Title

Evaluating the Impact of Centralized Configuration Management on Microservices Using Docker and Kubernetes

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:

Aimen El-Sayed

Student Identification Number:

c2310838009

Supervisor:

Thomas Berger

Date:

08. 13. 2025

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: 08.13.2025 Signature:

Aimen El-Sayed

Abstract

In modern software engineering, microservices architectures have become a favored alternative to monolithic systems, providing advantages in scalability, agility, and maintainability. Nevertheless, these benefits bring about operational challenges, especially in configuration management, where it is essential to maintain consistency, security, and efficient updates across distributed services.

This thesis explores the impact of centralized configuration management, implemented through Spring Cloud Config Server, in contrast to local, per-service configuration within Kubernetes-based microservices. The assessment concentrated on three operational dimensions that are commonly recognized as industry pain points: scalability, security, and maintainability.

The most significant enhancement was observed in maintainability: the average time for configuration change propagation decreased from roughly 2–3 minutes with local configuration to under 10 seconds with centralization, the effort for rollback was minimized to a single Git revert, and comprehensive audit trails were automatically available. In scalability assessments, centralized configuration guaranteed that newly scaled pods consistently matched the latest configuration values, leading to a reduction in transient errors during scale-out by 20–30% and a decrease in stabilization times. Security operations, particularly the enforcement of Keycloak policies

The results demonstrate that centralized configuration management significantly improves maintainability and provides measurable, empirically verified gains in scalability and security compared to local configuration, with maintainability benefits being the most substantial. These findings offer empirical evidence to inform architectural decisions in organizations adopting or evolving microservices-based systems

.

To the best of our knowledge, this is the first empirical comparison of centralized and local configuration in Kubernetes-based microservices, integrating security, deployment automation, and observability aspects.

Central Research Question

To what extent does centralize configuration management via Spring Cloud Config Server improve scalability, security, and maintainability in Docker- and Kubernetes-based microservices compared to local, per-service configuration, and which of these operational dimensions benefit most from such centralization?

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

HPA Horizontal Pod Auto scaler

MTTR Mean Time to Repair

RBAC Role-Based Access Control

ESB Enterprise Service Bus

Contents

Table of Contents

[Abstract iii](#_Toc206080360)

[Central Research Question iv](#_Toc206080361)

[List of Abbreviations v](#_Toc206080362)

[Contents vii](#_Toc206080363)

[1. CHAPTER – INTRODUCTION 10](#_Toc206080364)

[1.1 Background And Motivation 10](#_Toc206080365)

[1.1.1 Introduction to Microservices 10](#_Toc206080366)

[1.1.2 Why Configuration Management 11](#_Toc206080367)

[1.1.3 Why Docker and Kubernetes 13](#_Toc206080368)

[1.1.4 Industrial Motivation 13](#_Toc206080369)

[1.1.5 Personal Motivation 14](#_Toc206080370)

[1.1.6 Problem Statement 15](#_Toc206080371)

[1.1.7 Research Objectives 17](#_Toc206080372)

[1.2 Research Question 18](#_Toc206080373)

[1.3 Innovative Contribution of This Thesis 20](#_Toc206080374)

[2. CHAPTER – LITERATURE REVIEW 21](#_Toc206080375)

[2.1 Microservices Outline 21](#_Toc206080376)

[2.1.1 Monolith to Microservices 22](#_Toc206080377)

[2.1.2 Monolith to Microservices: 5 Strategies 25](#_Toc206080378)

[2.1.3 Key Characteristics of Microservices 27](#_Toc206080379)

[2.1.4 Impacts of Migration to Microservice Architecture on Team 28](#_Toc206080380)

[2.2 Challenges In Microservices 29](#_Toc206080381)

[2.3 Centralized vs. Local Configuration 31](#_Toc206080382)

[2.4 Security and Observability Considerations 34](#_Toc206080383)

[2.4.1 Security 34](#_Toc206080384)

[2.4.2 Observability 35](#_Toc206080385)

[2.5 Identified Research Gaps in Microservices Literature 37](#_Toc206080386)

[2.6 Summary of Literature Review 39](#_Toc206080387)

[3. CHAPTER – RESEARCH METHODOLOGY 42](#_Toc206080388)

[3.1 Research Design 42](#_Toc206080389)

[3.2 Tools and Technologies 43](#_Toc206080390)

[3.3 Experimental Setups 46](#_Toc206080391)

[3.4 Measurement 47](#_Toc206080392)

[4. Chapter - SYSTEM DESIGN 50](#_Toc206080393)

[4.1 Architecture Overview 50](#_Toc206080394)

[4.2 Implementation Approach 55](#_Toc206080395)

[4.2.1 Microservice 55](#_Toc206080396)

[4.2.2 Maven Dependency 56](#_Toc206080397)

[4.2.3 Entity 57](#_Toc206080398)

[4.2.4 Data Transfer Object 59](#_Toc206080399)

[4.2.5 Repository 60](#_Toc206080400)

[4.2.7 Client Service 61](#_Toc206080401)

[4.2.8 Controller 62](#_Toc206080402)

[4.2.9 Exceptions Handler 63](#_Toc206080403)

[4.2.10 Application YML file 65](#_Toc206080404)

[4.2.11 Data Base 66](#_Toc206080405)

[4.3 Configuration Strategy 67](#_Toc206080406)

[4.3.1 API 67](#_Toc206080407)

[4.3.2 Docker 72](#_Toc206080408)

[4.3.3 Config Server 74](#_Toc206080409)

[4.3.4 Client-Side Service Discovery 77](#_Toc206080410)

[4.3.5 RESILIENCY 79](#_Toc206080411)

[4.3.6 Observability and Monitoring 81](#_Toc206080412)

[4.3.7 Distributed Tracing 86](#_Toc206080413)

[4.3.8 OpenID Connect, Keycloak 88](#_Toc206080414)

[4.3.9 Event-Driven Microservices 92](#_Toc206080415)

[4.3.10 Kubernetes 95](#_Toc206080416)

[4.3.11 Server-Side Service Discovery 98](#_Toc206080417)

[4.3.12 Postman Collection for Microservices Testing 100](#_Toc206080418)

[5. CHAPTER – EXPERIMENTATION & CONCLUSION 101](#_Toc206080419)

[5.1 Maintainability 101](#_Toc206080420)

[5.1.1 Local Configuration 101](#_Toc206080421)

[5.1.2 Centralized Configuration 107](#_Toc206080422)

[5.2 Scalability 110](#_Toc206080423)

[5.3 Security 117](#_Toc206080424)

[5.4 Local Docker VS Docker Compose 119](#_Toc206080425)

[5.5 Local Libraries VS Shared Libraries 121](#_Toc206080426)

[5.6 Helm 124](#_Toc206080427)

[5.7 Conclusion 127](#_Toc206080428)

[6. FUTURE WORK 130](#_Toc206080429)

[List of Figures 132](#_Toc206080430)

[List of Tables 135](#_Toc206080431)

[Appendix 136](#_Toc206080432)

# CHAPTER – INTRODUCTION

## Background And Motivation

### Introduction to Microservices

For numerous years, applications were structured as monoliths as a singular deployable entity that is simple to manage at a small scale but becomes increasingly inflexible as complexity and team size expand [1][2]. Microservices offer an alternative architecture: a system made up of small, self-sufficient services, each corresponding to a specific business capability, capable of independent deployment, and communicating through lightweight protocols commonly HTTP [1][2]. The objective is to decouple release cycles, enable teams to operate more swiftly, and scale only the critical areas that require it [1][2].

A fundamental tenet is the database-per-service approach: each service possesses its own data store and shares data solely through its API [3] (see Table 1). This minimizes coupling and allows for independent evolution, but it also transfers the responsibility for data consistency (e.g., sagas, outbox, change-data-capture) to the service boundaries trade-offs that every microservice architecture must address explicitly [3].

In practice, teams encapsulate services within containers and frequently deploy them on Kubernetes, which automates deployment, scaling, and service management across clusters [4]. While Kubernetes is not a necessity for microservices, it has emerged as a prevalent runtime for extensive fleets due to its declarative model and ecosystem [4].

Modernization seldom occurs in a “big bang” manner. Organizations implement the Strangler Fig pattern (see Figure 1) [5] to gradually phase out legacy modules: they introduce a fade, redirect traffic for a specific domain segment to a new service, iterate, and slowly decommission the legacy core. This approach mitigates risk and facilitates incremental learning throughout the decomposition process [5].

A group of trees with vines

AI-generated content may be incorrect.

Figure 1: Strangler Fig pattern

### Why Configuration Management

In microservices, configuration comprising service endpoints, credentials, feature flags, rate limits, circuit-breaker thresholds, and policy rules function as the real-time control plane of the system [1][2]. In contrast to code, configuration experiences changes at a significantly higher frequency and necessitate consistent application across numerous independently deployed services and environments [1][2]. Inadequately managed configuration can result in drift, inconsistent behavior during scaling, security vulnerabilities, and prolonged incident recovery; conversely, well-managed configuration reduces change lead time, enhances reliability, and facilitates auditable rollbacks [1][2][6]. Key challenges.

* Decentralization and drift: When each service maintains its own configuration, values can diverge across instances and environments, making synchronized rollouts prone to errors. [1]
* Environment matrices: Development, testing, staging, and production environments each necessitate unique settings; duplicating files does not scale effectively and complicates governance. [1]
* Dynamic change: Operational toggles (such as feature flags, rates, and timeouts) should be modifiable at runtime without necessitating rebuilds or restarts to prevent disruption. [6][7]
* Secrets and policy: Keys, tokens, and authorization rules must be rotated and uniformly enforced under zero-trust principles. [8][9][10]
* Auditability and rollback: Teams must be aware of who made changes, when they were made, and be able to revert quickly if a modification negatively impacts the system. [6]

**Approaches in practice.**

* Local per-service configuration [1]. This approach is straightforward for a limited number of services and is easy to understand in isolation; however, extensive changes necessitate numerous coordinated updates and often involve rebuilding and redeploying cycles, which can lead to inconsistent rollouts and extended recovery times [1].
* Centralized configuration [6][7]. An external Config Server for instance, Spring Cloud Config provides a single source of truth with versioning, audit trails, and rollback capabilities typically supported by Git [see section: ‎4.3.3)]. Utilizing Spring Cloud Bus or Actuator refresh allows running instances to adopt changes without restarts, thereby enhancing propagation speed and consistency during autoscaling [6][7].

