Title

Subtitle

Master Thesis

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

**Master of Science in Engineering**

to the University of Applied Sciences FH Campus Wien

Master Degree Program: Software Design and Engineering

Author:

First name surname

Student Identification Number:

Number

Supervisor:

Degree firstname name

Date:

dd. mm. yyyy

Declaration of authorship:

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

Date: Signature:

Kurzfassung

Diese Arbeit untersucht die Auswirkungen eines zentralisierten Konfigurationsmanagements mittels Spring Cloud Config Server auf Microservices-Architekturen. Die Untersuchung konzentriert sich auf die Skalierbarkeit, Sicherheit und Wartbarkeit im Vergleich zu herkömmlichen lokalen Konfigurationen.

Abstract

This thesis investigates the impact of centralized configuration management using Spring Cloud Config Server on microservices-based architectures. It compares centralized and local configuration approaches with a focus on scalability, security, and maintainability.

List of Abbreviations

GSM Global System for Mobile communication

GPRS General Packet Radio Service

WLAN Wireless Local Area Network

Key Terms

GSM

Mobilfunk

Zugriffsverfahren

Contents

Table of Contents

Type chapter title (level 1)1

Type chapter title (level 2)2

Type chapter title (level 3)3

Type chapter title (level 1)4

Type chapter title (level 2)5

Type chapter title (level 3)6

1. INTRODUCTION

## Background and Motivation

### microservices

A decade ago, applications were deployed as a single unit, with all functionalities deployed together on a single server. This architecture approach, known as Monolithic, has pros such as simpler development and deployment for smaller teams, better performance, and less cross-cutting concerns. However, it also has cons like limited agility, difficulty in adopting new technologies, and a single code base. There are various forms of Monolithic, including Single-Process Monolith, Modular Monolith, and Distributed Monolith. Service-Oriented Architecture (SOA) emerged to address the challenges of large, monolithic applications by organizing software systems as interoperable services. SOA offers benefits like reusability, better maintainability, higher reliability, parallel development, but also has cons like complex management, high investment costs, and extra overload. Microservices, on the other hand, are independently releasable services modelled around a business domain, allowing for more complex systems to be constructed [1].

### My experience

From my experience as a developer at Allianz, I’ve seen how local configuration can lead to inconsistent setups across services, make harder to manage, especially when we deal with big project and slow down deployment. These frustrations motivated me to explore centralized solutions.

Through this thesis, I aim to connect my professional experience with academic research, using technologies like Spring Boot, Kafka, Docker, and Kubernetes to build and test a microservices system that’s scalable, secure, and easier to maintain.

## Problem Statement

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

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

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

Challenges in Configuration Management

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

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

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

## Research Objectives

This thesis sets out to answer the following:

* Scalability: Can centralized configuration improve how fast we deploy services or scale them on Kubernetes?
* Security: Does it reduce risks by storing and managing secrets more safely?
* Maintainability: Is it easier to update configs, roll back changes, and track what’s going on?

The goal is to find out whether using something like Spring Cloud Config Server really helps teams work faster, safer, and more confidently — or if it just adds complexity.

## Research Question

How does centralized configuration management with Spring Cloud Config Server improve scalability, security, and maintainability in microservices compared to local per-service configurations and Monolith?

# LITERATURE REVIEW

## Overview of Microservices

### Definition and Core Principles

Microservices are a method of developing a single application as a suite of small services, each running independently and communicating with lightweight mechanisms. Traditional methods for web applications, such as WAR or EAR files, can be challenging and practically impossible for microservices. To overcome this, organizations should consider using automated deployment machines [2].

A diagram of a computer service

AI-generated content may be incorrect.

### Monolith to Microservices

1. Incremental Refactoring

Incremental refactoring involves gradually transforming a monolithic system into a microservices architecture. This strategy allows you to progressively decompose a monolith into microservices, reducing the risk of business disruption.

With incremental refactoring, you can start by identifying the parts of the monolith that are most suitable for becoming independent microservices. These could be functionalities that are relatively isolated from the rest of the system or those that would benefit most from the advantages offered by microservices, such as scalability and speed of deployment [4].

