application with microservices while the monolith continues to operate. This pattern draws inspiration from the strangler fig tree, which envelops other trees and gradually supplants them.

The strangler pattern facilitates the incremental introduction of microservices into your system without interrupting the operation of the monolith. This methodology mitigates risk and promotes a more seamless transition process [6].

3. Decomposing by Business Capability

This strategy focuses on dismantling a monolith into microservices based on business functionalities. This approach aligns the technical elements of your system with your business goals, simplifying the management and evolution of your system in response to business demands.

When decomposed by business capability, it is crucial to ensure that each microservice is accountable for a singular business capability. This practice helps preserve the independence of microservices and diminishes the complexity of the system [6].

4. Anticorruption Layer (ACL)

The anticorruption layer (ACL) is a strategy employed to guarantee that the transition from a monolith to microservices does not compromise the business logic of your system. The ACL serves as a protective barrier between the monolith and the microservices, facilitating the conversion of data and requests between the two systems.

Utilizing an ACL can assist in ensuring that the integrity of the business logic is maintained throughout the transition process [6].

5. Domain-Driven Design (DDD)

Domain-driven design (DDD) is a software development approach that focuses on understanding the business domain and using this understanding to guide the design and

development of software. In the context of transitioning from monolith to microservices, DDD can be used to identify the boundaries of microservices and to ensure that the transition process aligns with business goals. [6].

### Key Characteristics of Microservices

Microservices are an increasingly popular approach to building and deploying software applications. This architectural style involves breaking down an application into a set of independent services that can be developed, deployed, and maintained separately. The goal of microservices is to make software development more agile and scalable, allowing teams to release new features and updates quickly and efficiently [6].

* Componentization via Services: Component is a unit of software that is independently replaceable and upgradeable.
* Organized around Business Capabilities: The microservice approach to division is splitting up into services organized by business capability.
* Products not Projects: This is Amazon’s notion of “you build, you run it” where a development team takes full responsibility for the software in production.
* Smart endpoints and dumb pipes: Microservices aim to be as decoupled and as cohesive as possible, so they own their own domain logic and receive a request, applying logic and producing a response with using Restful APIs.
* Decentralized Governance: Netflix is a good example of an organization that follows this philosophy. Sharing useful and all tested code as libraries encourages other developers to solve similar problems in similar ways.
* Decentralized Data Management: That means Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems.
* Infrastructure Automation: That means automate deployment to each new environment and for every microservice separately.
* Design for failure, Resilience: Microservices design by dealing with failures and try to manage failures by managing errors with proper actions. Microservices are also designed to be resilient, meaning that they can continue to operate even if one or more services fail
* Scalable: Each service operates independently, it is possible to scale individual services up or down as needed, without affecting the rest of the application. This allows teams to allocate resources more efficiently and ensure that the application can handle increased traffic or usage.
* Technology Agnostic: Different services can be written in different programming languages or use different technology stacks.

### Impacts of Migration to Microservice Architecture on Team

Overall, the role of the Product Owner has been significantly influenced by the transition from a monolithic to a microservice architecture. As a representative of customer and product interests, the Product Owner has prioritized enhancing the frequency of software deliveries. The shift to microservice architecture has markedly improved the continuous delivery process. Unlike before, when new functionalities were deployed to customers in fixed deployment cycles, even at the conclusion of each Sprint, they are now delivered continuously throughout a Sprint. Delivery to the customer has been established as a criterion for the completion of User stories and has been integrated into the Definition of Done. This increased frequency of deployments has resulted in more regular interactions with customers, thereby enhancing agility [7].

In comparison to the Product Owner, the Scrum Master indicated that the migration from a monolithic to a microservice architecture resulted in a reduced number of changes. The Scrum Master primarily highlighted the importance of communication and coordination among teams, as well as the necessity for enhanced motivation among team members.

The Scrum Master underscored that decomposing the application into smaller, independent

units facilitated the allocation of application responsibilities across teams and contributed to achieving cross-functionality within the team. This approach also mitigated issues related to inter-team dependencies, ultimately leading to improved efficiency.

Additionally, the Scrum Master noted that the ability to articulate Sprint goals with greater precision has bolstered team member motivation throughout the Sprint. However, the Scrum Master also acknowledged the complexities involved in transitioning to microservice architecture.

Regarding Scrum ceremonies and artifacts, the Scrum Master did not perceive any significant changes from his perspective. Specifically, while the Sprint retrospective was discussed in greater detail, it remained fundamentally unchanged according to the Scrum Master [7].

The Impacts of Migration to Microservice Architecture on the Developer Role

Naturally, the technological dimensions of migration from monolithic to microservice architecture were predominant in the interviews conducted with developers. They candidly expressed the challenges that accompanied the migration process, emphasizing that issues related to deployment, operation, and monitoring should not be underestimated. Moreover, the DevOps concept and the associated automation of processes were identified as critical. It became essential to fulfil the heightened demands for knowledge, experience, and technical expertise within the development team.

Conversely, microservices facilitated a more manageable workflow for development teams and reduced the risks associated with the creation of new functionalities. The process of adding or replacing individual microservices proved to be significantly simpler than redeploying the entire monolithic system [7].

## Challenges In Microservices

**Complexity**

Microservices offer flexibility and modularity. However, development teams tend to face many challenges, including service communication, data consistency, and distributed system management.

Developing, operating, and managing an application based on microservices requires specific expertise, tools, and sophisticated monitoring and orchestration functionalities. Organizations must allocate resources towards infrastructure, automation, and DevOps methodologies to effectively manage the intricacies linked to microservices. Distributed System Challenges [7].

In the microservice architecture, communication between services happens via a network, which leads to increased latency, networking overhead, and potential failure points.

Ensuring dependable communication, managing network disruptions, and preserving data consistency across distributed services can be challenging. Organizations need to establish robust communication strategies, including circuit breakers, retries, and protocols, to address these issues. [7].

**Operational Overhead**

The operational overhead associated with running many microservices in production environments is huge. Examples of tasks that become more complicated in a distributed system are monitoring, logging, debugging, and tracing.

Organizations necessitate dependable surveillance and insight into potential threats to acquire an understanding of the well-being and effectiveness of specific services and the overall system. Additionally, managing service dependencies, version control, and ensuring backward compatibility contribute to the operational intricacy of microservices.[7].

**Data Management**

In a microservices architecture, every service possesses its own data store, which may result in data duplication, consequently causing inconsistencies and synchronization challenges. Ensuring data consistency within distributed systems necessitates careful

Design and execution of data management strategies such as event sourcing, eventual consistency, and distributed transactions. Organizations must diligently oversee data access and uphold data integrity to avert data corruption and associated issues [7].

**Service Discovery and Communication**

Microservices are required to dynamically discover and communicate with one another. Consequently, an effective service discovery mechanism is essential. The management of service endpoints, load balancing, and failover among distributed services presents significant challenges. Organizations ought to implement service registry and discovery solutions, such as Consul or Eureka, to facilitate communication between services. Additionally, robust communication patterns, such as service mesh architectures, enhance reliability and fault tolerance [7].

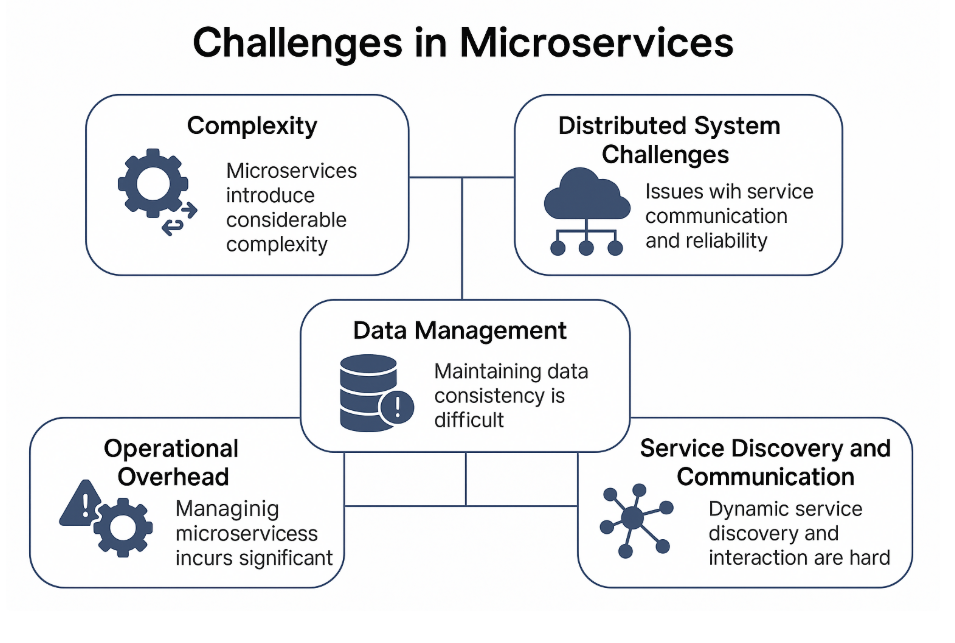


Figure 4: Challenges in Microservices

## Centralized vs. Local Configuration

1. Configurations play a crucial role in the operation of microservices. They include Environment Variables: Such as database URLs, API keys, and credentials. Feature Flags: Enabling or disabling features dynamically. Service Endpoints: The URLs for interaction with other services. Rate Limits: Regulating usage to avert system overload. Inadequately managed configurations can lead to downtime, unpredictable behavior, or security vulnerabilities. Thus, it is essential to implement a robust strategy. Challenges in Configuration Management include Decentralization: With a multitude of services, the local storage and management of configurations for each can lead to inconsistencies. Environment-Specific Configurations: Distinct configurations are required for development, staging, and production environments. Dynamic Updates: Certain configurations, like feature toggles or throttling limits, may necessitate updates during runtime. Security Concerns: The handling of sensitive information such as API keys or passwords demands careful management to avoid breaches. Key Practices for Configuration Management include Externalize Configurations. Avoid embedding configurations directly within your application code. Instead, use configuration files, environment variables, or configuration management tools to externalize settings. This approach guarantees consistency across different deployments and environments [7].

2.Centralized Configuration Management and Implement centralized configuration management systems such as Consul, etc., or Spring Cloud Config. These tools store configurations in a central repository, allowing services to dynamically access their settings [7]. Advantages: Consistency across services, Simplified updates without requiring service redeployment and Secure access control.

3. Utilize Environment-Specific Configurations

Maintain separate configuration files or entries for each environment (e.g., config.dev.json, config.prod.json). This approach aids in avoiding the inadvertent deployment of incorrect settings.

Do we genuinely need to externalize? It seems we are opening a Pandora's box in this situation. Let us evaluate the pros and cons of having my configuration file (e.g., config.json) in conjunction with my Docker image [7].

We will require a method to clone the configuration locally. It is more challenging to validate that a configuration change does not disrupt a service. Do we need to implement a rollback of configuration values? What if my configuration service is down, and my service cannot retrieve my external configuration? Taking a step back, what configuration values do we typically store, and what types of values would we modify at runtime? Let us consider, one could store the following information: Database Connection Information Timeout Values Service URLs https://service-a.com (yes, there is service discovery, but still, some URL needs to be stored) Feature Flags Other Constants? Which of these would we want to adjust to post-deployment? Feature Flags, but we would want this to be persisted across deployments. Timeout values could be utilized for experimentation. However, they might be temporary [7].

Table 3: Centralized vs. Local Configuration

|  |  |  |
| --- | --- | --- |
| Configuration Type | Advantages  or Disadvantages | Details |
| Embedded | Advantages | Easy to understand |
| Embedded | Advantages | Simplifies testing configuration for a specific state within the codebase |
| Embedded | Advantages | Local development is very convenient to initiate |
| Embedded | Advantages | Local changes to the configuration file do not impact on other developers. |
| Embedded | Advantages | Deployment is uncomplicated. |
| Embedded | Disadvantages | Secrets are exposed in the Git repository, which is not ideal. (Mitigation: AWS SSM) |
| Externalized | Advantages | Changing values is quick. |
| Externalized | Advantages | Solutions exist to poll for changes and apply them without needing to restart the container. |
| Externalized | Advantages | Shared configuration among services can be established in a single location. |
| Externalized | Disadvantages | There is uncertainty regarding how local development functions. |
| Externalized | Disadvantages | What happens if I am modifying values during development? |
| Externalized | Disadvantages | Do other individuals or services notice this private change? |

## Security and Observability Considerations

### Security

Security in microservices goes far beyond a perimeter firewall. In a distributed system with multiple independent services, it’s critical to implement defense-in-depth strategies that address identity, data flow, access control, and observability across all layers.

A key element of securing microservices is the API Gateway. Since microservices expose multiple endpoints, a gateway acts as a centralized access control point. It manages authentication, applies authorization policies, and protects against common threats using WAF.

