INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Migration of JEE Services to Cloud Native Architecture with Quarkus

TIAGO MIGUEL PAIVA NORA
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Migration of JEE Services to Cloud Native Architecture with Quarkus

Tiago Nora

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Information and Knowledge Systems

Academic Supervisor: Isabel Azevedo (ISEP)
Enterprise Supervisor: Ricardo Ferreira (Altice Labs)

Porto, June 29, 2025

Statement of Integrity

I hereby declare that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end. The exceptions are explicitly recognised in the section "Ethical considerations" of the first chapter. This section also states how AI tools were used and for what purpose.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 29, 2025

Dedicatory

Dedicated to all the people who helped me on this journey and trusted in my potential

Abstract

The transformation of services to cloud-native architecture is a must for the organization seeking better scalability, flexibility, and operational efficiency. This paper presents research about the migration of JEE services to a cloud-native framework with Quarkus, an open-source, Kubernetes-native Java solution.

The document presents an efficient way to migrate legacy systems, addressing some significant concerns in regard to integration, performance, and security, it also reinforces the importance of Quarkus as one of the main frameworks for developing scalable and resilient cloud-native applications and promoting best practices like Kubernetes and container orchestration to fully realize the benefits of this architectural paradigm.

A successful migration also depends very much on the project planning and capability building, where teams need to gain specialized knowledge in cloud-native technologies, DevOps, and microservices design. To ensure a comfortable experience without affecting operations, a phased migration approach should be supported with clear planning, resource allocation, and risk management.

Then there are ethical considerations involved including privacy of data, reliability of the system and observance of the law more importantly protection of user data and finally transparency to the stakeholders, this is the basis for an ethical migration process.

Keywords: Cloud-native architecture, Quarkus, JEE migration, Kubernetes

Resumo

A transformação de serviços para uma arquitetura nativa da nuvem é uma obrigação para as organizações que procuram uma melhor escalabilidade, flexibilidade e eficiência operacional. Este documento apresenta uma pesquisa sobre a migração de serviços JEE para uma framework cloud-native com a ferramenta Quarkus, uma solução Java open source nativa para Kubernetes.

O documento apresenta um método eficiente de migrar sistemas legados, abordando algumas preocupações significativas em relação à integração, desempenho e segurança, além de reforçar a importância do Quarkus como um dos principais frameworks para o desenvolvimento de aplicativos nativos da nuvem escaláveis e resilientes e promover as melhores práticas, como Kubernetes e orquestração de contentores, para aproveitar ao máximo os benefícios desse paradigma arquitetônico.

Uma migração bem sucedida também depende muito do planeamento do projeto e do desenvolvimento de capacidades, em que as equipas precisam de adquirir conhecimentos especializados em tecnologias nativas da cloud, DevOps e design de microsserviços. Para garantir uma experiência confortável sem afetar as operações, uma abordagem de migração faseada deve ser apoiada por um planeamento claro, atribuição de recursos e gestão de riscos.

Em seguida, são tidos em conta aspectos éticos, como a privacidade dos dados, a fiabilidade do sistema e a conformidade regulamentar, e, mais importante ainda, a proteção dos dados dos utilizadores e, por último, a transparência com as partes envolvidas, o que constitui a base de um processo de migração ético.

Acknowledgement

I'd like to thank my supervisor Isabel Azevedo for all her guidance and for always being available to help during the course of this work.

I would also like to thank my family, who provided me with all the materials I needed to succeed in life, my friends for their support and Altice Labs for the opportunity.

And finally, I would like to thank all my colleagues and teachers who have passed through my academic life at ISEP.

Contents

List of Figures

List of Tables

List of Source Code

List of Abbreviations

AI	Artificial Inteligence.
AOT	Ahead Of Time.
APIs	Application Programming Interface.
CD	Continuous Deployment.
CI	Continuous Integration.
CNCF	Cloud Native Computing Foundation.
GDPR	General Data Protection Regulation.
IEEE	Institute of Electrical and Electronics Engineers.
IoT	Internet of Things.
IPP	Instituto Politecnico do Porto.
IT	Information Technology.
JIT	Just In Time.
KPI	Key Performance Indicator.
KQI	Key Quality Indicator.
LLVM	Low Level Virtual Machines.
ML	Machine Learning.
OHM	Order Handling Management.
OPEX	Operational Expenditures.
OSS	Operations Support Systems.
RSS	Resident Set Size.
TOSS	Telecom Operations Support Systems.
WBS	Work Breakdown Structure.

Chapter 1

Introduction

This chapter includes a description of the problem, the objective's set, the research methodology, the ethical considerations, contributions, previous work, the use of AI tools and the structure of the document.

1.1 Context and Problem

Cloud-native systems are increasingly seen as essential for telecommunication operators seeking to deliver innovative services. In fact, most telecom providers now recognize that transitioning to a cloud-native architecture is necessary to gain the agility and scalability needed for launching new services in a cost-effective manner [1]. Legacy networks struggle to support the on-demand, real-time offerings required by modern customers, whereas cloud-native frameworks allow telcos to develop and deploy new capabilities much faster.

Industry analysis affirms that rising competition and technologies like 5G have made cloud-native transformation essential for operators to stay ahead and drive service innovation[2]. This architectural shift also enhances resilience and operational efficiency, cloud-native telecom platforms support features like automation and self-healing, which reduce downtime and improve service reliability.

By decoupling functions and utilizing container orchestration, a cloud-based OSS can adapt more readily to changing network conditions and optimize resource utilization, ultimately lowering costs through better utilization and automation [3]. These benefits make a strong case for modernizing OSS with cloud-based (and particularly cloud-native) principles to meet the demands of real-time, dynamic service management.

Modern telecom services build on industry-standard public cloud platforms and cloud-native development tools. Platforms such as Amazon Web Services (AWS) [4], Microsoft Azure [5], and Google Cloud Platform (GCP) [6] have become the industry standard for deploying digital services, together accounting for roughly two-thirds of the global cloud infrastructure market share [7]. On top of these public clouds, organizations rely on cloud-native technologies to develop and run their services with maximal agility.

Container orchestration systems like Kubernetes, a portable, extensible, open source platform for managing containerized workloads and services, are now ubiquitous for managing scalable services deployments, serving as the backbone for cloud-native applications [3]. Furthermore, modern frameworks such as Quarkus [8] have emerged to help developers create lightweight, high-performance cloud-native applications optimized for these environments, Quarkus, for example, is a Kubernetes-native Java framework designed for fast startup and low resource usage in containerized workloads [9].

Together, the use of leading public clouds alongside tools like Kubernetes and Quarkus enables telecom operators to build and operate innovative services with greater flexibility, portability, and efficiency across today's cloud-centric infrastructure.

NOSSIS ONE suite is an operations support system solution that, for business and technical reasons, needs to migrate some components to the cloud to enhance scalability, flexibility, and operational efficiency. The current architecture limits the ability to rapidly adapt to changing market demands, scale resources according to usage, and integrate with modern cloud-native technologies. Therefore, the migration to a cloud-based architecture is essential to enable a more agile, resilient, and efficient system that can support the growing volume of data and the evolving requirements of telecommunications networks.

It should be noted that most of these problems are not unique to NOSSIS ONE or even the telecommunications industry. Most monolithic applications in other industries experience similar issues when attempting to scale and modernize. Moreover, the challenges in telecommunications are further compounded by the rapidly increasing volumes of data—driven by technologies such as 5G, Internet of Things (IoT), and other emerging technologies and the constant need to deliver new services with low latency and high availability [10].

However, migration to a cloud-native architecture is not straightforward. It includes refactoring or rewriting considerable parts of the codebase, reviewing development and deployment practices, rethinking the storage and inter-module communication mechanisms, along with consideration of several issues regarding security and cost control in cloud environments. Also, teams have to handle with caution a wide variety of security, compliance, and cost management problems in the cloud. Other risks may also be encountered, such as potential loss or corruption of information, unplanned downtime, inappropriateness with business requirements, undeserved operational costs due to inefficient utilization of resources, and resistance within organizations to adopt new development and running practices.

According to the literature, this migration falls under the category of incremental migration [11]. This strategy entails progressively converting and implementing portions of the legacy application into the new cloud-native environment, rather than trying to replace the entire monolithic system all at once. In this project, some legacy JEE services were gradually incorporated into a containerized Kubernetes-based architecture after being refactored using Quarkus. This phased approach lowers the risks usually connected with large-scale system transitions, minimizes operational disruption, and permits thorough testing. When uninterrupted service availability is crucial and distinct modules can change independently over time, incremental migration is especially appropriate.

In the context of NOSSIS ONE, the need for a move to cloud solutions is justified by the ever-increasing demand for data in telecommunications networks, market competition, and the need for greater agility in bringing new functionalities to the market. Thus, a microservices-based approach, or, in general, cloud-native, allows for elastic scalability and more robust Continuous Integration (CI)/Continuous Deployment (CD) processes, and fosters the continuous adoption of new technologies, rendering the system more resilient, efficient, and in line with market demands.

1.2 Objectives and Research Methodology

This work uses a case study, as defined by Oates [12], as part of an empirical evaluation strategy designed to complement and validate the findings of the literature review. Oates

claims that a case study is an appropriate technique for examining current events in their actual setting, particularly in situations where it is difficult to distinguish between the two. This project's exploratory case study looks at how Quarkus and Kubernetes can be used to successfully migrate legacy JEE systems to a cloud-native architecture.

This study aims to add to the body of knowledge on best practices related to the migration of JEE Services to Cloud Native Architecture with Quarkus, explore common problems and solutions in legacy system modernization, and assess how implementing cloud-native technologies affects software scalability, maintainability, and performance.

Selected JEE services were either completely converted to services using Quarkus or re-implemented using the framework from the ground up in this exploratory case study, in which the target is some components that belong to the Nossis suite. Kubernetes orchestrated the deployment of these services in a containerized environment. The resulting services were tested for functionality, performance, scalability, and resilience as part of the study's empirical component. Latency, throughput, CPU/memory utilization, and fault tolerance were among the important metrics gathered. The baseline measurements from the original system were then contrasted with these.

By bridging theory and practice, this case study validates architectural approaches found in the literature review and provides useful insights into modernization processes. In the specific context of NOSSIS ONE, this work focuses on three of its components — the document manager service, network qualifier service and the module 360 view — as an exploratory case study, serving as a strategic foundation for further enhancements that may be implemented based on the lessons learned throughout this project.

1.3 Ethical Considerations

The ethical framework of this project was based on the Instituto Politecnico do Porto (IPP) Code of Conduct, ensuring academic integrity standards such as proper citation, originality, and truthful presentation of results [13]. Specific articles followed included a commitment to ethical conduct and principles of careful analysis, comprehensive citation, and reproducible findings. Besides, the work was developed in compliance with the Institute of Electrical and Electronics Engineers (IEEE) Code of Ethics [14], which includes honesty, openness, professional competence, and avoidance of real or perceived conflicts of interest. The ethical considerations also included realistic goals, the use of open-source resources, and guidance from an advisor on relevant ethical, economic, and cultural issues.

In relation to the migration of the services to a cloud-native, Quarkus based system introduces several ethical issues, apart from the development and release of responsible technology. Certainly, data privacy and data security should be core discussion points related to users' sensitive information. These should at least be in compliance with certain data protection laws, such as General Data Protection Regulation (GDPR). Compliance may be ensured with measures such as encryption of sensitive data in-transit and at-rest, strict access control, and anonymizing of user data where possible. Besides, monitoring tools like Grafana and Prometheus should be configured in such a way that unauthorized access to system logs and telemetry data is prevented. Be that as it may, systems ought not to be biased but show neutral behavior so as not to make some groups suffer. This helps balance technical development against the social and regulatory expectations of the projects, since consideration is taken into the design for the ethical principle at play [15].

It is important to note that the chosen code of ethics includes the requirement of examining the quality of created products and, at the same time, being aware of the possible limitations. In this respect, continuous improvement and maintenance of the system should include the practices and metrics to be implemented in the evaluation of quality with respect to especially performance, reliability, and usability. It is concurrently important to note any limitations, be they functional, technical, or related to scope, to ensure transparency and keep the project in line with ethical standards and user expectations. These measures will be integrated into the future work, enabling the responsible and sustainable evolution of the project.

1.3.1 Use of AI Tools

During the process of carrying out this dissertation, artificial intelligence (AI) tools were used selectively to enhance and support productivity, without in any manner compromising academic integrity or the originality of the work.

AI-driven language models, such as ChatGPT, were used primarily for the following:

- **Explanations of technical ideas:** AI was used as a supplementary instrument for explaining hard topics related to cloud-native architecture like Kubernetes, consolidating earlier acquired knowledge.
- **Text refinement and revision:** AI was used in certain portions to verify the grammar and readability of written English, which helped make the document much easier to read.
- **Structuring suggestions:** AI was consulted during planning for suggestions regarding how to structure particular chapters or how to approach methodological sections, which were then taken and developed fully by the author.