2. Strangler Pattern

The strangler pattern is a strategy that involves gradually replacing parts of a monolithic application with microservices while the monolith is still running. This pattern is inspired by the strangler fig tree, which grows around other trees and gradually replaces them.

The strangler pattern allows you to gradually introduce microservices into your system without disrupting the functioning of the monolith. This approach helps reduce risk and allows for a smoother transition process [4].

3. Decomposing by Business Capability

This strategy involves breaking down a monolith into microservices based on business functionalities. This strategy aligns the technical components of your system with your business objectives, making it easier to manage and evolve your system in response to business needs.

When decomposing by business capability, it is important to ensure that each microservice is responsible for a single business capability. This helps maintain the independence of microservices and reduces the complexity of the system [4].

4. Anticorruption Layer (ACL)

The anticorruption layer (ACL) is a strategy used to ensure that the transition from monolith to microservices does not corrupt the business logic of your system. The ACL acts as a barrier between the monolith and the microservices, converting data and requests between the two systems.

Using an ACL can help ensure that the transition process does not affect the integrity of your system’s business logic. This is particularly important when dealing with legacy systems, which often lack the necessary documentation to fully understand their business logic [4].

5. Domain-Driven Design (DDD)

Domain-driven design (DDD) is a software development approach that focuses on understanding the business domain and using this understanding to guide the design and development of software. In the context of transitioning from monolith to microservices, DDD can be used to identify the boundaries of microservices and to ensure that the transition process aligns with business goals.

DDD involves creating a model of the business domain and using this model to design the microservices. This approach helps ensure that each microservice corresponds to a specific part of the business domain, which can help ensure that the microservices architecture fully supports your business objectives [4].

### 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 [3].

**Componentization via Services:**Component is a unit of software that is independently replaceable and upgradeable.

**Organized around Business Capabilities:** The microservice approach to division is splitting up into services organized by business capability.

**Products not Projects:** This is Amazon’s notion of “you build, you run it” where a development team takes full responsibility for the software in production.

**Smart endpoints and dumb pipes:**Microservices aim to be as decoupled and as cohesive as possible, so they own their own domain logic and receiving a request, applying logic and producing a response with using Restful APIs.

**Decentralized Governance:**Netflix is a good example of an organization that follows this philosophy. Sharing useful and all tested code as libraries, encourages other developers to solve similar problems in similar ways.

**Decentralized Data Management:** Microservices also decentralize data storage decisions. We can say this approach as a Polyglot Persistence or Polyglot Databases. 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 microservices with separately.

**Design for failure, Resilience:**Microservices design by dealing failures and try to manage failures with 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. Because each service operates independently, a failure in one service should not affect the entire application.

**Scalable:** Each service operates independently, it is possible to scale individual services up or down as needed, without affecting the rest of the application. This allows teams to allocate resources more efficiently and ensure that the application can handle increased traffic or usage.

**Technology Agnostic:** Different services can be written in different programming languages or use different technology stacks. This makes it easier to choose the right tool for the job, rather than being tied to a particular technology stack.

### Benefits of Microservices

 Modularity and Decoupling

In a microservices architecture, each service is designed to handle a specific business function. This modularity helps in isolating services, making them easier to develop, test, and deploy independently [2].

Example:

Consider an e-commerce application. Instead of a monolithic application handling everything from user authentication to order processing, you have separate microservices:

User Service: Manages user accounts and authentication.

Product Service: Manages product listings and details.

Order Service: Handles order processing and payment.

[Scalability](https://www.geeksforgeeks.org/what-is-scalability-and-how-to-achieve-it-learn-system-design/)

Microservices can be scaled independently based on their individual demands. This allows you to allocate resources more efficiently and handle varying loads for different services [2].

Example:

In an online retail application, the Product Service might need to handle high traffic during sales events, while the User Service might not experience the same level of load. You can scale the Product Service separately to handle increased traffic without affecting the User Service.

Fault Isolation

Since services are independent, a failure in one service doesn’t necessarily impact others. This isolation helps maintain overall system stability and reliability [2].