Gateways like Amazon API Gateway or Spring Cloud Gateway consolidate access, reduce the attack surface, and ensure that traffic is properly filtered before reaching any internal services [8].

Even within private networks, assuming that internal communication is secure by default is a mistake. Microservices systems should adopt a Zero Trust approach, where internal service-to-service communication is encrypted using TLS. For enhanced identity verification and resistance against MITM attacks, mTLS is recommended. In mTLS, both services authenticate each other before any data exchange occurs [8].

For access control, microservices typically rely on authentication (who you are) and authorization (what you’re allowed to do). Real-world implementations often combine several access control models:

* Role-based access control for grouping user/service permissions.
* Attribute-based access evaluates conditions at runtime.
* Policy-based access based on defined business logic.
* Relationship-based access considering hierarchies and ownership.

In many cases, no single model is sufficient. Secure microservices systems blend these approaches, assigning unique identities to each service and limiting permissions according to the principle of least privilege.

To reduce load on authentication servers and improve response time, many architectures use JWTs. JWTs encode user identity and permissions, allowing services to validate them locally using JWKS without needing round-trip validation on every request. While efficient, JWTs should be short-lived or revocable to avoid stale or overly permissive tokens.

Rate limiting and DDoS protection are also essential, especially for public APIs. Techniques such as IP throttling, API key restrictions, and behavioral analysis can prevent malicious or accidental service overloads. These protections help maintain uptime and performance underload [8].

Internally, many systems now use service meshes like Istio or Linked to enforce security policies and route traffic. These tools use sidecar proxies to manage service discovery, mTLS enforcement, and telemetry collection.

They also provide observability features like traffic shaping, tracing, and access control — all essential for secure operations.

Secrets management is another foundational layer. API keys, database credentials, and tokens must never be hardcoded. Instead, secrets should be stored in dedicated tools like Hashi Corp Vault, AWS Secrets Manager, or Doppler. Secrets should be rotated regularly and scoped to the smallest set of permissions needed.

Lastly, a security system must be observable. Distributed tracing tools like Open Telemetry allow you to assign a unique ID to each request and trace it across multiple services. When combined with log aggregation platforms (e.g., Datadog, Splunk), these traces help detect suspicious patterns, debug failures, and respond to incidents quickly.

Together, these practices create a resilient and secure microservices architecture. They not only prevent unauthorized access and breaches but also ensure that incidents are detected early and mitigated efficiently [8].

### Observability

In a microservices environment, observability is all about knowing what’s happening inside your system — even when it’s made up of dozens or hundreds of small, independent services. The goal is to gain visibility into the internal state, performance, and health of your distributed application. To do that effectively, developers and operations teams rely on a set of observability patterns that provide actionable insights into system behavior.

Logging is the most common and foundational observability practice. Every Micro Service typically generates its own logs, recording key events, errors, and informational messages. These logs are then collected by a centralized logging service (e.g., ELK Stack, Loki) and sent to a searchable analytics tool.

This setup makes it possible to trace how one event flows through multiple services. For instance, if one service logs an error, centralized logs let you quickly check whether that error was triggered by an upstream service or caused a downstream failure [9].

**Application Metrics Pattern**

In addition to raw logs, metrics offer elevated numerical insights regarding system performance, such as CPU utilization, memory usage, response durations, or error frequencies. Metrics can be gathered from both specific microservices and the underlying infrastructure on which they operate.

For instance, if a particular service begins to consume more CPU than usual, effective metrics collection allows for immediate identification of whether this is a singular problem or indicative of a larger trend. Tools such as Prometheus, particularly when combined with Kubernetes, furnish this level of visibility and can even initiate alerts in the event of anomalies. [9].

**Distributed Tracing Pattern**

Distributed tracing monitors an individual user request as it moves through various microservices. This technique is particularly beneficial for identifying performance bottlenecks and determining the locations of failures within intricate systems.

For instance, when a user encounters an error, logs may indicate which service generated the report, yet they often fail to explain the underlying cause. A trace provides a comprehensive view of the request's journey, emphasizing which service experienced delays or triggered the error. Tools such as Open Telemetry, Jaeger, and Zipkin are frequently employed for tracing purposes. [9].

**Exception Tracking**

While logs and metrics provide an overview of general behavior, exceptions are instrumental in pinpointing specific application-level errors. Exceptions arise when the code does not function as anticipated as in the case of a failed database call or a null pointer.

Monitoring exceptions enables you to differentiate between infrastructure issues (such as a full disk and genuine bugs within the code. After isolating the service and the method that triggers the exception, developers can more efficiently debug and resolve the problem [9].

**Health Check APIs**

Each microservice ought to provide a health check endpoint that indicates the operational status of the service. These APIs offer valuable information regarding uptime, latency, error rates, and additional metrics.

Health checks serve a purpose beyond human oversight orchestration tools such as Kubernetes depend on them to determine whether to restart malfunctioning services. In their absence, services may seem functional to users, even when they are not performing correctly behind the scenes [9].

**Auditing**

In industries subject to regulation, the process of auditing is of paramount importance. It guarantees that applications operate in accordance with compliance standards, for instance, confirming that sensitive actions are recorded or that data access is adequately monitored.

Audit logs can be produced by services in a manner like standard logs, which can then be scrutinized to identify unauthorized access, atypical behavior, or breaches of policy. Tools for observability facilitate the automation of this analysis, thereby simplifying the process of ensuring compliance and addressing incidents. [9].

## Identified Research Gaps in Microservices Literature

**Maintainability Over Time**

While many studies highlight the short-term advantages of microservices—such as improved modularity and agility—the long-term effects on maintainability and technical debt remain understudied. Lenarduzzi et al. performed a four-year case study and found that although a migration to microservices initially spikes technical debt, over time debt growth slows significantly compared to the original monolith [DIVA Portal+13arXiv+13arXiv+13](https://arxiv.org/abs/1902.06282?utm_source=chatgpt.com).  
More recently studied a large-scale industrial microservices system (100+ services across 15,000 locations) and identified a “technical debt gamble,” where teams regularly accumulate and then pay down debt in cycles. Key drivers included poor communication and a misalignment between architecture and organizational structure [arXiv+1](https://arxiv.org/abs/2506.16214?utm_source=chatgpt.com). These findings underscore that while microservices can manage long-term debt more effectively, the dynamics are volatile and require governance strategies, a topic still lacking deep longitudinal research [21].

**Cost–Benefit Trade-offs**

Although microservices are widely touted for agility and scalability, quantitative cost analyses remain limited. One early study compared infrastructure costs of monolithic vs. microservices architectures and found that microservices hosted on cloud platforms can reduce   
However, anecdotal industry reports show the opposite: in one case, moving from microservices back to a monolith slashed infrastructure costs by over 90%   
Emerging “FinOps” practices, such as cost‑profiling Kubernetes-hosted microservices using open‑source APM stacks, promise more granular insights into deployment cost structures—but rigorous, peer-reviewed studies quantifying DevOps tooling investments and team‐training expenses are still sparse [22].

**Tooling and Observability Maturity**

Architectural literature frequently describes the need for observability—metrics, tracing, logging—but benchmark studies comparing tools (e.g., Prometheus vs. commercial APMs)   
Groundcover’s recent comparison highlights differences in feature breadth and pricing between Prometheus (open-source) and Datadog (commercial), but lacks performance   
Similarly, blogs describe APM features like distributed tracing and dependency mapping as crucial in microservices [blog.dreamfactory.com](https://blog.dreamfactory.com/how-apm-tools-monitor-microservices-data-flows?utm_source=chatgpt.com), but independent benchmarking in real-world, large-scale environments is largely absent from academic research or vendor-neutral studies [23].

**Impact on Team Productivity**

Several reports offer anecdotal insights into roles like Product Owners or Scrum Masters experiencing smoother deployments post-migration. Yet, empirical productivity metrics—like lead time, deployment frequency, or defect rates—before vs. after microservices   
One developer productivity study highlights that microservices’ complexity in cloud-native contexts can negatively impact team productivity, due to the overhead of tracking and   
Broader industry reports indicate that even AI-driven tooling only saves developers part of their time, with over 90% still hindered by organizational inefficiencies such as cross-team This suggests that while microservices offer structural advantages, their productivity impacts are highly context-dependent—and systematic, data-driven studies comparing before and after adoption within the same teams are missing [23].

# CHAPTER – RESEARCH METHODOLOGY

## Research Design

We will begin by developing the system in a monolithic fashion, then move towards microservices, each with its own configuration, and finally advance to a centralized microservice architecture. Building Microservices with Spring Boot: In this crucial section, we will explore the development of microservices using Spring Boot, focusing on configuration, RESTful APIs, and the essential concepts that are important for Java developers. Service Discovery and Load Balancing: This part will examine how services register and locate each other, as well as load balancing techniques designed to optimize resource use and improve performance. API Communication: This section covers the various ways in which microservices can communicate with one another, including REST calls and messaging solutions. Data Management: In this part, we will address the management of databases within microservices, emphasizing Spring Data JPA and the management of data transactions and consistency. Security in Microservices: Here, we will investigate the incorporation of security measures within microservices, employing JWT for API security and ensuring secure communication between services. Logging and Monitoring: This section is vital for production settings, discussing strategies for effective logging and monitoring microservices to maintain operational health. Testing and Deployment: Focusing on the deployment of microservices using Docker and Kubernetes, this section will cover key testing methodologies to ensure quality.

To facilitate fair comparison, both systems will be deployed in controlled settings with the same workloads. Configuration modifications will be implemented, and their impacts will be assessed using standardized performance metrics, system logs, and user experience indicators.

## Tools and Technologies

1.Docker  
Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production [10].

2.SpringBoot  
Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can just run. You can get started with minimum configurations without the need for an entire Spring configuration setup. Spring Boot offers the following advantages to its developers. Easy to understand and develop spring applications Increases productivity and reduces the development time [11].

3.Git  
is an open-source DVCS that allows developers to track and manage changes to their codebase. You can easily manage small as well as large projects with high speed and efficiency by Git. Unlike traditional version control systems, Git allows multiple developers to work on a project simultaneously without interfering with each other's work. We can use Git privately as well as publicly. Git offers numerous benefits to developers and development teams: Version Control: Git helps with tracking changes, allowing you to go back to previous states if something goes wrong. Collaboration: It enables multiple developers to work on a project simultaneously without interfering with each other’s work [12].

4.SpringCloud  
Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, short lived microservices and contract testing). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand-up services and applications that implement those patterns [11].

5.Keycloak  
is an open-source Identity and Access Management tool. Being an Identity and Access Management tool, it streamlines the authentication process for applications and IT services. The purpose of an IAM tool is to ensure that the right people in a company have appropriate access to resources. It usually enables the implementation of SSO, identity federation, and strong authentication [14].

6.Grafana  
Grafana open-source software enables you to query, visualize, alert on, and explore your metrics, logs, and traces wherever they are stored. Grafana OSS provides you with tools to turn your TSDB data into insightful graphs and visualizations. The Grafana OSS plugin framework also enables you to connect other data sources like NoSQL/SQL databases, ticketing tools like Jira or ServiceNow, and CI/CD tooling like GitLab [13].

7.Prometheus  
Prometheus is integrated for real-time monitoring and metrics collection. It provides visibility into application performance, resource usage, and system health, enabling objective measurement of scalability and maintainability impacts [13].

8.ApacheKafka  
is a distributed event streaming platform used to build real-time data pipelines and messaging systems. It allows microservices to communicate asynchronously by publishing and subscribing to events messages in a fault-tolerant and scalable way [15].

9.RabbitMQ  
is a lightweight message broker that enables services to send and receive messages using queues. It supports various messaging protocols and ensures reliable delivery, routing, and acknowledgment of messages between microservices [16].

10.Kubernetes  
is an open-source platform for automating deployment, scaling, and management of containerized applications. It manages clusters of containers and ensures applications run consistently, recover from failures, and scale as needed [17].

11.Helm  
Helm is a package manager for Kubernetes that simplifies deployment by using charts configured with application definitions. It allows you to define, install, and upgrade Kubernetes applications in a repeatable and manageable way [18].

12.Postman  
serves as the collaborative platform for teams to develop APIs collectively. Featuring integrated support for the MCP, Postman facilitates the design, testing, and management of APIs that drive both human workflows and intelligent agents [19].

13.Java  
Oracle Java stands as a prominent programming language and development platform. It effectively lowers expenses, accelerates development timelines, fosters innovation, and enhances application services. Java continues to be the preferred development platform for both enterprises and developers [20].

## Experimental Setup

This banking domain consists of three services: Accounts, Cards, and Loans, all managed through an API Gateway (Spring Cloud Gateway). Two variants are assessed as necessary: (i) a Monolithic baseline, and (ii) a Microservices deployment.

