UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

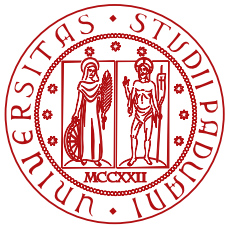
Optimizing Microservices Deployment with Kubernetes: A Performance and Scalability Analysis

Relatore: Prof. Matteo Ceccarello

Laureando/a: Muhammad Ali

ANNO ACCADEMICO: 2024 – 2025

Data di laurea : 22-03-2025



Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Optimizing Microservices Deployment with Kubernetes: A Performance and Scalability Analysis

Master Candidate

Muhammad Ali

Student ID 2071499

Supervisor

Prof. Matteo Ceccarello

University of Padova

Academic Year
2024/2025

Dedication

*I dedicate this thesis to my beloved **parents**, whose unwavering support, sacrifices, and endless prayers have been the foundation of my success. Their love and guidance have shaped me into the person I am today.*

*To my **teachers and mentors**, whose wisdom and encouragement have been instrumental in my academic and professional growth. Their guidance has inspired me to strive for excellence.*

*To my **friends and colleagues**, who have stood by me through challenges, offering motivation and companionship along this journey.*

*Lastly, to **Codebeex SRL** and my supervisors, for providing me with an opportunity to learn, grow, and explore new dimensions in the field of **microservices architecture and federated authentication**. Their support has played a significant role in shaping this study.*

This thesis is a testament to the collective efforts of all those who have been part of my academic and professional journey.

Abstract

The growing adoption of microservices architecture has revolutionized the development and deployment of distributed web services. However, ensuring seamless and secure authentication across multiple independent microservices remains a critical challenge. This thesis explores the design, development, and deployment of a federated authentication mechanism for microservices, specifically tailored for the PartsCoder Middleware system at Codebeex SRL. As organizations increasingly adopt microservices architectures, ensuring seamless and secure authentication across distributed services becomes crucial. This project investigated various federated authentication technologies to enable secure, efficient access control and user identity verification across multiple microservices without compromising system performance.

Through this study, I developed Python-based FastAPI microservices with a JWT (JSON Web Token)-based authentication model to achieve secure user verification across independent service endpoints. The JWT mechanism allows decentralized authentication across the PartsCoder application, minimizing risks associated with single-point failures and enhancing scalability and reliability. Key microservices for survey management in the PartsCoder Middleware were identified, designed, and implemented, including secure data exchange protocols to safeguard user information.

The deployment of these microservices leveraged Docker and Kubernetes, enabling efficient scaling and management of containers across clusters. Kubernetes facilitated robust orchestration and fault tolerance, streamlining the integration of the new authentication framework into the existing infrastructure at Codebeex SRL. This thesis demonstrates how the federated authentication model enhances both security and performance in microservices-based applications, providing insights and best practices for authentication across similar distributed systems in production environments.

Keywords: Microservices Architecture, Federated Authentication, Single Sign-On (SSO), JSON Web Token (JWT), Kubernetes, Docker, FastAPI, Secure Access Control.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Acronyms

API Application Programming Interface - A set of rules and protocols for building and interacting with software applications.

CSV Comma Separated Values - A simple file format used to store tabular data, such as a spreadsheet or database, where each line corresponds to a row and each field is separated by a comma.

DBMS Database Management System - Software for creating, managing, and manipulating databases. It provides a systematic way to store, retrieve, and manage data.

DNS Domain Name System - The hierarchical system that translates human-readable domain names (like www.example.com) into IP addresses that computers use to identify each other on the network.

HTTP Hypertext Transfer Protocol - The protocol used for transmitting hypertext over the internet, allowing for the retrieval of linked resources, such as web pages.

HTTPS Hypertext Transfer Protocol Secure - An extension of HTTP that uses encryption to secure the data transferred between the web server and the browser, enhancing security and privacy.

IDP Identity Provider - A service that creates, maintains, and manages identity information for users and provides authentication services to applications.

JWT JSON Web Token - A compact, URL-safe means of representing claims to be transferred between two parties, commonly used for authentication and information exchange.

K8s Kubernetes - An open-source platform for automating the deployment, scaling, and management of containerized applications, providing container orchestration.

OIDC OpenID Connect - An identity layer on top of the OAuth 2.0 protocol that allows clients to verify the identity of end-users based on the authentication performed by an authorization server.

RBAC Role-Based Access Control - A method for regulating access to computer or network resources based on the roles of individual users within an organization.

REST Representational State Transfer - An architectural style for designing networked applications, relying on a stateless, client-server communication protocol, typically HTTP.

SaaS Software as a Service - A software distribution model in which applications are hosted in the cloud and made available to users over the internet, eliminating the need for local installation.

SPA Single-Page Application - A web application that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server, leading to a more fluid user experience.

SSO Single Sign-On - An authentication process that allows a user to access multiple applications with one set of login credentials, simplifying user management and improving user experience.

TLS Transport Layer Security - A cryptographic protocol designed to provide secure communication over a computer network, replacing the earlier SSL (Secure Sockets Layer) protocol.

UI User Interface - The space where interactions between humans and machines occur, encompassing all elements that allow users to interact with a device or application.

URL Uniform Resource Locator - The address used to access resources on the internet, specifying the location of a resource on a computer network and the protocol used to retrieve it.

VM Virtual Machine - An emulation of a computer system that provides the functionality of a physical computer, allowing multiple operating systems to run on a single physical machine.

CI/CD Continuous Integration/Continuous Deployment - A set of practices that enable development teams to deliver code changes more frequently and reliably by automating the integration and deployment processes.

SQL Structured Query Language - A standard programming language used for managing and manipulating relational databases, enabling tasks such as querying, updating, and deleting data.

JSON JavaScript Object Notation - A lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate, commonly used in APIs.

IP Internet Protocol - The principal communication protocol used for relaying datagrams across network boundaries, ensuring data is sent from the source to the destination.

1

Introduction

In recent years, the software development landscape has been marked by a significant shift from monolithic applications to modular, scalable microservices architectures. This transformation is driven by the increasing need for agility, resilience, and maintainability in applications, especially as businesses adapt to rapid technological advancements and evolving market demands. Traditional monolithic architectures, while effective in certain contexts, often present challenges in scaling, fault tolerance, and adaptability to new technologies. As a response, microservices architecture where applications are decomposed into smaller, autonomous services that can be developed, deployed, and managed independentlyhas emerged as a leading solution for organizations aiming to achieve greater flexibility and speed in software development cycles.