Importantly, no content was generated or included directly from AI tools without extensive human oversight, critical analysis, and alteration. All literature reviews, code, analysis, and conclusions were developed independently by the author and based on academic literature, technical documentation, and implementation experience.

AI utilization was driven by ethical guidelines set by the institution for its use, promoting transparency, equity, and accountability in its usage. There were no AI tools utilized in data analysis, code writing, or experimental findings, and all results reported are a product of the author's own work.

1.4 Contributions

This research dissertation contributes greatly at an academic level as well as in practice within its application area of migrating Java EE (JEE) Services into Cloud-Native Architecture with Quarkus. It provides the comprehensive and systematic literature review assessing what is already known about cloud-native architecture, the Quarkus framework, best practices, common challenges, and applicable solutions in academic and practical aspects of migration. Further, the analysis considers the sufficiency of the Quarkus framework in relation to its counterparts by obtaining evidence on performances and scaling capabilities in addition to indigenous Kubernetes integration. This paper presents a case study that takes multiple services into account with a special emphasis on their technical architecture, implementation strategies, testing methodologies, and deployment practices in Kubernetes. In addition, performance metric analysis and result validation are incorporated in this study in

terms of scalability boundaries, response time, and resource usage. The best practices are also established in the research, with experiential descriptions providing value for organizations that plan to engage in such migration efforts. The dissertation also houses ethical considerations such that issues like data privacy, regulatory compliance, and fairness would be adequately addressed in the migration process.

1.5 Previous Work

This document can be considered as a continuation of the work developed during the first semester in a curricular unit called Preparação para Dissertação (PREPD). During the first stage, a preliminary study was set up to identify the area of research, formulate the problem statement, and define the objectives and methodology of the dissertation. A literature review was also initiated in PREPD to examine existing techniques and technologies on moving monolithic applications to Quarkus and cloud-native applications.

The PREPD work involved issues such as project planning and skills management, which were particularly important in informing the dissertation timetable, establishing the technical skills required, and ensuring that the project was realistic in terms of the available resources and expertise. The ethical dimensions were also addressed, namely to ensure that no private or sensitive data was used throughout all phases of development and implementation according to the most ideal practices in transparency, integrity, and security in software development. This initial preparation laid the ground for successful execution of the practical work outlined in the second semester.

1.6 Document Structure

This section provides an overview about the project structure. This document is divided in seven chapters, these being Introduction, Background, Skill Management and Project Planning, Literature Review, Design, Implementation and Conclusion.

The current chapter is the Introduction and its objective is to present the problem that is being analyzed, the objectives that were defined, what ethical considerations that can arise and finally the present section.

The Background chapter aims to provide a detailed background to the problem, where is presented the relevant factors and environment in which the project is inserted.

The Skill Management and Project Planning chapter identifies the relevant skills that are important to the objectives of the project and covers the methodologies and tools used to plan, execute and monitor the assigned tasks to make certain that the proposed targets are satisfied.

The Literature chapter presents a comprehensive analysis of existing research on the topic to provide a foundation for understating the practical and academic context.

The Design chapter describes the project design stage, explaining the architectural framework, design goals, essentials details, and the reasons for making major technical choices.

The Implementation chapter describes the practical aspects of building the system, discussing how the design specifications were implemented in code through modern development practices and tools.

And finally, the Conclusion chapter, reflects on how adopting a cloud-native architecture.It acknowledges the study's limitations (scope confined to stateless modules, the rapidly evolving toolchain, and added operational complexity) and lays out concrete paths for future research and closes by framing the transition to cloud-native practices as both a technical evolution and a strategic imperative, offering guidance and inspiration for practitioners and researchers alike.

Chapter 2

Background

This chapter provides the foundational context and theoretical basis necessary to understand the technologies, frameworks, and methodologies employed in this thesis. It outlines the key components of modern cloud-native architectures, focusing on the tools and platforms used to facilitate the migration of legacy systems to scalable, efficient, and high-performance environments.

2.1 Background Systems

The NOSSIS One from Altice Labs is an integrated OSS suite designed for the management and optimization of telecommunications' operations provided through various network technologies and services. All these features on cloud-native, open modular architecture that enables both traditional Telco networks, Information Technology (IT) networks, and new virtualized environments [16]. The previously mentioned article highlights the key features of NOSSIS One, which include:

- **Functional Areas**: Covered key operational processes, inventory, fulfillment, and assurance practices, thus helping to manage services and resources end-to-end.
- **Automatization and Artificial Inteligence (AI)/Machine Learning (ML)**: NOSSIS One embraces AI/ML-driven automation across operations, decision-making, and customer experience improvement via intelligent analytics and closed loop operations.
- **Integration and Flexibility**: The system provides standard Application Programming Interface (APIs) along with off-the-shelf multi-technology packs to quickly integrate with multiple network vendors. This warrants an easy onboarding of new services,
- **Focus on Customer Experience**: Designed in such a way that its focus is to provide a greater customer experience with a clear view in real time of performance monitoring and automated fault management.

The key takeaways with NOSSIS One are cost reduction, increased agility, and operational streamlining in the delivery of telecommunication services. It is very instrumental for service providers in their efforts to improve operational efficiency and service delivery.

The architecture of the Altaia system, part of the NOSSIS One Suite that was described previously, is designed to provide a powerful and unified end-to-end assurance platform operating in multiservice, multi-technology, and multivendor environments [17]. The previously mentioned article, highlights the detailed architectural diagram, where outlines several key components and functionalities that characterize the system:

- **Real time Monitoring**: Altaia provides a real time platform for full service continuity with network and service performance monitoring. It serves to enable proactive detection of service degradation and problems within the networks, important for maintaining high quality customer experiences.
- **Multi-Vendor Support**: The system is capable of integrating with several technologies and a number of other different vendors. The ability for it to be deployed across various network environments, from legacy to modern.
- **Data Aggregation and Analysis**: Altaia aggregates data through the collection and analysis from various sources, drawing Key Performance Indicator (KPI) and Key Quality Indicator (KQI) up to the cell level. The aggregation may be helpful in identifying service degradation issues and troubleshooting before it affects customers.
- **User Centric Dashboards**: The platform delivers powerful dashboards for real time indications and performance metrics. These can provide a personalized view to present information subscribers, geographical areas, devices, or with administrative perspectives to tune in to very specific monitoring of groups of customers.
- **Proactive Problem Resolution**: Altaia is able to baseline threshold and historical performance data to proactively identify and resolve network issues, thus ensuring that service availability is maintained and that customer experience is not degraded.
- **Integration with Operations**: The system is designed to support various operational teams, including network operations, engineering, and customer account management, facilitating a centralized and homogeneous view of service quality.

In other words, Altaia architecture provides proactive service quality management with support from data and analytics in real time, leading to improved customer satisfaction and operational efficiency.

The Document Manager system is a robust solution designed for efficient document management, belonging to the Atalaia system, namely to its assurance area. As a result of this integration, all the documents needed for operations like storage, retrieval, and version control are centralized within the Atalaia framework. This makes the system work better and more consistently.

It allows a variety of functions, such as the treatment of different document types, for flexibility in adapting to various needs that may be required by an organization. This integration will not only manage the life cycle of the documents, but also will perfectly align with Altaia's operational structure and maintain all documentation consistent and efficient.

While a number of Altaia's modules have been architected for seamless cloud integration, enabling deployment across private, public, or hybrid infrastructures, the Document Manager subsystem currently remains limited and is therefore not yet suitable for cloud-based deployments.

Another of the systems that needs to be mentioned is the NetQ system, it is an Operations Support System (OSS) utilized to diagnose the network systems and institute remedial actions. NetQ is intended to assist Communication Service Providers (CSPs) in streamlining their network operations.

The main features of this system are:

- **Network Problem Diagnosis**: NetQ makes it possible to determine problems throughout the network, both service provider network and customer site.
- **Corrective Actions**: It offers intelligent diagnosis capabilities and suggested actions to correct issues found, which helps in quick problem resolution.
- **End-to-End Testing**: The system conducts end-to-end testing from the service provider network right through to the customer home network, with complete service verification.
- **Integration with Other Systems**: NetQ may be easily integrated with Customer Relationship Management (CRM) and other operational support systems to facilitate integrated management of service delivery and support processes.

And, finally, the View 360 module acts as a single, centralized gateway to make it easier to access a set of underlying services by aggregating numerous disparate endpoints onto a single logical interface, it translates the burden of managing direct interaction with each service away, turning variable processes like authentication, logging, exception handling, and route management into requirements. Since client applications can access multiple services through a single entry point, this not only simplifies the system architecture overall but also improves security and maintainability.

Additionally, the modular nature of the module renders it simple to scale and integrate in the future, hence appropriate for environments with complex service ecosystems that need proper management and orchestration.

This module is tailored specifically for one of the clients, with a robust and future-proofed service orchestration and resource management strategy.

2.2 Kubernetes

Kubernetes is an open-source container orchestration platform that is powerful and meant to automate the deployment, scaling, and management of containerized applications. Initially developed by Google, Kubernetes is now developed under the auspices of the Cloud Native Computing Foundation (CNCF), where it has rapidly emerged as the de facto standard for managing distributed systems in cloud-native environments. It offers developers and operations teams various tools with which to efficiently manage applications in both on-premise and cloud-based infrastructures [18].

At its core, Kubernetes provides a declarative model of configuration, whereby a developer gets to declare how they want their application state to look and automatically aligns the system to recurrently realize that same state [19]. The key features include auto-scaling, load balancing, and self-healing, very important in guaranteeing reliability and the availability of applications driven by microservices. Such features become essential for monolithic legacy applications when migrating to a cloud-native setting, such as those built with Java Enterprise Edition, into an environment where services dynamically scale at will or by demand [20].

Kubernetes also supports multi-cloud and hybrid cloud environments, thus offering portability across a wide range of cloud providers, including AWS, Azure, and GCP. Because it is extensible, the system integrates well with a variety of tools and services, allowing organizations to customize their Kubernetes clusters for a wide range of specific operational needs.

A significant characteristic of the ecosystem is its support through a wide variety of plugins that help in managing the complexity of the microservices deployment and orchestration process [21].

Kubernetes thus provides the infrastructure to manage microservices in the context of migrating JEE services to cloud-native architectures. With the help of containers, Kubernetes enables each service to independently deploy, update, and scale; hence, it becomes easier to manage complex applications in a distributed environment [22].

2.3 GraalVM

According to Matija Sipek and colleagues, GraalVM is a high-performance, polyglot virtual machine that seeks to host a diverse range of programming languages while improving software development efficiency [23]. It provides wide-ranging benefits in terms of language interoperability, performance optimization, and deployment flexibility across various environments. Some of the key applications of GraalVM in modern software development include:

- **Polyglot Programming and Interoperability**: GraalVM enables developers to create applications using multiple languages Java, JavaScript, Python, Ruby, R, and Low Level Virtual Machines (LLVM) based languages such as C and C++ on one runtime. There is no isolation between these languages, thus providing the best interoperability and making it easy to use different technologies in one project.
- **Performance Optimization**: High performance due to advanced optimizing compiler participation. GraalVM could act either as Just In Time (JIT) or Ahead Of Time (AOT) Compiler and could apply different techniques at runtime: in lining and escape analyses, for the speeding up of application execution, and includes a native image — best performance at low footprint, thanks to reduced cold start and low memory, hence being best suited for micro and cloud services.
- **Cloud and Containerized Environments**: GraalVM does cloud computing pretty well, especially when combined with frameworks like Quarkus. It supports the creation of native images optimized for Kubernetes or Docker deployments that improve scalability and resource management in containerized environments.
- **Database Integration and Stored Procedures**: GraalVM works well with Oracle and MySQL, among other databases, to enable the use of languages like JavaScript for stored procedures and user-defined functions. This makes the development and maintenance of database logic easier while providing more flexibility in programming.
- **Tool Support and Instrumentation**: GraalVM offers a set of advanced tool support for debugging, profiling, and dynamic analysis with its framework called Truffle. Such tools are language agnostic and lightweight, thus making developers efficiently analyze and optimize application performance across different programming languages.
- **Web Development**: GraalVM supports modern web development using WebAssembly through the TruffleWasm project. It allows developers to combine JavaScript with lower-level languages to achieve near-native performance for web applications, which furthers high performance and versatility for web-based solutions.

Conclusively, GraalVM is the most versatile and powerful tool in modern contexts that affords software developers advantages never seen previously concerning language interoperability,

high performance, and deployability. Having diverse applications from polyglot programming, cloud-native architecture, integration with databases, up to web technologies, it gradually converts to an indispensable helper for people who deal with multi-technology landscapes.

2.4 Quarkus

In the following sections, the framework is going to be presented in detail, starting with an overview of its core features and capabilities then a detailed presentation of the framework integrated into a Kubernetes environment for efficient Cloud Native deployments, how it supports native compilation with GraalVM and finally how Quarkus optimizes this runtime for Java applications to scale in a cloud environment.