Platform. The environment is set up on Kubernetes using Docker Desktop (v1.32.x) with Container/Docker Engine; deployment is facilitated by Helm. Optionally, Istio/Linked is disabled unless a specific test necessitates mTLS.

Host hardware. The specifications include 8 vCPU, 16–32 GB RAM, and NVMe SSD; the operating systems supported are Windows 10/11 or macOS; the software stack includes Java 21, Spring Boot 3.4.x, and PostgreSQL 15 (containerized).

Network shaping. The network conditions are manipulated using Linux tc/netem (or the network controls of Docker Desktop) to introduce latency or packet loss for scenarios.

Observability stack. The monitoring framework comprises Prometheus and Grafana for metrics, Loki for logs, and the Open Telemetry SDK along with Jaeger for tracing.

Load generation. The load testing tools utilized are k6 (CLI) or JMeter.

Configuration management. The configuration is managed through Spring Cloud Config Server (HTTP) with a Git backend, and Kubernetes Secrets are employed for sensitive information.

All experiments are conducted with a minimum of 3 trials per condition, reporting the median and p95 latency where relevant. Between trials, the cluster is restored to a pristine state.

## Measurement

**Scalability Experiments**

Horizontal Scaling Responsiveness – Goal, procedure, and outputs.

Saturation Throughput & Knee Point – Goal, procedure, and outputs.

**Security Experiments**

Sec1. Secrets Rotation Drill – Goal, procedure, and outputs.

Sec2. mTLS Policy Enforcement – Goal, procedure, and outputs.

**Maintainability Experiments**

Configuration-Change  
Goal. Measure how quickly and consistently new configurations are applied across all services.  
Procedure. Deploy baseline configuration, introduce a config change in Git-backed config server, and trigger refresh endpoints. Measure time until all pods reflect the change.  
Outputs. Average and p95 propagation time; % of services updated without manual intervention.

Rollback-Complexity  
Goal. Assess how easily incorrect configurations can be reverted.  
Procedure. Apply a faulty config, initiate rollback using version control, and measure operational steps required.  
Outputs. Step count; total rollback duration.

Operational-Effort  
Goal. Quantify manual vs. automated effort needed for configuration updates.  
Procedure. Track all steps and commands issued during an update process.  
Outputs. Manual step ratio: time spent in human intervention.

# Chapter - SYSTEM DESIGN

## A diagram of a software system AI-generated content may be incorrect.Architecture Overview

Figure 5: Architecture Overview

1. The API Client refers to any external system, application, or tool (Postman) that initiates HTTP requests to engage with the backend system. It serves as the initial point for testing or utilizing APIs made available by microservices through a unified entry point As Show in Figure 5.

2. (Spring Cloud Gateway) As Show in Figure 5, The API Gateway functions as the sole entry point for all requests from external clients. It undertakes essential cross-cutting tasks such as: Routing requests to the corresponding microservice (e.g./accounts, /loans) Enforcing security measures (authentication & authorization) Implementing rate limiting, retries, logging, and load balancing Serving as an abstraction layer that conceals internal service implementation details from the client.

3. (Service Registry) As Show in Figure 5, Eureka Server acts as the service discovery mechanism offered by Spring Cloud Netflix. Each microservice: Registers itself upon initialization, providing its name, IP address, and port number. Dynamically discovers other services using the registry (for instance, the Accounts service can find and invoke the Loans service without hardcoding its address). This facilitates dynamic scaling and fault tolerance in distributed environments.

4. Config Server (Spring Cloud Config Server) As Show in Figure 5, The Config Server offers centralized and externalized configuration management. Rather than hardcoding configurations within each microservice, they: Retrieve configurations at startup or during refresh from the Config Server. Obtain consistent values for aspects such as database credentials, service ports, feature toggles, and environment-specific properties. Configurations are generally stored in a Git repository or a local file system overseen by the config server.

5. (Accounts, Cards, Loans) As Show in Figure 5, Each microservice is crafted with the Single Responsibility Principle in mind, meaning it possesses a distinct business capability: Accounts Service – Oversees user accounts and profile information. Cards Service – Manages card-related data and operations. Loans Service – Manages loan processing, approval, and related tasks.

6.Event Broker (RabbitMQ / Kafka) The Event Broker facilitates asynchronous communication among microservices using events: Publishing Events: Services disseminate domain events (e.g., "Loan Approved", "Card Created") to the broker. Decoupled Processing: Other services can subscribe to pertinent topics without establishing direct dependencies. Scalability & Resilience: This approach diminishes tight coupling and improves responsiveness in environments with high load.

7.Message Consumer A message consumer is a backend component (or service) that listens to and processes incoming events from the broker: Asynchronous Handling: This enhances performance by delegating processing tasks. Loose Coupling: It allows microservices to respond to events without awareness of their source. Use Cases: Examples include sending notifications, updating projections, or initiating workflows.

8.Observability & Monitoring (Prometheus, Grafana) This layer is essential for sustaining operational visibility: Metrics Collection: Prometheus collects service metrics (CPU, memory, request latency). Visualization: Grafana offers dashboards for monitoring health, usage, and trends. Distributed Tracing: Open Telemetry facilitates tracing across services to pinpoint performance bottlenecks. Health Checks: Services provide /actuator/health and tracing endpoints to oversee uptime and readiness.

As the system evolves from a monolithic architecture to a microservices-oriented design, Docker assumes a crucial role in encapsulating and managing each phase of this transformation. Initially, the application was constructed as a conventional monolith utilizing Spring Boot. During this phase, Docker is employed to package the entire application into a singular image, encompassing all dependencies and configuration files. Typically, the configuration is integrated within the application itself, often represented as an application. Properties file that is bundled directly within the JAR. This Docker image facilitates consistent execution across various environments; however, it lacks flexibility—any modification in configuration and requires the image to be rebuilt and redeployed.

As the architecture progresses towards microservices, the application is segmented into distinct business domains, such as Accounts, Cards, and Loans. Each microservice is crafted as an independent Spring Boot application, complete with its own Docker image. These images continue to include embedded configuration files, rendering them self-sufficient yet inflexible. A change in configuration for one service necessitates the rebuilding and redeployment of its Docker image, resulting in the duplication of environment specific values across services and prolonged feedback loops during development and testing.

The subsequent phase introduces a centralized configuration management system using Spring Cloud Config Server. This separate service, the Config Server, is also containerized using Docker. Configuration values are no longer embedded within each service image; instead, they are stored externally typically in a Git repository or a mounted local file system that is accessible to the Config Server. Microservices are reconfigured to dynamically retrieve their configuration at startup from the Config Server. This architecture allows the Docker images of the microservices to remain unchanged even when configuration values are modified. Environment-specific details, such as database URLs, ports, and credentials, are relocated outside the service container.

Steps:

API Client – Any external system or tool (e.g., Postman) that sends HTTP requests to the backend via a unified entry point (Figure 5).

Spring Cloud Gateway – Central entry points that routes requests (/accounts, /loans), enforces authentication/authorization, applies rate limits, logging, retries, load balancing, and hides internal service details (Figure 5).

Service Registry (Eureka) – Microservices register themselves (name, IP, port) and discover others dynamically, enabling scaling and fault tolerance (Figure 5).

Config Server (Spring Cloud Config) – Centralized configuration source. Services fetch environment-specific settings (DB credentials, ports, toggles) from Git-backed config server at startup or refresh (Figure 5).

Business Services (Accounts, Cards, Loans) – Each service implements a single business capability:

Accounts: Manages user accounts/profile data.

Cards: Handles card operations.

Loans: Processes and manages loans (Figure 5).

Event Broker (RabbitMQ/Kafka) – Enables asynchronous, decoupled communication by publishing domain events (“Loan Approved”, “Card Created”) to subscribed services.

Message Consumers – Services that process events from the broker for tasks like notifications, projections, or workflows, without direct coupling to the event source.

Observability & Monitoring (Prometheus, Grafana) – Collects metrics, visualizes dashboards, traces distributed requests, and performs health checks for system reliability.

Steps Flow Diagram

A diagram of a software company

AI-generated content may be incorrect.

## Implementation Approach

### microservice

This project is an independent, self-contained application that handles specific business capability (such as accounts, cards, or loans). Each microservice has its own codebase, database, and can be developed, deployed, and scaled independently. Microservices communicate with each other over the network (usually via REST APIs), often using DTOs to exchange data. Service discovery (Eureka), centralized configuration (Config Server), and monitoring/tracing are used to manage and observe the microservices. This architecture improves modularity, scalability, and maintainability of the overall system.

1. Config server: A centralized configuration server designed for all microservices, providing configuration properties.
2. Eureka server: A service registry that facilitates service discovery, enabling microservices to locate and interact with one another.
3. accounts: Oversees business logic and data related to accounts (e.g., user accounts, balances).
4. loans: Manages operations associated with loans, including applications, approvals, and overall management.
5. cards: Oversees services related to cards, such as the issuance and management of credit and debit cards.
6. message: Presumably responsible for messaging or notifications between services or directed towards users.
7. Gateway server: An API gateway that directs external requests to the corresponding microservices, manages authentication, and serves as a unified entry point.

All Microservice inside the project As Show in Figure 6.

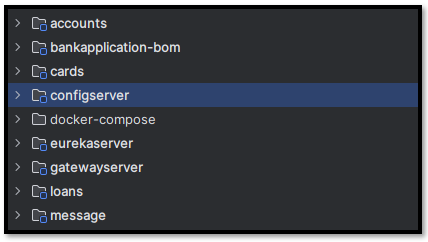


Figure 6: Microservices in Application

### ****Maven Dependance****

is a reference in your pom.xml file that tells Maven to download and include a specific library or module in your project. Dependences are defined inside the <dependencies> section. This will add the Spring Boot Web starter to your project. Maven will automatically download it and make it available for your build As Show in Figure 7.

The following are examples of various Spring Boot starters and their functionalities: spring-boot-starter-actuator: This starter adds endpoints that facilitate the monitoring and management of the application. opentelemetry-javaagent: This component enables distributed tracing, enhancing observability. micrometer-registry-Prometheus: This starter exposes metrics that can be utilized by Prometheus. spring-boot-starter-data-jpa: It provides support for JPA and Hibernate, allowing for database access. Spring-boot-starter-validation: This starter enables bean validation capabilities. Spring-boot-starter-web: It serves as the core dependency for constructing RESTful APIs. Spring-boot-dev tools: This tool enhances the development experience by allowing hot reload during runtime. h2: This is an in-memory database designed for development and testing purposes, applicable at runtime only. Spring Boot Starter Test: It offers testing support specifically for Spring Boot applications. Lombok: This library minimizes boilerplate code using annotations.

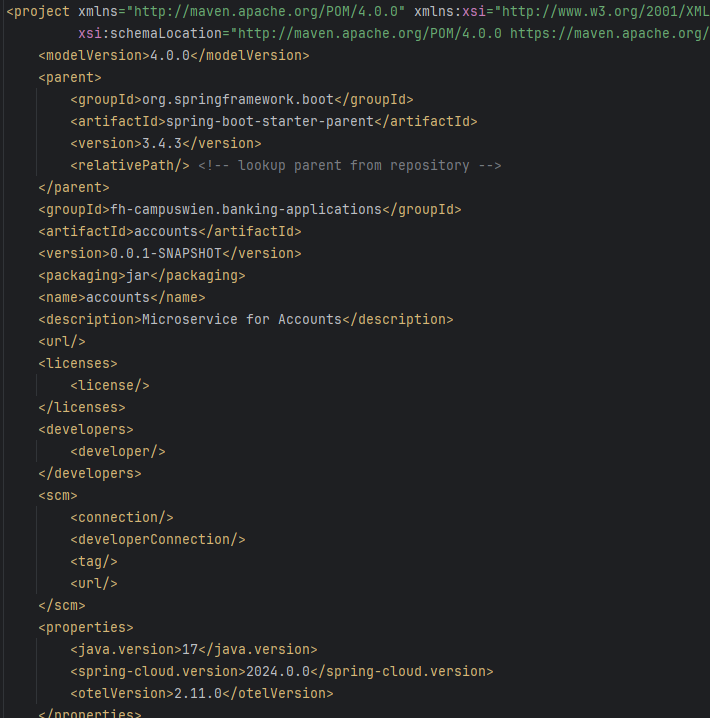


Figure 7: Account Maven Dependance Example

### Entity

In this project, an entity is defined as a Java class that corresponds to a table within the database. Entities are generally marked with the @Entity annotation and are integral to the persistence layer, overseen by Spring Data JPA. Each entity correlates its fields with the columns in the database, enabling the application to execute CRUD operations on the underlying data As Show in Figure 8.

Entities serve to represent fundamental business objects, including accounts, customers, loans, or cards. They frequently incorporate annotations such as @Id for primary keys and may utilize relationships like @OneToMany or @ManyToOne to establish connections between tables.