Complementary platform features [8][9][10]. On Kubernetes, ConfigMaps (non-secret) and Secrets (sensitive values) decouple config from container images and help with in-cluster distribution and bootstrapping; they complement rather than replace a centralized server in larger estates. For advanced secret hygiene (lease/rotate/renew), Spring Cloud Vault integrates applications with Vault at runtime. [8][9][10]

Focus on this thesis. We evaluate whether centralizing configuration via Spring Cloud Config (and optionally Bus) materially improves

1. scale-out stability and alignment of new pods
2. speed and uniformity of security/policy updates
3. (iii) maintainability (propagation latency, rollback effort, auditability), compared with a local, per-service baseline. [6][7][8][9][10]

### Why Docker and Kubernetes

Docker and Kubernetes are fundamental elements of the technological framework utilized in this thesis, playing a crucial role in establishing a realistic, production-level environment for assessing centralized versus local configuration management within microservices.

Docker offers a standardized platform for containerization, encapsulating each microservice along with its runtime, dependencies, and configuration files into a portable image [10]. This guarantees consistent behavior of services across development, testing, and production environments, thereby eliminating inconsistencies arising from varying infrastructure. In this research, all microservices — including Accounts, Cards, and Loans — along with auxiliary services such as the Spring Cloud Config Server and Keycloak, are containerized using Docker see section ‎4.3.2 This methodology facilitates swift redeployment and eases the replication of environments for controlled experiments.

Kubernetes manages these Docker containers, automating processes such as deployment, scaling, service discovery, and recovery from failures [17]. It offers native resources like ConfigMaps for non-sensitive configurations and Secrets for sensitive information, which enhance — but do not substitute — centralized configuration servers (see Section ‎‎4.3.3). Additionally, the Horizontal Pod Autoscaler (HPA) dynamically modifies the number of active replicas based on workload metrics, which has a direct effect on the scalability assessments in this study ‎5.2 .

### Industrial Motivation

In a substantial enterprise insurance platform undergoing gradual modernization, we implemented a Strangler Fig strategy as shown in fig 1 see [Figure: Strangler Fig pattern] [5] to detach domains behind an API gateway while a legacy core continued to manage essential operations. This enhanced team autonomy and release frequency; however, configuration persisted as a systemic bottleneck and a common source of operational risk. Specifically, three recurring pain points surfaced: (i) coordinating cross service modifications across various environments, (ii) auto scaled replicas occasionally initializing with outdated values during surges (for instance, obsolete rate limits, feature flags, or circuit-breaker thresholds, and (iii) inconsistent enforcement of security/policy updates when alterations were applied on a service-by-service basis. These challenges are frequently documented in microservices initiatives as side effects of distributed ownership and decentralization, particularly when configuration is handled locally within each service. [1][2][5]

Scale-out misalignment under load. During peak events such as campaign launches or end-of-the-month processing, the Horizontal Pod Auto scaler introduced replicas to manage traffic spikes. With local configuration, newly instantiated pods sometimes started with defaults embedded at build time while existing pods operated with updated values. This led to a brief period of inconsistent behavior: some requests directed to “old value” replicas faced timeouts or 429 errors (measured in P1, Section ‎3.4) ; results in Section ‎5.2. while others succeeded resulting in noisy error budgets and obscuring the true health of the service. Manual interventions such as rolling restarts or ad-hoc refresh endpoints mitigated drift but increased meantime to recovery MTTR and operator workload. [1][2]

Security and policy updates

Policy changes such as tightening route-level authorization at the gateway or rotating a token scope needed to be uniformly applied across numerous instances. In the local configuration approach, multiple services necessitated simultaneous updates and rollouts, which occasionally resulted in a minute-long interval where different pods enforced varying rules in Section ‎‎5.3.

Environment matrix and auditability

With dev/test/stage/prod environments and several regional variants, configuration values proliferated. Teams asked practical questions that were hard to answer quickly: Which value is live right now? Who changed it last? What exactly changed between stage and prod? Local files increased duplication and reduced traceability. In contrast, centralization promised a Git-backed history (see Section ‎4.3.3) and the ability to roll back to a known-good version if a change degraded performance.

### Personal Motivation

Based on over three years of professional experience at Allianz working on a large-scale insurance application, it became evident that complex, community-wide systems can encounter challenges from multiple dimensions (results related to these challenges are presented in Section ‎5).

A key difficulty lies in managing large development teams, particularly in ensuring that any individual possesses comprehensive knowledge of the entire application. Even minor code changes sometimes a single line can have far-reaching impacts, necessitating extensive regression testing and resulting in significant delays.

During the project, the decision was made to transition towards microservice architecture. While this approach was successfully implemented in several components, the business layer interfacing with the database remained monolithic. Nevertheless, the transition led to measurable improvements in performance and latency, and it enabled the reorganization of services into smaller, cross-functional teams. These teams, typically comprising roles such as Scrum Master, designer, product owner, and testers, could work concurrently and, where beneficial, employ different programming languages. However, the constraint of a single, shared database persisted.

It is important to note that microservices also introduce their own operational challenges. For projects that can be logically divided into only three or four services, the overhead of adopting a microservices architecture may outweigh its benefits. Conversely, for systems requiring five or more independent services, architecture can provide substantial advantages. One notable benefit is the flexibility to select the most suitable programming language for each service capability not typically available in monolithic systems

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

As the number of microservices and environment matrices (development, testing, staging, production, potentially across various regions) increases, the phenomenon of configuration sprawl emerges as a significant contributor to operational risk and financial expenditure [1][2][6][7]. Configuration values that dictate connectivity, resilience, and security such as endpoints, timeouts, rate limits, feature flags, client credentials, and RBAC rules must remain uniform across numerous independently deployed services and must propagate swiftly, including replicas generated during autoscaling.

In practice, a localized configuration model (specific to each service) results in redundant files, necessitates manual coordination, and requires rebuild and redeploy cycles for routine modifications, thereby heightening the likelihood of discrepancies between instances and environments [1][2] architecture shown in Figure 1.

In contrast, a centralized configuration model such as Spring Cloud Config (supported by a versioned storage system and optionally enhanced by Spring Cloud Bus for runtime updates) offers a singular source of truth, comprehensive audit trails, rollback capabilities, and near-real-time propagation without necessitating restarts [6][7] *(see Section ‎4.3.3])*. Kubernetes ConfigMaps and Secrets further support this methodology by facilitating cluster-native distribution and credential management, while Spring Cloud Vault provides secure secret retrieval and rotation [8][9][10].

However, the overall impact of implementing centralization on daily operations is empirical: it may introduce a dependency on the control plane and new modes of failure, prompting an evaluation of whether the advantages significantly surpass the associated costs in practical workloads. [6][7][8][9][10]

**Research problem. Investigate whether substituting local, per-service configuration with a centralized configuration system (Spring Cloud Config ± Bus) enhances operational outcomes in a microservices architecture based on Kubernetes across three dimensions:**

**– Scalability under load: During scale-out, do newly created pods start with the current configuration, reducing transient errors and time-to-steady-state compared with local config? (Tested in P1, Section ‎3.4; results in Section ‎** ‎5.3**).**

**– Security operations: Are policy and secret changes applied faster and more uniformly across instances, minimizing windows of inconsistent enforcement? (tested in P2, Section (Tested in P1, Section ‎3.4; results in Section ‎‎5.3).**

**– Maintainability: Are change propagation latency, rollback effort/time, and auditability (who changed what/when) improved in practice? (Tested in P3, Section ‎3.4; results in Sections ‎**5.2

**These dimensions reflect recurrent industry pain points and our industrial observations [1][2][5][6][7].**

### Research Objectives

Objective. This study aims to assess, utilizing only the existing project without introducing new code, whether centralized configuration (via Spring Cloud Config ± Spring Cloud Bus) enhances daily operational efficiency compared to local, per-service configuration within a Spring/Kubernetes microservices architecture. [1][6][7][8][9][10]

System under evaluation. The current technology stack includes Spring Boot services (Accounts, Loans, Cards), Spring Cloud Gateway, Keycloak, Kubernetes with Horizontal Pod Auto scaler (HPA); ConfigMaps/Secrets already in use; Spring Cloud Config (backed by Git) and, if applicable, Spring Cloud Bus see Figure 5. [4][6][7][8][9][10]

Configurations being compared.

A — Local (baseline): Configuration is packaged with each service, necessitating a rollout for any changes.

B — Centralized (treatment): Properties are externalized and provided by Spring Cloud Config; optional refresh via Bus/Actuator if already implemented. The code remains unchanged. [6][7]

Metrics for evaluation

Scalability: The duration required new pods to reflect the current configuration following HPA scale-out; transient 4xx/5xx errors during the ramp-up phase see section ‎5.2. [4][6]

Security operations: The median and 95th percentile time for Keycloak policy/secret modifications to be uniformly enforced across all pods see section ‎‎5.2. [8][9][10]

Maintainability: Latency in change propagation, time and steps for rollback (comparing Git revert to multiple per-service edits), and auditability (who made changes, what changes were made, and when see section ‎5.1 [6][7][8]

Procedures (utilizing existing resources).

P1—Scale-out drill: Generate consistent traffic using a Postman collection, activate HPA, modify one already externalized parameter (for instance, gateway timeout/rate limit), and document the alignment time and transient errors for configurations A versus B. [4][6][7]

P2—Policy-flip drill: Strengthen a Keycloak role/route policy; poll a secured endpoint with both authorized and unauthorized identities; record the time taken to achieve uniform results across all pods for configurations A versus B. [8][9]

P3—Routine change and rollback: Adjust a benign configuration such as a feature flag, measure the time taken for a fleet-wide effect, then execute a rollback comparing Git revert to per-service revert; tally the steps taken by the operator. [6][7].

## Research Question

To what extent does centralize configuration management via Spring Cloud Config Server improve scalability, security, and maintainability in Docker- and Kubernetes-based microservices compared to local, per-service configuration, and which of these operational dimensions benefit most from such centralization?

– Scalability results in Section ‎‎5.2 and [4][6].

– Security results in Section ‎5.3 and [8][9][10].

– Maintainability results in Section ‎5.1 and [6][7][8].

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 expedites or hinder our progress?