Example:

If the Order Service encounters a problem, it won't affect the Product Service or the User Service. Customers can still browse products and manage their accounts while the order processing issue is addressed.

Technology Diversity

Different services can be developed using different technologies and programming languages that best fit their specific requirements [2].

Example:

The User Service could be developed using Java for its strong support for security features, while the Product Service might use Node.js for its fast, non-blocking I/O, and the Order Service could use Python for its ease of writing and maintaining complex business logic.

Independent Deployment

Each microservice can be deployed independently, allowing for more frequent updates and quicker iteration without affecting other services [2].

Example:

If you need to update the Product Service to add new filtering options, you can deploy this update without touching the User Service or the Order Service. This reduces deployment risk and accelerates release cycles.

Improved Developer Productivity

Teams can work on different services simultaneously without waiting for others to complete their work. This parallel development speeds up overall progress and fosters innovation [2].

Example:

A development team focused on the Order Service can add new payment methods while another team enhances the User Service with additional authentication features, allowing both services to evolve independently

## 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 [4].

Building, running, and governing a microservices-based app demands specialized skills, tools, and advanced monitoring and orchestration capabilities. Companies need to invest in the infrastructure, automation, and DevOps practices to handle the complexity associated with microservices.

### Distributed System Challenges

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

Securing reliable communication, coping with network failures and maintaining data consistency among the distributed services can be hard. Organizations must develop resilient communication patterns, such as circuit breakers, retries, and procedures, to mitigate such challenges [4].

### 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.

Companies require reliable monitoring and observability of threats to gain knowledge of the health and efficiency of individual services and the system. Moreover, service dependencies management, versioning, and backward compatibility increase the operational complexity of microservices [4].

### Data Management

In a microservices architecture, each service has its data store, and there can be a duplication of data, thereby leading to inconsistency and synchronization issues.

Ensuring data consistency across distributed systems involves meticulously designing and implementing data management techniques like event sourcing, eventual consistency, and distributed transactions [4].

Organizations are required to meticulously govern data access and maintain data integrity to prevent data corruption and related problems.

### Service Discovery and Communication

Microservices must discover and talk to each other dynamically. Therefore, a strong service discovery mechanism is needed. Managing service endpoints, load balancing, and failover across distributed services is hard [4].

Organizations should apply service registry and discovery solutions, for instance, Consul or Eureka, to simplify the communication between services. Furthermore, resilient communication patterns, like the service mesh architectures, improve reliability and tolerance to faults.

## Centralized vs. Local Configuration

## Configurations are essential for the functioning of microservices. These encompass: Environment Variables: Such as database URLs, API keys, and credentials. Feature Flags: Allowing for the dynamic enabling or disabling of features. Service Endpoints: The URLs of other services for interaction. Rate Limits: Managing usage to prevent system overload. Poorly managed configurations can result in downtime, erratic behavior, or security risks. Therefore, implementing a strong strategy is imperative. Challenges in Configuration Management Decentralization: With numerous services, the local storage and management of configurations for each can result in inconsistencies. Environment-Specific Configurations: Different configurations are necessary for development, staging, and production environments. Dynamic Updates: Certain configurations, such as feature toggles or throttling limits, may need to be updated at runtime. Security Concerns: The storage of sensitive data like API keys or passwords requires careful handling to prevent breaches. Key Practices for Configuration Management 1. Externalize Configurations Avoid embedding configurations directly in your application code. Utilize configuration files, environment variables, or configuration management tools to externalize settings. This practice ensures consistency across various deployments and environments. Example: In Node.js, leverage environment variables with libraries such as dotenv: require('dotenv').config(); const dbHost = process.env.DB\_HOST; const apiKey = process.env.API\_KEY; 2. Centralized Configuration Management Implement centralized configuration management systems like Consul, etcd, or Spring Cloud Config. These tools maintain configurations in a central repository, enabling services to dynamically retrieve their settings. Advantages: Uniformity across services. Simplified updates without the need for service redeployment. Secure access control. 3. Utilize Environment-Specific Configurations Keep distinct configuration files or entries for each environment (e.g., config.dev.json, config.prod.json). This practice helps prevent the accidental deployment of incorrect settings [6].