By employing entities, the project capitalizes ORM, facilitating interaction with the database through Java objects rather than direct SQL queries. This methodology enhances maintainability, readability, and compatibility with Spring Boot’s data management capabilities.

The Customer class represents a customer in the database for the accounts service. It is a JPA entity mapped to a database table, with fields for customer ID, name, email, and mobile number. By extending Base Entity, it also inherits auditing fields (created/updated timestamps and user info). This class is used to persist, retrieve, and manage customer data within the application.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 8: Customer Class Entity Example

### Data Transfer Object

A DTO in a microservices architecture is used to transfer data between services, often over the network (e.g., via REST APIs). DTOs help decouple internal domain models from external representations, ensuring that only necessary data is shared between microservices. This improves security, versioning, and maintainability. For example, when the accounts service calls the cards service to fetch card details, it uses a DTO to structure the request and response payloads, making inter-service communication clear and consistent. Each microservice defines its own DTOs for both incoming and outgoing data.

Example: The CardsDto class serves as a Data Transfer Object (DTO) for card information within the application. Its main purpose is to encapsulate and transfer card-related data between different layers (such as controller and service) or between microservices, ensuring that only the necessary card data is exposed and validated. It also supports API documentation and input validation through annotations, promoting clear contracts and data integrity in the system As Show in Figure 9.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 9: Data Transfer Object Example

### Repository

In this project, a repository is defined as a Java interface that facilitates data access for entities, usually by extending JpaRepository from Spring Data JPA.

Repositories serve to abstract the database layer, enabling the execution of CRUD operations (create, read, update, delete) on entities without the necessity of writing SQL queries.

The responsibilities of repositories include:

* Interacting with the database to retrieve, save, update, or remove entity records.
* Providing custom query methods through method naming conventions.
* Supporting pagination, sorting, and complex queries.
* This methodology fosters a clear distinction between business logic and data access, thereby enhancing the maintainability and testability of the codebase.
* A computer screen shot of a program

  AI-generated content may be incorrect.They define the contract for account-related operations in the application. It specifies methods for creating, fetching, updating, and deleting accounts, as well as updating account status. Implementations of this interface provide the business logic for managing account data, ensuring a clear separation between the service layer and other layers (such as controllers or repositories). This promotes modularity, testability, and maintainability in the codebase As Show in Figure 10.

Figure 10: Account Repository Example

### Controller

In a Spring Boot application, the controller functions as a fundamental element of the web layer, tasked with managing incoming HTTP requests, processing them (often by delegating responsibilities to service classes), and delivering suitable HTTP responses. Controllers are marked with @RestController or @Controller and establish request mappings through annotations such as @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping as Show in Figure 11.

Controllers act as the initial point of contact for client interactions, providing RESTful APIs that enable clients to execute operations like creating, reading, updating, or deleting resources. They are responsible for validating input data, managing exceptions (frequently with the assistance of global exception handlers), and returning responses in a well-structured format, usually in JSON.

A screen shot of a computer program

AI-generated content may be incorrect.The Accounts Controller in fh\_campuswien.banking\_applications.accounts.controller exposes REST APIs for managing account-related operations. It provides endpoints to create, fetch, update, and delete accounts, as well as to retrieve build info, Java version, and contact information. The controller uses validation, integrates with the service layer, and leverages resilience patterns like retry and rate limiting. It ensures that all account operations are accessible via well-defined HTTP endpoints, returning clear and consistent responses.

Figure 11: Account Controller Class Example

### Exceptions Handler

The purpose of the is to centralize and manage exception handling for the accounts microservice. It contains custom exception classes and a global exception handler to: Define specific exceptions for business errors (e.g., customer already exists, resource not found).

Intercept and handle exceptions thrown during request processing.

In fh\_campuswien.banking\_applications.accounts.exception package as Show in Figure 12, serves as a centralized component for exception handling within your Spring Boot application. It is annotated with ControllerAdvice, allowing it to intercept exceptions raised by controllers and deliver custom responses, thereby enhancing error reporting and the overall client experience. Key features: Validation Error Handling: This feature overrides the handleMethodArgumentNotValid method to capture validation errors (such as invalid request bodies) and returns a map detailing field error along with a 400 Bad Request status. Global Exception Handling: The handleGlobalException method is designed to catch all unhandled exceptions, providing a structured error response accompanied by a 500 Internal Server Error status. Custom Exception Handling: There are specific handlers for CustomerAlreadyExistsException and ResourceNotFoundException that return meaningful error messages along with the appropriate HTTP status codes (400 and 404).

Benefits: Ensures uniform error responses throughout the application. Enhances maintainability by consolidating exception logic. Improves API usability by offering clear and structured error messages to clients.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 12: Global Exception Class Example

### Application YML file

The configures your Spring Boot application. It defines settings for the server, database, logging, service discovery, messaging, monitoring, and resilience as Show in Figure 13. This file centralizes environment-specific and service-specific properties, enabling features like:

* Setting server port and application name
* Configuring the H2 in-memory database and JPA/Hibernate
* Enabling the H2 console for development
* Importing external configuration from a config server
* Setting up Kafka messaging and (optionally) RabbitMQ
* Exposing and customizing Spring Boot Actuator endpoints for monitoring and health checks
* Registering with Eureka for service discovery
* Defining logging levels and patterns
* Configuring Resilience4j for circuit breaking, retrieving, and rate limiting
* This file ensures the application runs with the correct infrastructure, resilience, and monitoring settings.

A screen shot of a computer

AI-generated content may be incorrect.

Figure 13: Application.yml file Example

## Configuration Strategy

### API

The Swagger UI functions as a web-based interface that presents the technical specifications of the services within a software system. The accompanying screenshot illustrates the Accounts microservice, which is a part of a larger banking system tasked with functions such as opening accounts, updating customer information, and retrieving account data. Below is a clear explanation of what the Swagger UI reveals:

Purpose of the Page: It aids users (especially software developers or testers) in comprehending: What operations the system can perform (for example, creating a new account or retrieving account details) What information is required (such as name, email, and account number) What type of response will be generated (whether it is a success or an error, along with the relevant message)

Main Features of the Accounts Microservice: Create an Account: You can direct the system to create a new customer account by providing necessary information such as: Full name, Email address, Phone number, Account type (for instance, Savings), Branch address. Update Account Information: This feature allows for the alteration of a customer's contact or account details. Fetch Account Details: This function enables you to access all information related to a specific account. Delete an Account: This action removes a customer and their account information from the system. View Build and Version Info: This displays the current version of the service in use along with technical build information. Fetch Customer Details: This retrieves both customer and account information at once, which is advantageous for comprehensive profile views.

What Kind of Data Is Involved? The system expects and returns data in a structured format that includes Customer Info: name, email, mobile number; Account Info: account number, type, and branch; Other Info: error messages (in case of issues), success codes, or system responses.

Why It’s Useful (Even for Non-Technical Stakeholders): It ensures that all components of the system interact effectively. It supports developers, testers, and architects in their tasks.

**All Microservices API documents PDF will be in folder API-Document in GIT-Hub Repository.**

**This example for API Account microservice as Show in Figures 14 – 17.**

**Link:** [**AimenFH/Banking-Applications: Developing Resilient and Scalable Event-Driven Microservices for Banking Applications**](https://github.com/AimenFH/Banking-Applications)

**A screenshot of a computer

AI-generated content may be incorrect.**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 14: API Example for Account micro service 1

Figure 15: API Example for Account micro service 2

Figure 16: API Example for Account micro service 3

Figure 17: API Example for Account micro service 4

A screenshot of a computer program

AI-generated content may be incorrect.

### Docker

In modern software development, Docker is used to package and run applications in isolated environments called containers. These containers bundle everything an application needs: the code, dependencies, and configuration. For microservices-based applications, Docker plays a critical role in ensuring portability, consistency, and scalability across development, testing, and production environments.

In this setup, we containerize Spring Boot applications (like Accounts, Loans, and Cards microservices) using three different methods, each suited for different developer needs and automation levels.

Three Ways to Containerize a Spring Boot Microservice

1. Docker file

2. Build packs

3. Google Jib

We decided to go Build packs in project as it is a secure and easier way for all microservice images As Show in Figure 19.

This section defines the account microservice as Show in Figure 18 in Docker Compose setup. It specifies:

The Docker image to use (aimendocker/account:s14).

The container name (docuker-ms).

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 18: Account Info for Docker Image

A screenshot of a computer

AI-generated content may be incorrect.

Figure 19 : Docker Image for Microservices

Docker Compose is utilized in this project to manage sophisticated microservices architecture, allowing developers to define, configure, and operate multiple interrelated services through a single YAML file. The Compose configuration encompasses essential business microservices (accounts, loans, cards, message, Gateway server), infrastructure elements (Kafka, Redis, RabbitMQ, Keycloak, Prometheus, Grafana, Loki, Tempo, Minio, Nginx), and auxiliary services (Config Server, Eureka Server). Each service is characterized by its image, ports, environment variables, health checks, dependencies, and a shared network. The implementation of extends and common configuration files (such as common-config.yml) adheres to DRY principles, centralizing resource limits, environment variables, and network configurations. This methodology streamlines scaling, testing, and deployment, facilitating the rapid initiation of the entire ecosystem for development, testing, or production. Docker Compose further oversees service dependencies and health checks, guaranteeing that services like Eureka and Config Server are operational before dependent microservices commence. Observability is incorporated through Prometheus, Grafana, Loki, and Tempo, delivering monitoring, logging, and tracing capabilities out of the box. The configuration promotes resilience, security (with Keycloak for authentication), and event-driven communication (via Kafka and RabbitMQ).

Three Distinct Environments:

Development Environment: Employs in-memory or lightweight databases (e.g., H2, local volumes). Opens all ports for straightforward access and debugging. Facilitates hot-reload and developer tools.

Testing/Staging and Production Environment: Replicates production but may utilize reduced resource limits. Health checks and dependencies are rigorously enforced. Data persistence is enabled through Docker volumes.

### Config Server

The configuration file delineates a Spring Boot application intended to function as a centralized configuration server within a microservices architecture. Its primary role is to externalize and oversee configuration properties for all related microservices from a singular, version-controlled source, thereby guaranteeing consistency, improving manageability, and enabling dynamic updates across diverse environments.

The application is given a logical name for identification when it registers with other components in a distributed system, such as a discovery server. This naming convention assists client applications in finding the configuration server and obtaining their specific configuration files based on their service identifiers.

The configuration profile activates the Git backend, signifying that the application will derive configuration properties from a remote Git repository. This selection permits version control of configuration files, ensuring that modifications are traceable over time, while also allowing rollback capabilities and promoting collaboration by treating configuration as code. An alternative native profile is available but commented out, which would facilitate loading configurations directly from the local file system or class path, beneficial in testing scenarios or when Git is not accessible.

Within the Git configuration section, various properties govern the application's interaction with the remote repository, including the repository's location, the default branch, and performance settings such as timeouts. The server is set to automatically clone the Git repository upon startup and to pull the latest changes with each refresh, ensuring that the configuration server consistently delivers the most up-to-date properties to client services without necessitating manual updates.

In summary, this setup creates a secure, scalable, and centralized system for overseeing externalized configurations across various microservices. It effectively tackles issues such as configuration drift, manual property overrides, and inconsistent deployments, thus enhancing the reliability, maintainability, and agility of the overall software architecture.

Developer Initiates Configuration Update:

A developer pushes new or modified configuration files to the centralized configuration repository (for instance, GitHub) as Show in Figure 20.

Webhook Activation:

A designated webhook on the Git repository identifies changes and automatically alerts the Config Server.

Broadcast of Configuration Change Event:

The Config Server transmits a configuration refresh event to a message broker (such as Kafka or RabbitMQ), which informs all subscribed microservices as Show in Figure 21.

Dynamic Configuration Reload:

Subscribed microservices (like Accounts, Loans, and Cards) receive the event and update their configurations dynamically without necessitating a restart, thereby facilitating uninterrupted updates.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 20: Config Server GitHub

A screen shot of a computer

AI-generated content may be incorrect.

Figure 21: Config Application YAML File

### Client-Side Service Discovery

1. As Show in Figure 22 Client-side service discovery is an architectural strategy utilized in distributed systems, where client applications dynamically locate and interact with services during runtime. Rather than depending on a central load balancer or gateway, each client queries a centralized service registry to gather information about the service instances it needs to connect with. Upon initialization, every microservice instance registers itself with the service registry, providing essential details such as its IP address, port number, and other relevant metadata. When a service instance is terminated or fails, it unregisters or is eventually removed from the registry, ensuring that the discovery data remains current.

A computer screen shot of a black background with orange text

AI-generated content may be incorrect.