2.4.1 Presentation

Quarkus is a modern cloud-native framework targeted at Kubernetes but designed to work in both native and traditional environments, for developing microservices and cloud applications in Java. Quarkus, under the Apache License 2.0, is open-source and kubernetes oriented but supporting native compilation with GraalVM and also OpenJDK HotSpot. By being natively integrated with Kubernetes, Quarkus paves the way for the containerization of Java-based applications in order to avoid typical pitfalls and deployment complexities. Because Quarkus is based on well-known Java libraries, there is minimal learning curve adjustment to be made by a Java developer who already knows such technologies; thus, it is easy to start working with it using your knowledge [24].

2.4.2 Main Features and Advantages

According to Alex Soto Bueno and Jason Porter, Quarkus has numerous striking features to improve productivity and fine tune the performance of the application on Kubernetes [24]. In the book is included, features and benefits such as:

- **Developer Productivity**: Quarkus's development tools allow high productivity out of the box on account of low learning curve for Java developers who use dependency injections, JAX-RS, and other common Java technologies. Quarkus encourages a smooth and efficient development cycle with tools for scaffolding, live reloading, and a save and refresh workflow called dev mode.
- **Kubernetes Integration**: Quarkus makes deploying to kubernetes lots simpler because it will generate automatic Dockerfiles and kubernetes YAML configuration optimized for either the OpenJDK JVM or native executables with GraalVM. Kubernetes extensions in Quarkus reduce the amount of manual configuration work traditionally required, making it possible to package and deploy applications in Kubernetes clusters with minimal setup.
- **Performance Optimizations**: Backed by its motto "supersonic, subatomic", Quarkus focuses on low memory usage as well as fast startup times. Quarkus applications boot in milliseconds and run on very low memory, normally less than 50 MB, owing to the compilation of Java applications into a native executable. It makes this one very effective in any Kubernetes cluster as it would mean resources are effectively utilized by scaling up real quick.

- **Scalability and Resource Efficiency**: Quarkus is ideal for high-density deployment of applications running on Kubernetes. Quarkus brings applications into Kubernetes clusters with low Resident Set Size (RSS) and fast response times, which in turn ensures effective scaling within the cluster. This further optimizes resources in that it is still able to scale up speedily for unexpected loads.
- **Cloud-Native Enhancements**: Quarkus supplies cloud-specific enhancements like reading Kubernetes ConfigMaps and Secrets directly from the API server without file mounts. It allows developers to configure applications dynamically in Kubernetes without extra manual setup.

In conclusion, Quarkus provides an effective means of deploying Java applications to Kubernetes by offering deployment simplicity, first class resource efficiency, and numerous developer utilities aimed at best practices throughout the full development and operational cycle.

2.4.3 Comparison with Other Frameworks

This section proceeds to give a detailed comparative analysis of some of the frameworks in the Java ecosystem that are specifically tailored for microservices development. The frameworks under review are Spring Boot, Micronaut, and Quarkus, each having different advantages depending on various performance, architecture, and community-focused criteria. This comparison is based on the article wrote by Lukasz Wyciślik, Lukasz Latusik, and Anna Malgorzata Kamińska [25].

Spring Boot is a very popular framework that helps to simplify the creation of stand-alone, production-grade Spring applications. It supports both traditional blocking models and reactive programming through Spring WebFlux. Its large active community, in addition to extensive documentation, attracts both experienced and novice developers [26].

Micronaut focuses on building modular, easy-to-test microservices. It emphasizes low memory consumption and fast startup times, making it appropriate for resource-constrained environments and cloud-native scenarios. In the way that Micronaut uses annotation metadata at compile time, it avoids reflection at runtime—hence, better performances [27].

Quarkus is intended for Kubernetes-native and serverless environments, with Java optimization for GraalVM and HotSpot. It has achieved fast startup times with low memory usage—features that are very critical when it comes to microservices deployments, which need fast scaling. Quarkus also supports reactive programming models, which aligns quite well with modern asynchronous application designs [8].

In relation to reactive programming, Micronaut and Quarkus come with reactive methodologies inherently, while Spring Boot offers reactive support via Spring WebFlux. This flexibility lets developers choose between reactive or traditional (blocking) paradigms based on application needs and developer expertise. While all three frameworks are capable of operating in blocking and non-blocking modes, Micronaut and Quarkus are more consciously optimized for non-blocking I/O operations. This may be beneficial in high-throughput, event-driven applications with demands for good resource utilization efficiency.

In relation to computational performance, Quarkus and Micronaut tend to have faster cold-start times, which are quite valuable in serverless environments. Spring Boot remains competitive, with an advantage of mature ecosystem and tooling.

The design philosophies of Micronaut and Quarkus are oriented toward fast startup times. While Spring Boot is somewhat slower at starting up for some configurations, it has vast plugin and integration options, which can give it an edge regarding ease of deployment.

Regarding database interaction, all frameworks have strong support, most notably, through ORM frameworks such as Hibernate. This database interactions can significantly impact the overall performance of microservices when there is a high throughput of data.

In the context of community and ecosystem, each of these frameworks has an active community that means frequent updates, strong tooling, and a lot of documentation:

- **Spring Boot**: Size and maturity of the Spring ecosystem means most problems have well documented solutions. A large user base also drives community-driven innovation.
- **Micronaut**:Although younger, Micronaut has rapidly grown in popularity due to its performance features and strong cloud-native focus. Its community is growing and provides responsive support channels.
- **Quarkus**: With Red Hat behind it, the community is growing rapidly, and its adherence to Kubernetes and modern cloud infrastructures supports it; the ecosystem of plugins is evolving along with new releases.

In green computing contexts, low-resource frameworks become more relevant. The attention placed by Micronaut and Quarkus on minimal memory consumption and fast startup times may, therefore, translate to lower energy consumption, especially at scale. Beyond that, the maturity of Spring Boot supports a high degree of tuning and optimization of resource utilization.

All three frameworks offer horizontal scaling strategies for microservices-based applications in order to deal with increased loads and ensure high availability. The paper demonstrates that the architecture of each framework supports the development of scalable systems by decoupling services and letting them scale independently, following best practices of microservices.

The frameworks are generic and can be applied in various application domains:

- **Cloud-Native Web Services**: Especially in cloud-native environments where containerization and serverless deployments are popular, Quarkus and Micronaut are great.
- **Resource-Constrained Environments**: Given its small footprint, Micronaut is well suited to IoT or edge computing devices.
- **Enterprise-Grade Systems**: The mature ecosystem of Spring Boot and wide integration with enterprise tools strongly justifies its use in large mission-critical applications.

The comparative analysis shows that Spring Boot, Micronaut, and Quarkus are strong solutions for developing microservices within the Java ecosystem, each has its own strength. These could be, as follows:

- **Spring Boot**: Well established with a big community and ecosystem; suitable for complex, enterprise-grade projects where broad library support and developer familiarity are critical.
- **Micronaut**: Emphasizes light-weight architecture and fast start-up times; excels in environments where modular design and efficient resource usage are key requirements.

- **Quarkus**: Optimized for containerization, cloud-native environments, and the use of GraalVM to bring extremely fast startup times along with low memory consumption—very modern, serverless architectures.

The final choice of the most appropriate framework will depend, after all, on specific project requirements that include performance goals, deployment environments, and team expertise. One could imagine future research investigating the efficacy of such frameworks in even more specialized contexts, like edge computing or resource-constrained IoT devices, to further expand this body of knowledge.

2.5 Monolith Decomposition Patterns Into Microservices

The migration from a monolithic system to microservices should be incremental, avoiding the famous "big bang" approach of rewriting everything at once [28]. Sam Newman, author of *Building Microservices* [29] and *Monolith to Microservices* [30], emphasizes that the main objective of this migration is to achieve independent deploys, i.e., to be able to change and deploy parts of the system without affecting the whole. To this end, Newman suggests using Domain-Driven Design (DDD) techniques to identify bounded contexts and service boundaries within the monolith, which guide where to begin the decomposition. Below, we summarize the main decomposition patterns described by Newman, explaining when and why to use each, with illustrative examples and the pros and cons of each approach. Some decomposition patterns are as follows:

- Strangler Fig Pattern
- UI Composition Pattern
- Branch by Abstraction Pattern
- Backends For Frontends

2.5.1 Strangler Fig Pattern

Inspired by a tropical fig tree, this pattern citted by the author involves encircling the monolith with a new application and migrating functionality piece by piece. First, identify a set of functionalities in the monolith to extract; then, develop that functionality in a new microservice; finally, intercept calls destined for the monolith and redirect them to the new service. In this way, the old and new systems coexist, allowing the microservice to grow around the monolith until it eventually replaces it completely. A typical implementation includes placing a proxy (e.g., an HTTP reverse proxy) in front of the monolith, which initially forwards all traffic, and as functionalities are migrated, routes specific requests to the new microservices. Importantly, migration does not imply immediately removing monolith logic—Newman suggests not deleting legacy code right away to serve as a rollback plan if something goes wrong.

Adopt the Strangler Fig Pattern under these circumstances:

- Useful when the functionality to extract is well-bounded and calls can be intercepted at the system perimeter (e.g., API or UI level).
- Ideal for web applications (HTTP-driven), where redirecting specific endpoints is straightforward.

- Enables gradual migration without interrupting feature delivery, giving time for the new service to mature before taking on real load.
- Minimizes risk by allowing easy rollback: since the monolith remains intact, calls can be redirected back if issues arise.

A practical example of the implementation of this pattern: In an e-commerce monolith, suppose you decide to extract the billing module. Using the Strangler Fig pattern, you place a proxy in front of the monolith and implement a billing microservice. Initially, all requests to `/billing` are handled by the monolith. Once the new service is ready and tested, configure the proxy to route billing requests to the microservice. Other features remain in the monolith until migrated, page or endpoint by endpoint.

Related to the usage of the pattern, the advantages can be the following:

- Incremental and controlled migration, avoiding long downtimes or risky rewrite projects.
- Easy rollback by redirecting traffic back to the monolith.
- Migration can be paused or halted if expected benefits do not materialize.
- Allows testing the new component in production without exposing all users immediately.

In the other hand, the disadvantages can be the following:

- Limited applicability if functionality is deeply embedded without a clear interception point.
- Introduces extra latency via the proxy, which must be monitored.
- Requires managing routing and proxy configurations as more pieces are migrated.
- Development teams must understand and manage a hybrid architecture during migration.

2.5.2 UI Composition Pattern

This pattern addresses migration at the presentation layer. Instead of rebuilding the entire UI in one step, divide it into independent components or pages, allowing parts of the interface to be served by the new microservice while the rest remains with the monolith. To the user, the application appears monolithic, but under the hood different UI elements are powered by different services [31]. Composition can occur at the page level or via widgets (page fragments). A technique cited by Newman is using Edge-Side Includes (ESI), where the web server assembles pages by aggregating content from multiple sources—e.g., an HTML template from the monolith with placeholders where microservices insert specific widgets. Modern micro-frontend approaches also fit here, enabling teams to develop isolated UI pieces that are later aggregated client-side or server-side.

Adopt the UI Composition Pattern under these circumstances:

- Ideal when the monolith has a large or tightly coupled presentation layer.
- Allows gradual migration of the user experience, reducing user disruption.
- Useful to validate new frontend technologies at small scale before a full rollout.
- Enables independent deployments of UI components aligned with backend services.

A practical example of the implementation of this pattern: A travel website initially implements only a "Top 10 Destinations" widget from a new service via ESI, while the rest of the page is served by the monolith. This low-risk component allows the team to learn in practice about performance and integration before migrating more UI parts.

Related to the usage of the pattern, the advantages can be the following:

- Incremental evolution of the user interface.
- Isolation by business module, enhancing cohesion.
- Independent deployments enable faster release cycles for UI parts.
- Facilitates A/B testing or canary releases on interface segments.
- Opens the door to micro-frontend architectures and diverse frontend frameworks.

In the other hand, the disadvantages can be the following:

- Potential inconsistent user experience during migration.
- Additional composition layer adds complexity and integration challenges.
- Client-side composition may impact performance; server-side adds aggregation logic.
- Deployment coordination required for contract changes between teams.
- End-to-end testing becomes more complex across multiple services.

2.5.3 Branch by Abstraction Pattern

Refactor the monolith internally by creating an abstraction layer that allows both old and new implementations to coexist in production code [32]. Steps: (1) introduce an abstraction (e.g., an interface) for the target functionality; (2) change callers to use the abstraction; (3) develop the new implementation behind the abstraction (e.g., calling an external service); (4) switch the abstraction to the new implementation via a feature toggle; (5) remove the old code and possibly the abstraction once migration is complete.

Adopt the Branch by Abstraction Pattern under these circumstances:

- When functionality is deeply embedded without an external interception point.
- Allows gradual development of the new implementation within the main codebase.
- Reduces integration conflicts by continuous merging in the main branch.
- Improves modularity and testability even before completing migration.