Microservices provide distinct advantages, such as the ability to scale individual services independently, enhance fault isolation, and adopt diverse technology stacks across various parts of an application. These benefits align with the principles of Continuous Integration and Continuous Deployment (CI/CD), which encourage frequent releases and faster innovation. However, the adoption of microservices architecture also introduces new challenges. Issues such as complex inter-service communication, dependency management, and heightened resource demands necessitate efficient orchestration and management of services. Consequently, organizations increasingly turn to Kubernetes, an open-source container orchestration platform, to address these challenges and streamline the deployment and management of containerized microservices across diverse infrastructure environments.

Originally developed by Google and now managed by the Cloud Native Computing Foundation (CNCF), Kubernetes offers robust tools to automate deployment, scaling, and operations of application containers. It provides a flexible framework for managing distributed applications, enhancing service resilience, and maintaining optimal resource utilization in production environments. Kubernetes' features, such as self-healing, load balancing, service discovery, and automated scaling, make it an ideal platform for deploying microservices at scale.

1.1 Thesis Structure

This thesis is structured as follows:

1. **Chapter 1- Introduction:** Provides an introduction to the study, outlining motivation, objectives, research questions, and contributions.
2. **Chapter 2 - Background:** Discusses foundational concepts, including microservices architecture, Kubernetes, federated authentication, and security mechanisms.
3. **Chapter 3- Methodology**: Details the implementation of the authentication mechanism, microservices architecture, and deployment strategy.
4. **Chapter 4- Results**: Presents the evaluation of system performance, scalability, and security.
5. **Chapter 5- Conclusion and Future Work**: Summarizes findings, discusses limitations, and provides directions for future work.

1.2 Motivation

The adoption of microservices architecture offers substantial benefits, including improved modularity, increased scalability, and enhanced fault tolerance. These advantages stem from the ability to decompose applications into smaller, independent services that can be developed, deployed, and scaled independently. However, these benefits come with their own set of challenges, particularly in the areas of deployment, resource allocation, interservice communication, authentication, and authorization.

In a Kubernetes environment, managing these complexities requires careful orchestration and coordination. Traditional authentication mechanisms, such as

session-based authentication, often struggle to scale across multiple services due to the decentralized nature of microservices. This can create issues with session persistence and security as users interact with different services. Federated authentication addresses this challenge by enabling Single Sign-On (SSO) across independent microservices. This method enhances both the user experience and security by allowing users to authenticate once and seamlessly interact with multiple services without needing to log in repeatedly.

This study explores federated authentication techniques within a microservices ecosystem and examines the impact of Kubernetes on the performance and scalability of microservices. It focuses on how these technologies were integrated into a real-world application developed at Codebeex SRL.

1.2.1 PartsCoder® Middleware at Codebeex SRL

Codebeex SRL specializes in developing distributed web applications and middleware solutions for enterprise systems. During my internship at Codebeex, I contributed to the development of **PartsCoder® Middleware**, a scalable and efficient system designed to manage spare parts for vehicles.

PartsCoder® Middleware serves as an intermediary layer, integrating and managing a wide range of vehicle spare part data, including part specifications, pricing, and availability. The system is built on a microservices architecture, where each service handles specific aspects of spare parts management, such as part identification, inventory tracking, and order processing. This modular design allows for better scalability and flexibility, enabling the system to handle growing data volumes and transaction loads as the business expands.

The middleware consolidates data from multiple sources, providing a centralized platform for vehicle spare parts information. One of its key features is its integration with **Odoo**, an open-source enterprise resource planning (ERP) system. This integration allows PartsCoder® Middleware to publish product information across different e-commerce platforms and ensures real-time synchronization between the database and online storefronts.

1.2.2 Business Operations and Customer Experience

PartsCoder® Middleware enables businesses to streamline their operations by providing a comprehensive view of vehicle parts inventory, customer details,

and order status. The integration with Odoo ensures that spare parts information is consistently updated across various e-commerce websites, reducing the risk of errors and improving the accuracy of product listings. This helps businesses maintain a seamless customer experience by ensuring that product availability and pricing are up-to-date.

The middleware architecture ensures secure and efficient data flow between all components, supporting seamless interactions with customers and business partners. It allows businesses to handle transactions, manage inventory, and provide customers with accurate, timely information on the availability and specifications of spare parts.

1.2.3 Authentication Challenges in PartsCoder® Middleware

As the PartsCoder® Middleware involves interactions across multiple microservices, it required an efficient authentication mechanism to ensure secure interactions between these services while maintaining a seamless user experience. Given the distributed nature of the system, traditional session-based authentication mechanisms were not suitable for ensuring scalable and secure communication. This necessitated the implementation of federated authentication to provide a robust, unified authentication solution across all microservices.

1.3 Objectives of the Study

This study aims to explore and implement federated authentication in a multi-microservice environment, focusing on scalable deployment using Kubernetes. The key objectives are:

- **Develop Python FastAPI-based microservices** that support federated authentication.
- **Implement JWT-based authentication** for secure token management across independent services.
- **Deploy microservices on Kubernetes** to ensure scalability and reliability.
- **Analyze federated authentication mechanisms** and their impact on security and performance.
- **Evaluate the effectiveness of federated authentication** by assessing its usability, security, and scalability.

- **To evaluate the impact of Kubernetes** on the scalability and performance of microservices architectures.

1.3.1 Study Scope

The scope of this study is centered on the authentication mechanisms required for securing distributed services in a microservices-based ecosystem. Specifically, it investigates:

Key Areas of Focus

- **Microservices Architecture:** Understanding its benefits, challenges, and authentication requirements.
- **Federated Authentication:** Exploring techniques such as JSON Web Tokens (JSON Web Token - A compact, URL-safe means of representing claims to be transferred between two parties, commonly used for authentication and information exchange. (JWT)), OAuth 2.0, and OpenID Connect.
- **Kubernetes Deployment:** Implementing and managing authentication mechanisms in a cloud-native environment.
- **PartsCoder® Middleware:** Analyzing authentication integration within Codebeex's middleware.

1.4 Background

Microservices architecture has become a popular choice for designing complex, large-scale applications due to its emphasis on loose coupling, high cohesion, and modularization. By allowing individual services to operate independently, microservices enable faster development cycles, greater autonomy for development teams, and enhanced collaboration through the use of diverse technology stacks. This autonomy, however, comes with challenges such as maintaining data consistency across distributed services, ensuring seamless communication between services, and implementing consistent security policies.

Kubernetes addresses many of these operational complexities by abstracting infrastructure management and automating deployment, scaling, and maintenance tasks. Kubernetes also provides built-in mechanisms for managing communication between services, balancing traffic loads, and monitoring the health of applications. These capabilities make Kubernetes a powerful tool for managing microservices, helping organizations achieve scalability, reliability, and agility in a cloud-native environment.