Figure 22: Eureka Discovery

2.framework, when a client intends to communicate with another service, it queries the service registry to obtain a list of available instances, as shown in Figure 21. The client then selects one of these instances based on a load-balancing strategy defined within its logic, which may include methods like round-robin selection, least-connections, or more sophisticated latency-aware approaches. While this model enhances flexibility and efficiency by removing centralized bottlenecks, it also adds complexity to the client side, which must now incorporate service discovery and load balancing logic.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 23: Eureka Example

3.TheSpring Cloud ecosystem provides strong support for this client-side discovery pattern. Eureka, a popular service registry created by Netflix, serves as the foundation for service registration and discovery. Clients utilize the Spring Cloud Load Balancer, which replaces the now-obsolete Netflix Ribbon, to manage load balancing among service instances. Additionally, feign clients are frequently employed in Spring applications to streamline HTTP communications between services and integrate smoothly with the discovery mechanism. Although Eureka is predominantly used in Spring-based applications, other service registries like Consul, etc., and Apache Zookeeper offer similar functionalities and are utilized in non-Spring or polyglot environments.

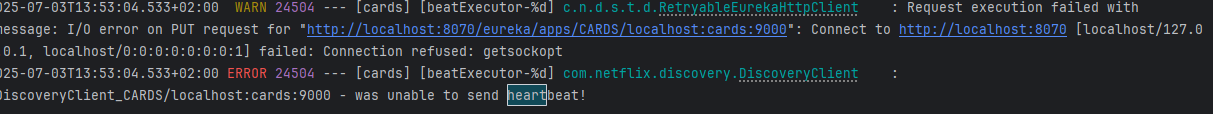
4. As Show in Figure 24, heartbeat in a service discovery system like Eureka is a periodic signal sent by service instances to the service registry to indicate that they are active and healthy. These signals allow the registry to maintain an up-to-date list of available services. If a heartbeat is missed for a configured duration, the registry may assume the service is unavailable and remove it. However, to prevent false removals due to temporary issues, Eureka uses self-preservation mode, which delays eviction if too many heartbeats are missed system-wide, ensuring continued stability.

Figure 24: Eureka heartbeat Example

### RESILIENCY

Ensuring resilience and stability in microservices architecture requires careful handling of failures and network disruptions between services. A single slow or failed service should not cause cascading failures across the system. To mitigate this risk, fallback mechanisms are employed, allowing the system to return default values, fetch data from caches, or reroute to alternative services or databases when a dependent service is unavailable. This approach ensures continuity even in partial failure scenarios.

In addition to fallbacks, it is crucial for services to exhibit self-healing behavior. This involves configuring timeouts and automatic retries, allowing temporarily failing services to recover without manual intervention. Tools such as Resilience4J, which has become a popular alternative to the now-deprecated Hystrix library, provide robust support for such resilience patterns in the Java ecosystem.

The Circuit Breaker pattern, inspired by its counterpart in electrical systems, plays a key role in protecting services from being overwhelmed by repeated failed calls. It monitors remote calls and interrupts them if they become unresponsive or consistently fail, thereby preventing further strain on the system. Once the issue is resolved, the circuit breaker allows traffic to flow again, ensuring minimal downtime and improved fault tolerance.

A screen shot of a computer program

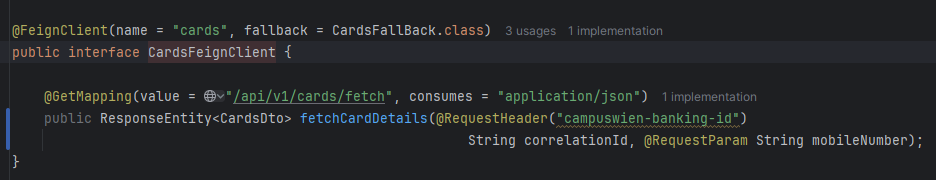
AI-generated content may be incorrect.Complementing this is the Retry pattern, which allows operations to be retried after transient failures, particularly useful in scenarios like brief network disruptions. It includes strategies such as exponential backoff to avoid overwhelming resources and integrates well with circuit breakers to halt retries after a certain threshold, maintaining system efficiency. To avoid unintended consequences, operations subject to retrieves must be idempotent, producing the same result regardless of how often they are executed.

Figure 25:Card Service Feign Client

Another crucial resilience strategy, like Show in Figure 26 is the Rate Limiter pattern, which protects services from being overloaded by excessive or abusive requests. By limiting the number of allowed requests within a specific time frame—based on IP address, user, session, or subscription tier—the system ensures fair usage and maintains availability. When limits are exceeded, requests are rejected with a standard response, typically HTTP 429 (Too Many Requests). This not only protects system performance but also supports differentiated service levels for various user groups, such as basic and premium users.

Together, these patterns—fallbacks, circuit breakers, retries, and rate limiting—form the foundation for building resilient, self-healing, and stable microservices systems capable of withstanding failures and continuing to deliver reliable user experiences.

These settings help your service handle failures gracefully, avoid cascading errors, and protect downstream systems by controlling retries and request rates.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 26: Circuit breaker Example

### Observability and Monitoring

Observability refers to the ability to understand the internal state of a system by analyzing its external outputs. In the context of microservices, observability is achieved by collecting and evaluating data from various sources such as metrics, logs, and traces. Metrics provide quantitative insights into the health of a system, including CPU usage, memory usage, and response times. Logs serve as chronological records of events occurring within the system, helping to trace errors, exceptions, and other unexpected behaviors. Traces capture the journey of a request across the microservices architecture, helping to identify latency and performance bottlenecks. These three pillars together enable comprehensive visibility into a system's behavior and health.

Monitoring complements observability by continuously examining telemetry data and generating alerts for known failure states. It focuses on identifying and resolving problems, tracking the health of individual services, and optimizing system performance. Monitoring is reactive in nature, responding to issues as they arise, while observability takes a more proactive approach by offering insights that help us understand and resolve root causes in real time.

Logs are essential tools for diagnosing problems and understanding system behavior. Each log entry includes a timestamp and contextual information about specific events. Log levels such as trace, debug, info, warning, and error allow filtering of log data based on severity. While monolithic applications store all logs in a centralized manner due to their single-codebase nature, microservices architectures require centralized logging systems to consolidate logs from multiple services. This is crucial for tracking issues across services and reconstructing the sequence of events for individual requests.

Grafana is a widely used open-source tool that provides visualization for metrics, logs, and traces. It enables interactive dashboards and alerting mechanisms by connecting to various data sources. Grafana Loki, designed for scalable and cost-effective log aggregation, works alongside Grafana Alloy, a lightweight log agent that ships logs from containers to Loki. Together, they offer a powerful and scalable logging solution, allowing centralized management and analysis of logs across distributed microservices environments.

We will be going now to show examples of Configuration and Result we can log on to our App by Prometheus Grafana. And Loki as Show in Figures 27 – 32.

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 27: Prometheus Configuration

A screenshot of a computer

AI-generated content may be incorrect.Checking Hardware used by Prometheus

Figure 28: Account Micro Service Actuator

A screen shot of a graph

AI-generated content may be incorrect.A screenshot of a computer

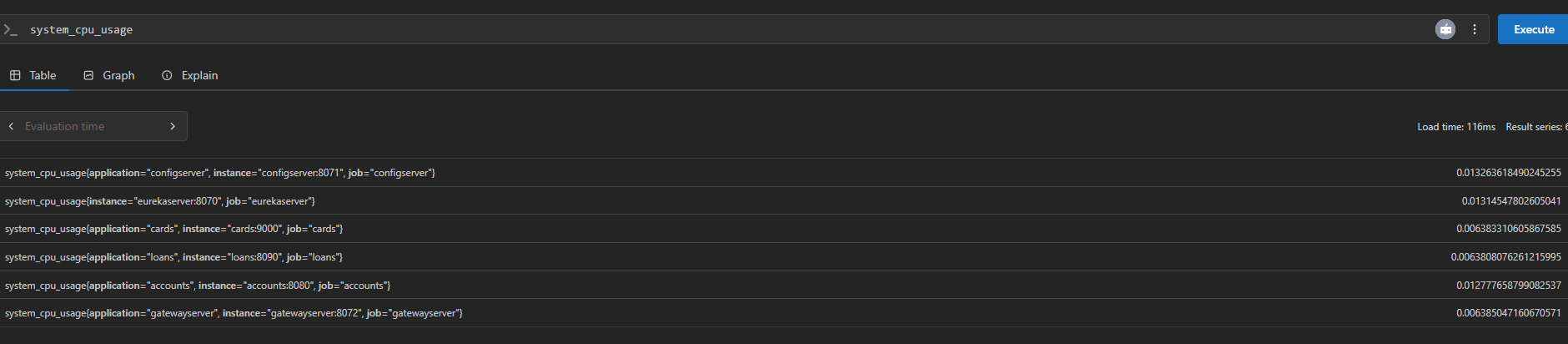
AI-generated content may be incorrect.

Figure 29: Checking CPU Usage for Microservices in Prometheus Dashboard

Figure 30: Checking CPU Usage Graph for Microservices in Prometheus Dashboard

Figure 31:All Microservices State shown in Prometheus Dashboard

Now we will use Grafana to better view Data of Prometheus.

Figure 32: Grafana View of CPU Usage of Microservices

A screenshot of a computer

AI-generated content may be incorrect.

by using Grafana we have two options to create our own dashboard to take a ready created one which we can have better overview on overall information microservice and adjust other Information we want to view as Shown in Figure 33.

A screenshot of a computer

AI-generated content may be incorrect.We are going now to set alert on Account microservice inside graphene and mock it with Hook deck, after setting everything we are going to shut down Account microservice and A screenshot of a computer

AI-generated content may be incorrect.we expect to get alert on Hook deck about the status of down Account microservice as Shown in Figures 33 – 34.

Figure 33: JVM (Micrometer) Dashboard for Account Microservice

Figure 34:Grafana Alert when Account Micro service down

A screenshot of a computer

AI-generated content may be incorrect.

Figure 35: Hook deck Response When Account Micro service down

### Distributed Tracing

While conventional tools such as event logs, health probes, and metrics provide significant insights into the internal state of an application, they are inadequate in cloud-native environments where a single user request frequently traverses multiple distributed services. These tools encounter difficulties in correlating data across service boundaries.

Distributed tracing effectively addresses this issue by monitoring the path of a request as it moves through various microservices. It enables developers to comprehend request flows, identify performance bottlenecks, and resolve issues within complex, interconnected systems. A fundamental component of distributed tracing is the correlation ID—a distinct identifier assigned to each request at its entry point.

This ID is transmitted to all services involved, facilitating the linking of logs and traces throughout the system. Distributed tracing is founded on three essential components: Trace: A comprehensive representation of a request, identified by a trace ID and consisting of multiple spans. Span: Denotes a single step in the request’s journey, encompassing its start and end times, associated with a trace ID and its own span ID. Tags: Metadata that offer context, such as the user’s identity, request URL, or tenant ID as Shown in Figure 37.

By adopting distributed tracing, teams achieve enhanced visibility into the collective behavior of their microservices, resulting in expedited debugging, improved performance monitoring, and more robust systems.

When a client request is received at the edge server or the first service within the network, a trace ID, such as 29cdbe2e21bc, will be generated, and it will remain consistent throughout the process as Shown in Figure 36.

We made a request that will get information from 3 microservice with all customer details and now we will check trac id of request to see it path throw our micro service

Figure 36: Open Telemetry Pattern

**A screenshot of a computer

AI-generated content may be incorrect.**A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 37: Trace ID from Account Microservice

Figure 38: Trace ID from Card Microservice

Figure 39: Temp show Path of Request

By firing request from post man we got an id, with id we could trace request through each microservice also see graph of path through LOKI as Shown in Figure 39.

### OpenID Connect, KeyCloak

OpenID Connect serves as a straightforward identity layer that is constructed upon the OAuth 2.0 framework. While OAuth 2.0 primarily focuses on authorization granting applications the ability to access user data OpenID Connect incorporates the crucial element of authentication. This indicates that OpenID Connect not only informs an application of its permission to access a resource but also identifies the user.

The principal distinction is found in the ID Token that OpenID Connect introduces. This token is a JSON Web Token that encompasses structured identity information (referred to as claims regarding the authenticated user, including their name, email, and profile information. Consequently, applications can ascertain the identity of the user without the need to manage separate login credentials.

The elegance of OpenID Connect is evident in its ability to standardize identity sharing. Through a single protocol, numerous applications and services can securely share and validate user identities across the internet. This is particularly vital in contemporary times, as users anticipate the ability to utilize their Google, Facebook, or corporate credentials to log in to various services without the necessity of repeatedly creating accounts.