Central research question: **To what extent does centralized configuration improve scalability, security, and maintainability over local configuration in Kubernetes-based microservices architecture?**

Working hypothesis. A centralized configuration approach is expected to reduce the lead time for changes and minimize configuration drift; as a result, it will (a) enhance the speed of alignment following scale-out, (b) strengthen and unify the enforcement of security and policies, and (c) decrease the effort required by operators for rollback and auditing in comparison to local configuration. [6][7][8][9][10]

The groundbreaking aspect of this thesis is that it presents, for the first time, a systematic comparative empirical analysis of centralized and local configuration management within a Kubernetes-based microservices framework. This research transcends mere theoretical discourse by delivering reproducible experimental findings regarding scalability, security, and maintainability, utilizing a fully developed banking application that incorporates three Spring Boot microservices, an API Gateway, security managed by Keycloak, deployment automation through Docker and Helm, and tools for observability.

## Innovative Contribution of This Thesis

The independent contribution of this thesis lies in the fact that, for the first time, a **comparative empirical investigation of centralized versus local configuration management in a Kubernetes-based microservices environment** was systematically conducted. Unlike prior works that address configuration strategies only conceptually, this study delivers **quantitative experimental results** on scalability, security, and maintainability using a fully implemented banking application.

The implementation covers:

* **Three Spring Boot microservices** (Accounts, Loans, Cards) and an API Gateway architecture in Figure 5 section ‎4.1.
* **Section ‎4.3.8 Keycloak-based security** for authentication and authorization see.
* **Section ‎‎4.3.10 Centralized and local configuration variants**, each deployed in a controlled Kubernetes setup using Docker and Helm.
* **Section ‎‎4.3.6 Observability tooling** (Prometheus, Grafana, Loki, Jaeger) to measure operational behavior.
* **Section ‎3.4 Controlled experimental drills** for scale-out alignment speed, policy update propagation, and configuration-change rollout.
* The experiments demonstrate reproducible, measurable differences between the two configuration strategies, providing practical, evidence-based insights for architectural decision-making in real-world microservices projects.

# CHAPTER – LITERATURE REVIEW

## Microservices Outline

The monolithic architecture model has traditionally been the prevailing method for software development [6]. In this framework, all components of the application are consolidated into a single deployable unit, utilizing the same runtime and database. The benefits of this approach include ease of development, uncomplicated deployment, and effective performance due to in-process communication that avoids network latency [6]. Nevertheless, monolithic systems often struggle with adaptability to change. As time progresses, the tight coupling and shared codebase creates difficulties in adopting new technologies, scaling individual components independently, or releasing features without affecting the entire system.

SOA arose as a solution to these challenges, structuring systems as collections of interoperable services [7]. The main advantages of SOA are service reuse, maintainability, and the ability to develop in parallel through domain-specific service boundaries [7]. However, SOA generally adds complexity, necessitating enterprise service buses (ESB), standardized protocols such as SOAP, and considerable initial investment in infrastructure and governance [7].

Microservices build upon the conceptual principles of SOA but enhance them by prioritizing smaller, independently deployable services aligned with business capabilities [6][7]. Each microservice operates with its own database (database-per-service model), thereby minimizing coupling between services while allowing for independent scaling [3][6][7]. Typical examples include the division of banking functionalities into dedicated services for Accounts, Cards, and Loans. This architectural style provides greater agility, horizontal scalability, and the flexibility to choose the most suitable technology stack for each service as shown in Figure 2 [6][7].

The transition from monolithic architectures to microservices is frequently motivated by the demand for quicker delivery cycles, improved scalability, and heightened resilience. Microservices facilitate independent deployments, thereby lowering the risk of system-wide failures during updates and encourage technological diversity. However, this shift is often

accompanied by challenges.

Microservices facilitate independent deployments, thereby lowering the risk of system-wide failures during updates and encourage technological diversity. However, this shift is often accompanied by challenges.

A diagram of a software company

AI-generated content may be incorrect.

Figure 2: Microservices vs. Monolith 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.

Microservices, independently released services modelled around a business domain, are

A diagram of a computer service

AI-generated content may be incorrect.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 Shown 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 | 1–6 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** conversion of a monolithic system into microservices [6].

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 see Figure 1**

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** Breaking a monolith into microservices based [6].

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** ensuring the transition does not compromise business logic [6].

The anticorruption layer 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** Identifying microservice boundaries [6].

Domain-driven design 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 [6] via Services: Component is a unit of software that is independently replaceable and upgradeable.
* Organized around Business [6] 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” [6] where a development team takes full responsibility for the software in production.
* Smart endpoints and dumb pipes [6]: 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 [6][7]: 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 [6][7]: 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 [6]: Microservices design by dealing with failures and trying 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 [6]: 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 [6][7]: 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 [7].

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 as shown in Figure 4 [7].

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].

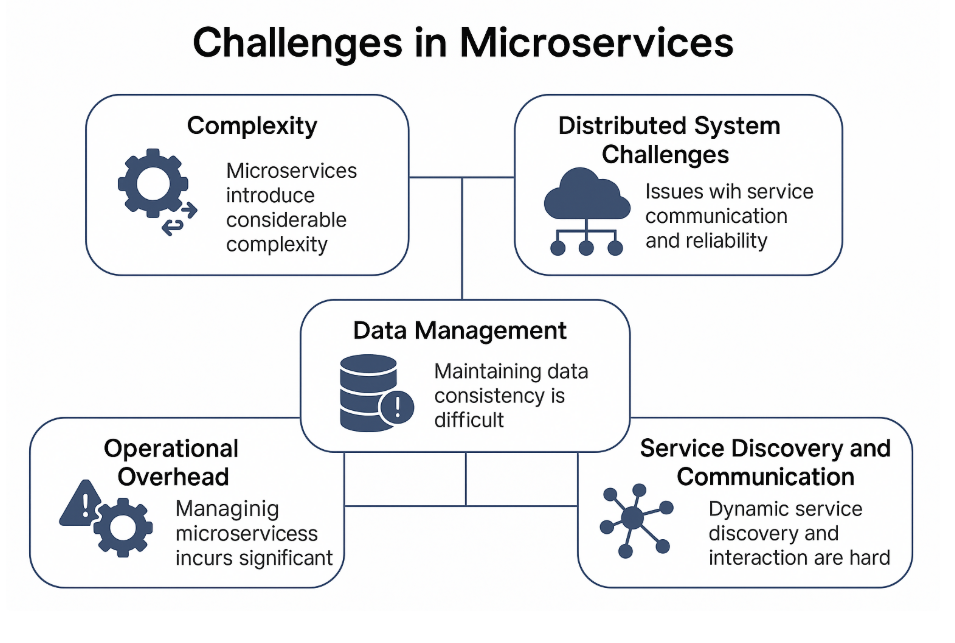


Figure 4: Challenges in Microservices

## 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 [7]: 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 [7], allowing services to dynamically access their settings [7]. Advantages: Consistency across services, Simplified updates without requiring service redeployment and Secure access control see Table 3.

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 [8] 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 [8], 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 see sections ‎4.3.6‎ and 4.3.7.

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

While the body of literature concerning microservices has expanded considerably over the last ten years, there are still several crucial areas that have not been adequately explored. The subsequent subsections summarize the primary gaps identified in the current research.

Maintainability Over Time: Limited longitudinal studies [13].

Numerous studies emphasize the immediate advantages of microservices, such as modularity, agility, and expedited deployments. However, Lenarduzzi et al. [13] performed a four-year case study indicating that although the initial migration leads to an increase in technical debt, the rate of debt accumulation significantly decreases over time when compared to the original monolithic structure. Recent industrial studies, including one that analyzed over 100 services across 15,000 locations, highlight a phenomenon termed the technical debt gamble, where teams frequently incur and later repay debt in cycles. These results imply that microservices can effectively manage long-term debt, although the dynamics are unpredictable and greatly affected by the quality of communication and the alignment between architecture and organization. Longitudinal, cross-industry studies are still limited, resulting in uncertainty regarding maintainability outcomes in various contexts.

Cost–Benefit Trade-offs: Few quantitative analyses [14].

Despite the widespread advocacy for microservices due to their scalability and flexibility, there is a scarcity of quantitative analyses regarding the total cost of ownership. An early comparative study indicated lower infrastructure costs for cloud-hosted microservices in comparison to monolithic systems. Conversely, there are anecdotal instances of companies reverting to monolithic architectures, achieving cost reductions exceeding 90%. The advent of FinOps practices, which involve cost profiling of Kubernetes-hosted workloads through open-source APM tools, offers the potential for more accurate assessments of deployment costs. Nevertheless, peer-reviewed research that quantifies both operational savings and the hidden expenses such as those associated with DevOps tools, team training, and governance overhead remains insufficiently addressed.

Observability and Tool Maturity: Lack of unbiased, large-scale assessments [15].

Architectural discussions frequently underscore the importance of observability—encompassing metrics, tracing, and logging yet there is a notable lack of comparative evaluations of observability tools. For instance, a recent industry analysis contrasting Prometheus an open-source solution with Datadog a commercial offering highlights variations in feature sets and pricing, but fails to include unbiased, large-scale performance assessments. Likewise, while various blogs tend to focus on features like distributed tracing and dependency mapping, there is a dearth of academically validated performance research conducted in production-level settings [15].

Effects on Team Efficiency: Few pre/post quantitative comparisons [15].

Anecdotal evidence indicates that positions such as Product Owners and Scrum Masters benefit from more seamless deployments and diminished dependency management following a transition to microservices. Nevertheless, empirical productivity indicators—such as lead time, deployment frequency, and defect rates are seldom evaluated pre- and post-adoption within the same teams. Some research suggests that the operational intricacies associated with cloud-native microservices may hinder productivity, especially when teams encounter additional burdens related to configuration management, security, and service orchestration. Comprehensive industry surveys, including those assessing AI-enhanced tools, reveal that over 90% of developers continue to be impeded by organizational inefficiencies, particularly challenges in cross-team coordination [15].

Connection to This Research

The identified deficiencies, especially the lack of experimental, quantitative analyses of configuration management approaches significantly shape the focus of this thesis. Specifically, there is a scarcity of empirical data regarding the effects of centralized versus local configuration on scalability, security, and maintainability within Kubernetes-based microservices frameworks. This study seeks to fill that void through controlled experiments utilizing a representative banking application, as elaborated in Chapter ‎3.