Microservices in Modern Applications

Microservices have gained traction in modern applications, particularly in organizations that prioritize agility, flexibility, and resilience. These microservices are often deployed inside containers, which are lightweight, standalone, and executable software packages that include everything needed to run a servicesuch as the code, runtime, libraries, and dependencies. Containers provide a consistent environment across different computing environments, making them ideal for microservice-based architectures. Architecting applications as collections of independent, loosely coupled services provides the following benefits:

- Rapid deployment of new features and updates with minimal impact on overall system stability.
- Fault isolation, ensuring that issues in one service do not propagate across the entire application.
- Enhanced team productivity and collaboration by allowing teams to work independently on distinct services using technologies suited to the needs of each service.

Despite these advantages, managing microservices poses unique challenges, particularly in areas such as consistent security implementation, dependency management, and data synchronization. Addressing these issues requires advanced orchestration tools such as Kubernetes, which streamline microservice deployment and management.

The Role of Kubernetes

Kubernetes simplifies the operational complexities of managing microservices by offering features that automate tasks such as deployment, scaling, and monitoring. Key Kubernetes functionalities include:

- **Automated Deployment and Scaling:** Kubernetes enables dynamic scaling of services based on demand, ensuring optimal resource utilization and high availability.
- **Load Balancing:** Kubernetes manages the distribution of incoming traffic to maintain optimal service performance and availability.
- **Self-Healing:** Kubernetes monitors container health, automatically restarting failed containers and replacing unhealthy instances to ensure application reliability.
- **Service Discovery and Communication:** Kubernetes supports service discovery, enabling seamless communication between microservices through dynamic endpoint management and internal DNS resolution.

1.5 Contributions of the Study

This study, conducted during an internship at Codebeex SRL, directly contributes to the companys ongoing efforts to improve the scalability, security and maintainability of its PartsCoder® middleware. By implementing a microservices-based authentication system with federated identity management, the study addresses key challenges in distributed web service environments. The primary contributions of this study to Codebeex are as follows.

- **Secure Federated Authentication for Microservices:** The study introduces an authentication architecture leveraging JWT-based authentication. This enables Single Sign-On (SSO) across microservices, simplifying user management and improving security.
- **Fine-Grained Authorization with RBAC:** By implementing Role-Based Access Control (Role-Based Access Control - A method for regulating access to computer or network resources based on the roles of individual users within an organization. (RBAC)), the study ensures that access to microservices is granted based on predefined roles and permissions. This enhances security and access control in the PartsCoder® middleware.
- **Scalable and Efficient Microservices Deployment:** Deploying microservices on Kubernetes - An open-source platform for automating the deployment, scaling, and management of containerized applications, providing container orchestration. (K8s) allows Codebeex to scale its services dynamically, ensuring optimal resource utilization and high availability. The study provides a structured deployment strategy using Kubernetes, making the system more resilient to failures.

- **Standardized Microservices Development with FastAPI:** By using `FastAPI` to develop microservices, the study demonstrates an efficient, high-performance framework for building RESTful APIs. This establishes a best-practice approach for future microservices development at Codebeex.
- **Centralized Service Discovery with `mw-dashboard`:** A dedicated microservice, `mw-dashboard`, has been developed to list and manage all available microservices in the PartsCoder® ecosystem. This simplifies service discovery and improves maintainability.
- **Containerized Deployment with Docker and Kubernetes:** The study establishes a complete DevOps pipeline for containerized microservices deployment, ensuring seamless integration and automated scaling. Codebeex can now efficiently manage and deploy its services across multiple environments.
- **Performance and Security Evaluation:** The study includes an in-depth analysis of authentication performance, latency, and security vulnerabilities. The insights gained from this evaluation provide recommendations for further improvements in codebeex microservice security and efficiency.

1.6 Methodology Overview

To achieve the research objectives, the following methodology is followed:

1. **Microservices Development:** Python-based FastAPI microservices are developed to support independent authentication and business logic.
2. **Authentication Implementation:** JWT-based authentication is integrated, ensuring stateless and secure user verification.
3. **Federated Authentication Setup:** A Single Sign-On (SSO) mechanism is configured to allow seamless authentication across all microservices.
4. **Deployment on Kubernetes:** All services, including authentication, are containerized using Docker and orchestrated using Kubernetes.
5. **Performance Evaluation:** Authentication latency, system scalability, and token security are analyzed to validate the effectiveness of the approach.

"Microservices enable developers to build scalable, flexible systems by breaking down complex applications into loosely coupled services, each responsible for a specific business function." — Industry Expert

This study aims to contribute to the domain of scalable authentication in microservices-based environments, demonstrating how federated authentication mechanisms can enhance security, scalability, and user experience. By leveraging Kubernetes for deployment, this work provides a practical and efficient approach for managing authentication in distributed architectures.

2

Background

In recent years, software development has seen a rapid evolution in methodologies and architectures, moving towards systems that prioritize scalability, fault tolerance, and agility. This shift is largely enabled by the adoption of microservices architecture, which allows for the decomposition of complex applications into smaller, independent services. Each service encapsulates a distinct business capability and can be independently developed, deployed, and scaled. This chapter provides essential background information on microservices architecture, Kubernetes as a container orchestration tool, and an overview of Codebeex, the organization where I conducted my internship. These elements are foundational to understanding the scope and objectives of this study.

2.1 Codebeex: Company Overview

Codebeex SRL is a technology-focused company specializing in software development, cloud solutions, and IT consulting services. The company emphasizes creating scalable, resilient, and high-performing applications that meet the specific needs of its clients. With a strong emphasis on innovation, Codebeex prioritizes adopting modern technologies and architectures, including microservices, containerization, and cloud-native solutions, to deliver reliable and cutting-edge products [15].

2.1.1 Core Services and Expertise

At Codebeex, the primary services span across various aspects of software development and IT consulting, with a focus on the following:

- **Custom Software Development:** Codebeex designs and implements software solutions that are tailored to client needs, leveraging the latest frameworks and technologies to ensure performance and scalability [15].
- **Cloud Solutions:** The company offers comprehensive cloud consulting and implementation services, aiding clients in transitioning to and optimizing cloud-native infrastructures [15].
- **Microservices Development:** Codebeex has a proven track record in developing applications using microservices architecture, allowing clients to benefit from systems that are modular, maintainable, and highly scalable [15].
- **DevOps Practices:** The company integrates DevOps principles to streamline software delivery pipelines, incorporating CI/CD pipelines, automated testing, and monitoring to enhance reliability and accelerate deployment [15].

Codebeex's commitment to innovation, efficiency, and high standards in software development aligns with industry best practices and supports the successful adoption of microservices and Kubernetes.