A practical example of the implementation of this pattern: In an ERP monolith where tax calculation logic is scattered, create an `TaxCalculator` interface and update all calls to use it. Initially, implement the interface with the legacy logic. In parallel, develop a new implementation that calls a tax microservice, and switch via a feature toggle once validated.

Related to the usage of the pattern, the advantages can be the following:

- Minimizes development interruption; old and new coexist safely.
- Improves internal code quality by enforcing clear interfaces.
- Supports safe testing with fallback to legacy behavior.

- Decouples deploy and release phases via feature toggles.

In the other hand, the disadvantages can be the following:

- Temporary code complexity with parallel implementations.
- Maintenance overhead of keeping both versions until cleanup.
- Risk of prolonging the process and leaving dead code.
- Potential performance/memory impact with dual logic.
- Requires strong refactoring skills and adequate automated tests.

2.5.4 Backends For Frontends

Inspired by the need to optimize and tailor data-flows for different client UIs, this pattern introduces a dedicated backend service for each type of frontend. First, identify the specific data-shaping, performance and security requirements of each client (e.g. web SPA, mobile app, IoT device); then, build a thin BFF service that aggregates, orchestrates and transforms calls to your core microservices; finally, have each client talk exclusively to its own BFF. In this way, you isolate UI-specific logic and payload shaping in per-client edge services, letting the underlying services remain focused on business concerns [33].

A typical implementation includes placing each BFF as an edge service (for example, a lightweight Node.js or Go HTTP server) in front of your suite of microservices. Clients authenticate against—and make all requests to—their BFF, which in turn fans out calls (or queries a gateway/federation layer) to the appropriate microservices, then composes and returns a tailored response. Importantly, you avoid duplicating heavyweight domain logic in each BFF by reusing shared libraries or delegating to common services, keeping BFFs thin and maintainable.

Adopt Backends For Frontends under these circumstances:

- Multiple client types with divergent needs. When different UIs require different payload shapes, call patterns or performance profiles (e.g. mobile vs. desktop).
- Desire to reduce chattiness. When client apps are making many fine-grained calls to microservices, a BFF can batch or aggregate requests.
- Independent UI team delivery. Enables each frontend team to own its BFF and iterate without coordinating changes across all clients.
- Security and QoS boundaries. When you want per-client authentication, rate-limiting or caching policies applied at the edge.

A practical example of this pattern in action: In an e-commerce platform, you have both a rich React web storefront and a native mobile app. The web UI needs product details, image galleries, recommendations, and user reviews in one payload, while the mobile app requires a leaner response (omitting reviews) and must include an offline-sync token. You implement two BFFs, bff-web and bff-mobile. Both call the same product, catalog and review services, but bff-mobile filters out heavy media, adds sync metadata and enforces mobile-specific rate limits. The React app is configured to use https://api.example.com/web/*, and the mobile client to use https://api.example.com/mobile/*.

Related to the usage of the pattern, the advantages can be the following:

- Tailors APIs to UI needs, avoiding over- and under-fetching.
- Improves client performance (fewer round-trips, smaller payloads).
- Decouples UI and service teams, accelerating delivery.
- Centralizes client-specific security, caching and rate-limiting.

In the other hand, the disadvantages can be the following:

- Introduces extra services to build, deploy and operate.
- Risk of duplication if BFFs aren't kept thin (shared logic may get copied).
- Increases overall operational complexity and monitoring surface.
- Requires governance to prevent API divergence and ensure consistency.

Chapter 3

Skills Management and Project Planning

This chapter in the first section emphasizes skill management as crucial for project success, focusing on identifying, assessing, and developing essential skills while in the second section covers project planning, including the project charter, scope, deliverables, scheduling, and risk management.

3.1 Skill Management

This section introduces the chapter related to skill management, explaining its importance to the successful completion of a project. It identifies that the development and harnessing of relevant skills are important in relation to the objectives of the project. To begin with, skill management concerns technical, analytical, and soft skills in studying challenges, optimizing processes, and realizing desired outcomes. It also provides a summary of the structured approach that this chapter undertakes: identifying the essential skills, assessing current proficiencies, developing strategies for improving skills, and understanding how the acquisition of skills may impact project success.

3.1.1 Identification of Required Skills

The following section describes the skill identification necessary and required to the successful delivery of the project. We can categorize skills based on their specific needs: technical, analytical, and soft. Technical skills cover knowledge of cloud-native architecture, Quarkus, database management and other tools or technologies. In the case of analytical skills, this involves the ability to evaluate complex problems, interpret data and make informed decisions. And finally, interpersonal skills include project management and effective collaboration, from initial planning to the completion of all ongoing tasks.

3.1.2 Assessment of Current Skills

In this section is assessed the current skills where is identified the current skills, required proficiency, current proficiency, gap between the required proficiency and proficiency and to finish a comment about the skill where is given more information about it. In the Table 3.1 is shown the topics that were mention before. In this table, the main components are:

- **Skills**: The first step to the assessment of the current skills is to make a list of them. These skills my include technical expertise, soft skills and the ability to work in a specific tool or software.

- **Required Proficiency**: For each skill in the skills list is associated a required proficiency that is necessary for the achievement of the work in hands.
- **Current Proficiency**: For each skill in the skills list is associated a current proficiency. This proficiency level can be quantified through some criteria like years of experience, ability to work the problem independently.
- **Gap**: Once the required proficiency and current proficiency of each skill have been identified, the gap is evaluated. These gaps are important to identify the critical areas for individual development, in other words, the areas that need for attention and study by the person in question.
- **Comment**: And finally, after the gap assessment, a comment is provided to detail and give more context about the skill.

Skill	Required Proficiency	Current Proficiency	Gap	Comments
Quarkus	4	2	High	Basic understanding of Quarkus, with limited hands-on experience. Further learning is needed in Quarkus-specific cloud-native practices.
Java Language	5	4	Low	Strong proficiency in Java, though specific Java features in cloud-native contexts require review.
Gradle	3	1	High	Limited experience with Gradle; training will be prioritized to improve build automation skills.
PostgreSQL Database	4	3	Medium	Competent in PostgreSQL basics but needs practice with advanced queries and cloud integration.
Docker	3	2	Low	Some experiences with docker but with some gaps of knowledge.
Kubernetes	3	1	High	Basic understanding for what kubernetes is used with non previous experiences.
Github	4	3	Low	Competent in github with some academic years of experience using this tool
Github Actions	3	1	High	Basic understanding for what github actions are used with non previous experiences.
Grafana	3	1	High	Basic understanding for what grafana is used with non previous experiences.
Prometheus	3	1	High	Basic understanding for what prometheus is used with non previous experiences.
Playwright	3	1	High	Basic understanding how to make tests and basic understanding for what playwright is used with non previous experiences.
Time Management	4	3	Low	Experience in managing smaller projects; needs to build on strategies for larger, collaborative projects.
Communication	4	2	Medium	Effective in written communication but seeking improvement in presenting complex technical information to non-specialists.
Critical thinking	4	3	Low	Contributes to analytical tasks in an initial exposure to professional settings and needs improvement in critical assessment of situations.
Problem-Solving	4	3	Low	Needs enhancement in the ability to identify, analyze and solve problems in a professional setting with the capability of operating in dynamic and complex environments.
Continuous Learning	4	2	Medium	Demonstrates the rudiments of how important it is to adapt and learn new skills.
Collaboration	4	3	Low	Demonstrates some potential for teamwork and collective problem-solving but needs development in terms of fostering collaborative relationships.
Decision-Making	4	3	Low	Requires improvement in assessing risks and benefits to make informed and timely decisions

Table 3.1: Assessment of Current Skills

3.1.3 Strategies for Skill Development

In this section is discussed strategies for skill development. There are numerous forms that one skill can be development, and the right approach often depends on the nature of the skill and from individual to individual and resources available [34]. Some of the strategies that can be used to improve a skill are the following:

- **Education**: One of the most common strategies to develop skills is through formal education, meaning individual courses, universities, workshops and certifications.
- **On the job learning**: Other way to develop skills is to learn on the job by doing some work and experience the troubles that appear in the way of testing and working with a specific tool or technology and this can include, too, shadowing a colleague with more work experience and knowledge.
- **Mentorship**: A mentorship provided from a knowledge person in the same area can be an alternative where this individual can share valuable experiences and can give target guidance and advice to an individual. With that, a mentorship can help individuals navigate challenges with their professional and person development.
- **Self learning and practice**: Individuals can take charge of their own development without the intervention of superiors or other entity to individually learn about a given topic. This individual learning can be accomplice with the help of books, videos, practicing exercises, online courses and exploring themselves the tool or technology. Normally this is done by individuals with high motivation and that can define their own path.
- **Feedback**: Feedback can be a great way to develop skills and to identify areas of improvement and skills that need refining. This feedback can come from superiors, colleagues, clients and others entities. After this feedback is received, an analysis and reflection of the content of the feedback is necessary to help pinpoint where development is needed.
- **Job Rotation**: And finally, job rotation helps an individual gain experience and exposure to different roles within the organization. This rotation can help the individual broader their skill set and at the same time the person gains valuable insights about how the company works, for example in others departments.

3.1.4 Identification of Skills to Improve

As seen in the Section 3.1.2, there are skills that are identified that require some attention, with both High and Medium ratings in the gap column, given the context of the present work. Some examples of that are:

- Quarkus
- Gradle
- Kubernetes
- Github Actions
- Grafana
- Prometheus

- Playwright
- Communication
- Time Management
- Critical Thinking
- Continuous Learning
- Problem-Solving

For hard skills or technical skills, one can try practicing the task on his/her own, take part in appropriate courses, read technical manuals, work with specialists of greater experience and apply various applications or methodologies. For example, in case of Quarkus it would be beneficial to create services and watch tutorials, while in relation to Kubernetes it would be helpful to set up local clusters and deploy applications for better comprehension of the subject.

And in terms of soft skills, one can aim for better communication through engagement in feedback practices, present new tools and technologies to other teams, workshops, and clear articulation of ideas and goals, while gaining experience in a commercial environment can be done through personal goal setting that matches the organization's goal, participation in multiple functional teams and being guided by more experienced professionals.

3.1.5 Ongoing Skills Management Throughout the Project

In any project, the need for ongoing skills' management throughout the project is an important factor for the success of the project and to ensure that the team has the necessary expertise to meet project objectives and overcome challenges as they arise. Periodic assessments should be done to assess if the project's requirements align with the skills of the team in project as training and development programs so that the members of the team can acquire new skills or enhance existing ones. Other factor for ongoing skills management is related to the shifting project requirements or new market conditions, the ability to quickly re-skill or up-skill the team becomes vital to staying on schedule and within budget. Finally, throughout the project, it is necessary to do a skill gap identification and mitigation to monitor the skill set available and identify gaps that might cause problem in the execution of the project [35].

3.1.6 Impact of Skills on Project Success

The success of any project is highly correlated to the individual's skills that work inside it, from project managers to team members, without exception [36]. Some of the key skills that can impact the project outcome are:

- **Technical Skills**: Technical skills are essential for performing the work that is needed successfully
- **Project Management Skills**: Planning in a general way, ensuring that the project runs smoothly on time and on budget, is an important step to the project's success.
- **Interperson Skills**: Collaboration, effective communication and conflict resolution are majors skills to work with people, this contributes to the relationship with every team

member of the team. Any wrong misstep can alter the team's environment and cause harm to the project's success.

- **Domain Knowledge**: Having experience or any sort of other means of knowledge in the industry domain enables better decision-making and consequently better solutions
- **Financial Knowledge and Time Management**: Two of the most important factors to measure the project's success is whether it is completed on schedule and within the allocated budget.

3.2 Project Planning

This part of the chapter deals with project planning, providing elements of the project charter such as stakeholders, benefits, constraints, and assumptions among others. The other topics covered include the project scope, phases, and deliverables, which include WBS, the integrated project schedule in terms of a Gantt chart that highlights scope, milestones, and costs, monitoring and controlling mechanisms, and identification and management of risks for successful project execution.

One of the most critical phases in the process management process is the project planning, this phase acts as a foundation for the project it shelf and plays a crucial role for determining the good outcome of the project's success or failure. In this chapter is going to be defined the project scope, where is outlined what is included and not included in the project, the timelines and the milestones.

The project planning phase provides a structured framework for the implementation of JEE services migration to a cloud-native architecture. Precisely defining what is to be delivered, establishing a detailed schedule, and putting in place strong monitoring and risk management practices are how this plan provides the basis upon which a successful project will be delivered within the constraints and scope set.