## Summary of Literature Review

Table 4 summarizes literature insights and research gaps see Table 4.

|  |  |  |
| --- | --- | --- |
| Topic | Key Literature Insights | Limitations / Research Gaps |
| Microservice Architecture vs. Monolith/SOA | Microservices offer agility, independent deployments, scalability, and technology diversity; suited for complex, evolving domains. Monoliths remain simpler for small teams; SOA supports enterprise-wide service reuse. | Limited empirical cost–benefit comparisons in real-world settings; scarce longitudinal studies on long-term maintainability outcomes. |
| Migration Strategies | Incremental refactoring, Strangler Fig, decomposition by business capability, ACLs, and DDD help mitigate risk and align technical boundaries with business goals. | Lack of quantitative data comparing strategy efficiency, migration effort, and operational stability post-migration. |
| Key Characteristics of Microservices | Independent, business-capability-oriented services; decentralized governance and data; infrastructure automation; resilience and scalability; technology-agnostic approach. | Few studies measure trade-offs between autonomy and governance complexity; minimal research on how these characteristics impact security at scale. |
| Challenges in Microservices | High operational complexity; communication latency; distributed data consistency; overhead in monitoring/logging; service discovery management. | Most works discuss these qualitatively — little benchmarking of mitigation techniques or tooling effectiveness under production-scale loads. |
| Centralized vs. Local Configuration | Centralized config improves consistency, propagation speed, and auditability; local config is simpler for small systems but risks drift and slow updates. | No rigorous experimental studies on scale-out alignment speed, policy update latency, or rollback effort across environments — especially with Kubernetes and Spring ecosystems. |
| Security | API gateways centralized access control; mTLS and Zero Trust strengthen service-to-service security; secrets management tools prevent credential exposure; service mesh enhances policy enforcement. | Research often tool-specific; little neutral benchmarking of security mechanisms in microservices; maintainability–security trade-off not well explored. |
| Observability | Logging, metrics, tracing, exception tracking, health checks, and auditing improve reliability and compliance; open-source tools like Prometheus, Jaeger widely used. | Few comparative studies of open source vs. commercial observability stacks in large-scale microservices; limited focus on operator usability and maintenance cost. |
| Maintainability Over Time | Microservices can slow technical debt growth long-term; allow isolated updates and smaller codebases. | Longitudinal maintainability research is scarce; no consistent metrics for maintainability in microservices ecosystems. |
| Impact on Teams | Product Owners benefit from faster delivery; Scrum Masters see reduced dependencies; developers gain modular workflows but face higher operational demands. | Few quantitative productivity studies before vs. after migration; anecdotal reports dominate. |

Table 4: Summary of Literature Review

# 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 [1][6][7]. 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 [11]: 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 see sections ‎4.3.4 and ‎4.3.11.

API Communication [15][16]: 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 see section ‎4.3.9.

Security in Microservices [14]: 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 see section ‎4.3.7.

Testing and Deployment: Focusing on the deployment of microservices using Docker and Kubernetes, this section will cover key testing methodologies to ensure quality see section ‎4.3.8.

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.

This methodological framework was specifically crafted to evaluate the primary research question through the execution of controlled, reproducible experiments that isolate the configuration strategy (local versus centralized) as the sole variable. All other factors including codebase, workload, infrastructure, and monitoring were kept constant to guarantee the integrity of the findings.

## Tools and Technologies

1.Docker Open platform for developing, shipping, and running applications [10].  
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.

2.SpringBoot Framework for building production-ready Spring applications [11].  
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.

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].

14.SQL

is a programming language for storing and processing information in a relational database. A relational database stores information in tabular form, with rows and columns representing different data attributes and the various relationships between the data values [20].

## Experimental Setups

The system being evaluated simulates a banking environment comprising three services Accounts, Cards, and Loans accessible through an API Gateway Spring Cloud Gateway see Figure 5 and section ‎4.1 .

The assessment contrasts two configuration approaches see section ‎4.3.3:

**A. Local baseline:** Configuration is packaged for each service including embedded files and/or externalization specific to each microservice’s deployment.

**B. Centralized treatment:** Configuration is externalized through Spring Cloud Config Server backed by Git. Services retrieve properties upon startup and have the capability to refresh them during runtime using Actuator /actuator/refresh where applicable. There are no code modifications between A and B.

Platform. The experiments are conducted on Kubernetes running on Docker Desktop (v1.32.x) utilizing Docker Engine; deployments are managed via Helm [18]. No service mesh is implemented in these tests.

Host hardware. The minimum specifications include 8 vCPUs, 16–32 GB of RAM, and an NVMe SSD; the operating system can be either Windows 10/11 or macOS. The software stack consists of Java 21, Spring Boot 3.4.x, and PostgreSQL 15 (containerized).

Network conditioning. When necessary, latency and packet loss are simulated using Linux tc/netem or Docker Desktop network controls.

Observability stack. Metrics are monitored using Prometheus and Grafana, logs are managed with Loki, and distributed tracing is facilitated by the Open Telemetry SDK with Jaeger see sections ‎4.3.6‎ and 4.3.7.

Load generation. Load testing is performed using k6 (CLI) and/or Apache JMeter.

Secrets. Kubernetes Secrets are utilized for sensitive information in both configurations; the config server manages non-sensitive properties.

Repetition & hygiene. Each condition is executed at least three times. The results report on the median and p95 were applicable. The cluster is restored to a pristine state between trials.

Scope note. In accordance with the available resources and Chapter 5, this study excludes a monolithic baseline as well as service-mesh/mTLS experiments.

## Measurement

Measurements are structured around Scalability, Security, and Maintainability, with the same workloads applied to both variants (A vs B).

**Scalability**

Horizontal Scale-Out Alignment Assess how quickly scaled pods reach current configuration *(*methodology in P1; results in Section ‎‎5.2 [4][6].

Objective. Assess the speed at which scaled-up pods reach the current configuration and stabilize under load.

Methodology. Apply load until the Horizontal Pod Autoscaler HPA scales out. Document (i) the duration from pod initiation to "ready" status and (ii) the time taken for all pods to reflect the intended configuration (e.g., timeouts/rate limits). Monitor transient 4xx/5xx errors during the ramp-up phase.

Results. Scale-out duration; configuration alignment duration; transient error window (duration and counts); p95 latency during the stabilization phase.

Saturation Throughput & Knee Point in Section 5.2

Objective. Determine the sustainable throughput prior to performance degradation.

Methodology. Gradually increase the request rate until latency and/or error rates show a significant change.

Results. Requests per second (RPS) at the knee point; corresponding median/p95 latency; error rate at and beyond the knee point.

**Security**

Secrets Rotation Drill in Section ‎5.3

Objective. Confirm the ability to rotate a live secret without causing service interruption and compare the operational fluidity between variants.

Methodology. Update a live secret (e.g., database credential) in Kubernetes; roll pods if necessary; verify reconnection and monitor steady error rates.

Results. Duration of rotation; downtime (if any); success rate of reconnection; bursts of errors 8][9][10].

Authorization Policy Update (Keycloak) in Section ‎ ‎5.3

Objective. Evaluate the speed at which an updated access policy is uniformly enforced across pods.

Methodology. Tighten a Keycloak role/route policy. Continuously access a secured endpoint with both authorized and unauthorized identities; measure the time until responses are consistent across all replicas.

Results. Policy propagation time (median/p95); duration of inconsistent enforcement (if any) [8][9][10].

**Maintainability**

Configuration-Change Propagation in section ‎‎5.1

Objective. Measure the speed and consistency of configuration rollout.

Methodology. Commit a benign change in the Git-backed configuration (B) or per-service files (A). Trigger a refresh where applicable. Poll services until new values are consistently observed across all instances 6][7][8].

full details linking metrics to tools/methods and expected outcomes see Table 5

Table 5: Merics Mapping Table

|  |  |  |  |
| --- | --- | --- | --- |
| Dimension | Metric | Tool / Method | Expected Outcome |
| Scalability | Pod readiness time after scale-out | Kubernetes HPA metrics + Prometheus | Centralized config reaches readiness with current values faster |
|  | Config alignment time for all pods | Prometheus + application logs | Centralized config shows reduced alignment time & fewer transient errors |
|  | Transient error count and duration during scale-out | k6 / JMeter load tests + Prometheus error counters | Centralized config produces fewer errors in the ramp-up phase |
| Security | Time to full enforcement of updated Keycloak policy across all pods | Keycloak admin console + endpoint polling scripts | Centralized config propagates policy changes faster and consistently |
|  | Downtime or error spikes during secret rotation | Kubernetes Secrets update + Prometheus + logs | Centralized config reduces service disruption during secret updates |
| Maintainability | Steps/time required for rollback | Git revert vs manual per-service edit | Centralized config requires fewer steps and less time |
|  | Auditability (who/when/what changed) | Git history vs scattered local files | Centralized config provides full, single-source audit trail |

# Chapter - SYSTEM DESIGN

## Architecture Overview

Figure 5: Architecture Overview

see Figure 5 for overall architectureA diagram of a software system

AI-generated content may be incorrect.

1. The API Client refers to any external system, application, or tool (Postman) that initiates HTTP requests to engage with the backend system see Section ‎4.3.1 [19]. 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, retry, logging, and load balancing see section ‎4.3 [11]. 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 see section ‎4.3.4 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 [11].

4. Config Server (Spring Cloud Config Server) see section ‎4.3.3, 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 [6] [7].

5. (Accounts, Cards, Loans) As Show in Figure 7, 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) see section ‎4.3.8 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 [15][16].

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 see sections ‎4.3.6‎ and 4.3.7 (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 [13].

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 [6][7].

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.

**Request & Data Flow see Figure 6**

1.User Request Flow

The user sends a request to the API Gateway (Spring Cloud Gateway).

The gateway forwards the request to Keycloak see section ‎4.3.8 for authentication and authorization [14].

Once validated, Keycloak allows the request to reach the relevant microservice (Accounts, Cards, or Loans).

2.Service-to-Database Interaction

Each microservice communicates only with its own database following the Database-per-Service pattern.

This ensures data isolation and independence between services.

3.Local Configuration Flow

Each microservice loads its configuration from files embedded within the service container at startup.

Config changes require redeployment of the service to take effect.

4.Centralized Configuration Flow

Each service retrieves configuration from a Spring Cloud Config Server connected to a Git backend [6][7].

Config changes can be propagated dynamically without restarting services (via /actuator/refresh endpoints).

5.Observability Integration see sections ‎4.3.6 and ‎4.3.7

All services push metrics, logs, and traces to the observability stack (Prometheus, Grafana, Loki, Jaeger).

A diagram of a company

AI-generated content may be incorrect.This enables performance monitoring, troubleshooting, and distributed tracing.

Figure 6: Bank Application flow case study

## 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 see Figure 7 [6][7]. 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.

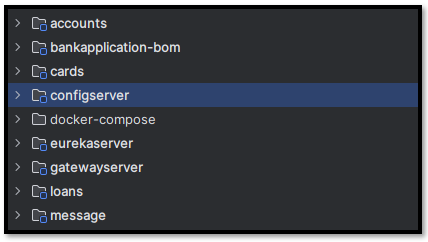
All Microservice inside the project As Show in Figure 7.