2.1.2 Internship Projects and Insights

During my internship at Codebeex, I contributed to multiple projects that involved the development of microservices architecture and Kubernetes orchestration. One key project focused on building a federated authentication system for multi-service applications. This involved configuring microservices to communicate securely while ensuring seamless user access across services, implemented using JWT-based authentication. The project emphasized security, user experience, and modularity, allowing users to authenticate once and access various services without repeated login requirements [1].

Working within Codebeex's collaborative environment, I had the opportunity to gain hands-on experience with Kubernetes and Docker, from container management to deploying microservices across distributed environments. Working on DevOps technologies, I learned the intricacies of automated deployment, service orchestration, and the critical role of Kubernetes in enhancing application robustness and scalability.

2.2 Microservices Architecture

Microservices architecture is a modular approach to software design where applications are constructed as a suite of small, self-contained services. Each service addresses a specific business requirement and operates independently, communicating with other services through lightweight mechanisms such as HTTP or messaging protocols. This architectural style contrasts with traditional monolithic structures, where all application components are tightly integrated, resulting in several notable advantages:

- **Scalability:** Individual microservices can be scaled according to their resource demands. For example, a high-traffic service can be scaled without affecting other parts of the application, leading to efficient resource utilization [10].
- **Resilience:** Faults within one service are less likely to cascade and impact other services, as each service operates independently, thereby enhancing the fault tolerance of the entire application [7].
- **Agility:** Since microservices are loosely coupled, development teams can work on different services in parallel. This independence supports shorter development cycles, quicker feature rollouts, and a streamlined continuous deployment process [9].

2.2.1 Microservices in the Context of Codebeex

At Codebeex, microservices are used to build the PartsCoder® middleware, which consists of the following microservices:

- `mw-dashboard`: Lists all available services in the system.
- `mw-auth`: Handles authentication and JWT token generation.
- `mw-realm-info`: Manages authentication tokens and Odoo server details.
- `mw-part-mapper`: Stores and processes vehicle part data.
- `mw-photo-shark`: Provides image guidelines for vehicle part documentation.
- `mw-survey`: Manages survey-related functionalities.

Despite its benefits, microservices architecture brings new challenges related to service orchestration, inter-service communication, and data consistency across distributed systems [12]. Efficient orchestration is essential to managing service interactions, scaling, and optimizing resources, which leads to the need for a robust platform like Kubernetes.

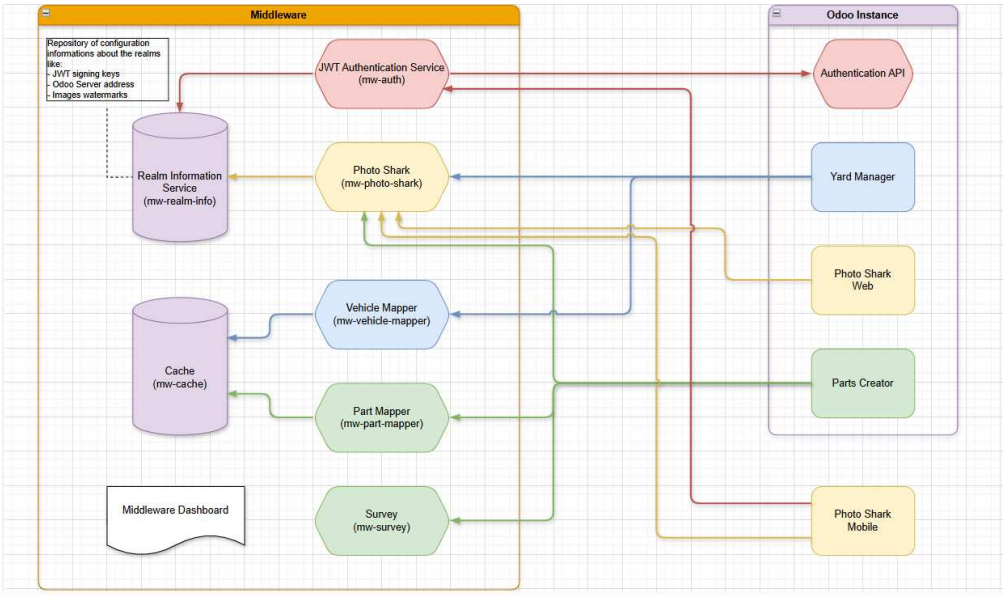


Figure 2.1: PartsCoder® Middleware Architecture at Codebeex SRL

2.3 Kubernetes: A Container Orchestration Platform

Kubernetes, an open-source platform developed by Google and maintained by the Cloud Native Computing Foundation (CNCF), is widely adopted for orchestrating and managing containerized applications. It provides a comprehensive set of features that automate the deployment, scaling, and operation of applications in containerized environments. Its use is particularly beneficial in managing the complexities of microservices-based architectures [4]. Through YAML or JSON configuration files, Kubernetes allows developers to specify the desired state of their applications, enabling reproducible and reliable deployments [13].

2.3.1 Core Components of Kubernetes

- **Pods:** The smallest deployable units that encapsulate containers.
- **Nodes:** Physical or virtual machines that run pods.
- **Cluster:** A set of nodes managed by Kubernetes.

- **Services:** Abstract network endpoints that allow microservices to communicate.
- **Ingress:** Manages external access to services within the cluster.
- **ConfigMaps and Secrets:** Store configuration data and sensitive information securely [3].

2.3.2 Deployment of Microservices on Kubernetes

At Codebeex, all microservices are containerized using Docker and deployed on Kubernetes. This ensures:

- **Automated Scalability:** Kubernetes can dynamically scale services up or down based on real-time demand, ensuring that application resources are optimally utilized [4].
- **Self-Healing Capabilities:** Kubernetes monitors container health, automatically replacing or restarting failed containers, ensuring the robustness and resilience of applications [4].
- **Service Discovery and Load Balancing:** Built-in mechanisms allow services to locate each other efficiently, facilitating seamless inter-service communication [13]. Load balancing distributes traffic evenly, enhancing performance and maintaining service availability.

Kubernetes abstracts away much of the infrastructure management burden, allowing developers to focus on application functionality rather than operational logistics. Its ability to efficiently manage resources and provide high availability makes it an ideal choice for deploying and managing microservices [2].

2.4 Federated Authentication

2.4.1 Definition and Importance

Federated authentication is an identity management approach that enables users to authenticate once and access multiple services without requiring separate credentials. It is based on industry standards such as OAuth 2.0 and OpenID Connect [6].

Advantages of Federated Authentication

- **Single Sign-On (SSO):** Users log in once and gain access to all connected services.
- **Improved Security:** Reduces the risk of credential reuse and phishing attacks.
- **User Convenience:** Eliminates the need for multiple passwords.

2.4.2 Authentication Mechanisms Implemented at Codebeex

In the PartsCoder® middleware, federated authentication is implemented using:

- **JWT-based Authentication:** Ensures stateless authentication between microservices [8].