3.2.1 Key Elements

The stakeholders of the project are the sponsor, Ricardo Santos Ferreira, team leader of the BOS41 Department and who is responsible for the projects, João Amaral, Vítor Tavares, Carlos Araújo, Cátia Felizardo, André Santos, team members that supported the techical and fuctional aspects of the projects and finally Altice Labs itself, which provides the infrastructure, resources, and strategic direction for the project, ensuring alignment with its broader organizational goals and supporting the deployment of the new system architecture across its operations. The users would include internal engineering teams, operational teams, customer support teams and cloud management teams who will be using the Altaia platform following improvement. The project shall bring a host of benefits like improved scalability, resilience of the system, flexibility in adapting to the needs of the client, and optimization of costs by effective resource utilization. However, the project must be strictly oriented to final deliverables no later than June 30, 2025.

These key assumptions include: effective communication with the stakeholders in respect of fixing an agreed-upon timeline for key deliverables, system documentation and engineering support should be available, access to the original repository where the code is hosted and finally availability of tools and technologies required, such as: Quarkus, Kubernetes, relational databases and MinIO systems.

3.2.2 Scope of Work

The project is structured to correspond with well-defined phases of the research methodology. These phases include initiation, planning, execution, testing, validation, and close-up. In the initiation phase, the objectives, scope, and resourcing have to be defined. The planning therefore involves developing a very detailed schedule and risk management plan. Execution involves JEE service migration into cloud-native services.

The inclusion of tests and validation ensures that the deliverables are of quality, whereas its closure phase will cover such areas as the submission of documentation and final presentations. The requirements analysis and the Work Breakdown Structure (WBS) will include the migration of the services, the integration of the new services, testing and deployment, documentation and knowledge transfer. The expected results are the services code bases, the test results and the first and second delivery of the document.

3.2.3 WBS

A WBS is a project management tool used for breaking down the scope of a project into smaller elements that are easier to handle. It hierarchically decomposes the work from the overall objective to specific tasks, hence, making it easier to plan, execute, and control the project.

The key features of a WBS are the following:

- **Hierarchy**: The WBS is organized by levels. The highest level is the whole project, and the lower levels detail components and tasks.
- **Deliverables**: There is something specific that every WBS element delivers. The WBS hence focuses on delivering to meet the project objectives.
- **Uniqueness**: Each element should be unique, and there should not be any overlap between components or activities. Scope-focused: It ensures that all the work required is identified for inclusion in the planning.

The project presented in Figure 3.1 is structured into four main phases: Planning, Requirements Gathering, Development, and Closure. Each phase consists of tasks and subtasks that detail the activities necessary to achieve the project objectives.

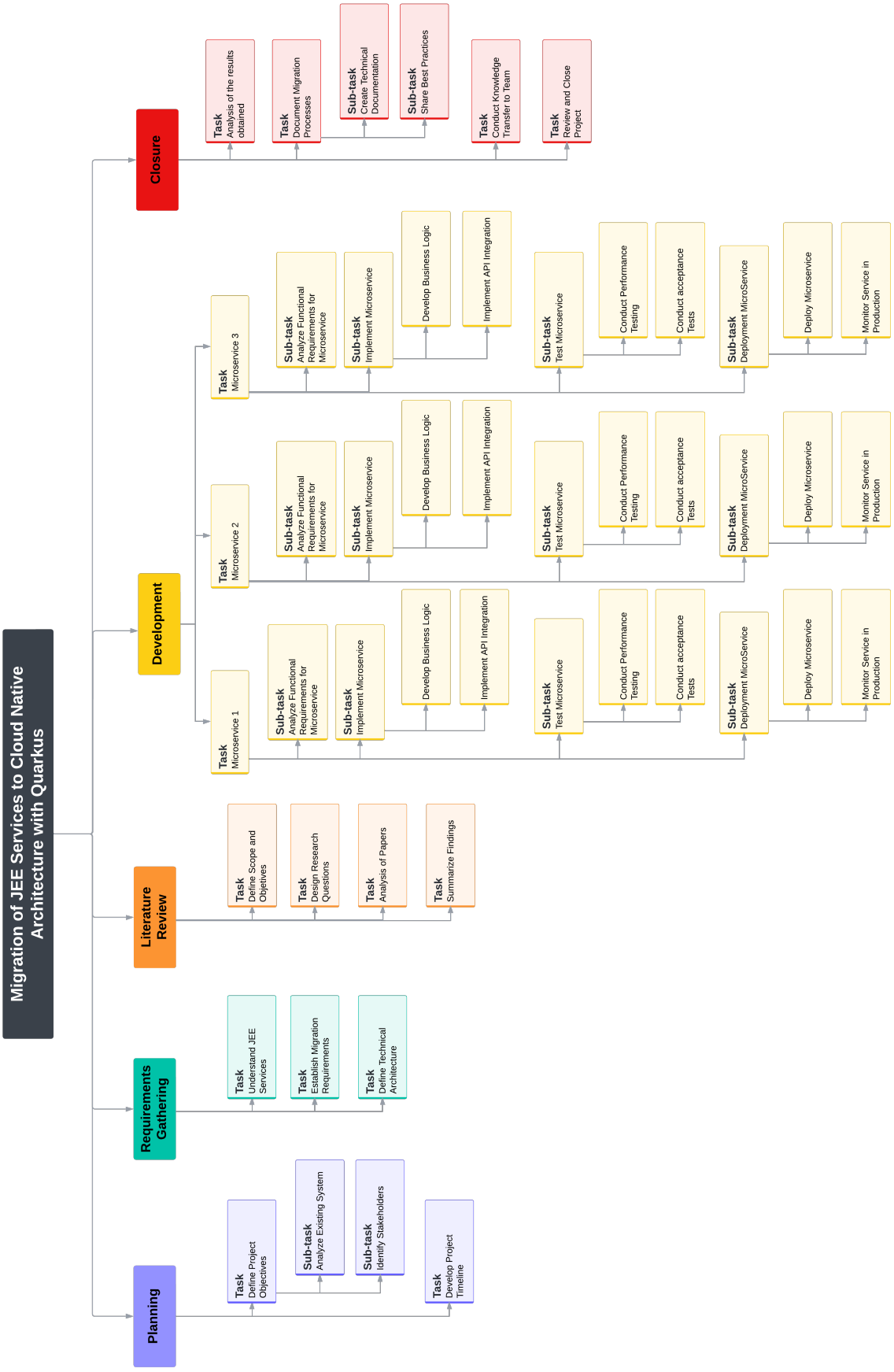


Figure 3.1: Work Breakdown Structure (WBS) for the migration project.

The planning phase, Figure 3.2, is dedicated to defining the project objectives and developing a detailed timeline. This phase includes the following tasks:

- Define Project Objectives
  - Analyze the existing system: Review the current architecture and functionalities of the system. Identify legacy components and dependencies.
  - Identify stakeholders: List all stakeholders involved in the migration process and that be affected by this process
- Develop Project Timeline: Create a detailed timeline with milestones for each phase of the project and identify key deadlines and deliverables.

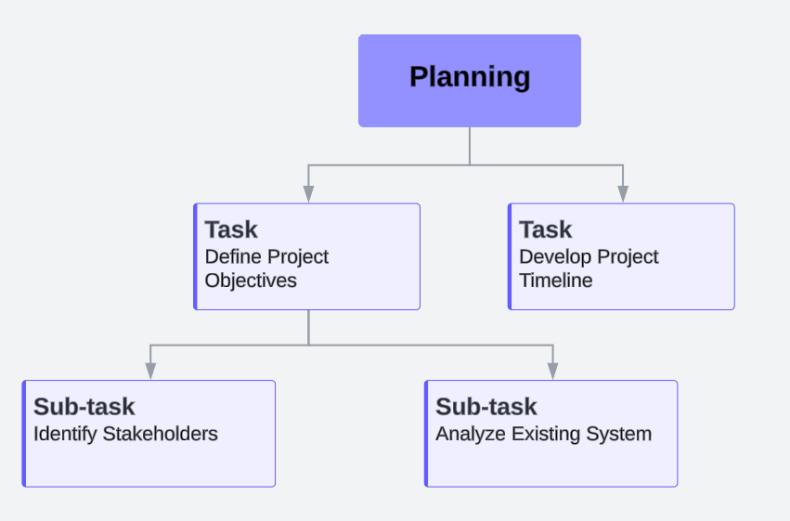


Figure 3.2: Planning Phase

The requirements gathering phase, Figure 3.3, focuses on identifying the requirements and technical architecture needed for the migration of services. The tasks include:

- Understand JEE Services: Analyze the current JEE (Java Enterprise Edition) services and document the existing system's functional and non-functional requirements.
- Establish Migration Requirements: Define goals and objectives for the migration to a cloud-native architecture. Identify constraints and risks associated with the migration. Determine the expected outcomes of the migration process.
- Define Technical Architecture: Outline the target architecture based on cloud-native principles.

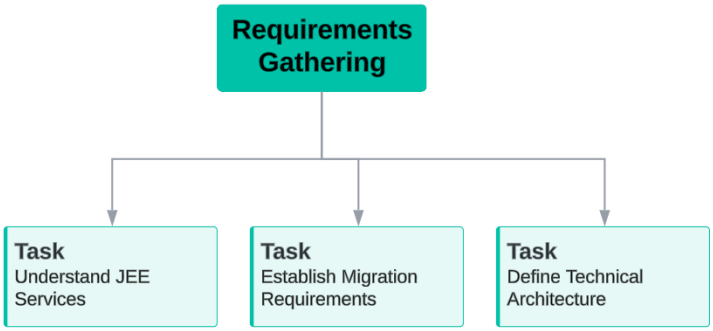


Figure 3.3: Requirements Gathering Phase

The literature review phase, Figure 3.4, outlines the key tasks involved in conducting a comprehensive review. The tasks include:

- Define Scope and Objectives: Clearly identify the boundaries of the study and the specific objectives of the literature review.
- Design Research Questions: Formulate questions that will guide the search and selection of relevant materials.
- Analysis of Papers: Critically evaluate the selected papers to extract relevant insights.
- Summarize Findings: Compile and synthesize the key findings from the analysis conducted.

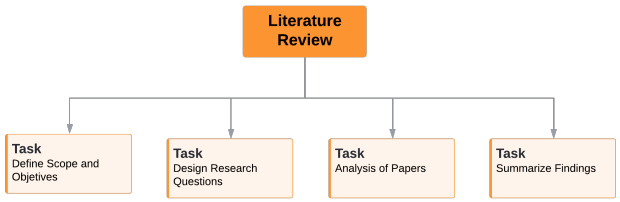


Figure 3.4: Literature Review Phase

The development phase, Figure 3.5, consists of multiple tasks aimed at the implementation and testing of services. This phase is divided into activities for three services (Document Manager, Network Qualifier and Module 360 View):

- Document Manager:
  - Analyze Functional Requirements: Review and prioritize the functionalities required for Document Manager. Define API endpoints, database schemas, and integration points.
  - Implement Service

        * Develop Business Logic: Implement core functionalities based on defined requirements. Ensure scalability and modularity in the design.
        * Implement API Integration: Design and implement REST APIs for communication. Ensure APIs adhere to security standards and best practices.
    - Test service
        * Unit Testing: Evaluate the service's response time, throughput, and resource utilization.
        * Integration Testing: Validate functionalities against user requirements and verify compliance with system specifications.
    - Deployment Service
        * Deploy Service: Set up the service in a staging/production environment. Configure CI/CD pipelines for automated deployments.
        * Monitor Service in Production: Implement monitoring tools to track performance and errors. Ensure system stability post-deployment.
- Network Qualifier:
    - Analyze Functional Requirements: Review and prioritize the functionalities required for Network Qualifier. Define API endpoints, database schemas, and integration points.
    - Implement service
        * Develop Business Logic: Implement core functionalities based on defined requirements. Ensure scalability and modularity in the design.
        * Implement API Integration: Design and implement REST APIs for communication. Ensure APIs adhere to security standards and best practices.
    - Test service
        * Unit Testing: Evaluate the service's response time, throughput, and resource utilization.
        * Integration Testing: Validate functionalities against user requirements and verify compliance with system specifications.
    - Deployment service
        * Deploy service: Set up the service in a staging/production environment. Configure CI/CD pipelines for automated deployments.
        * Monitor Service in Production: Implement monitoring tools to track performance and errors. Ensure system stability post-deployment.
- Module 360 View:
    - Analyze Functional Requirements: Review and prioritize the functionalities required for Module 360 View. Define API endpoints, database schemas, and integration points.
    - Implement service

    * Develop Business Logic: Implement core functionalities based on defined requirements. Ensure scalability and modularity in the design.
    * Implement API Integration: Design and implement REST APIs for communication. Ensure APIs adhere to security standards and best practices.
  - Test service
    * Unit Testing: Evaluate the service's response time, throughput, and resource utilization.
    * Integration Testing: Validate functionalities against user requirements and verify compliance with system specifications.
  - Deployment service
    * Deploy service: Set up the service in a staging/production environment. Configure CI/CD pipelines for automated deployments.
    * Monitor Service in Production: Implement monitoring tools to track performance and errors. Ensure system stability post-deployment.