Do we truly need to externalize? It appears that we are opening a Pandora's box here. Let us assess the advantages and disadvantages of having my configuration file (e.g., config.json) alongside my Docker image. Advantages & Disadvantages of embedded configuration: Advantages Simple to comprehend Facilitates testing configuration for a specific state in the codebase Local development is quite convenient to initiate Local modifications to the configuration file do not affect other developers. Deployment is straightforward. No need for versioning of configuration state. Disadvantages Secrets are exposed in the Git repository, which is undesirable. However, this can be alleviated by utilizing AWS SSM (Parameter Store). Modifying values is cumbersome: the entire CI pipeline must be triggered (dev, int, stage, prd). This can take up to 1.5 hours :( Requires redeployment of containers. Advantages & Disadvantages of externalizing to a service Let us examine the opposite side.... Advantages: Modifying values is swift. Solutions are available to poll for changes and reflect them without restarting the container. — That is convenient! Shared configuration among services can be defined in one location. Disadvantages: Uncertain about how local development operates. What if I am altering values during development? Do other individuals or services observe this private change? We will require a method to clone the configuration locally. 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 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.

## Security and Observability Considerations

### Security

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

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

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 Transport Layer Security (TLS). For enhanced identity verification and resistance against Man-in-the-Middle (MITM) attacks, mutual TLS (mTLS) is recommended. In mTLS, both services authenticate each other before any data exchange occurs.

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:

* RBAC: Role-based access control for grouping user/service permissions.
* ABAC: Attribute-based access that evaluates conditions at runtime.
* PBAC: Policy-based access based on defined business logic.
* ReBAC: 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 JSON Web Tokens (JWTs). JWTs encode user identity and permissions, allowing services to validate them locally using JWKS (JSON Web Key Sets) 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 under load.

Internally, many systems now use service meshes like Istio or Linkerd 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 HashiCorp Vault, AWS Secrets Manager, or Doppler. Secrets should be rotated regularly and scoped to the smallest set of permissions needed.

Lastly, a secure system must be observable. Distributed tracing tools like OpenTelemetry 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 microservice 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, or Fluentd) and sent to a searchable analytics tool.

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

Application Metrics Pattern

Beyond raw logs, metrics provide high-level numerical insights about system performance — like CPU usage, memory consumption, response times, or error rates. Metrics can be collected from both individual microservices and the infrastructure they run on.

Let’s say one service suddenly uses more CPU than normal. With good metrics collection, you can instantly see whether this is an isolated issue or part of a broader pattern. Tools like Prometheus (especially when integrated with Kubernetes) provide this visibility and can even trigger alerts when anomalies occur [9].

Distributed Tracing Pattern

Distributed tracing tracks a single user request as it travels through multiple microservices. It’s especially useful for diagnosing performance bottlenecks and pinpointing where failures happen in complex systems.

For example, a user hits an error. Logs might tell you which service reported it, but not why. A trace shows you the full path of that request — highlighting which service was slow or caused the error. Tools like Open Telemetry, Jaeger, and Zipkin are commonly used for tracing [9].

Exception Tracking

While logs and metrics capture general behavior, exceptions help identify specific application-level bugs. Exceptions occur when code doesn’t behave as expected — like a failed database call or a null pointer.

Tracking exceptions helps you distinguish between infrastructure failures (like a full disk) and actual bugs in the code. Once you isolate the service and the method triggering the exception, developers can debug and patch the issue more effectively [9].

Health Check APIs

Every microservice should expose a health check endpoint that reports whether the service is operational. These APIs give insights into uptime, latency, error rates, and more.

Health checks are not just useful for humans — orchestration tools like Kubernetes rely on them to decide whether to restart failing services. Without them, services might appear healthy to users — even when they’re not working properly under the hood [9].

Auditing

In regulated industries, auditing is critical. It ensures that the application behaves according to compliance rules — for example, ensuring that sensitive actions are logged or that access to data is properly tracked.