2.5 Role-Based Access Control (RBAC)

2.5.1 Overview

Role-Based Access Control (RBAC) is a security model that restricts access to resources based on predefined roles assigned to users [5].

Key Components of RBAC

- **Users:** Individuals or entities requesting access to a system.
- **Roles:** Defined sets of permissions (e.g., admin, user, manager).
- **Permissions:** Allowed actions (e.g., read, write, delete).
- **Policies:** Rules that enforce access control.

2.5.2 RBAC Implementation in PartsCoder®

RBAC is used in Codebeexs authentication system to manage access control across microservices. The `mw-auth` microservice enforces RBAC policies using JWT claims to ensure users have the necessary permissions.

This background chapter establishes the foundational concepts necessary for understanding the study's focus on microservices architecture and Kubernetes. The microservices approach enables scalable and resilient software applications, while Kubernetes simplifies the orchestration and management of these distributed services. Codebeex, as an organization dedicated to modern software practices, provided a practical setting for applying these technologies, offering insights into real-world challenges and solutions.

The following chapters will further explore the specifics of deploying microservices on Kubernetes, with an emphasis on optimization techniques, best practices, and lessons learned from my experiences at Codebeex.

3

Methodology

This chapter elaborates on the methodology employed during my internship at Codebeex SRL, focusing on the design and implementation of a microservices architecture integrated with federated authentication technologies. The approach was grounded in agile software development principles and combined theoretical frameworks with practical applications. The process involved several steps, including system design, development, integration, and deployment using various technologies such as Python, FastAPI, JWT-based authentication, Kubernetes, and Docker. This chapter outlines the methodology followed to achieve the goals set for this study. The focus was on creating secure, scalable, and efficient solutions to handle user authentication across multiple services within a distributed environment, specifically the PartsCoder® Middleware at Codebeex SRL.

3.1 Sytem Design

The PartsCoder® survey application follows a microservices architecture, where individual services are responsible for specific functionalities. The system architecture is designed to include multiple independent services interacting with each other through well-defined APIs and streamline the authentication process. A detailed design approach is adopted to ensure scalability, flexibility, and efficient communication between services.

The study was designed with the goal of improving secure access manage-

ment in a multi-microservice architecture, leveraging federated authentication to simplify and streamline the authentication process. The study was structured into multiple phases, each focusing on a specific aspect of the project:

- **Problem Analysis**: Identifying the need for federated authentication in multi-microservice environments to ensure secure and seamless user authentication across different services.
- **System Design and Architecture**: Architecting a microservices-based middleware platform for PartsCoder®'s application, incorporating JWT-based authentication and various inter-service communications.
- **Development and Implementation**: Developing microservices using Python and FastAPI, implementing JWT authentication, and containerizing the services using Docker.
- **Deployment and Testing**: Deploying the services to a Kubernetes cluster, ensuring scalability, security, and proper communication among services.
- **Evaluation**: Assessing the security, scalability, and efficiency of the deployed system, including a performance analysis of service interactions.

3.2 System Architecture

The microservices architecture followed a **loosely coupled** approach, where each service handled a specific task and communicated with others via APIs. The middleware was designed to serve the PartsCoder® survey application by providing a set of RESTful APIs.

Each microservice is designed to be self-contained and independently deployable, ensuring flexibility and scalability. This approach is essential in an environment where numerous small services handle specific functions, allowing for easier maintenance and scalability [11]. The system is designed using a Python-based framework, FastAPI, to ensure high performance and asynchronous operations, which are crucial for the microservices environment [14].

3.2.1 Federated Authentication

Federated authentication was implemented to enable a user to authenticate once and access multiple services within the ecosystem.**JSON Web Token (JWT)** is a compact, URL-safe token format used for securely transmitting information between parties as a JSON object. It is digitally signed, ensuring integrity

and authenticity. JWT was used for secure, stateless authentication, allowing different microservices to validate user tokens without having to store session data centrally.

3.2.2 Architecture Diagram

The overall architecture of the middleware is represented below:

Service Name	Functionality
`mw-dashboard`	Lists all available services and their specifications.
`mw-auth`	Handles user authentication and issues JWT tokens.
`mw-realm-info`	Stores user-specific token and Odoo server details.
`mw-part-mapper`	Manages vehicle part information.
`mw-photo-shark`	Provides photo guidelines for parts and vehicles.
`mw-survey`	Manages surveys, allowing users to create and edit them.

Table 3.1: Microservices Overview

3.2.3 Technology Stack

Component	Technology Used
Programming Language	Python
Framework	FastAPI
Authentication	JWT (for secure authentication)
Containerization	Docker
Orchestration	Kubernetes
Database	PostgreSQL

Table 3.2: Key System Components

3.3 Implementation Process

The implementation phase consisted of several key steps, as detailed below:

3.3.1 Microservices Development

The development process was structured into iterative cycles, where each cycle involved the completion of specific functionalities followed by review and

feedback. This iterative process facilitated continuous improvement and allowed for adjustments based on real-time feedback, ensuring alignment with user expectations. Each iteration generally followed these steps:

Agile Methodology Practices

The following Agile practices were adopted throughout the project:

1. **Sprint Planning:** Involved defining the goals and deliverables for the sprint, selecting user stories from the backlog, and assigning tasks to team members.
2. **Daily Stand-ups:** Short meetings were held daily to discuss progress, roadblocks, and plans for the day. This promoted transparency and team collaboration.
3. **Sprint Review:** At the end of each sprint, the completed work was demonstrated to stakeholders for feedback. This ensured that the team was aligned with project goals and could adjust as necessary.
4. **Retrospective:** The team discussed what went well, what could be improved, and actionable items for the next sprint to enhance efficiency and communication.

Microservices were developed using **Python** with the **FastAPI** framework. FastAPI was chosen for its efficiency in creating high-performance APIs and its support for asynchronous programming, which is essential for efficiently managing multiple requests. The following microservices were implemented:

- **mw_auth:** This service is responsible for user authentication and token generation. It verifies user credentials and issues JSON Web Tokens (JWT) for subsequent requests.
- **mw_dashboard:** This service provides an overview of available services and their status, acting as a central point for service management and monitoring.
- **mw_realm_info:** It manages and stores user-specific token information and associated user details, ensuring secure and easy access to the application.
- **mw_part_mapper:** This service contains comprehensive details related to vehicle parts, allowing users to access and manage this information efficiently.
- **mw_photo_shark:** This microservice offers guidelines and specifications for vehicle photographs, facilitating a structured approach to managing image assets.
- **mw_survey:** It manages the creation, editing, and distribution of surveys, providing users with tools to collect and analyze feedback effectively.