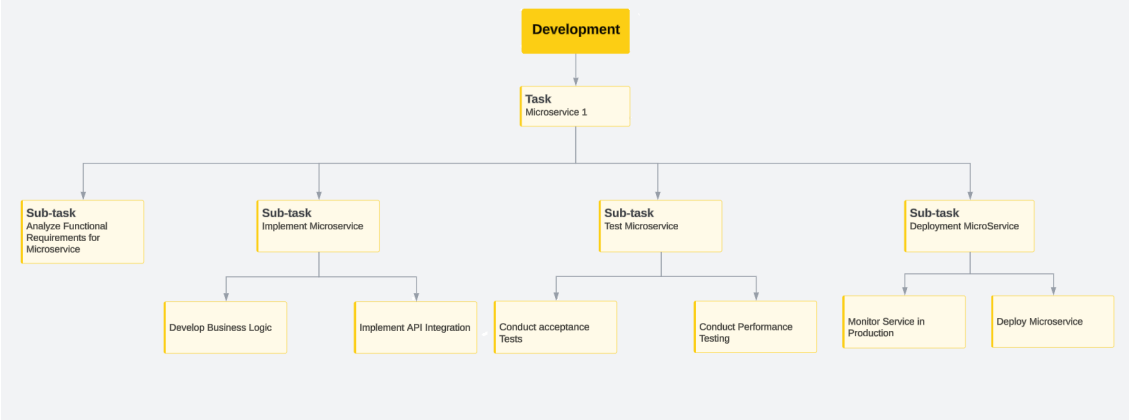


Figure 3.5: Development Phase

The closure phase, Figure 3.6, focuses on analyzing results, documentation, and knowledge transfer. The tasks include:

- Analyze the Results Obtained: Compare migration outcomes against defined goals and objectives. Prepare a report summarizing the performance and efficiency improvements.
- Document Migration Processes
  - Create Technical Documentation: Document system architecture, deployment processes, and operational guidelines.
  - Share Best Practices: Compile a list of lessons learned and best practices for future projects.
- Conduct Knowledge Transfer to the Team: Organize training sessions to familiarize the team with the new architecture. Provide documentation and resources for ongoing system maintenance.

- Review and Close the Project: Review the project timeline, deliverables, and overall success. Gather feedback from stakeholders and finalize the project closure report.

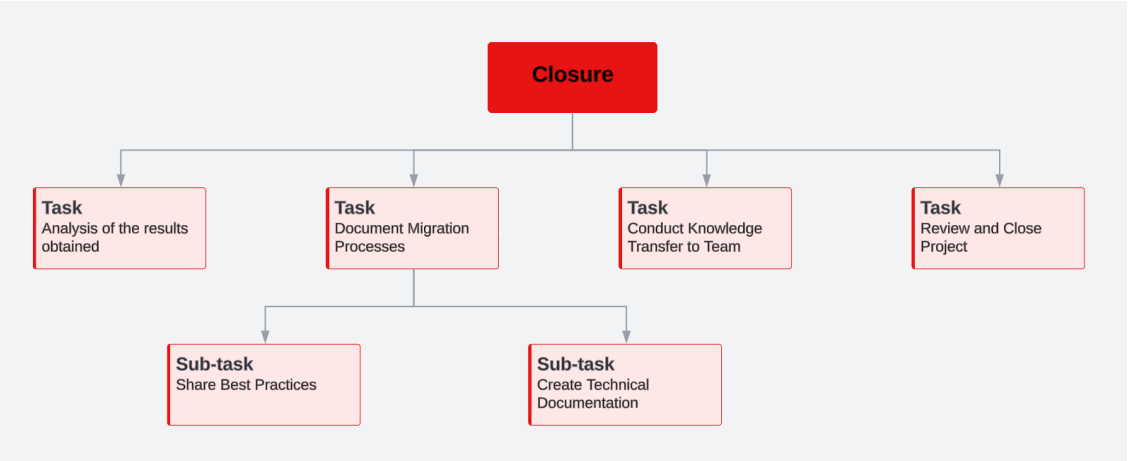


Figure 3.6: Closure Phase

3.2.4 Integrated Project Schedule

The project's key milestones were established to make sure that the progress of the project timeline stays on track. In this section is presented a Gantt chart where all the activities are presented. A Gantt chart is a graphical representation of a timeline that includes critical milestones, start and end date of each activity. This chart provides a road map for the project. Figure 3.7 and 3.8, represent detailed timelines for PREPD and DIMEI, respectively. Additional charts with the milestones and detailed task breakdowns are available in the annex chapter, specifically Figures A.1, A.2, A.3 and A.4.

Task 1.1 - PREPD	47 dias?	Sex 01/11/24	Sáb 04/01/25		Tiago
Task 1.1.1 Project Inicialization	5 dias	Sex 01/11/24	Qui 07/11/24		Tiago
Task 1.1.2 Investigation	47 dias	Sex 01/11/24	Sáb 04/01/25		Tiago
Task 1.1.3 Development Microservice 1	46 dias?	Sex 01/11/24	Sex 03/01/25		Tiago + ALB
Task 1.1.4 Working on Skills	46 dias	Sex 01/11/24	Sex 03/01/25		Tiago
Soft Skills	25 dias	Sex 01/11/24	Qui 05/12/24		Tiago
Hard Skills	46 dias	Sex 01/11/24	Sex 03/01/25		Tiago

Figure 3.7: Gantt chart dissertation project related to PREPD

Task 1.2 - DIMEI	125 dias	Seg 06/01/25	Sex 27/06/25	2	Tiago
Task 1.2.1 Development Microservice 2	35 dias	Seg 06/01/25	Sex 21/02/25		Tiago + ALB
Task 1.2.2 Development Microservice 3	70 dias	Seg 24/02/25	Sex 30/05/25	47	Tiago + ALB
Task 1.2.3 Closure	20 dias	Seg 02/06/25	Sex 27/06/25	57	Tiago

Figure 3.8: Gantt chart dissertation project related to DIMEI

The Agile methodology would be recommended for this project because it supports iterative development and adaptability, which agrees well with the complexity in migrating JEE services to a cloud-native architecture using Quarkus. Here is why Agile would be the best fit for your project, and why other methodologies such as Waterfall, Scrum alone, or Kanban alone wouldn't be ideal:

Why Agile?

- **Flexibility and Adaptability**: Migration of legacy systems is often plagued by unforeseeable problems. Agile allows process adjustments whenever new issues or opportunities arise; its short development cycles, or sprints, provide frequent opportunities to iteratively refine approaches and adapt requirements.
- **Incremental Deliveries**: Agile allows the delivery of smaller, functional parts of the system, like migrated services or new integrations, much more frequently. This ensures that the project shows continuous value addition and keeps the stakeholders actively engaged.
- **Early Feedback Loops**: Agile encourages frequent interaction with the stakeholders and testing continuously during the development. This reduces the possibility of major errors remaining unnoticed for a long period and allows ample opportunity for early course corrections.
- **TDD Support**: Agile practices and the focus on testing go perfectly with tools like Playwright and Quarkus to ensure quality is brought into every stage of development.

Why not Other Methodologies?

- **Waterfall**: Waterfall is too rigid for projects involving unknown challenges because it can't easily handle changes. Requirements are to be fixed very early, which is practically impossible in migration projects, where unexpected issues usually pop up. Also, testing in Waterfall is usually done at the end, which means there is an increased risk of finding critical errors late, resulting in costly rework. Further, long feedback cycles in Waterfall mean stakeholders see results only at the end, which leaves little room for adjustments during development.
- **Scrum Alone**: While Scrum gives way to team autonomy, using it in isolation from other Agile practices may result in fragmented efforts. The complexity of a migration project requires more structured approaches in order that there is no discontinuity in integration and delivery. Moreover, the focus on sprints in Scrum can lead to neglecting Kanban-like workflows, which balance testing and system integration efforts evenly.
- **Kanban Alone**: Kanban does not have well-defined milestones, which may cause delays in making progress with complex and interdependent tasks, such as database migration or integration of services. Moreover, a focus on flow and reduction of waste in Kanban does not inherently support the long-term planning required for a strategic migration project.

Implement an Agile Hybrid approach by adding Scrum to have organized iterations with Kanban for better workflow control. This integrated approach offers adaptability but continues to emphasize delivering high-quality, validated increments of the project. Steer clear of the Waterfall methodologies that are too rigid, or Kanban only that does not bring in a structured way of milestone planning, since they do not match and mold with the changing dynamics and technical complexities of your migration project.

3.2.5 Project Execution

The actual executing out of the planned activities specified in the WBS and in the Gantt charts is referred to as the project execution phase. In this section, the project's execution timeline is summarized.

Most of the planned tasks were completed within the anticipated time frames during the project's duration. Technical limitations, resource availability, or the necessity to redefine some requirements during the migration process, however, caused slight delays in some activities.

The delays were mostly constrained to the development of the services. These delays were largely due to:

- Unexpected complexity in decoupling legacy dependencies led to adjustments
- Temporary unavailability of access to test environments or internal resources
- The need to redefine and clarify requirements for specific services after integration challenges emerged.

All tasks were finished by the project's final deadline of June 30, 2025, in spite of these delays. Effective planning, risk management, and adaptation techniques to address practical technical and operational challenges have been demonstrated by the project's execution, which has stayed relatively close to its scope.

3.2.6 Monitoring and Controlling Procedures

In addition, mechanisms for monitoring and controlling will be provided to monitor the performance of the project to ensure that it is completed on time. The performance evaluation will focus on adherence to the project schedule, quality of deliverables, and compliance with the budget. Tools like JIRA for task tracking, and Grafana will be vital tools used in monitoring performance. Progress meetings will be held routinely to discuss and address any challenges that come up. The weekly progress meetings will discuss the issue closure and task completion status.

3.2.7 Risk Identification and Management

Some identified risks in the project include delayed timelines due to integration challenges of such services, unavailability of tools or technologies to effect it. These are mitigated by undertaking reviews on a weekly basis. Additionally, clear communication among team members and stakeholders is maintained to promptly address any unforeseen issues. During the execution of the project to migrate JEE services to a cloud-native architecture using Quarkus, various potential risks were identified that could negatively impact the progress and expected outcomes. Below are the main risks analyzed in detail:

Risk 1: Services Integration Failure Due to Compatibility Issues

- **Description:** Failure in integrating migrated services with existing systems.
- **Cause:** Incompatibility between services and APIs.
- **Effect:** Delays in project delivery due to necessary rework.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 1.
- **Impact (1-5):** 4.
- **PI Score:** 4.

- **Expected Result, No Action:** Services fail to communicate effectively, causing schedule delays.
- **Risk Response Type:** Mitigate.
- **Response Description:** Conduct integration tests after each migration and use automated validation tools.

Risk 2: Kubernetes or Cloud Infrastructure Misconfiguration

- **Description:** Misconfigured infrastructure results in instability.
- **Cause:** Complexity of manual setup or lack of automation.
- **Effect:** Instability in production services.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 2.
- **Impact (1-5):** 5.
- **PI Score:** 10.
- **Expected Result, No Action:** Services become unavailable, affecting end-users.
- **Risk Response Type:** Mitigate.
- **Response Description:** Automate configurations using tools like Terraform and perform validation tests.

Risk 3: Non-Compliance with GDPR Privacy Regulations

- **Description:** Sensitive data is compromised during the development process.
- **Cause:** Inadequate management of sensitive data.
- **Effect:** Significant fines and reputational damage.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 1.
- **Impact (1-5):** 4.
- **PI Score:** 4.
- **Expected Result, No Action:** The organization faces legal and financial penalties for non-compliance.
- **Risk Response Type:** Transfer.
- **Response Description:** Implement data masking tools and regular audits to ensure GDPR compliance.

Risk 4: Communication Failures Between services

- **Description:** Services fail to communicate reliably.
- **Cause:** Poorly implemented APIs or inconsistent contracts.
- **Effect:** Unavailability of critical functionalities.
- **Risk Owner:** Team Leader.

- **Probability (1-5):** 2.
- **Impact (1-5):** 4.
- **PI Score:** 8.
- **Expected Result, No Action:** System failures affect overall application performance.
- **Risk Response Type:** Mitigate.
- **Response Description:** Use tools like Pact to validate API contracts and implement continuous integration.

Risk 5: Inadequate Service Scalability

- **Description:** Services fail to handle increasing loads.
- **Cause:** Improper auto-scaling configuration.
- **Effect:** Performance degradation and user dissatisfaction.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 3.
- **Impact (1-5):** 5.
- **PI Score:** 15.
- **Expected Result, No Action:** Services slow down under high demand.
- **Risk Response Type:** Mitigate.
- **Response Description:** Configure auto-scaling policies in Kubernetes and conduct load tests.

Risk 6: Excessive Dependence on a Critical Service

- **Description:** The system heavily relies on a single critical service.
- **Cause:** Poorly distributed architecture without adequate redundancy.
- **Effect:** Total system downtime if the critical service fails.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 2.
- **Impact (1-5):** 5.
- **PI Score:** 10.
- **Expected Result, No Action:** Complete unavailability of the system during a critical failure.
- **Risk Response Type:** Mitigate.
- **Response Description:** Ensure redundancy and implement fallback mechanisms for critical services.