Audit logs can be generated by services just like regular logs, then analyzed to detect unauthorized access, unusual behavior, or policy violations. Observability tools help automate this analysis, making it easier to ensure compliance and respond to incidents [9].

# RESEARCH METHODOLOGY

## Research Design

This thesis adopts a comparative experimental approach to evaluate the impact of configuration management strategies on microservices-based systems. The research is designed around building and analyzing two parallel system implementations that are identical in functionality and architecture but differ in how they manage configuration.

The first system uses a local configuration approach, where each microservice manages its configuration internally via static files bundled with the application. The second system utilizes a centralized configuration approach, where all configuration data is externalized and managed through a dedicated configuration server. This setup enables real-time configuration updates, version control, and secure management of sensitive data.

Both systems will consist of multiple microservices, a discovery service, an API gateway, and a supporting infrastructure stack including databases, monitoring tools, and container orchestration. The configurations being tested will include properties such as service ports, database credentials, logging levels, and feature toggles.

The evaluation will focus on three primary areas:

Maintainability: Assessed by measuring the time, complexity, and risk involved in updating configuration values across services. This includes rollback mechanisms and the ease of auditing changes.

Scalability: Observed by analyzing how configuration changes affect service behavior during scaling operations under simulated load conditions. Key metrics include system responsiveness, configuration propagation time, and consistency.

Security: Evaluated by comparing how each approach handles sensitive data, such as credentials and API keys, and how exposed these values are during storage, transport, and access.

To ensure a fair comparison, both systems will be deployed in controlled environments with identical workloads. Configuration changes will be introduced, and their effects will be monitored using standardized performance metrics, system logs, and user experience indicators.

This research design enables a focused and measurable comparison of local versus centralized configuration management strategies, providing empirical evidence to support architectural decision-making in microservices development.

## Tools and Technologies

Docker

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

Spring Boot

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can just run. You can get started with minimum configurations without the need for an entire Spring configuration setup.

Spring Boot offers the following advantages to its developers −Easy to understand and develop spring applications Increases productivity and reduces the development time [11].

Git

Git is an open-source distributed version control system (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 in 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].

Spring Cloud

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].

Key cloak

Key cloak is an open-source Identity and Access Management (IAM) tool. Being an Identity and Access Management (IAM) 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 Single Sign-On (SSO), identity federation, and strong authentication [14].

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 time-series database (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].

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].

Apache Kafka

Kafka 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].

RabbitMQ

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].

Kubernetes

Kubernetes (K8s) 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].

Helm

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

## Evaluation Metrics

### Scalability

* Deployment Time: How quickly services can be configured and restarted after a change.
* Autoscaling Behavior: How well services respond to increased load when horizontally scaled.
* Config Consistency Across Instances: Whether all replicas apply the same configuration during scale-out events.

### Security

* Secret Exposure Risk: Evaluates how securely sensitive data (e.g., credentials, tokens) is handled and stored.
* Access Control: Use of Role-Based Access Control (RBAC) to limit who can change configurations and access secrets.
* Auditability: Ability to trace who made configuration changes and when, for compliance and forensics.

### Maintainability

* Change Propagation Time: Measures how long it takes for new configurations to be applied across services.
* Rollback Capability: How easily incorrect configurations can be reverted.
* Operational Effort: The number of manual steps or interventions required to apply or manage configuration changes.

# SYSTEM DESIGN

## Architecture Overview

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

### API-Gateway

### Eureka

Eureka: Service Discovery and Self-Preservation in Microservices

In a microservices architecture, services often need to find and communicate with each other dynamically. This is where Eureka, a service discovery tool from Spring Cloud Netflix, plays a central role. Eureka enables client-side service discovery and load balancing, allowing microservices to register themselves at startup and locate each other using logical service names.

How Client-Side Service Discovery Works

Service Registration:

Each microservice, when it starts, registers its IP address and port with the Eureka service registry and sends regular heartbeat signals to indicate it's alive.

Service Discovery:

When another microservice wants to communicate (e.g., the Accounts service calling Loans service), it queries Eureka for the latest instance details of the target service.