Although OAuth 2.0 is effective for permitting applications to access limited resources such as calendars or photos, it does not concern itself with the identity of the user. This is where OpenID Connect excels. By integrating identity on top of OAuth 2.0, it assists developers in implementing comprehensive IAM

The Importance of OpenID Connect:

* Enhances OAuth 2.0 by adding identity, thereby completing the authentication and authorization framework.
* Standardizes login processes across various applications through a secure and reliable mechanism.
* Enhances user experience by facilitating SSO across different applications.
* Protects user data, as identity claims are both verified and cryptographically signed.
* Accommodates mobile, web, and API-based applications with a contemporary authentication flow.
* OpenID Connect also introduces several key features:
* It establishes standardized scopes such as OpenID, profile, email, and address.
* It provides ID Tokens that allow for the verification of user identity.

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 40: Security Config

Establishes a Spring Security configuration tailored for a Spring Boot Web Flux application, functioning specifically as a gateway server. It secures endpoints through JWT-based authentication and role-based access control, which is integrated with an OAuth2 provider like Keycloak.

The class is designated as a configuration component for reactive security by utilizing the @Configuration and @EnableWebFluxSecurity annotations. Within this class, the springSecurityFilterChain bean delineates the security behavior. It permits all HTTP GET requests to proceed without any restrictions. However, access to certain API paths such as /campuswien-banking/accounts/, /cards/, and /loans/\*\* is limited to users possessing the respective roles ACCOUNTS, CARDS, and LOANS as Shown in Figures 40 and 41.

The configuration activates OAuth2 resource server capabilities, employing JWT tokens for authentication purposes. To manage the extraction of roles from the JWT tokens, it utilizes a custom converter referred to as KeycloakRoleConverter. This mechanism guarantees that the roles embedded within the token are accurately interpreted and enforced by Spring Security.

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 41: Key Clock Filter Chain

In this configuration, CSRF protection is disabled, as the gateway processes stateless API requests where CSRF protection is deemed unnecessary. The grantedAuthoritiesExtractor method is tasked with customizing the way authorities (roles) are extracted from the JWT token, ensuring alignment with the structure utilized by Keycloak as Shown in Figures 42 and 43.

This configuration is implemented to guarantee that only authorized users with designated roles can access protected resources via the gateway, thereby serving as a pivotal element of the system’s authentication and authorization framework.

Established two global filters within a Spring Cloud Gateway server to facilitate distributed tracing by managing a correlation ID. These filters are part of the package fh\_campuswien.banking\_applications.gatewayserver.filters and are utilized to monitor requests and responses across microservices.

The RequestTraceFilter functions as a global pre-filter that examines incoming HTTP requests for the presence of a correlation ID. If the correlation ID is found in the headers, it logs the ID for traceability purposes. Conversely, if it is absent, a new correlation ID is generated using a UUID and appended to the request exchange. This guarantees that each incoming request is assigned a unique identifier that can be utilized for tracking throughout the system.

The ResponseTraceFilter is designated as a post-filter. Once the request has been processed by downstream services, this filter extracts the correlation ID from the request headers and incorporates it into the response headers. This enables clients or other services that receive the response to trace the request back through its processing path as well.

Both filters depend on a utility class known as FilterUtility, which manages the retrieval and assignment of the correlation ID. Logging is conducted using SLF4J to assist in debugging and tracking request flows. Collectively, these filters implement a distributed tracing mechanism, aiding in the correlation of logs and the monitoring of request behavior as they navigate through various components of the microservices architecture.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 42: Postman Using KeyCloack Role

Figure 43: KeyCloack Client Roles

A screenshot of a computer

AI-generated content may be incorrect.

### Event-Driven Micro Services

Apache Kafka is a distributed event streaming platform that is open-source and designed to manage high-volume, real-time data streams with exceptional throughput, fault tolerance, and scalability. It facilitates the creation of real-time data pipelines and applications that react to ongoing data flows.

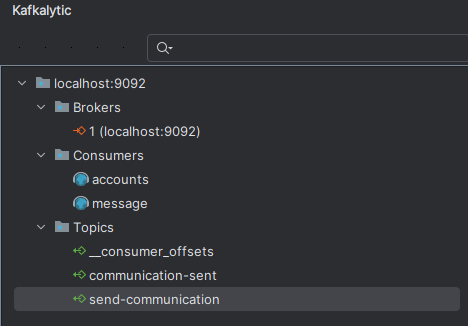
Kafka functions based on several fundamental components. Producers send messages to Kafka topics, which serve as logical channels for organizing data. Each topic can be segmented into multiple partitions, which allows for parallel processing and load balancing. Messages within a partition are sequenced and assigned a unique offset that acts as an identifier. Kafka brokers are the servers responsible for managing these topics and partitions, overseeing message storage, replication, and delivery to consumers.

Offsets are crucial as they enable consumers to monitor their progress through the data stream. Kafka guarantees fault tolerance through replication, where the data of topics is duplicated across several brokers. Consumers retrieve messages from topics, subscribing to partitions and keeping track of their offsets to regulate how and when they access data.

Kafka permits the grouping of consumers into consumer groups. Within a group, each partition is allocated to a single consumer, which allows for parallel processing while ensuring that no message is read more than once per group. Kafka Streams, a client library, facilitates stream processing within the Kafka ecosystem, allowing for real-time data transformation and aggregation as Shown in Figure 45.

A Kafka cluster generally comprises multiple brokers to enhance redundancy and scalability. Topics are utilized to classify messages, and partitions are employed to distribute the storage and processing workload. As producers send data into topics, Kafka utilizes optional partition keys or round-robin algorithms to determine the storage location for each message as Shown in Figure 46.

On the producer side, configuration entails establishing broker addresses and serialization formats. Messages are dispatched to topics and directed to specific partitions. Upon receipt, messages are appended to the partition log with an offset.

**A screen shot of a computer program

AI-generated content may be incorrect.**A screen shot of a computer

AI-generated content may be incorrect.

Figure 44: Kafka Lytic Details

Figure 45: Kafka Connection

Figure 46: Kafka Maven Dependance

### Kubernetes

Container orchestration addresses key operational challenges in managing modern containerized applications, such as automating deployments, rollouts, and rollbacks, ensuring services are self-healing, and supporting autoscaling based on metrics like CPU utilization. Kubernetes is the most widely adopted open-source platform designed to handle these needs. Originally developed by Google and now maintained by CNCF, Kubernetes automates the deployment, scaling, and management of containerized applications, providing a cloud-neutral solution.

Kubernetes enables organizations to run distributed systems resiliently, handling service discovery, load balancing, storage orchestration, automated updates, rollback mechanisms, self-healing through automatic restarts and health checks, and centralized configuration management. Its architecture comprises a control plane and worker nodes. The control plane, or master node, consists of core components including the API server, scheduler, controller manager, and etcd. These components coordinate the cluster, manage workloads, store configurations, and maintain desired system states.

Worker nodes are the infrastructure where containers run. Each node hosts podsKubernetes’ smallest deployable unit containing one or more containers. Key components in a worker node include the kubelet, which communicates with the control plane; kube-proxy, which handles networking; and the container runtime, such as Docker or container, that runs the containers as Shown in Figure 48.

Configuration in Kubernetes is handled using resources like ConfigMaps, which store key-value pairs for configuration data, keeping them separate from the application code. Deployments manage the lifecycle of applications, ensuring the right number of replicas are running and handling updates or rollbacks automatically. Deployment manifests define the desired state of applications, including image versions, ports, environment variables, and replica counts.

Services provide stable networking and load balancing to pods, abstracting away pod IPs which can change over time. The main types of Kubernetes services are ClusterIP, NodePort, and LoadBalancer. ClusterIP is used for internal access within the cluster. Node Port opens a specific port on each node for external access. Load Balancer provides an external cloud-based load balancer to distribute traffic across nodes and pods.

Kubernetes deployment and service definitions are tied together using labels and selectors. The Deployment defines which pods to create and their labels, while the Service uses selectors to target those pods and expose them through a stable endpoint.

In production environments, Kubernetes clusters consist of multiple brokers and nodes to support replication, fault tolerance, and high availability. Helm is often used alongside Kubernetes as a package manager to simplify the deployment of complex applications by templating configuration files and managing releases in a standardized way.

Altogether, Kubernetes provides a robust platform for operating microservices-based applications, automating every aspect of container lifecycle management from deployment to scaling to recovery.

Now we will show examples of how we configure Account microservice for Kubernetes as Shown in Figure 47.

This YAML configuration outlines Kubernetes Deployment and Service. The Deployment operates a single replica of the account’s container utilizing the aimendocker/accounts:s12 image, configures environment variables sourced from a ConfigMap, and makes port 8080 available. The Service facilitates external access by exposing the Deployment on port 8080 through a Load Balancer.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 48: nods for all Micro Service

Figure 47: Kubernetes config for Account Microservice

### Server-Side Service Discovery

In server-side service discovery, K8s assumes the role of monitoring and managing instances of microservices. When a microservice, such as the Accounts service, needs to communicate with another service, like the Loans service, it simply invokes the Kubernetes service URL linked to the target. Kubernetes eliminates the necessity for the calling service to handle load balancing, as it manages this at the server level by directing requests to one of the available and healthy instances of the target microservice.

In contrast to conventional service registries like Eureka, Kubernetes does not mandate that services explicitly register themselves. Instead, it utilizes its internal API to automatically identify and manage service endpoints. When a request is directed to the service URL, Kubernetes' integrated discovery mechanism guarantees that the request is routed to a suitable instance of the target service, facilitating high availability and effortless scaling without the need for any client-side load-balancing logic.

A Service that exposes the application within the cluster on port 80, directing traffic to container port 8761 as Shown in Figure 49.

A Service Account that enables the application to securely interact with the Kubernetes API.

A Role and Role Binding that provides the service account with the necessary permissions to access services, endpoints, and pods within the namespace.

A Deployment that facilitates the operation of the discovery server container, incorporating probes for health assessments and utilizing the specified service account.

A screen shot of a computer

AI-generated content may be incorrect.

Figure 49:Server-Side Service Discovery YML

### Postman Collection for Microservices Testing

To facilitate systematic and automated testing of the microservices architecture, a Postman collection has been developed that encompasses all essential services: Accounts, Cards, Loans, Config Server, Eureka Server, and API Gateway. This collection offers a structured and reusable method for invoking API endpoints, validating service behavior, and testing updates to runtime configurations.

The Accounts service comprises endpoints for creating, updating, retrieving, and deleting account records, in addition to fetching customer details and checking system metadata such as build version and Java version. The Cards service permits similar operations tailored to card data, while the Loans service is responsible for the creation, updating, and deletion of loans. Each of these services features actuator endpoints like /actuator/refresh, which facilitate dynamic configuration updates without necessitating a service restart.

The Config Server endpoints are designed to support the encryption and decryption of sensitive configuration data, thereby validating the centralized configuration management process. The Eureka Server request confirms registered service discovery at runtime, ensuring that all services remain dynamically connected and accessible without the need for hardcoded service URLs.

The Gateway Server section includes requests that are routed through the API Gateway, validating endpoint mapping, security integration, and token-based authentication utilizing OAuth2. Numerous test cases encompass both Bearer tokens and client credential flows, effectively simulating real-world authentication scenarios.

This Postman collection functions as both a validation tool for functional testing and a demonstration of the impacts of configuration strategies. By toggling between embedded and centralized configurations and invoking the same endpoints, one can observe the effects of configuration.

**Full Postman Collection will be available on GitHub Repository for project.**

**Link:** [**AimenFH/Banking-Applications: Developing Resilient and Scalable Event-Driven Microservices for Banking Applications**](https://github.com/AimenFH/Banking-Applications)

# CHAPTER – EXPERIMENTATION & RESULTS

## Maintainability

### Local Configuration

1. @Value Annotation – Local Configuration

This approach allows you to inject individual property values directly into specific fields within your application components. It is suitable for simple cases but can lead to hard-coded property keys scattered throughout the codebase.

This screenshot demonstrates how a Spring Boot application reads a configuration property and exposes it through a REST API.

The build. Version value is retrieved from the application's configuration file and injected into a variable using Spring's property injection mechanism. This allows the application to dynamically read values such as the current build version without hardcoding them.

An HTTP GET endpoint is provided at /build-info, which returns the value of the build. Version property. When a request is made to this endpoint, the application responds with an HTTP 200 status and includes the information in the response body.

This setup is useful for external systems or developers who need to check which version of the service is currently running. It supports better traceability, monitoring, and verification in environments where multiple deployments are involved.

A screen shot of a computer

AI-generated content may be incorrect.A screen shot of a computer

AI-generated content may be incorrect.

Figure 51: Pass Data for Local Configuration

Figure 52: Call Hard Code Data

2. Using Environment Interface – Local Configuration

The Environment interface enables access to application properties programmatically. By auto wiring the Environment bean, developers can retrieve property values dynamically, offering greater flexibility. However, this method still involves manually specifying property keys in the code.

This screenshot demonstrates how a Spring Boot application uses the Environment interface to access configuration properties dynamically.

Instead of injecting individual values, the application declares an Environment object, which provides access to all properties within the application's environment. This approach allows the application to retrieve configuration values programmatically at runtime.