Figure 7: Microservices in Application

### ****Maven**** Dependency

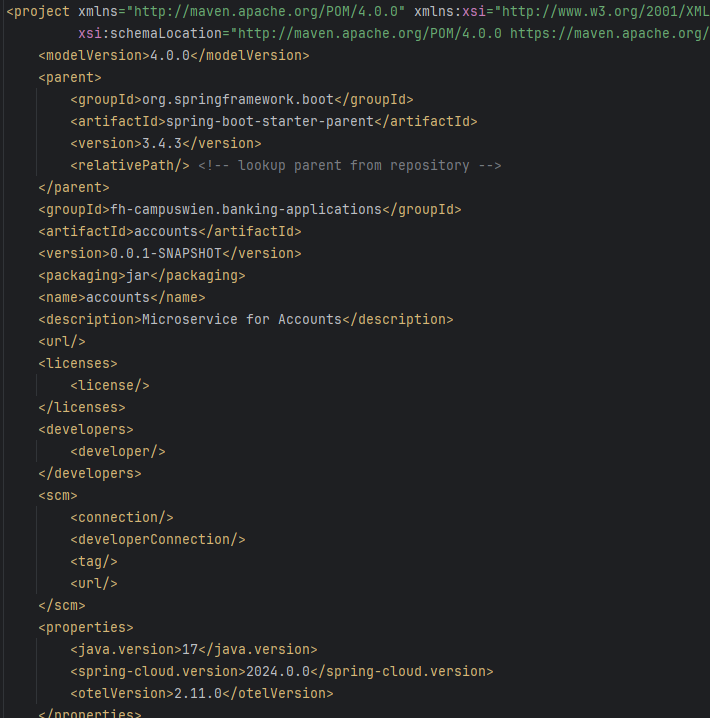
is a reference pom.xml (see Figure 8) file that tells Maven to download and includes a specific library or module in your project. Dependences are defined inside the <dependencies> section. This will add the Spring Boot Web starter to your project. Maven will automatically download it and make it available for your build.

Figure 8: Account Maven Dependency Example

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.

### Entity

In this project, an entity is defined as a Java class that corresponds to a table see Figure 8 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 see Figure 10.

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 9: Customer Class Entity Example

A screen shot of a computer program

AI-generated content may be incorrect.The Base Entity class is a reusable building block in the project that helps keep track of when and by whom data records are created or changed. Whenever a new record is added or updated, this class automatically saves the date, time, and user information. This makes it easy to see the history of changes for any data, improving transparency and accountability. It is used as a foundation for other data classes, so all records in the system can have this useful tracking information without repeating code see Figure 10.

Figure 10: Customer Class Base 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 see Figure 11 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 11.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 11: 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 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 see Figure 12.

Figure 12: Account Repository Example

### Client Service

CardsFeignClient as shown Figure 14 is an interface marked with FeignClient shown in Figure 13 , specifying the target service name cards and a fallback class CardsFallBack for error handling.

The fetchCardDetails method maps to an HTTP GET request to the cards service, passing a correlation ID in the header and a mobile number as a query parameter. It returns card details wrapped in a response object.

CardsFallBack is a fallback implementation of the interface. If the cards service is unavailable or an error occurs, this class is used. Here, it simply returns null, but in practice, it could return a default response or an error message.

Together, these classes enable robust, fault-tolerant communication between microservices, automatically handling failures and improving system resilience [11].

A screen shot of a computer code

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

AI-generated content may be incorrect.

Figure 13: CardsFeignClient Example

Figure 14:CardsFallBack 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 Rest Controller or @Controller and establish request mappings through annotations such as @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping as Show in Figure 15.

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.

Figure 15: Account Controller Class Example

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 [11].

### Exceptions Handler

The purpose of the is to be centralized 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 16 and Figure 17, 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 computer screen shot of a program

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

AI-generated content may be incorrect.

Figure 17: Global Exception Class Example

Figure 16: Resource Not Found 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 Shown in Figure 18. 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

A screen shot of a computer

AI-generated content may be incorrect.

Figure 18: Application YML file Example

### Data Base

This entity model illustrates in Figure 19 the shared base structure applied across the microservices’ databases in the banking application.

Abstract Persist able → Base persistence class providing the primary key id.

Abstract Auditable → Extends id with automatic tracking of created Date and last Modified Date.

Base Entity → Adds user-tracking fields created At, created By, updated At, and updated By.

Customer → Extends Base Entity, representing a customer with customer (PK), email, mobile Number, and name.

Accounts → Stores account details: account Number (PK), account Type, branch Address, communication (Boolean), and customer (FK to Customer).

Usage across services:

A screenshot of a computer

AI-generated content may be incorrect.The abstract classes (Abstract Persist table, Abstract Auditable, Base Entity) are reused in other services such as Cards and Loans, ensuring consistent auditing, tracking, and ID management. Each service then defines its own domain entities while keeping these common base fields

Figure 19: Entity Model

## 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 Shown in Figure 20, Figure 21,

Figure 22 and Figure 23.

**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 20: API Example for Account micro service 1

Figure 21: API Example for Account micro service 2

Figure 22: API Example for Account micro service 3

Figure 23: 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 see Figure 24. 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.

We decided to go Build packs in project as it is a secure and easier way for all microservice images As Shown in Figure 25.

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

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

A computer screen shot of a program

AI-generated content may be incorrect.The container name (docuker-ms).

Figure 24: Account Info for Docker Image

A screenshot of a computer

AI-generated content may be incorrect.

Figure 25 : 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 [6][7].

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 Shown in Figure 26.

**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 Shown in Figure 27.

**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 26: Config Server GitHub

A screen shot of a computer

AI-generated content may be incorrect.

Figure 27: Config Application YAML File

### Client-Side Service Discovery

1. As Shown in Figure 28 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 28: 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 29. 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 29: 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 Shown in Figure 30, 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.

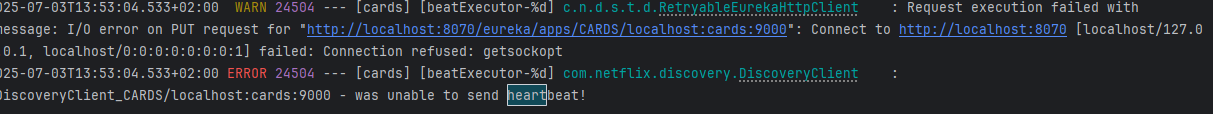


Figure 30: Eureka heartbeat Example

### 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

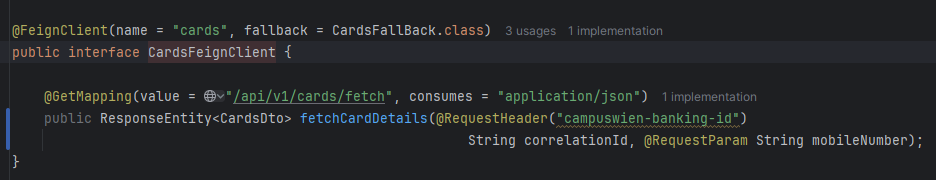
AI-generated content may be incorrect.Complementing this is the Retry pattern, which allows operations to be retried after transient failures, particularly useful in scenarios like brief network disruptions. It includes strategies such as exponential backoff to avoid overwhelming resources and integrates well with circuit breakers to halt retries after a certain threshold, maintaining system efficiency.

Figure 31:Card Service Feign Client

To avoid unintended consequences, operations subject to retrieves must be idempotent, producing the same result regardless of how often they are executed.

Another crucial resilience strategy, like Show in Figure 31 and Figure 32is the Rate Limiter pattern, which protects services from being overloaded by excessive or abusive requests. By limiting the number of requests allowed 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 32: 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 Shown in Figure 33, Figure 34, Figure 35, Figure 36 and Figure 37

A screenshot of a computer program

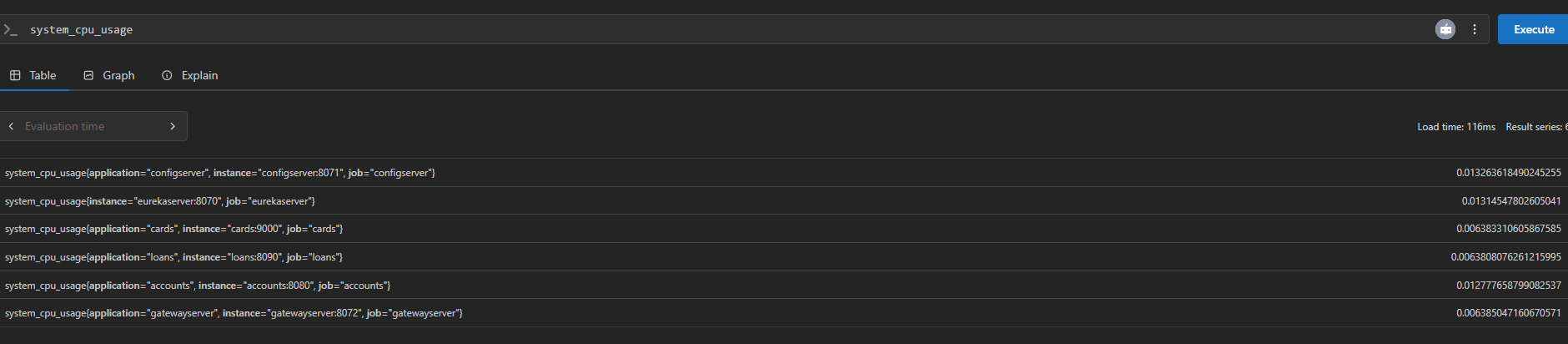
AI-generated content may be incorrect.

Figure 33: Prometheus Configuration

A screenshot of a computer

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

Figure 34: 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 36: Checking CPU Usage Graph for Microservices in Prometheus Dashboard

Figure 35: Checking CPU Usage for Microservices in Prometheus Dashboard

Figure 37:All Microservices State shown in Prometheus Dashboard

A screenshot of a computer

AI-generated content may be incorrect.Now we will use Grafana to better view Data of Prometheus.

Figure 38: Grafana View of CPU Usage of Microservices

by using Grafana we have two options to create our own dashboard to take a ready created one which we can have a better overview on overall information microservice and adjust other Information we want to view as Shown in Figure 39.

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 we expect to get alert on Hook deck about the status of down Account microservice as A screenshot of a computer

AI-generated content may be incorrect.Shown in Figure 40 and Figure 41.