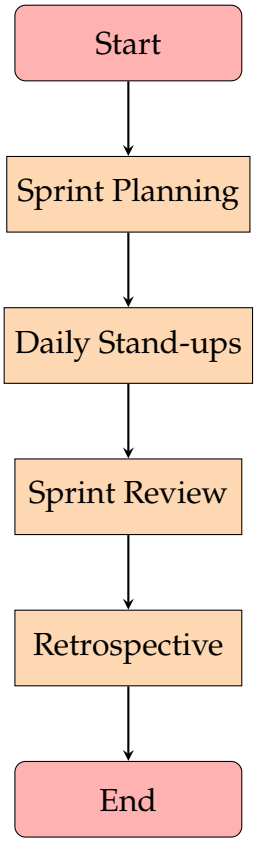


Figure 3.1: Agile Methodology Practices Flow

The development of each microservice followed the principles of single responsibility and loose coupling, ensuring that each service could be developed, deployed, and scaled independently. This modular design also facilitates easier debugging and maintenance.

3.3.2 JWT Authentication

Each microservice, upon receiving a request, verifies the JWT token sent by the client. If the token is valid, the service processes the request; otherwise, it responds with an HTTP 401 Unauthorized status.

```
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from jwt import JWTError, decode

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

def verify_token(token: str = Depends(oauth2_scheme)):
```

```
    try:
        payload = decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except JWTError:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Invalid token",
        )
```

Code 3.1: JWT Token Verification

The token is sent by the client in the `Authorization` header, and each microservice verifies the token before allowing any action. This ensures the integrity of the authentication process.

3.4 Containerization with Docker

Each microservice was containerized using Docker, which provides the flexibility to deploy and manage services in isolated environments. The Docker configuration for the `mw-auth` microservice is outlined below:

```
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt /app/

RUN pip install -r requirements.txt

COPY . /app/

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Code 3.2: Dockerfile for mw-auth Service

This configuration builds a Docker image for the `mw-auth` service, installing the necessary dependencies and running the FastAPI application using `uvicorn` as the ASGI server.

3.5 Deployment Using Kubernetes

The deployed microservices were orchestrated with Kubernetes, which provides scalability, load balancing, and automated deployment management.

3.5.1 Kubernetes Setup

The Kubernetes setup consisted of several components:

- **Pods**: Each microservice was deployed as an individual pod.
- **Services**: A service was defined for each microservice to expose its endpoints and enable communication between them.
- **Ingress**: A Kubernetes ingress resource was set up to manage external access to the services.

```
apiVersion: v1
kind: Service
metadata:
  name: mw-auth-service
spec:
  selector:
    app: mw-auth
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8000
```

Code 3.3: Kubernetes Service for mw-auth

This configuration ensures that the `mw-auth` service can be accessed internally by other services via a Kubernetes service.

3.5.2 Auto-scaling

The Kubernetes cluster was configured to automatically scale the pods based on resource utilization. This ensured that the system could handle varying loads effectively.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: mw-auth-hpa
```

```
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: mw-auth
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

Code 3.4: Kubernetes Horizontal Pod Autoscaler for mw-auth

The next chapter discusses the results of the system's deployment, including performance metrics and security evaluations.

3.5.3 Containerization and Deployment with Kubernetes

To facilitate deployment and scaling, all microservices were containerized using **Docker**. The use of Docker allows for consistent environments across development, testing, and production stages. The containerization process involved the following steps:

1. **Creating Docker Images:** Each microservice was packaged into a Docker image with all necessary dependencies defined in a Dockerfile. This file specifies how the image should be built, including the base image, the installation of libraries, and the command to run the service.
2. **Kubernetes Configuration:** Deployment configurations were created using YAML files, specifying resource requirements, replica counts, and service definitions. This includes defining **Deployments** to manage the desired state of applications and **Services** for enabling communication between pods.
3. **Service Discovery:** Kubernetes facilitates service discovery, allowing microservices to communicate with each other seamlessly through internal DNS. Each service can be accessed by its name, eliminating the need for hard-coded IP addresses.
4. **Load Balancing:** Kubernetes provides built-in load balancing capabilities, ensuring traffic is evenly distributed across service instances. This prevents any single instance from becoming a bottleneck, improving the overall performance of the application.

5. **Persistent Storage:** For services that require persistent data storage, such as the **mw_realm_info** service, Kubernetes volumes were used to ensure data persistence beyond the lifecycle of individual pods.

The following YAML snippet provides an example of a Kubernetes deployment configuration for the **mw_auth** microservice:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mw-auth
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mw-auth
  template:
    metadata:
      labels:
        app: mw-auth
    spec:
      containers:
      - name: mw-auth
        image: codebeex/mw-auth:latest
        ports:
        - containerPort: 80
        env:
        - name: SECRET_KEY
          value: "mysecretkey"
```

Code 3.5: Kubernetes Deployment Example

3.5.4 Testing Strategy

Testing was an integral part of the development process, ensuring that each microservice functioned correctly and met performance expectations. The testing strategy included:

1. Unit Testing: Individual components of each microservice were tested to verify their functionality using testing frameworks. Unit tests focused on specific functions and methods to ensure they produced expected results.

2. Integration Testing Integration tests were conducted to evaluate the interaction between different microservices. These tests confirmed that services

could communicate correctly and that data was shared seamlessly across the architecture.

3. End-to-End Testing: End-to-end tests simulated real user scenarios to validate the entire application flow, from user authentication to data retrieval across multiple services.

4. Load Testing: Load testing was performed to assess the system's performance under various traffic conditions.

The comprehensive testing strategy ensured that the final deployment was robust, efficient, and capable of handling real-world usage scenarios.

3.6 Monitoring and Logging

To ensure the continuous performance, reliability, and stability of the microservices architecture, a comprehensive monitoring and logging system was put in place. This system allowed for proactive identification of issues, efficient troubleshooting, and the ability to track the system's health and operational metrics in real-time. The implementation of effective monitoring and logging mechanisms was crucial for the success of the project, as it enabled the team to maintain high service availability and performance standards. The following strategies were implemented:

3.6.1 Centralized Logging

One of the key strategies for effective monitoring was the implementation of centralized logging. All microservices were configured to forward their logs to a centralized logging platform, enabling the development and operations teams to access and analyze logs from a single location, regardless of the number of microservices running or the complexity of the system.

3.6.2 Performance Monitoring

Another critical aspect of ensuring the reliability and efficiency of the deployed system was real-time performance monitoring. Tool such as **Kubernetes Dashboard** were integrated into the environment to provide a comprehensive view of the operational health of each microservice and the underlying infrastructure.