In this case, the environment is used to read a system or environment-specific variable and expose its value through a REST endpoint. When a request is made to the specified endpoint, the application responds with the value of the requested property.

This method is particularly useful when the property key is dynamic, or when conditional logic is needed to decide which property to access. It also provides more flexibility compared to direct injections but requires manual handling of property keys.



A screen shot of a computer

AI-generated content may be incorrect.

Figure 53: Environment interface to access configuration

3. Using @ConfigurationProperties – Local Configuration

This is the most robust and scalable approach. The Configuration Properties annotation binds groups of related properties to a strongly typed bean, eliminating the need to hard-code property keys. This improves maintainability, readability, and allows validation of configuration values.

This example demonstrates how to use @ConfigurationProperties in a Spring Boot application to read grouped configuration values in a structured and scalable way.

Instead of injecting individual values or accessing properties directly through keys, this approach defines a dedicated class that represents a group of related configuration settings. The class is annotated with @ConfigurationProperties and linked to a specific prefix from the configuration file, allowing Spring Boot to automatically bind the corresponding values.

This strongly typed class is then made available in the application by enabling it through the @EnableConfigurationProperties annotation. The values can be returned or processed as needed, such as exposing them through a REST API.

A screenshot of a computer screen

AI-generated content may be incorrect.This method avoids hard-coding property keys, making the configuration more maintainable, readable, and easier to validate.

Figure 54: Methode to Call Configuration Properties

A screen shot of a computer code

AI-generated content may be incorrect.

Figure 55: Configuration Properties Class

4.Profile - Local Configuration

**Using Command-Line Arguments**

Spring Boot automatically maps command-line arguments into key-value pairs and makes them available through the Environment object. These arguments take the highest precedence, meaning they override values from any other source like property files or environment variables.

This is especially useful in production environments where configurations need to be injected dynamically at runtime. The naming convention matches the Spring property names and uses double hyphens (--) for passing values.

**Using JVM System Properties**

JVM system properties provide another way to externalize configuration. These are passed using the -D prefix and also override configuration from files.

This method allows properties to be injected during application startup without rebuilding the JAR file. When both JVM properties and command-line arguments are provided, command-line arguments take precedence.

This feature is useful for customizing builds or temporary overrides, especially in script-driven or automated deployment environments.

**Using Environment Variables**

Environment variables are a universal and platform-agnostic method for configuration. They work across operating systems and are often used in containerized or cloud environments.

To map environment variables to Spring Boot properties, variable names must be written in the uppercase and use underscores (\_) instead of dots (.). This is handled automatically by Spring Boot using relaxed binding.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 56: Example for Account Configuration

Figure 57: Different Environment in Configuration

5. Local Configuration Problem

CLI arguments, JVM properties, and environment variables are effective ways to externalize configuration and maintain the immutability of the application build. However, using these approaches often involves executing separate commands and manually setting up the application, which can introduce potential errors during deployment.

Given that configuration data evolves and require changes, similar to application code, what strategies should be employed to store, track revisions and audit the configuration used in a release?

In scenarios where environment variables lack granular access control features, how can you effectively control access to configuration data?

When the number of application instances grows, handling configuration in a distributed manner for each instance becomes challenging. How can such challenges be overcome?

Considering that neither Spring Boot properties nor environment variables support configuration encryption, how should secrets be managed securely?

After modifying configuration data, how can you ensure that the application can read it at runtime without necessitating a complete restart?

### Centralized Configuration

In a distributed microservices architecture, managing configuration consistently across multiple services is crucial. Spring Boot, together with Spring Cloud Config, Spring Cloud Bus, and Spring Boot Actuator, provides a robust mechanism to externalize, manage, and dynamically refresh configuration without needing to restart applications.

The configuration values for each service are stored in a centralized Git repository. This repository includes multiple YAML files organized by service name and environment (e.g., accounts.yml, cards-prod.yml, etc.). These files are accessed by the Spring Cloud Config Server, which acts as a central source of truth for all configuration data. The Config Server is connected to the Git repository using a URI defined in its configuration. It pulls values from the specified branch (such as main) and serves them to client microservices like accounts, loans, and cards.

To allow applications to refresh their configuration at runtime, the Spring Boot Actuator module must be added to each microservice. This exposes endpoints such as /refresh or /bus refresh, which can be triggered to reload the configuration without restarting the application. The /refresh endpoint is a basic option that works for individual services, while /bus refresh enables broadcast-style refreshes across multiple services using a message broker like RabbitMQ or Kafka.

The basic refresh process begins when a developer commits changes to the configuration files in the Git repository. These updates are detected and fetched by the Config Server. A POST request to the /refresh endpoint of a running service triggers it to reload the updated configuration. However, in systems with many services or multiple instances per service, this manual process becomes inefficient.

To address this, Spring Cloud Bus comes into play. It connects the services via a messaging infrastructure, allowing configuration change events to be broadcast. When a POST request is sent to /actuator/bus refresh on any one instance, the event is distributed to all connected services, which then reload their configuration from the Config Server. This reduces the need to trigger refreshments manually for each instance.

For full automation, Spring Cloud Config Monitor can be used. This component exposes a /monitor endpoint on the Config Server. External systems like GitHub can be configured with a webhook that automatically sends a POST request to this endpoint after each push to the configuration repository. Once triggered, the monitor initiates a configuration refresh via Spring Cloud Bus, ensuring that all services reload their latest configuration in real time.

Additionally, to enhance visibility and security, management endpoints are enabled in the configuration to expose health checks and readiness/liveness probes. Sensitive properties can also be encrypted and stored securely in the configuration repo, with decryption handled at runtime using a shared encryption key.

A computer screen shot of a code

AI-generated content may be incorrect.In summary, the process begins with a change pushed to the Git configuration repository. This change is either manually or automatically propagated to the Config Server, which then refreshes connected services either individually or collectively using the message bus. This architecture supports seamless, scalable, and automated configuration management with minimal downtime and operational overhead.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 58: Mange Secrets Inside Account Microservice

Figure 59: Spring Cloud Config Example

A screenshot of a computer

AI-generated content may be incorrect.

Figure 60: GitHub Repository for Configuration

A screenshot of a webhook

AI-generated content may be incorrect.

Figure 61: Connection between GitHub Repository and Spring Micro Service

### Local Docker VS Docker Compose

In modern microservice-based architectures, applications are often composed of multiple services that need to run together and communicate with one another. Managing these services individually using Docker CLI can become repetitive, error-prone, and difficult to maintain, especially when services depend on each other and require specific configurations. This is where Docker Compose becomes essential.

Docker Compose is a tool that simplifies the process of defining and running multi-container Docker applications. It allows developers to describe services, networks, volumes, environment variables, and dependencies in a single YAML configuration file. With one command, all services can be built, configured, and started together, eliminating the need to run and link each container manually.

Without Docker Compose, developers would need to manually run each container with the correct port mappings, volume mounts, network settings, and startup order. This creates a high risk of errors, increases deployment complexity, and makes automation difficult—especially when configuration needs to change across different environments like development, testing, and production.

In your current setup, Docker Compose is being used to orchestrate services such as RabbitMQ and the Config Server, along with shared configurations and environment-specific settings. Each environment (default, QA, production) has its own folder with a docker-compose.yml and common-config.yml, allowing for modular and reusable configuration.

The rabbit service includes health checks and port mappings, and it extends configuration from a shared file to define networking rules. The configserver depends on RabbitMQ, uses health checks to ensure it only starts when RabbitMQ is healthy, and imports properties from the centralized config server using Spring Cloud Config.

Shared configuration blocks such as microservice-base-config and microservice-configserver-config handle common settings like memory limits, environment variables, and Spring profiles. This promotes reuse, consistency, and easier management of resources.

Thanks to Docker Compose, all these services can be started with a single command. Dependence is managed automatically, services wait for each other based on health checks, and all configuration details are version-controlled and environment-specific.

A computer screen shot of a program

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 62: Rabbit Docker Compos Example

Figure 63: Docker Compose Folder

A screen shot of a computer

AI-generated content may be incorrect.

Figure 64: Common Configuration Inside Docker Compose

### Local Libraries VS Shared Libraries

In microservice architecture, managing dependencies and shared code across multiple services can quickly become complicated. Each service might use different versions of the same library, which can lead to version conflicts, bugs, and inconsistent behavior. To solve this, Spring Boot and Maven offer a solution called BOM, or Bill of Materials. A BOM helps define all dependency versions in one central place, so each microservice can automatically use the same versions without specifying them individually. This makes it much easier to keep everything consistent and simplifies upgrades when library versions change.

Another common challenge is code duplication. Many services might share the same utilities, configuration classes, or logging setup. Without a good structure, this shared code could be copied across projects or managed in a way that becomes difficult to maintain. One option is to create a shared library, but this can lead to large files with unused code. A better approach is to use a multi-module Maven project, where common code is placed in a separate module that other services can include. This avoids duplication and keeps the system modular.

To support this structure, the project includes helpful metadata such as developer information, organization URLs, and source code links, making it easy to track ownership and contribute across teams. It also uses automated Docker image naming based on the project name and version, helping keep deployments consistent.

A screen shot of a computer

AI-generated content may be incorrect.Altogether, this setup improves the way microservices are developed and maintained. It ensures consistent dependency versions, avoids repeated code, and keeps configuration clean and reusable. This makes the development process more efficient and reduces the chances of issues during buildings, testing, and deployment.

A screen shot of a computer

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.A computer screen shot of a program

AI-generated content may be incorrect.

### Kubernetes VS Helm

Without Helm, deploying microservices within Kubernetes necessitates the upkeep of distinct YAML manifest files for each individual service. While these files may exhibit slight variations—such as differing service names, ports, and labels, much of the content remains repetitive across the various services. This situation results in redundancy, potential manual errors, and operational inefficiencies, particularly in large-scale projects.

Helm addresses these challenges by implementing a templating system. Rather than creating multiple similar YAML files for each microservice, developers can utilize a single template file that incorporates placeholders. Dynamic elements such as service name, type, or port are specified in a separate value.yaml file. During the deployment process, Helm populates placeholders with the defined values, thereby automatically generating tailored manifests for each service.

In addition to templating, Helm provides a range of other powerful features. It enables the packaging of all Kubernetes manifest files associated with a service into a single chart, which can be stored and shared through public or private repositories. Helm streamlines the deployment and lifecycle management of applications within Kubernetes, allowing developers to install, upgrade, rollback, or delete entire applications with a single command—thus removing the necessity to manually apply each individual file.

Furthermore, Helm maintains a version history of deployments, facilitating a straightforward rollback to a previous state. Like a conventional package manager for software, Helm functions as the package manager for Kubernetes, overseeing complex applications and their configurations in a consistent and automated manner.

A standard Helm chart structure comprises a top-level folder (for instance, wordpress), a Chart.yaml file containing metadata, a values.yaml file for dynamic configuration, a templates directory that holds manifest templates, and an optional charts directory for dependencies. This organizational structure promotes reusability, modularity, and more efficient management of Kubernetes deployments.

### Result

## Scalability

## Security

### KeyCloak

In a secure microservices environment, OAuth2 plays a crucial role in controlling and managing access to protected resources. The flow typically begins when a user initiates a request to access specific resources. This request is sent to the client application, which acts on behalf of the user. The client contacts the authorization server (e.g., Keycloak) to obtain an authorization code. This code proves the user has granted permission. Once received, the client uses this code along with its own credentials to request an access token from the authorization server.

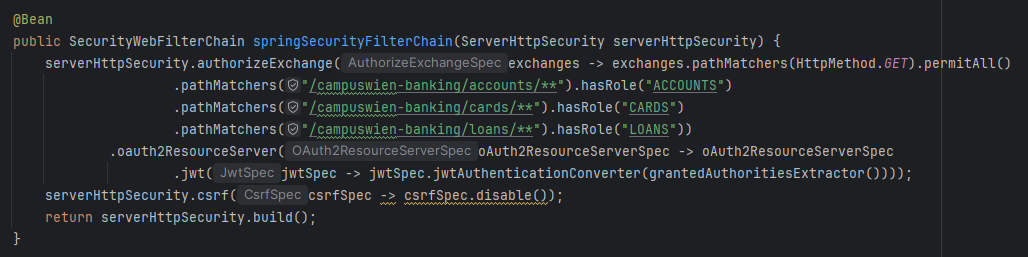
After receiving the access token, the client includes it in subsequent requests to the resource server (for example, a microservice behind Spring Cloud Gateway). Before granting access, the gateway validates the token by consulting the authorization server. If valid, the request proceeds, and the user gains access to the necessary data. This validation ensures that only authorized users can access protected endpoints.

On the resource server side, Spring Security is configured to control access to specific endpoints based on user roles. These roles are embedded within the access token as claims. To extract and interpret these roles, a custom converter is implemented. The KeycloakRoleConverter reads the realm\_access.roles claim from the JWT token and converts each role into a Spring GrantedAuthority, prefixed with ROLE\_. This allows fine-grained access control to be enforced through annotations or route-based rules.