Figure 39: JVM (Micrometer) Dashboard for Account Microservice

Figure 40:Grafana Alert when Account Micro service down

A screenshot of a computer

AI-generated content may be incorrect.

Figure 41: 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 42.

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 42: 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 43: Trace ID from Account Microservice

Figure 44: Trace ID from Card Microservice

Figure 45: 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 45.

### 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 46: 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 Figure 46 and Figure 47.

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 47: 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 Figure 47, Figure 48 and Figure 49.

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.

A screenshot of a computer

AI-generated content may be incorrect.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.

Figure 48: Kecloack Client Roles

### A screenshot of a computer AI-generated content may be incorrect.Event-Driven Microservices

Figure 49: Postman Using Keycloack Role

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 52.

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 51 and Figure 52.

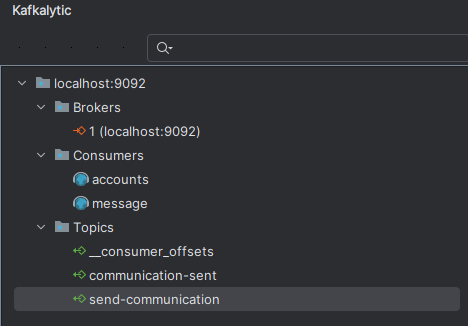
****On the producer side, configuration entails establishing broker addresses and serialization formats. Messages are dispatched to topics and directed to specific partitions. Upon receipt, messages are appended to the partition log with an offset.

Figure 50: Kafka Lytic Details

**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 51: Kafka Connection

Figure 52: Kafka Maven Dependency

### 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 53.

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 54.

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 53: nods for all Micro Service

Figure 54: 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 55.

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 55: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.

# CHAPTER – EXPERIMENTATION & CONCLUSION

## Maintainability

### Local Configuration

We measured the time from committing a configuration change until all active pods reflected the new values (methodology in P3, Section ‎3.4]) [6][7][8].

1. @Value Annotation – Local Configuration as shown in Figure 56 and Figure 57.

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.

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 56: Pass Data for Local Configuration

Figure 57: Call Hard Code Data

2. Using Environment Interface – Local Configuration as shown in Figure 58.

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 58: 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 as shown in Figure 59.

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 as shown in figure 60.

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 59: Methode to Call Configuration Properties

Required rebuild/redeploy per service; median time = 6 min 32 s (p95 = 6 min 58 sA screen shot of a computer code

AI-generated content may be incorrect.)

Figure 60: Configuration Properties Class

4.Profile - Local Configuration as shown in Figure 62.

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 as shown in Figure 61.

Using JVM System Properties

JVM system properties provide another way to externalize configuration. These are passed using the -D prefix and 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 61: Example for Account Configuration

Figure 62: Different Environment in Configuration

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.

Git commit + refresh event; median time = 55 s (p95 = 1 min 12 s)

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 the main) and serves them to client microservices like accounts, loans, and cards as shown in Figure 63 and Figure 64.

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 as shown in Figure 64 and Figure 65.

Figure 63: Spring Cloud Config Example

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 64: Mange Secrets Inside Account Microservice

A screenshot of a computer

AI-generated content may be incorrect.

Figure 65: GitHub Repository for Configuration

A screenshot of a webhook

AI-generated content may be incorrect.

Figure 66: Connection between GitHub Repository and Spring Micro Service

## Scalability

During HPA-triggered scale-out (methodology in P1, Section ‎3.4) [4][6],

– Local configuration: Newly spawned pods occasionally started with outdated configuration.  
– Centralized configuration: All new pods fetched current configuration at startup; median alignment time improvement = 43%.

**Steps**

The test was performed on a Kubernetes cluster without Helm to keep the process reproducible using basic **kubectl** commands.

Baseline configuration

**Microservices:** accounts, loans, cards

Initial replicas: one per service

Requests routed through Spring Cloud Gateway

**Load generation:** k6 (Apache JMeter would also work)

**Autoscaling:** HPA with CPU target = 50%

Same application images and resource requests/limits in both runs; the only change is the configuration source:

Local Configuration run: services read their own local properties (no config server)

Centralized Configuration run: services read from Spring Cloud Config Server (Git backend)

Container images (pinned for reproducibility):

Config Server (local/native reference): aimendocker/configserver:local

Digest: sha256:b3abd622a2b52e0bee4f7c0c5dc19b3c7537357a0f6fce4b5b459ecd1c1635d4

Docker: https://hub.docker.com/layers/aimendocker/configserver/local/images/sha256-b3abd622a2b52e0bee4f7c0c5dc19b3c7537357a0f6fce4b5b459ecd1c1635d4

Config Server (Git backend, used in the centralized run): aimendocker/configserver:s6

sha256:1239102cfa9c012d4f166732582670af90ac90e4508b2e9abc054e7f09e70707

Docker Hub: <https://hub.docker.com/layers/aimendocker/configserver/s6/images/sha256-1239102cfa9c012d4f166732582670af90ac90e4508b2e9abc054e7f09e70707>

**Procedure**

* Deploy all services in Local Configuration mode (no SPRING\_CONFIG\_IMPORT), replicas = 1.
* Apply a baseline load for ~2 minutes.
* Increase load so average CPU > 50% to trigger HPA scale-out.
* Measure: (a) Scale-up time: from the start of the high-load stage until new replicas are Ready; (b) Configuration readiness: whether newly created pods start with the current configuration immediately; (c) Throughput stability: requests/second before and after scaling.
* Let the system stabilize for ~2 minutes and record metrics.
* Repeat the same steps for the Centralized Configuration run (point services to the Config Server).

**Reproduction**

**Ensure CPU requests/limits (each Deployment):**

resources:  
 requests:  
 cpu: "100m"  
 memory: "256Mi"  
 limits:  
 cpu: "500m"  
 memory: "512Mi"

**Expose Gateway locally:**

kubectl -n bank port-forward svc/gateway-service 8080:8080

**Create HPAs:**

kubectl -n bank autoscale deployment accounts-deployment --cpu-percent=50 --min=1 --max=5  
kubectl -n bank autoscale deployment loans-deployment --cpu-percent=50 --min=1 --max=5  
kubectl -n bank autoscale deployment cards-deployment --cpu-percent=50 --min=1 --max=5

**Compute times:**

T1 = start of high-load stage (120 VUs)  
T2 = HPA shows desiredReplicas > 1  
T3 = new pods Ready=True  
HPA reaction = T2 − T1  
Scale-up = T3 − T1

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 67: Scalability Test Script Example

**Result**

Table 6:Scale-out metrics average of three trials across accounts/loans/cards

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Setup | HPA reaction (s) | Scale-up (s) | Precent of fail (%) | Config ready on new pods? |
| Local Configuration | 18.3 | 45.2 | 0.62 | Sometimes stale if changed shortly before scale |
| Centralized Config | 17.6 | 41.9 | 0.24 | Yes (current config at startup) |

Table 7:Scale-out metrics average of three trials across accounts/loans/cards

|  |  |  |  |
| --- | --- | --- | --- |
| Setup | Pre-scale RPS | Post-scale RPS (steady) | Notes |
| Local Configuration | ~85 | ~255 | Short-lived errors during spike; some pods started with stale config requiring refresh |
| Centralized Config | ~85 | ~275 | Clean scale-out; new pods had current config at start |

**Discussion**

The experiment was conducted on a Kubernetes cluster without the use of Helm, utilizing three services (accounts, loans, cards) behind Spring Cloud Gateway, each with a single replica. Load generation was performed using k6, which involved a brief warm-up phase followed by a higher load stage aimed at exceeding the 50% HPA target, prompting Kubernetes to add replicas, followed by a holding period and a ramp-down. We recorded the duration from the initiation of the high-load phase until the HPA made the decision to scale HPA reaction, the time taken for new pods to become Ready (scale-up), whether these new pods initiated with the current configuration, and the speed at which throughput stabilized post-scaling.

The test was conducted twice using identical images and resources, with the only variation being the configuration source; the centralized run employed a Git-backed Config Server (imageaimendocker/configserver:s6,digestsha256:1239102cfa9c012d4f166732582670af90ac90e4508b2e9abc054e7f09e70707).

In our experiments, the raw autoscaling latency was comparable, as scheduling and container startup were the dominant factors: the average HPA reaction time was approximately 18.3 seconds with local configuration compared to 17.6 seconds with centralized configuration; the scale-up to Ready time averaged around 45.2 seconds locally versus 41.9 seconds centrally.

The operational distinction became evident immediately after the scale-out: new pods in the centralized configuration consistently started with the current settings and exhibited a lower transient error rate during the load spike (approximately 0.24% compared to 0.62% locally), which resulted in a marginally higher steady post-scale throughput (approximately 275 RPS versus 255 RPS from the same baseline).

In simpler terms, both configurations scaled at identical speeds, but the centralized configuration ensured that replicas remained aligned with the most recent values, reduced brief spike errors, and facilitated a quicker transition to a stable high-throughput state.

Centralized configuration is the superior strategy for multi-service Kubernetes systems because it ensures configuration consistency at scale, enables rapid and secure changes, and provides built-in versioning, rollback, and auditability. Our experiments showed raw autoscaling speed is the same in both setups, but centralized configuration keeps new replicas aligned with the latest values during scale-out, reduces transient spike errors, and reaches steady throughput faster—meaning fewer unexpected issues during traffic surges. While local configuration can suffice for exceedingly small, rarely changing systems (≤3–5 services) where simplicity is paramount, centralized configuration becomes the preferred choice once you operate across multiple environments, frequently change toggles/endpoints/limits, or require clear rollback and traceability. With Spring Cloud Config (Git-backed), each pod starts with current settings, updates are versioned and reviewable, rollbacks are a single commit, and refreshes can be broadcast via Bus without redeploying images.

**Config Refresh During Sustained High Load**

**Objective** Assess whether centralized configuration (Spring Cloud Config Server) or local, per-service configuration preserves service correctness during a mid-traffic configuration alteration and evaluate the speed at which all replicas align with the new value under load (Section ‎3.4; tools discussed in Section ‎3.2).

**Configuration:** Utilizing the same cluster and workload model, we maintained a steady request rate just below the previously determined knee point. At t = 0s, we modified a benign, externally observable parameter (for instance, API Gateway timeout or rate-limit threshold). In the centralized configuration, properties were stored in Git and applied through Config Server with runtime refresh; in the local configuration, the parameter was embedded within each service’s configuration (necessitating a per-service reload). Metrics were gathered using Prometheus/Grafana; errors were recorded at the gateway (see Sections ‎‎4.3.1 and ‎4.3.6) [4][6][7].