- **Kubernetes Dashboard**:Since the system was deployed on a **Kubernetes** cluster, the Kubernetes Dashboard was also used to monitor the health of the individual pods, deployments, and nodes within the cluster. It provided a high-level overview of the systems state, allowing the team to quickly identify any issues with containerized services, such as pod crashes, failed deployments, or resource exhaustion.

  - **Response Time**: The average time taken to process requests, which allowed for the identification of any latency issues that could affect user experience.
  - **Error Rates**: The frequency of errors encountered across the system, such as 4xx or 5xx HTTP errors. Monitoring error rates allowed for the immediate detection of problems, including authentication failures or service crashes.
  - **Resource Utilization**: The consumption of CPU, memory, and disk space by each microservice, which helped ensure that no service was overburdened or starved of resources. High resource utilization could indicate potential scaling issues or inefficient code.
  - **Service Uptime**: Real-time monitoring of the availability of the services, ensuring that any service downtime or unresponsiveness was immediately flagged.

  By continuously tracking these metrics, the team could assess the overall health of the microservices architecture and identify performance bottlenecks, ensuring that the system met the expected performance standards.

  With Kubernetes Dashboard, the team had full visibility into both the application layer (microservices) and the infrastructure layer (Kubernetes cluster), facilitating a more holistic approach to performance monitoring.

3.6.3 Benefits of Monitoring and Logging

The implementation of robust monitoring and logging mechanisms played a crucial role in ensuring the efficiency, reliability, and maintainability of the microservices-based system. These strategies provided valuable insights into system performance, resource utilization, and potential failure points, enabling the development and operations teams to proactively address issues before they could negatively impact the end users. The key benefits of these monitoring and logging mechanisms are outlined below:

- **Proactive Issue Detection**: One of the primary advantages of real-time monitoring and logging was the ability to detect performance degradation, failures, or bottlenecks before they escalated into critical incidents. By continuously collecting and analyzing system metrics such as CPU usage, memory consumption, request latency, and error rates, the system was

able to identify anomalies early. Automated alerts ensured that the development and operations teams were notified of unusual behavior, allowing them to take preventive measures before the issue affected system availability. This proactive approach significantly reduced system downtime and improved the overall user experience.

- **Faster Troubleshooting and Debugging**: Centralized logging provided a structured and efficient way to collect logs from all microservices in a single repository, making it easier to track and analyze system events. Instead of manually searching for logs across multiple servers or containers, developers could use tools like Elasticsearch and Kibana to query logs, filter results, and visualize patterns. This streamlined debugging process enabled faster identification of the root causes of errors, reducing the mean time to resolution (MTTR). Furthermore, structured log data, including request traces, timestamps, and error codes, helped in reproducing issues accurately and applying targeted fixes.

- **Optimized Resource Management**: Continuous performance monitoring allowed for dynamic resource allocation based on real-time system demands. By tracking CPU and memory utilization of each microservice, the system could detect underutilized resources, preventing overprovisioning and reducing unnecessary operational costs. Conversely, if a microservice exhibited high resource consumption, alerts could trigger autoscaling mechanisms to allocate additional resources dynamically, ensuring smooth system operation during traffic spikes. This level of resource optimization led to improved cost efficiency and enhanced system scalability.

- **Improved System Reliability and Availability**: The combination of real-time monitoring, alerting, and centralized logging ensured that system reliability was maintained at a high standard. Monitoring tools like Grafana and Kubernetes Dashboard provided a comprehensive overview of the systems health, enabling teams to identify trends and predict potential failures. Additionally, automated alerting mechanisms ensured that service disruptions were promptly addressed, minimizing their impact on users. With these monitoring strategies in place, the system was able to recover quickly

4 Results

This chapter presents the results obtained from the implementation of microservices and federated authentication technologies during the internship at Codebeex SRL. The outcomes are organized into several sections, highlighting the performance of the microservices architecture, the effectiveness of the federated authentication system, user feedback, and key performance metrics. The chapter also includes tables and figures to visually represent the findings.

4.1 Microservices Performance

The microservices architecture was designed to enhance scalability, maintainability, and overall performance. Each microservice was developed and deployed independently, allowing for efficient resource utilization and load management. The following subsections detail the performance metrics collected during testing.

4.1.1 Response Time Analysis

The response time of each microservice was measured under varying load conditions. The tests simulated different numbers of concurrent users accessing the services to evaluate how the architecture scaled. The results are summarized in Table 4.1.

The data indicates that all microservices maintained acceptable response times even under high load, with the **mw_auth** service exhibiting the highest latency due to the additional overhead associated with user authentication and token generation.

Microservice	1 User (ms)	10 Users (ms)	100 Users (ms)
mw_auth	120	150	300
mw_dashboard	80	90	180
mw_realm_info	95	120	250
mw_part_mapper	70	75	160
mw_photo_shark	85	100	220
mw_survey	110	135	280

Table 4.1: Response time analysis of microservices under different load conditions

4.1.2 Error Rate Analysis

In addition to response times, the error rates of the microservices were analyzed during the load testing phase. The results are presented in Table 4.2.

Microservice	1 User	10 Users	100 Users
mw_auth	0%	0%	2%
mw_dashboard	0%	0%	1%
mw_realm_info	0%	1%	3%
mw_part_mapper	0%	0%	1%
mw_photo_shark	0%	0%	2%
mw_survey	0%	1%	4%

Table 4.2: Error rates observed during load testing

The **mw_auth** service demonstrated an increase in error rates under heavy load, primarily due to the strain on the authentication process, while other services maintained a low error rate.

4.2 Federated Authentication Effectiveness

The effectiveness of the federated authentication system was assessed through user testing and feedback. The authentication flow was designed to streamline user experience while ensuring security. Key findings include:

4.2.1 User Experience Feedback

A survey was conducted among users to evaluate their experience with the authentication process. The survey included questions about ease of use, perceived security, and overall satisfaction. The results are summarized in Figure 4.1.

Figure 4.1: User feedback on federated authentication experience

The feedback indicated that 85% of users found the authentication process straightforward, and 90% felt secure using the system. This high level of satisfaction reflects the effectiveness of the JWT-based authentication mechanism and the overall design of the federated authentication workflow.

4.2.2 Authentication Latency

The latency introduced by the authentication process was measured to ensure that it did not negatively impact the user experience. The average time taken for users to log in and receive their JWT token was approximately 150 ms, as shown in Table 4.3.

User Count	Average Login Time (ms)	Token Generation Time (ms)
1	120	30
10	140	30
100	160	35

Table 4.3: Authentication latency during user login

The results indicate that the authentication latency remains consistent even with an increasing number of concurrent users, ensuring that the user experience is not adversely affected.

4.3 Key Performance Metrics

In addition to response times and error rates, several key performance metrics were tracked to evaluate the overall health of the microservices architecture. These metrics included CPU and memory usage.