Instance Selection:

Eureka returns a list of available service instances. The calling service then uses a load-balancing strategy (e.g., round-robin) to select one and make the call directly.

Client Cache:

To reduce network dependency, services cache instance data locally and periodically refresh it from the registry. This ensures resilience even if Eureka is temporarily unavailable.

Load Balancing in Client-Side Discovery

Client-side load balancing allows microservices to distribute requests across multiple instances efficiently. The load balancing decision is made by the calling service, based on data received from Eureka and cached locally. If no cached instance is available, the service fetches the latest information from Eureka.

Eureka Self-Preservation Mode

To protect against false instance removals caused by temporary network issues or delays:

Eureka enters Self-Preservation Mode when it detects that too many instances are failing to send heartbeats.

While in this mode, it temporarily stops evicting instances, even if they miss heartbeats.

This ensures Eureka doesn't mistakenly remove healthy services due to transient failures

eureka.instance.lease-renewal-interval-in-seconds = 30

eureka.instance.lease-expiration-duration-in-seconds = 90

eureka.server.enable-self-preservation = true

eureka.server.eviction-interval-timer-in-ms = 60000

eureka.server.renewal-percent-threshold = 0.85

Key Benefits of Eureka

Dynamic discovery of services without hardcoded IPs.

Built-in client-side load balancing.

Supports fault tolerance through self-preservation.

Reduces downtime caused by network glitches.

### Gateway

Spring Cloud Gateway serves as the entry point to a microservices system, handling external client requests and routing them to the appropriate internal services. It acts as a gatekeeper, enforcing policies, managing traffic, and enabling security across services.

Key Functions

Centralized Routing: Instead of clients calling individual services directly, all requests are routed through the gateway.

Security Enforcement: The gateway handles authentication and authorization before passing the request to a service.

Traffic Filtering: Requests and responses pass through pre-filters and post-filters, allowing for modification, logging, and validation.

Load-Aware Routing: The gateway supports smart routing strategies (e.g., based on headers, paths, or API versions).

Internal Architecture

When a client makes a request:

Gateway Handler Mapping checks if the request matches a configured route.

If the route matches, the request passes through pre-filters.

The gateway forwards the request to the correct microservice.

The response returns through post-filters, and finally back to the client.

This reactive model, built on Spring WebFlux and Spring Reactor, makes the gateway highly scalable and efficient for edge service handling.

How to Create a Spring Cloud Gateway

Set up a Spring Boot project with the dependencies:

spring-cloud-starter-gateway

spring-cloud-starter-config

spring-cloud-starter-netflix-eureka-client

Add configuration in application.yml to enable discovery and service registration with Eureka:

A screenshot of a computer code

AI-generated content may be incorrect.

### Config Server

#### Micros 1. Accounts Microservice

This service manages all customer account data including personal information, contact details, and banking metadata. It exposes REST endpoints for CRUD operations and plays a central role in user identification across the system. It fetches its configuration from the centralized Config Server and registers itself with Eureka for discovery. It also publishes relevant events (e.g., account updates) to the messaging broker for downstream services to consume.

Loans Microservice

Focused on loan management, this service handles loan applications, repayment schedules, and status tracking. It communicates with the Accounts service to verify customer eligibility and updates loan records. It is also integrated with the configuration server and participates in inter-service communication via REST and Kafka for asynchronous operations.

Cards Microservice

This service is responsible for managing credit and debit card data. It includes functionality for issuing new cards, updating card limits, and retrieving transaction summaries. Like other services, it leverages centralized configuration, service discovery, and event publishing.

Message Processor

This is a specialized microservice designed to consume messages from Kafka topics. It listens for domain events published by other services (e.g., new account creation or loan approvals) and processes them asynchronously. This service enhances decoupling and eventual consistency within the architecture, making it easier to scale and adapt individual services.

Spring Cloud Config Server

This server externalizes configuration for all microservices. Configurations are stored in a centralized Git repository and fetched by services at runtime. This allows dynamic updates, environment-specific settings, and secure handling of sensitive properties (e.g., database credentials), supporting maintainability and consistency across services.