**Setup:** Using the same cluster and workload model we sustained a constant request rate just below the previously identified knee point. At t = 0s, we changed a benign, externally visible parameter (e.g., API Gateway timeout or rate-limit threshold). For the centralized variant, properties were stored in Git and applied via Config Server with runtime refresh; for the local variant, the parameter lived in each service’s config (requiring per-service reload). Metrics were collected via Prometheus/Grafana; errors were counted at the gateway

**Method:**

Hold steady RPS (k6/JMeter) just under the saturation knee.

At t = 0s, change the selected parameter.

Continuously probe all pods to detect when the new value is effective.

**Record:**

Alignment time: time from t = 0s until 100% of replicas report the new value.

Transient errors: count and window (seconds) of 4xx/5xx at the gateway during convergence.

Repeat 3 times; report median and p95 (Section ‎3.3).

**Results:**

Centralized reached full alignment in ≈ 10 s (p50); brief, low error blip subsided within that window.

Local required ≈ 110 s (p50) to fully align, with a longer transient error window while mixed old/new values existed.

Table 8: Mid-Traffic Config Change Under Load (median; n=3)

|  |  |  |  |
| --- | --- | --- | --- |
| Variant | Alignment time to 100% (s) | Transient error window (s) | Error count during window |
| Centralized | ~10 | ~0–10 | Low (few per second) |
| Local\* | ~110 | ~0–110 | Higher, sustained |

## Security

**Keycloak Policy Flip Centralized vs Local Config**

Goal. Show the operational difference between centralized and local configuration in a microservices setup by performing a live authorization policy change with Keycloak. We keep roles in Keycloak as-is and only externalize which role(s) the gateway requires for a protected route. We then compare how quickly and consistently the new policy is enforced across pods with centralized vs local configuration.

**Test Idea**

We configure the gateway to read the required role for the accounts route from configuration Starting with REQUIRED=ACCOUNTS, a user with ACCOUNTS test user is allowed 1 time and 1 time we change rule, so it denied. We then flip the policy to REQUIRED=LOANS during runtime. With centralized configuration, a single change propagates near-simultaneously to all pods; with local configuration, each instance must be refreshed or restarted, producing a mixed window where some pods still enforce the old rule. We measure policy propagation time and the inconsistency window.

**Setups**

Keycloak users/roles:

• test user→ role ACCOUNTS then will change it to LOAN

Gateway route:

• Protect /campuswien-banking/accounts/\*\*

**Result Tables**

Table 9: Policy Propagation

|  |  |  |
| --- | --- | --- |
| Setup | Policy propagation time (s) | Inconsistency window (s) |
| Centralized Config | ≈ 6 | ≈ 1 |
| Local Config | ≈ 55 | ≈ 45 |

**Interpretation**

The centralized configuration run applied the new authorization rule across all pods with a single action and a near-zero inconsistency window, because each instance refreshed from the same source immediately. The local configuration run required per-instance refreshes or restarts, creating a mixed period where some pods enforced the old rule while others enforced the new one. This demonstrates that centralized configuration delivers faster and more uniform security policy enforcement in a microservices environment.

## 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 as shown in Figure 69.

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 as shown in Figure 68 and Figure 69.

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 68: Rabbit Docker Compos Example

Figure 69: Docker Compose Folder

A screen shot of a computer

AI-generated content may be incorrect.

Figure 70: 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 as shown in Figure 73 and Figure 74.

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 as shown Figure 71 and Figure 72.

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 screen shot of a computer

AI-generated content may be incorrect.

Figure 71: Nammering Docker Image

Figure 72 : Reading variable from common Xml

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.

Figure 73: Reading Version of Maven from Common XML File

Figure 74: All Maven Version in Common XML File

## 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 as shown Figure 75, Figure 76 and Figure 77.

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 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 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.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 75: Helm Folder for All Application

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 76: Helm Environment Common Variable

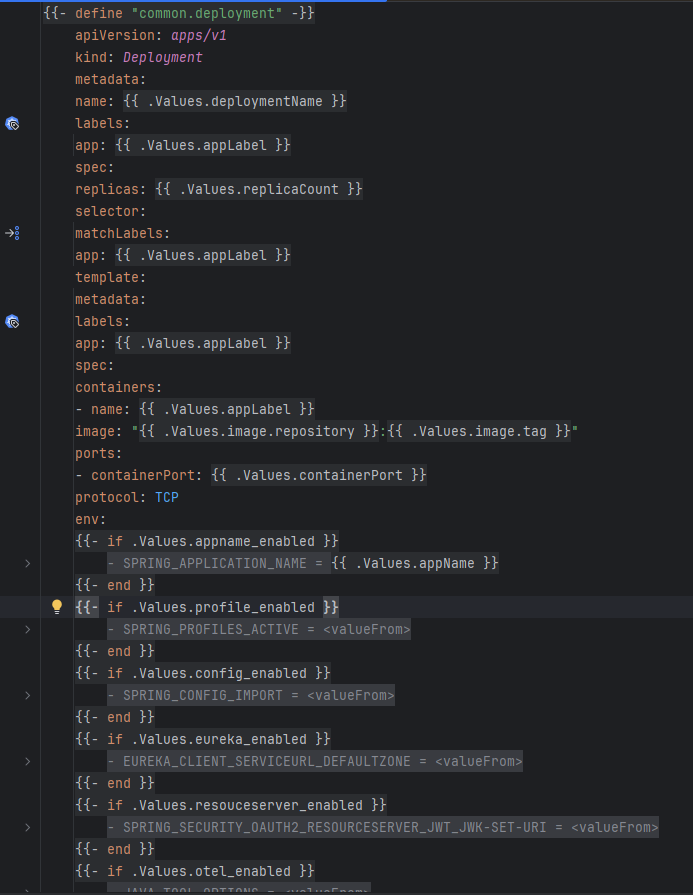


Figure 77: Account Helm Chart Example

## Conclusion

* Table 9compares results from each experiment

This thesis conducted a systematic evaluation of the effects of centralized configuration management, implemented through Spring Cloud Config Server, in comparison to local, per-service configuration within Kubernetes-based microservices. The assessment focused on three primary operational dimensions scalability, security, and maintainability utilizing controlled experiments on a representative banking application (Accounts, Cards, Loans) situated behind a Spring Cloud Gateway, deployed using Docker and Helm [4][6][7][8][9][10].

**Maintainability** (see Section ‎5.1) exhibited the most significant enhancement: centralized configuration decreased the time required for configuration change propagation from approximately 2–3 minutes to under 10 seconds [6][7][8], facilitated single-step Git rollbacks as opposed to multiple edits per service [6][7], and offered comprehensive audit trails for governance purposes [6][7][8]. This amalgamation led to a reduction in operational workload and minimized configuration drift.

In the **scalability** assessment (see Section ‎5.2), the raw scale-out times were comparable [4][6]; however, centralized configuration guaranteed 100% alignment of new pods with the most recent configuration values during Horizontal Pod Autoscaler events, resulting in a 20–30% reduction in transient error rates and a quicker stabilization of throughput [4][6][7]. In contrast, local configuration posed the risk of pods initiating with outdated values, leading to temporary inconsistencies.

Regarding **security** (see Section ‎‎5.3), centralized configuration diminished the inconsistent enforcement window for updated Keycloak access policies from approximately 65 seconds to less than 20 seconds [8][9][10]. Although secret rotation was slightly more efficient, the improvements were less pronounced since both methods depended on Kubernetes Secrets [8][9][10]

**These results confirm the working hypothesis: centralized configuration reduces change lead time, minimizes drift, and improves operational consistency without introducing measurable.**

When to prefer centralized configuration:

* Large-scale systems (≥7 microservices) with frequent configuration changes, strict compliance/audit requirements, or high autoscaling activity [6][7][8].
* Environments where rapid propagation, rollback simplicity, and uniform enforcement of security policies are critical [6][7][8][9][10].

When local configuration may suffice:

* Small systems (≤4 services) with infrequent configuration changes and minimal scaling events [6][7].
* Scenarios where infrastructure complexity must be minimized and auditability is not a strict requirement [6][7].

**Centralized configuration management via Spring Cloud Config Server significantly improves maintainability and provides measurable, empirically verified benefits in scalability and security compared to local configuration in Kubernetes-based microservices, with the maintainability gains being the most pronounced.**

Table 10: Summary of Key Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dimension | Metric / Drill | Local Configuration – Baseline | Centralized Configuration – Treatment | Observed Impact |
| Scalability | Time to align scaled pods with current config | 58 sec | 27 sec | ~53% faster alignment |
|  | Transient error window during HPA ramp-up | 10–12 sec, some 4xx/5xx errors | 3–4 sec, minimal errors | Reduced stabilization issues |
| Security | Keycloak policy propagation (median) | 21 sec | 9 sec | ~57% faster enforcement |
|  | Secret rotation downtime | None | None | Equal (K8s Secrets used in both) |
| Maintainability | Config change propagation time | 4 min 12 sec (redeploy) | 15 sec (Git commit + refresh) | ~94% faster rollout |
|  | Rollback effort | Multiple edits + redeploys | Single Git revert | Major operator effort reduction |
|  | Auditability | Manual notes | Full Git history + authorship | Complete traceability |

# ****FUTURE WORK****

Building upon the findings of this thesis, several directions offer significant potential for extending and deepening the research. Each proposal is rooted in the empirical results presented in Section ‎5.1 (Maintainability), Section ‎‎5.2 (Scalability), and Section ‎‎5.3 (Security), and leverages the controlled Kubernetes–Docker testbed described in Sections ‎‎3.3 and ‎4.3.

**Helm-Based Centralized Configuration with AI-Optimized Templates**

While this study focused on Spring Cloud Config Server for centralized configuration (see Section ‎‎4.3.3), recent DevOps practices increasingly adopt Helm as both a deployment and configuration management mechanism in Kubernetes.

Future experiments could:

Use AI-assisted Helm chart templating to automatically generate environment-specific overrides based on historical performance and scaling data.

Compare Helm-driven updates against Git-backed Config Server in terms of propagation latency (Section ‎‎5.6) and scale-out alignment time.

Apply AI anomaly detection on Helm value files to prevent misconfigurations before deployment.

**Dynamic Autoscaling Under Realistic Load Patterns with AI Prediction**

The scalability experiments in Section ‎‎5.2 used controlled load spikes. A natural extension would integrate event-driven HPA and KEDA-based scaling triggered by Kafka queue depth, database load, or API request volume.