The CPU and memory usage data demonstrate that all microservices operated within acceptable limits, with peak usage occurring during periods of high load. The microservices architecture effectively managed resources, highlighting the efficiency of the implemented design.

4.4 Summary of Results

The results of the implemented microservices and federated authentication technologies showcase their effectiveness in achieving the desired goals of scalability, performance, and user satisfaction. The comprehensive analysis of response times, error rates, user feedback, and system metrics reflects a robust and efficient system capable of handling the demands of the application environment.

5 Conclusion

This chapter concludes the study conducted during the internship at Codebeex SRL, focusing on federated authentication technologies within a multi-microservice architecture. It synthesizes the findings, emphasizing the significance of microservices architecture, JWT-based authentication, and containerization technologies in building scalable and secure middleware systems. Additionally, the chapter outlines the contributions of this study to addressing existing gaps in federated authentication solutions.

5.1 Key Findings

The study successfully developed a comprehensive Middleware system, integrating eight specialized microservices tailored to meet distinct functional requirements. These include:

- **mw-dashboard**: A centralized interface for accessing all available services.
- **mw-auth**: Facilitates secure user authentication and token issuance.
- **mw-realm-info**: Manages user-specific tokens and service configurations.
- **mw-part-mapper** and **mw-photo-shark**: Handle domain-specific requirements, such as vehicle part management and photo guidelines.

The adoption of JSON Web Tokens (JWT) ensured secure, stateless communication between microservices. This approach minimized performance overhead and eliminated the need for centralized session storage, aligning with the scalability requirements of the microservices model.

5.2 Technological Contributions

The implementation of Kubernetes and Docker played a pivotal role in deploying and managing the containerized microservices. Kubernetes features such as automated scaling, self-healing, and service discovery streamlined the operational workflow and enhanced system reliability. The use of Python FastAPI for microservice development demonstrated its efficiency in creating lightweight, high-performance REST APIs, further strengthening the Middleware's architecture.

5.3 Addressing Gaps in Existing Solutions

This study addressed several challenges associated with federated authentication in multi-microservice environments:

- **Token Lifecycle Management**: By leveraging JWT, the system enabled efficient token issuance and validation across services.
- **Inter-Service Security**: Secure inter-service communication was ensured through token-based authentication, preventing unauthorized access.
- **System Scalability**: The modular design and containerization supported seamless scaling of individual services.

Despite these advancements, the study recognizes the ongoing need for further refinement in areas such as advanced authentication protocols and dynamic load balancing to improve system robustness and efficiency.

5.4 Future Directions

The results of this study pave the way for future exploration in federated authentication and microservices architecture. Potential areas of focus include:

- Integration of advanced protocols such as OAuth 2.0 and OpenID Connect to enhance interoperability.
- Adoption of distributed tracing tools to monitor and optimize inter-service communication.
- Implementation of dynamic resource allocation mechanisms to further enhance scalability.

5.5 Significance and Impact

From an academic perspective, this study bridges theoretical concepts and practical implementation, providing a scalable framework that can be adapted to similar use cases. For Codebeex SRL, the Middleware system underscores the companys commitment to innovation, modularity, and security, strengthening its position as a leader in developing cutting-edge software solutions.

In conclusion, the findings of this study contribute meaningfully to the field of federated authentication in multi-microservice environments. The proposed solutions and methodologies address critical challenges while laying the groundwork for future advancements, ensuring the Middleware remains aligned with evolving industry standards and requirements.

References

[1] Auth0. *JWT (JSON Web Tokens)*. Accessed: 2023-11-25. 2023. URL: https://jwt.io/.

[2] Kubernetes Authors. *Kubernetes Documentation*. Accessed: 2023-11-25. 2019. URL: https://kubernetes.io/docs/.

[3] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes: Up and Running*. O'Reilly Media, 2018.

[4] Brian Burns, Joe Beda, and Lachlan Evenson. "Kubernetes: Up and Running: Dive into the Future of Infrastructure". In: *O'Reilly Media*. 2016, pp. 1–250.

[5] David Ferraiolo et al. "Proposed NIST standard for role-based access control". In: *ACM Transactions on Information and System Security* (2001).

[6] Dick Hardt. "The OAuth 2.0 Authorization Framework". In: *RFC 6749* (2012).

[7] Stefan Hensel and Mark Herndon. "Microservices: A Software Architectural Style". In: *Journal of Software Engineering* 33.2 (2016), pp. 122–136. DOI: 10.1007/s10115-016-0978-x.

[8] Michael Jones. "JSON Web Token (JWT)". In: *RFC 7519* (2015).

[9] Nadine Kratzke and Christian K. Prehofer. "Microservices: A Systematic Mapping Study". In: *Journal of Software Maintenance and Evolution* 31.4 (2019). DOI: 10.1002/smr.1972.

[10] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media, 2015.

[11] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[12] Claudia Pahl and David J. Sell. "Microservices: Concepts, Design, and Implementation". In: *Proceedings of the International Conference on Software Engineering*. Springer, 2015, pp. 39–49.

[13] Daniel Petri. "Understanding Kubernetes: Orchestrating Containers for DevOps". In: *Packt Publishing* (2018).

[14] Sebastián Ramírez. "FastAPI: The modern, fast (high-performance) web framework for building APIs with Python 3.7+". In: *FastAPI Documentation* (2018). Accessed: 2025-02-03. URL: https://fastapi.tiangolo.com/.

[15] Codebeex SRL. *About Us*. Accessed: 2023-11-25. 2023. URL: https://www.codebeex.com/en/.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my thesis supervisor, **Prof. Matteo Ceccarello**, for their invaluable guidance, continuous support, and insightful feedback throughout this study. Their expertise and encouragement have played a pivotal role in shaping my understanding of microservices architecture and federated authentication.

I extend my sincere appreciation to **Codebeex SRL** for providing me with the opportunity to undertake this internship. The enriching experience at Codebeex allowed me to work on real-world applications of microservices, Kubernetes, and federated authentication. I am grateful to my colleagues and mentors at Codebeex, particularly **Marco Tosato**, for their patience, technical insights, and constructive discussions, which greatly contributed to the successful completion of this work.

I would also like to acknowledge the faculty and staff of the **University of Padova**, whose support and resources have been instrumental in my academic journey. Their commitment to excellence in education and research has provided me with the foundation to pursue this thesis.

A special thanks to my family and friends, whose unwavering encouragement and belief in my abilities have been a constant source of motivation. Their support has been invaluable throughout this challenging yet rewarding journey.

Finally, I am grateful to everyone who, in one way or another, contributed to this thesis. Your support and contributions are deeply appreciated.

Muhammad Ali
March 22, 2025