To make this work, the application’s security configuration specifies which paths require which roles. For instance, requests to /campuswien-banking/accounts/\*\* might require the ACCOUNTS role. The Spring configuration also includes a URI pointing to the Keycloak public key set, which is used to verify the signature of JWT tokens.

On the client side, such as Postman, the access token is generated and attached to HTTP requests using the OAuth 2.0 mechanism. This token must be refreshed when it expired. Postman allows auto-refresh and easy inspection of the bearer token.

In Kafka-enabled services, security is equally critical. Messages may carry sensitive data or require role-based access. The token passed through HTTP headers may include roles that define what messages a consumer can read or produce. The extracted correlation ID in the custom logic helps trace and log requests across distributed services. It is injected into the request headers to maintain consistent context throughout the service flow.



A screenshot of a computer

AI-generated content may be incorrect.A computer screen with green and white text

AI-generated content may be incorrect.

### Docker Compose

In the Docker Compose configuration for the accounts microservice, you can observe that there is no port mapping defined. This means the container does not expose its internal ports directly to the host machine or the outside world. As a result, it is not possible to access the accounts service directly via browser or REST client by calling something like localhost:8080.

Instead, all access must go through the secure gateway, which acts as the single-entry point to all backend services. This architectural decision enforces centralized security, routing, and access control policies.

Additionally, in the environment configuration, the JWT Token verification is configured through the property:



"http://keycloak:8080/realms/master/protocol/openid-connect/certs"

This URI tells Spring Security where to fetch the public key needed to validate JWT tokens issued by Keycloak. It ensures that only requests carrying valid access tokens from a trusted source can reach the internal microservices through the gateway. This setup provides a strong security layer, combining network isolation with token-based authentication and authorization.

### Result

Ensures consistent identity/authentication configuration across services.

4. Service-to-Service Authorization

Local: 1. Managing Secrets (Passwords, Keys, Tokens)

Local Configuration:

Secrets (e.g., database passwords, API keys, OAuth credentials) are stored in each service’s application.yml or. properties file.

Risk of inconsistency or leaking secrets increases, especially if they are committed to version control by mistake.

Hard to rotate or revoke credentials across multiple services at once.

Centralized Configuration:

All secrets are stored securely in one place (e.g., Spring Cloud Config Server).

You can use encrypted values (with {cipher}) and an encryption key that the config server uses to decrypt at runtime.

Much easier to update or rotate secrets across the entire system, and access is centralized and auditable.

2. Protecting Configuration Endpoints

Local:

No central control over who accesses config files. If a service exposes actuator endpoints, they may leak info unless secured individually.

Each service must protect its own /actuator endpoints, leading to duplicated security config.

Centralized:

Config Server exposes /actuator/health, /encrypt, /decrypt, and possibly /monitor.

Only the Config Server needs to be secured tightly; clients simply consume config via secure internal communication.

Easier to implement role-based access control or authentication at a single point.

3. Gateway & OAuth2 Security Configuration

Local:

OAuth2 settings (client ID, secret, auth server URL) must be added manually to each service.

Risk of drift between services or incorrect settings.

If a secret changes, all services need to be redeployed manually.

Centralized:

Shared OAuth2 settings stored in Git or Vault, loaded by the Config Server.

One change in the Git config and a /refresh or /busrefresh will update all services at once.

Hardcoded or mismatched tokens and credentials may be used between services.

Manually updated, increasing the risk of broken authentication or leakage.

Centralized:

Shared service credentials can be managed centrally and rotated securely.

Policies and tokens can be enforced consistently across services.

# CONCLUSION and Future work

List of Figures

[Figure 1 .Strangler Fig Pattern 3](#_Toc205645339)

[Figure 2: Microservices Outline 10](#_Toc205645340)

[Figure 3 : Monolithic vs. SOA vs. Microservices 11](#_Toc205645341)

[Figure 4: Challenges in Microservices 19](#_Toc205645342)

[Figure 5: Architecture Overview 33](#_Toc205645343)

[Figure 6: Microservices in Application 37](#_Toc205645344)

[Figure 7: Account Maven Dependance Example 38](#_Toc205645345)

[Figure 8: Customer Class Entity Example 39](#_Toc205645346)

[Figure 9: Data Transfer Object Example 40](#_Toc205645347)

[Figure 10: Account Repository Example 41](#_Toc205645348)

[Figure 11: Account Controller Class Example 42](#_Toc205645349)

[Figure 12: Global Exception Class Example 43](#_Toc205645350)

[Figure 13: Application.yml file Example 44](#_Toc205645351)

[Figure 14: API Example for Account micro service 1 46](#_Toc205645352)

[Figure 15: API Example for Account micro service 2 47](#_Toc205645353)

[Figure 16: API Example for Account micro service 3 48](#_Toc205645354)

[Figure 17: API Example for Account micro service 4 49](#_Toc205645355)

[Figure 18: Account Info for Docker Image 50](#_Toc205645356)

[Figure 19 : Docker Image for Microservices 51](#_Toc205645357)

[Figure 20: Config Server GitHub 53](#_Toc205645358)

[Figure 21: Config Application YAML File 54](#_Toc205645359)

[Figure 22: Eureka Discovery 55](#_Toc205645360)

[Figure 23: Eureka Example 56](#_Toc205645361)

[Figure 24: Eureka heartbeat Example 56](#_Toc205645362)

[Figure 25:Card Service Feign Client 57](#_Toc205645363)

[Figure 26: Circuit breaker Example 58](#_Toc205645364)

[Figure 27: Prometheus Configuration 60](#_Toc205645365)

[Figure 28: Account Micro Service Actuator 60](#_Toc205645366)

[Figure 29: Checking CPU Usage for Microservices in Prometheus Dashboard 61](#_Toc205645367)

[Figure 30: Checking CPU Usage Graph for Microservices in Prometheus Dashboard 61](#_Toc205645368)

[Figure 31:All Microservices State shown in Prometheus Dashboard 61](#_Toc205645369)

[Figure 32: Grafana View of CPU Usage of Microservices 62](#_Toc205645370)

[Figure 33: JVM (Micrometer) Dashboard for Account Microservice 62](#_Toc205645371)

[Figure 34:Grafana Alert when Account Micro service down 63](#_Toc205645372)

[Figure 35: Hook deck Response When Account Micro service down 63](#_Toc205645373)

[Figure 36: Open Telemetry Pattern 64](#_Toc205645374)

[Figure 37: Trace ID from Account Microservice 65](#_Toc205645375)

[Figure 38: Trace ID from Card Microservice 65](#_Toc205645376)

[Figure 39: Temp show Path of Request 65](#_Toc205645377)

[Figure 40: Security Config 67](#_Toc205645378)

[Figure 41: Key Clock Filter Chain 68](#_Toc205645379)

[Figure 42: Postman Using KeyCloack Role 70](#_Toc205645380)

[Figure 43: KeyCloack Client Roles 70](#_Toc205645381)

[Figure 44: Kafka Lytic Details 72](#_Toc205645382)

[Figure 45: Kafka Connection 72](#_Toc205645383)

[Figure 46: Kafka Maven Dependance 72](#_Toc205645384)

[Figure 48: nods for all Micro Service 75](#_Toc205645385)

[Figure 47: Kubernetes config for Account Microservice 75](#_Toc205645386)

[Figure 49:Server-Side Service Discovery YML 77](#_Toc205645387)

[Figure 51: Pass Data for Local Configuration 79](#_Toc205645388)

[Figure 52: Call Hard Code Data 79](#_Toc205645389)

[Figure 53: Environment interface to access configuration 80](#_Toc205645390)

[Figure 54: Methode to Call Configuration Properties 81](#_Toc205645391)

[Figure 55: Configuration Properties Class 81](#_Toc205645392)

[Figure 56: Example for Account Configuration 83](#_Toc205645393)

[Figure 57: Different Environment in Configuration 83](#_Toc205645394)

[Figure 58: Mange Secrets Inside Account Microservice 86](#_Toc205645395)

[Figure 59: Spring Cloud Config Example 86](#_Toc205645396)

[Figure 60: GitHub Repository for Configuration 87](#_Toc205645397)

[Figure 61: Connection between GitHub Repository and Spring Micro Service 87](#_Toc205645398)

[Figure 62: Rabbit Docker Compos Example 89](#_Toc205645399)

[Figure 63: Docker Compose Folder 89](#_Toc205645400)

[Figure 64: Common Configuration Inside Docker Compose 90](#_Toc205645401)

List of Tables

[Table 1: Monolithic vs. SOA vs. Microservices Comparison 12](#_Toc205642688)

[Table 2: Critical Decision Points – Monolith vs. SOA vs. Microservices 13](#_Toc205642689)

[Table 3: Centralized vs. Local Configuration 21](#_Toc205642690)

Appendix and Bibliography

[1] Sam Newman, "Building Microservices (2nd Edition)," O’Reilly Media, 2021. [Online]. Available: https://samnewman.io/books/building\_microservices\_2nd\_edition/

[2] E. Solberg, "Evolution of Microservices Identification in Monolith Decomposition: A Systematic Review," ResearchGate, 2024. [Online]. Available: https://www.researchgate.net/publication/378111158\_Evolution\_of\_Microservices\_Identification\_in\_Monolith\_Decomposition\_A\_Systematic\_Review

[3] Ahrefs, "Paragraph Rewriter Tool," Ahrefs, 2025. [Online]. Available: https://ahrefs.com/writing-tools/paragraph-rewriter

[4] A. Das, "Managing Configuration in a Microservice Environment," Medium, 2022. [Online]. Available: https://article.arunangshudas.com/managing-configuration-in-a-microservice-environment-b2fddba31f28

[5] GeeksforGeeks, "Microservices," GeeksforGeeks, 2024. [Online]. Available: https://www.geeksforgeeks.org/microservices/

[6] E. Solberg, "Master Thesis," University of Oslo, 2023. [Online]. Available: https://www.duo.uio.no/bitstream/handle/10852/95663/1/Master-thesis--Eivind-Solberg.pdf

[7] M. Ezzat, "Microservices Architecture: Benefits, Challenges, and Use Cases," Medium, 2021. [Online]. Available: https://miladezzat.medium.com/microservices-architecture-benefits-challenges-and-use-cases-9cec05adcb57

[8] JKU, "Microservices Security," Johannes Kepler University, 2022. [Online]. Available: https://epub.jku.at/download/pdf/8073606.pdf

[9] Solo.io, "Microservices Security," Solo.io, 2023. [Online]. Available: https://www.solo.io/topics/microservices/microservices-securitys

[10] vFunction, "Microservices Architecture Guide," vFunction, 2023. [Online]. Available: https://vfunction.com/blog/microservices-architecture-guide/

[11] Docker, "Docker Overview," Docker Inc., 2025. [Online]. Available: https://www.docker.com/

[12] Pivotal Software, "Spring Boot Project," Spring.io, 2025. [Online]. Available: https://spring.io/projects/spring-boot

[13] Git SCM, "Git Version Control," Git, 2025. [Online]. Available: https://git-scm.com/

[14] VMware, "Spring Cloud Project," Spring.io, 2025. [Online]. Available: https://spring.io/projects/spring-cloud

[15] Red Hat, "Keycloak Identity and Access Management," Keycloak.org, 2025. [Online]. Available: https://www.keycloak.org/

[16] Apache Software Foundation, "Apache Kafka," Apache.org, 2025. [Online]. Available: https://kafka.apache.org/

[17] Pivotal Software, "RabbitMQ," RabbitMQ.com, 2025. [Online]. Available: https://www.rabbitmq.com/

[18] CNCF, "Kubernetes," Kubernetes.io, 2025. [Online]. Available: https://kubernetes.io/

[19] CNCF, "Helm," Helm.sh, 2025. [Online]. Available: https://helm.sh/

[20] Postman, "Postman API Platform," Postman.com, 2025. [Online]. Available: https://www.postman.com/home

[21] Oracle, "Java," Java.com, 2025. [Online]. Available: https://www.java.com/de/

[22] L. Bass, P. Clements, and R. Kazman, "Software Architecture in Practice," Addison-Wesley, 2012. [Online]. Available: https://arxiv.org/abs/1902.06282

[23] MDPI, "Future Internet Journal," MDPI, 2023. [Online]. Available: https://www.mdpi.com/1999-5903/15/1/37

[24] ITPro, "AI Paradox for Developers," ITPro, 2024. [Online]. Available: https://www.itpro.com/software/development/atlassian-says-ai-has-created-an-unexpected-paradox-for-software-developers-theyre-saving-over-10-hours-a-week-but-theyre-still-overworked-and-losing-an-equal-amount-of-time-due-to-organizational-inefficiencies