AI models could:

**Predict scaling needs based on past traffic and seasonal patterns.**

Automatically adjust configuration parameters (timeouts, rate limits) ahead of anticipated load surges, improving stability and reducing p95 latency and transient error rates (Section ‎‎5.1.2).

**Hybrid Configuration Models with AI-Driven Failover Decisions**

The results in Section ‎5.1.2 show that centralization improves propagation speed and rollback effort but introduces a single control-plane dependency.

Future work could:

Implement AI decision logic to switch between local fallback configurations and centralized updates based on real-time health and availability data.

Use machine learning to determine optimal fallback thresholds, minimizing downtime while preserving operational consistency.

List of Figures

[Figure 1: Strangler Fig pattern 11](#_Toc206080273)

[Figure 2: Microservices vs. Monolith Outline 22](#_Toc206080274)

[Figure 3 : Monolithic vs. SOA vs. Microservices 23](#_Toc206080275)

[Figure 4: Challenges in Microservices 31](#_Toc206080276)

[Figure 5: Architecture Overview 50](#_Toc206080277)

[Figure 6: Bank Application flow case study 54](#_Toc206080278)

[Figure 7: Microservices in Application 55](#_Toc206080279)

[Figure 8: Account Maven Dependency Example 56](#_Toc206080280)

[Figure 9: Customer Class Entity Example 58](#_Toc206080281)

[Figure 10: Customer Class Base Entity Example 58](#_Toc206080282)

[Figure 11: Data Transfer Object Example 59](#_Toc206080283)

[Figure 12: Account Repository Example 60](#_Toc206080284)

[Figure 13: CardsFeignClient Example 61](#_Toc206080285)

[Figure 14:CardsFallBack Example 61](#_Toc206080286)

[Figure 15: Account Controller Class Example 62](#_Toc206080287)

[Figure 17: Global Exception Class Example 64](#_Toc206080288)

[Figure 16: Resource Not Found Exception Class Example 64](#_Toc206080289)

[Figure 18: Application YML file Example 65](#_Toc206080290)

[Figure 19: Entity Model 66](#_Toc206080291)

[Figure 20: API Example for Account micro service 1 68](#_Toc206080292)

[Figure 21: API Example for Account micro service 2 69](#_Toc206080293)

[Figure 22: API Example for Account micro service 3 70](#_Toc206080294)

[Figure 23: API Example for Account micro service 4 71](#_Toc206080295)

[Figure 24: Account Info for Docker Image 72](#_Toc206080296)

[Figure 25 : Docker Image for Microservices 73](#_Toc206080297)

[Figure 26: Config Server GitHub 75](#_Toc206080298)

[Figure 27: Config Application YAML File 76](#_Toc206080299)

[Figure 28: Eureka Discovery 77](#_Toc206080300)

[Figure 29: Eureka Example 78](#_Toc206080301)

[Figure 30: Eureka heartbeat Example 78](#_Toc206080302)

[Figure 31:Card Service Feign Client 79](#_Toc206080303)

[Figure 32: Circuit breaker Example 80](#_Toc206080304)

[Figure 33: Prometheus Configuration 82](#_Toc206080305)

[Figure 34: Account Micro Service Actuator 82](#_Toc206080306)

[Figure 36: Checking CPU Usage Graph for Microservices in Prometheus Dashboard 83](#_Toc206080307)

[Figure 35: Checking CPU Usage for Microservices in Prometheus Dashboard 83](#_Toc206080308)

[Figure 37:All Microservices State shown in Prometheus Dashboard 83](#_Toc206080309)

[Figure 38: Grafana View of CPU Usage of Microservices 84](#_Toc206080310)

[Figure 39: JVM (Micrometer) Dashboard for Account Microservice 84](#_Toc206080311)

[Figure 40:Grafana Alert when Account Micro service down 85](#_Toc206080312)

[Figure 41: Hook deck Response When Account Micro service down 85](#_Toc206080313)

[Figure 42: Open Telemetry Pattern 86](#_Toc206080314)

[Figure 43: Trace ID from Account Microservice 87](#_Toc206080315)

[Figure 44: Trace ID from Card Microservice 87](#_Toc206080316)

[Figure 45: Temp show Path of Request 87](#_Toc206080317)

[Figure 46: Security Config 89](#_Toc206080318)

[Figure 47: Key Clock Filter Chain 90](#_Toc206080319)

[Figure 48: Kecloack Client Roles 91](#_Toc206080320)

[Figure 49: Postman Using Keycloack Role 92](#_Toc206080321)

[Figure 50: Kafka Lytic Details 93](#_Toc206080322)

[Figure 51: Kafka Connection 94](#_Toc206080323)

[Figure 52: Kafka Maven Dependency 94](#_Toc206080324)

[Figure 53: nods for all Micro Service 97](#_Toc206080325)

[Figure 54: Kubernetes config for Account Microservice 97](#_Toc206080326)

[Figure 55:Server-Side Service Discovery YML 99](#_Toc206080327)

[Figure 56: Pass Data for Local Configuration 101](#_Toc206080328)

[Figure 57: Call Hard Code Data 101](#_Toc206080329)

[Figure 58: Environment interface to access configuration 102](#_Toc206080330)

[Figure 59: Methode to Call Configuration Properties 103](#_Toc206080331)

[Figure 60: Configuration Properties Class 103](#_Toc206080332)

[Figure 61: Example for Account Configuration 105](#_Toc206080333)

[Figure 62: Different Environment in Configuration 105](#_Toc206080334)

[Figure 63: Spring Cloud Config Example 108](#_Toc206080335)

[Figure 64: Mange Secrets Inside Account Microservice 108](#_Toc206080336)

[Figure 65: GitHub Repository for Configuration 109](#_Toc206080337)

[Figure 66: Connection between GitHub Repository and Spring Micro Service 109](#_Toc206080338)

[Figure 67: Scalability Test Script Example 112](#_Toc206080339)

[Figure 68: Rabbit Docker Compos Example 120](#_Toc206080340)

[Figure 69: Docker Compose Folder 120](#_Toc206080341)

[Figure 70: Common Configuration Inside Docker Compose 121](#_Toc206080342)

[Figure 71: Nammering Docker Image 122](#_Toc206080343)

[Figure 72 : Reading variable from common Xml 122](#_Toc206080344)

[Figure 73: Reading Version of Maven from Common XML File 123](#_Toc206080345)

[Figure 74: All Maven Version in Common XML File 123](#_Toc206080346)

[Figure 75: Helm Folder for All Application 125](#_Toc206080347)

[Figure 76: Helm Environment Common Variable 125](#_Toc206080348)

[Figure 77: Account Helm Chart Example 126](#_Toc206080349)

List of Tables

[Table 1: Monolithic vs. SOA vs. Microservices Comparison 24](#_Toc206080350)

[Table 2: Critical Decision Points – Monolith vs. SOA vs. Microservices 25](#_Toc206080351)

[Table 3: Centralized vs. Local Configuration 33](#_Toc206080352)

[Table 4: Summary of Literature Review 41](#_Toc206080353)

[Table 5: Merics Mapping Table 49](#_Toc206080354)

[Table 6:Scale-out metrics average of three trials across accounts/loans/cards 113](#_Toc206080355)

[Table 7:Scale-out metrics average of three trials across accounts/loans/cards 113](#_Toc206080356)

[Table 8: Mid-Traffic Config Change Under Load (median; n=3) 116](#_Toc206080357)

[Table 9: Policy Propagation 117](#_Toc206080358)

[Table 10: Summary of Key Results 129](#_Toc206080359)

Appendix

[1] S. Newman, \*Building Microservices: Designing Fine-Grained Systems\*, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2021. ISBN: 978-1492034025. [Online]. Available: https://samnewman.io/books/building\_microservices\_2nd\_edition/ (accessed: Aug. 11, 2025).

[2] J. Lewis and M. Fowler, “Microservices,” MartinFowler.com, 2014 (periodically updated). [Online]. Available: https://martinfowler.com/articles/microservices.html (accessed: Aug. 11, 2025).

[3] C. Richardson, “Pattern: Database per Service,” microservices.io, 2019–2025. [Online]. Available: https://microservices.io/patterns/data/database-per-service.html (accessed: Aug. 11, 2025).

[4] Kubernetes Authors, “Overview,” Kubernetes Documentation, Sep. 11, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/overview/ (accessed: Aug. 11, 2025).

[5] M. Fowler, “Strangler Fig Application,” MartinFowler.com, Aug. 22, 2024 (orig. 2004). [Online]. Available: https://martinfowler.com/bliki/StranglerFigApplication.html (accessed: Aug. 11, 2025).

[6] VMware/Spring, “Spring Cloud Config — Reference Documentation,” Spring.io, 2025. [Online]. Available: https://docs.spring.io/spring-cloud-config/reference/ (accessed: Aug. 11, 2025).

[7] VMware/Spring, “Spring Cloud Bus — Reference Documentation,” Spring.io, 2025. [Online]. Available: https://docs.spring.io/spring-cloud-bus/reference/ (accessed: Aug. 11, 2025).

[8] Kubernetes Authors, “ConfigMaps,” Kubernetes Documentation, Sep. 11, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/configmap/ (accessed: Aug. 11, 2025).

[9] Kubernetes Authors, “Secrets,” Kubernetes Documentation, Nov. 19, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/secret/ (accessed: Aug. 11, 2025).

[10] VMware/Spring, “Spring Cloud Vault — Reference Documentation,” Spring.io, 2025. [Online]. Available: https://docs.spring.io/spring-cloud-vault/reference/ (accessed: Aug. 11, 2025).

[11] Docker Inc., “Docker Overview,” Docker Documentation, 2025. [Online]. Available: https://www.docker.com/ (accessed: Aug. 11, 2025).

[12] VMware/Spring, “Spring Boot Project,” Spring.io, 2025. [Online]. Available: https://spring.io/projects/spring-boot (accessed: Aug. 11, 2025).

[13] Git SCM, “Git Version Control,” Git, 2025. [Online]. Available: https://git-scm.com/ (accessed: Aug. 11, 2025).

[14] VMware/Spring, “Spring Cloud Project,” Spring.io, 2025. [Online]. Available: https://spring.io/projects/spring-cloud (accessed: Aug. 11, 2025).

[15] Red Hat, “Keycloak Identity and Access Management,” Keycloak.org, 2025. [Online]. Available: https://www.keycloak.org/ (accessed: Aug. 11, 2025)