Risk 7: Obsolescence of Tools or Technologies Adopted

- **Description:** Rapid technological advancements render adopted tools outdated.

- **Cause:** Lack of future-proof planning in tool selection.
- **Effect:** Difficulty in maintaining or expanding the project.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 2.
- **Impact (1-5):** 3.
- **PI Score:** 6.
- **Expected Result, No Action:** The project becomes harder to maintain as technology evolves.
- **Risk Response Type:** Accept.
- **Response Description:** Monitor technological trends and schedule regular updates for critical tools.

Risk 8: Security Vulnerabilities in services

- **Description:** Security vulnerabilities compromise sensitive data or system functionality.
- **Cause:** Misconfigured security settings or vulnerable third-party libraries.
- **Effect:** Unauthorized access, data theft, or system breaches.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 3.
- **Impact (1-5):** 5.
- **PI Score:** 15.
- **Expected Result, No Action:** Significant data breaches result in fines and reputational damage.
- **Risk Response Type:** Mitigate.
- **Response Description:** Conduct regular security audits and implement tools like OWASP for vulnerability detection.

Risk 9: Delays in Learning New Tools by the Team

- **Description:** The team requires more time than planned to learn new tools and frameworks.
- **Cause:** Limited experience with tools like Quarkus and Kubernetes.
- **Effect:** Delay in task completion and project timeline extensions.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 2.
- **Impact (1-5):** 3.
- **PI Score:** 6.
- **Expected Result, No Action:** Team performance remains low, leading to project delays.

- **Risk Response Type:** Mitigate.
- **Response Description:** Provide training sessions for the team before the start of implementation.

Risk 10: Failures in Monitoring and Observability

- **Description:** Insufficient monitoring makes it difficult to diagnose and resolve issues.
- **Cause:** Misconfiguration of tools like Grafana and Prometheus.
- **Effect:** Production issues remain unresolved, leading to service degradation.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 3.
- **Impact (1-5):** 2.
- **PI Score:** 6.
- **Expected Result, No Action:** Issues in production persist unnoticed, causing customer dissatisfaction.
- **Risk Response Type:** Mitigate.
- **Response Description:** Configure appropriate dashboards and alerts for continuous monitoring of services.

Risk 11: Unexpected Costs When Using Cloud Infrastructure

- **Description:** Costs exceed the planned budget due to unmonitored usage.
- **Cause:** Inefficient configurations or lack of resource allocation policies.
- **Effect:** Budget overruns make it difficult to sustain the project.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 2.
- **Impact (1-5):** 4.
- **PI Score:** 8.
- **Expected Result, No Action:** Cloud costs spiral out of control, exhausting financial resources.
- **Risk Response Type:** Transfer.
- **Response Description:** Implement cost monitoring tools and simulate resource usage before deployment.

Risk 12: Compatibility Issues with Library Versions

- **Description:** Outdated or conflicting dependencies cause unexpected bugs.
- **Cause:** Use of incompatible library versions in the project.
- **Effect:** Errors and instability in the system.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 3.

- **Impact (1-5):** 4.
- **PI Score:** 12.
- **Expected Result, No Action:** System instability leads to delays and user dissatisfaction.
- **Risk Response Type:** Mitigate.
- **Response Description:** Automate dependency validation and run regression tests before deployment.

Risk 13: Lack of Access to Realistic Test Data in Development

- **Description:** Testing is incomplete due to unavailable or restricted production data.
- **Cause:** Data access policies or limitations in test data availability.
- **Effect:** Bugs go undetected, leading to production failures.
- **Risk Owner:** Team Leader.
- **Probability (1-5):** 1.
- **Impact (1-5):** 4.
- **PI Score:** 4.
- **Expected Result, No Action:** Undetected issues in testing result in production problems.
- **Risk Response Type:** Mitigate.
- **Response Description:** Create synthetic datasets that mimic real production data for development environments.

Chapter 4

Literature Review

This writing discusses the methodology adopted for the research and relevant literature on migrating JEE services to Cloud Native solution. The research design and data collection methods were covered along with Quarkus, Kubernetes, and more technologies. This chapter covers the systematic process and the technological context. It draws attention to how the tools help in improving a system's scalability, resilience and efficiency while transforming a monolithic system into a microservice system.

4.1 Research Design and Approach

This research follows a mixed-methods design with qualitative and quantitative strands. On the qualitative side, a systematic literature review was conducted to ground the work in existing knowledge and answer theoretical research questions. On the quantitative side, an empirical case study was conducted by migrating JEE applications belonging to the Nossis suite to a cloud-native environment with Quarkus and quantifying the outcomes. This dual approach is justified to offer both a deep understanding of the problem domain and practical validation of proposed solutions. The research questions (RQs) capture this combination: they deal with migration issues and best practices (RQ1, RQ3 – more qualitative) and Quarkus' role in implementation (RQ2 – technical) and the case study provides empirical responses to them. Through the use of two methods, the research gains both scope and depth, the systematic literature review delivers rigor and coverage of current work, and the case study delivers contextual richness and measurable outcomes. Such triangulation increases the validity of results, since literature findings can be confirmed or exemplified by experimental migration results. Overall, the research design is appropriate for an applied software engineering dissertation, where understanding and improving a real process requires theoretical research and experimentation.

4.2 Systematic Literature Review Using PRISMA Framework

A systematic review was made in the literature to give a sound theoretical basis for the migration of JEE services into cloud-native architecture. This systematic review was conducted accordingly with the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) framework, which guarantees completeness and transparency in the processes of identification, screening, and synthesis of the research findings.

Searches have been conducted on IEEE Xplore, ACM Digital Library, Google Scholar, ResearchGate, SpringerLink, MDPI, arXiv and Elsevier. A database aggregator called b-on was used to help with the search. It has information on the various databases mentioned

above, i.e. it is possible to access articles from these databases on a single platform, but it is also possible to use this site to facilitate the search, since the platform identifies and eliminates repeated publications that may exist, since several articles may have been published by various publishers.

Following the PRISMA guidelines, a four-step literature review was conducted:

- Identification: There were extensive searches on main scholarly databases using synonymous keywords for JEE monolith migration, microservices, Quarkus, and cloud-native architecture, resulting in hundreds of preliminary hits.
- Screening: Duplicate entries were removed and the title/abstract was reviewed to exclude non-pertinent work.
- Eligibility: Full texts of the remaining papers were assessed against pre-specified inclusion/exclusion criteria. Non-empirical or tangentially related studies were excluded.
- Inclusion: A final filtered list of relevant studies was selected for each research question.

To determine the validity and quality of the studies incorporated, clear inclusion and exclusion criteria were defined and applied consistently during the review process. The criteria were tailored to the focus of each research question (RQ).

Research studies were considered for inclusion based on the following criteria:

- Discussing the migration of legacy monolithic systems into microservices architecture, in particular those that reported challenges, solutions, or re-architecting strategies.
- Resolved technical problems such as integration, scalability, or performance during the migration.
- Evaluated or showcased Quarkus in a cloud-native or microservices environment (such as feature analysis, performance benchmark, or case studies).
- Described best practices, methodologies, or design patterns for microservices migration (such as reviews, surveys, or industry expert reports).
- Were of empirical taste (such as experiment, benchmark, case study) or coming from reputable industry sources, preferably published in the past 3–5 years to track recent trends.
- Published in English and freely available.

Studies were to be excluded if:

- Did not produce valuable information concerning the study questions (e.g., the generic microservices debate without meaning in regard to legacy migration)
- Based exclusively on theory or not having any empirical foundation (e.g., lacking implementation, assessment, or case study).
- Touched upon tangent topics (e.g., performance tuning for microservices independent of legacy or Java-based stacks).
- Were published in any other language except English or unavailable through scholarly databases.

With the aid of these criteria, an informed and quality body of literature was gathered in order to provide the basis of the research of JEE to cloud-native migration, the case for Quarkus, and microservice best practice.

From each study sampled, pertinent data were systematically abstracted from an agreed template. Key information taken included:

- Study context (industry project, experimental study),
- Techniques or frameworks proposed or evaluated (migrating methods, performance assessments),
- Main findings and conclusions, especially those related to the prescribed RQs.
- Qualitative findings in terms of success factors, challenges, and best practices were also recorded.

Due to the diverse and primarily qualitative nature of the literature, a narrative synthesis approach was taken rather than a meta-analysis. The results were grouped thematically under each research question:

- RQ1 – Migration Challenges and Solutions: Studies were analyzed in attempting to identify repeated issues with making the transition from monolithic to microservices systems. They were categorized under elements such as integration, scalability, and performance. Similar solutions as well as mitigating strategies were also synthesized within the literature surveyed.
- RQ2 – Quarkus in Microservices: Literature regarding the use of Quarkus was researched to assess its technical merits and comparative advantages. Topics addressed included its cloud-native features, performance tests, memory usage, boot times, and compatibility with the Java ecosystem.
- RQ3 – Migration Best Practices: Migration best practices were derived from both academic and industry sources. They included best practices for service decomposition, planning and risk mitigation, architectural principles (e.g., domain-driven design), automation tools, and monitoring practices.

4.3 Characteristics of Research Questions

A good research question is supposed to be clear, in what exactly it seeks to investigate. It is also required to be broad enough without being excessive or, on the contrary, excessively thinned down, since any of the two pitfalls would compromise the analysis or relevance of the content. It should further be challenging such that it promotes analysis in depth and avoids answers that might be too simple, for example, "Yes" or "No". However, it should be answerable within the allotted time and resources. It should be researchable, so that the question may draw from a quality rich field of material, such as academic books and peer-reviewed material, to ground the research. Lastly, it should be analytic rather than just descriptive so that it is open to critical analysis [37].

4.4 Research Questions

Carefully formulated research questions have been identified that help in exploring and analyzing the migration from monolithic to microservices architecture. The following set of

questions forms the very basis of this study, as it guides the scope and objectives by addressing the critical aspects of the migration. In this context, the study systematically reviews best practices, challenges, and application performance, scalability, and maintainability impacts of the different strategies for migration. The following research questions summarize its key areas of interest and give a clear overview for the realization of the objectives:

- RQ1 — Migrating a legacy monolithic system to microservices involves re-architecting and addressing integration, scalability, and performance challenges?
- RQ2 — How does Quarkus facilitate the implementation of microservices architectures?
- RQ3 — What are the best practices for migrating an application from monolithic to microservices?

4.5 Literature Review Methodology

This section addresses the research questions defined, presents the research process, and discusses the reviewed literature, the discussion is fundamentally based in the in-depth investigation and analysis of the literature, industry standards, and valuable information accumulated throughout the research process.

4.5.1 Research Question 1

This research question seeks to investigate the pitfalls and best practices involved in migrating legacy monolithic systems to microservices architectures, in most cases, migration involves navigating technical challenges such as rearchitecting legacy systems, combining diverse services, and ensuring the conditions for scalability and performance improvement. By identifying factors that lead to successful migrations, this study will be very important to both academia and industry. To ensure effectiveness and rigor, a set of inclusion and exclusion criteria has been applied to guide the selection of research studies. The inclusion criteria were carefully defined to ensure that the studies to be reviewed provide answers relevant to the research question. These criteria are focused on research that really provides measurable and useful information on the transformation from monolithic systems to microservices architectures, dealing with critical challenges and identifying good practices. The inclusion criteria are listed as follows:

- Studies focusing on the migration of legacy monolithic systems to microservices architectures.
- Research addressing challenges in rearchitecting, integration, scalability, and performance during migration.
- Empirical studies, case studies, and industry reports published in peer-reviewed journals or reputable conferences.

Exclusion criteria have been designed to remove studies that do not directly contribute to the research objectives or do not yield important insight into the migration process. The following criteria ensure a focused and high-quality body of research to be reviewed:

- Studies not directly related to system architecture migration.
- Articles lacking empirical data or practical insights.

- Publications not available in English.

The following search terms were constructed systematically and used to identify relevant studies for this research. They have been carefully selected to include key concepts, technical challenges, and best practices on how to migrate legacy monolithic systems into microservices architectures. By combining a range of terms and phrases, the search aimed to accommodate a broad scope of studies from different sources. The search terms include:

- "monolithic to microservices migration"
- "legacy system re-architecting"
- "scalability challenges in microservices"
- "performance optimization in microservices architecture"

An initial search across IEEE Xplore, SpringerLink, MDPI, arXiv, and B-On yielded numerous articles. For RQ1, the initial search yielded 250 articles. After deletion, of 30 duplicitous records with the same study which is appearing in the database, 220 unique articles were left for screening. In the title/abstract screening stage, 170 articles were excluded for being off-topic (e.g. not specifically about monolith to microservices migration), not being in English, and/or lacking empirical content (e.g. opinion papers without data). This left 50 articles for full-text review. Upon further screening of the full texts, 30 were excluded as many did not address the challenges of integration, scalability or performance of migration in sufficient detail (some focused on microservices generally not legacy migration challenges). As a result, a total of 20 articles were selected for RQ1.