Eureka Discovery Server

Eureka acts as a service registry, allowing microservices to register themselves and discover others. This eliminates the need for hardcoded service endpoints and supports dynamic load balancing. Each microservice registers with Eureka on startup and periodically sends heartbeat signals to confirm availability.

API Gateway (Spring Cloud Gateway)

Although technically not a business service, the API Gateway is a critical edge service. It receives client requests, handles authentication (via Keycloak), and routes them to the appropriate internal service. It also applies filters for logging, rate limiting, and header manipulation.

### database

1. Accounts Database

The Accounts service maintains its own relational database to store customer profile data, contact information, and banking details. Tables are normalized to ensure data integrity and efficient querying. Typical entities include:

* Customer
* Address
* AccountDetails

This separation ensures that changes to account logic or schema do not affect other services.

2. Loans Database

The Loans microservice operates its own dedicated database containing loan application records, repayment schedules, and interest configurations. Key entities include:

* Loan
* PaymentHistory
* LoanType

Storing loan data separately enforces domain boundaries and supports independent scaling of storage and query performance.

3. Cards Database

The Cards service uses its own schema to manage card issuance, limits, and transaction logs. Core tables might include:

* Card
* Transaction
* CardLimit

Having a dedicated database supports high availability and fine-tuned performance tuning for card-specific operations.

4. Message Service Database (optional)

If the Message Processor service requires persistence (e.g., for event logs, retry queues, or idempotency keys), it uses a lightweight store such as PostgreSQL, MongoDB, or Redis. This ensures reliability and traceability in event processing.

* 🔐 Security and Configuration

All databases are:

* Connected via credentials stored in Spring Cloud Config Server.
* Accessed via Spring Data JPA repositories in the services.
* Protected via network policies in Kubernetes and environment-based secrets.

## Implementation Approach

## Configuration Strategy

Bibliography

|  |  |
| --- | --- |
| [1] | J. Schimel, Writing Science: How to Write Papers That Get Cited and Proposals That Get Funded, USA: Oxford University Press, 2012. |
| [2] | U. Alker and U. Weilenmann, *Sprachleitfaden, Gendergerechter Sprachgebrauch an der FH Campus Wien,* FH Campus Wien, 2006. |
| [3] | A. Petz and R. Oberpertinger, *Checkliste Wissenschafltiche Arbeiten,* FH Campus Wien, 2018. |
| [4] | M. Young, The Technical Writer's Handbook, University Science Books, 2002. |

List of Figures

[Abb. 1: Grafiklayout “Mit Text in Zeile“ und “Zentriert“ 1](#_Toc227742115)

[Abb. 2: Zweites Bild 1](#_Toc227742116)

[Abb. 3: Drittes Bild 4](#_Toc227742117)

List of Tables

[Tab. 1: Beispiel für eine Tabelle 3](#_Toc55980479)

[Tab. 2: Zweite Tabelle 3](#_Toc55980480)

Appendix

1-[Sam Newman, Building Microservices](https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/)

2 - <https://www.geeksforgeeks.org/microservices/>

3-https://medium.com/design-microservices-architecture-with-patterns/top-10-characteristics-of-microservices-12b046a59bfc

4-https://komodor.com/learn/monolith-to-microservices-5-strategies-challenges-and-solutions/

5-https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

6-https://article.arunangshudas.com/managing-configuration-in-a-microservice-environment-b2fddba31f28

7-https://ahrefs.com/writing-tools/paragraph-rewriter

8-https://www.osohq.com/learn/microservices-security#:~:text=What%20are%20the%20security%20challenges,between%20services%20are%20also%20vulnerable.

9- <https://www.groundcover.com/microservices-observability>

10- <https://docs.docker.com/get-started/docker-overview/>

11- <https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm>

12-https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F

13- <https://grafana.com/docs/grafana/latest/introduction/>

14- <https://sennovate.com/the-mssp-guide-to-keycloak/>

15- <https://kafka.apache.org/>

16- <https://www.rabbitmq.com/>

17- <https://kubernetes.io/docs/>

18- https://helm.sh/docs/