4.5.1.1 Results

In this section, it is synthesized the key findings from the selected studies mentioned in the previous section.

Migration to microservices architecture from the monolithic legacy systems means core architectural and operational changes. Such transformation aims at making use of modularity, scalability, and agility in microservices while preserving the existing capabilities, yet it opens sources of challenges with regard to integration, scalability, and performance. Next, it is listed each category of challenges with strategies to tackle each.

Integration challenges arise because there are intrinsic differences between how monolithic and microservices architectures are designed to handle data, state, and especially requirements specific to a tenant. Monolithic systems have a centralized and tightly coupled design, hence microservices have a decentralized and distributed architecture. In microservices, each is an independent service, hence, all issues regarding multitenancy and state management become very important to handle.

- **Multitenancy and Statefulness**: Migration to microservices involves dealing with multitenancy, supporting different organizations or tenants using shared services but meeting diverging needs, statefulness management is also crucial to ensure that the migration does not result in failures of availability and reliability.
  - Solution: Stateless service design and session management patterns can be used to reduce these complexities, increasing resilience and scalability [38].
- **Data Consistency**: The shift from centralized monolithic databases to microservices decentralized data stores makes it difficult to ensure data consistency.

  - Approach: Distributed transaction patterns such as the Saga pattern, along with event consistency mechanisms, can be followed in order to achieve data integrity across services [39].

When systems move from monolithic architecture, where scalability is usually done vertically, to microservices architecture, which can scale horizontally, scalability challenges arise. However, although microservices provide the potential of fine-grained scalability, their distributed nature brings complexities related to workload distribution, resource management, and infrastructure utilization in the system.

- **Horizontal and Vertical Scalability**: Microservices become especially useful for systems in which horizontal scalability is required; individual services can scale independently. However, for applications with limited concurrency demands or where vertical scaling suffices, the benefits of scalability may not be that visible.
  - Consideration: Detailed analysis of the workload pattern of the system is necessary to evaluate the feasibility and benefits of migration to microservices [40] [41].
- **Cloud-Native Infrastructure**: Containerization and orchestration provide a strong foundation for microservices on cloud platforms, which reduces network overhead and improves efficiency in communications. However, distributed database management and consistency of data still remain highly challenging.
  - Solution: The orchestration with Kubernetes and cloud-native databases can simplify the operation and solve some of those issues [39].

The performance challenges of microservices come from the fact that they are naturally distributed. While microservices bring the benefits of fault isolation and independent scaling, they might also introduce added latency and resource overhead due to inter-service communication and orchestration requirements. Understanding these trade-offs is important for tuning a system's performance at each migration step and after migration has finished.

- **Performance Trade-offs**: In the case of single-machine deployments, monolithic systems usually show better performance because there is lesser inter-process communication. That aside, microservices have an upper hand in distributed environments, providing both scalability with fault tolerance. Note that too much scaling can be at the cost of performance degradation.
  - Approach: Using optimized inter-service communication protocols like gRPC or message brokers, may help mitigate such trade-offs, together with performance monitoring tools [40] [41].
- **Resource Overhead**: The typical microservices architecture needs more computational resources due to increased overhead in maintaining isolated service environments.
  - Approach: Lightweight containerization, coupled with efficient orchestration strategies, contributes to the optimization of resource utilization while maintaining performance [41].

4.5.2 Research Question 2

The question being researched now aims at the extent to which Quarkus, a cloud-native framework, achieves the effective realization of microservices architecture. The study places much emphasis on the distinctive features, improved performance, and real-world applications of Quarkus that enable smooth development, deployment, and scaling of microservices—trying to explore the role of Quarkus in modernizing application development by examining empirical data and case studies. To enable focused scrutiny, a set of inclusion and exclusion criteria are put in place. The inclusion criteria have been set carefully to ensure the selection of studies that contribute significantly and practically to the understanding of Quarkus' role in establishing microservices architectures. These criteria aim to exclude non-relevant material while emphasizing studies that investigate, in-depth, the unique features and benefits of Quarkus. More specifically, the inclusion criteria stress the following:

- Studies and articles focusing on Quarkus in the context of microservices implementation.
- Research discussing Quarkus features, performance benchmarks, and case studies.
- Publications from reputable sources, including peer-reviewed journals, conferences, and authoritative web articles.

The exclusion criteria have been methodically developed in order to exclude studies that do not contribute directly to the research objectives or that will not produce meaningful and applicable findings. This process ensures a focused review of relevant and good-quality literature. More specifically, the exclusion criteria were set out to:

- Studies not directly related to Quarkus or microservices.
- Articles lacking empirical data or practical insights.
- Publications not available in English.

In order to systematically identify relevant studies that contribute to the objectives of the research, a search set of terms was constructed carefully. The terms chosen should cover several different dimensions of Quarkus: its unique features, performance analysis, and real-world applications in microservices. The search terms used include the following:

- "Quarkus microservices implementation"
- "Quarkus performance in microservices"
- "Quarkus features for cloud-native development"
- "Quarkus integration with Kubernetes"

An initial search across IEEE Xplore, SpringerLink, MDPI, arXiv, and B-On yielded numerous articles. RQ2 investigated the Quarkus framework with a more narrow search. In the beginning, we identified 40 papers utilizing Quarkus for microservices. After eliminating 4 duplicates, 36 unique records remained. We excluded 26 articles from the analysis based on screening of the title and abstract. The incidents were either off-topic. For instance, papers on Quarkus but not on the implementation of microservice architectures. Brief mentions of a tool without an evaluation qualified as a low-quality article. This led to 10 articles eligible for full-text analysis. After doing the full-text screening, 4 more papers were left out – usually, because they do not give concrete evidence of how Quarkus enables microservices (they mention Quarkus only in passing for instance or they do not have an empirical evaluation

of Quarkus in a microservice context). In the end, six articles were included that took RQ2 directly into account.

4.5.2.1 Results

In this section, it is synthesized the key findings from the selected studies mentioned in the previous section.

Quarkus is the next-generation framework for building microservices architectures, specially designed to be optimized in the cloud, it comes with a lot of features and benefits, making it a first choice for developers wanting to build fast, scalable, and cloud-native applications.

The Quarkus framework brings with it many features and advantages, making it one of the most popular frameworks for developing modern applications, it was designed with efficiency, scalability, and easily accessible integration within cloud-native environments to let developers build applications that are both efficient with resources and highly effective. Using its innovative capabilities, Quarkus enables organizations to optimize their microservices deployments while addressing the challenges of modern software development. Here are the major features and benefits showing why Quarkus is a popular choice for developing robust and scalable microservices:

- **Fast Startup and Low Memory Footprint**: Quarkus is designed to boot up fast and consume low memory. These two features are really important for microservices, which are usually scaled out horizontally and need to run efficiently in resource-restrained environments like containers or serverless platforms. Research proves that Quarkus can save up to 80% of memory and 95% of CPU compared with frameworks like Spring Boot, ensuring resource efficiency and cost-effectiveness [25] [42] [43].
- **Cloud-Native Design**: Designed from the ground up on the principles of cloud-native design, Quarkus seamlessly integrates with cloud services and infrastructure. This allows developers to fully take advantage of the modern cloud capabilities, offering elasticity and scalability, thus making Quarkus well-suited for dynamic cloud deployments [44].
- **Better System Performance**: Quarkus provides significant performance improvements over traditional frameworks and hence enables faster deployment and higher scalability. For example, it enhances the system uptime by 119%, provides fast startup times, and hence enables the rapid response to variable demands in microservices systems [43].
- **Standards-Based Development**: With strong support for standards-based microservice development, Quarkus allows for easier migration of existing services and guarantees compatibility with other systems and technologies. It leverages well-known Java libraries and tooling, so developers can rapidly build, configure, and manage microservices [42].
- **Integration with the Java Ecosystem**: As part of the mature Java ecosystem, Quarkus benefits from a wealth of libraries and tools but offers modern optimizations for microservices architectures. This combination provides a good balance between innovation and reliability, which is appropriate in enterprise-class software development [42] [44].

- **Comparative Advantage**: Comparative studies always put Quarkus at the top among the most popular frameworks like Spring Boot and Micronaut. Computational efficiency, memory utilization, and deployment speed give reasons to developers and software architects for the choice of Quarkus [43] [25].

Quarkus stands out in the world of microservices by offering incredibly low startup times coupled with very low resource consumption, plus cloud-native capabilities. The integration of Quarkus into the Java ecosystem and demonstrations of its resource-using efficiency make it enormously appealing to developers who have to develop scalable yet cost-efficient microservices. With Quarkus, organizations could further optimize their microservices deployments for modern cloud environments to ensure strong performance and reliability.

4.5.3 Research Question 3

The research question here seeks to identify the best practices for migrating applications from monolithic architectures to microservices frameworks. Mainly, this research aims at evaluating various strategies, methodologies, and empirical case studies that provide practical knowledge on how to migrate effectively. This study aims to synthesize accumulated knowledge from academic literature and industry practices in order to develop a framework that can help organizations navigate the migration process. Inclusion and exclusion criteria were established to identify high-quality studies relevant to the research question. The inclusion criteria should be set in such a manner that they allow only studies that offer information concerning the migration process. Such criteria are to select research works whose knowledge will apply directly in approaches and methodologies pertaining to the migration into microservices architectures. These criteria include:

- Studies and articles focusing on best practices for migrating from monolithic to microservices architectures.
- Research discussing strategies, methodologies, and case studies related to the migration process.
- Publications from reputable sources, including peer-reviewed journals, conferences, and authoritative web articles.

Exclusion criteria had therefore been clearly outlined so that studies which either did not fall within the scope of this research or those whose contribution to the understanding of the migration process was less are excluded. The process helps preserve focus and integrity of the research by allowing comprehensive analysis of the literature relevant and influential in nature. The established criteria aim to exclude those studies that do not align with the research objectives or do not contribute to the understanding of the migration process. Specifically, the following are considered:

- Studies not directly related to migration strategies or best practices.
- Articles lacking empirical data or practical insights.
- Publications not available in English.

In order to achieve this, a systematic and comprehensive search strategy is employed, including carefully developed search terms designed to include a wide and diverse scope of perspectives related to best practices of migration from monolithic to microservices architectures. Search terms are designed to represent all aspects of the migration process,

including technical challenges, strategies, methodologies, and case studies. Combining general terms with specific phrases and contextually relevant keywords, the search attempted to ensure that it captured all studies that would answer the research questions. Search terms include but are not limited to:

- "best practices for migrating monolithic to microservices"
- "monolith to microservices migration strategies"
- "microservices migration methodologies"
- "case studies on microservices migration"

An initial search across IEEE Xplore, SpringerLink, MDPI, arXiv, and B-On yielded numerous articles. To find evidence for RQ3, studies discussing best practices, guidelines and lessons learned for migration were searched. The initial search returned 180 articles. After 20 duplication removal, we had 160 unique articles to screen. Title/abstract screening excluded 120 articles outside the scope (papers on microservices which did not deal with the best practices for migration or generally technical discussions without migration guidance). At this stage we removed all non-English articles and those that did not add anything of substance, e.g. not empirical or practical. This screening left 40 candidate articles. In the full text review, 22 articles were excluded as the articles have not actually proposed any best practices for migration. Specifically, many articles were merely theoretical or were not focused on strategies of migration. A few other articles only mentioned migration in passing without proposal of guidelines. In the end, 18 articles were included in the review for RQ3 that presented some best practices or recommendations for moving from a monolith to microservices.

4.5.3.1 Results

In this section, it is synthesized the key findings from the selected studies mentioned in the previous section.

The refactoring of an application from a monolithic architecture to a microservices framework involves several best practices that help to make the migration seamless. In an effort to modernize their software infrastructures and systems, organizations are turning to a microservices architecture as a strategic decision to deliver greater scalability, maintainability, and operational adaptability. However, there are several uncertainties involved in this migration, an effective change requires careful planning, organized implementation, and the use of existing methodologies. Organizations should use the most updated tools and methodologies, so that they can overcome these challenges easily, ensuring that their systems are ready to meet the demands of contemporary software landscapes.

Complete understanding of the existing system is needed to initiate application migration. This is a very critical step in both the technical and business dimensions of the application and is used in discovering major opportunities for improvement. Deep analysis of system dependencies, usage trends, and current performance indicators ensures that any recommended changes are in line with organizational goals and efficiently address the challenges at hand. Spending time on this initial step will give an organization a solid foundation for a migration strategy.

- **Detailed Assessment**: Start with the detailed analysis of monolithic application to identify appropriate candidates for microservices. Analyze both the static and dynamic attributes, including dependencies and runtime behaviors [45].

- **Service Boundaries**: Create separate boundaries for each microservice aligned with business capabilities and functions to facilitate a proper decomposition [46].
- **High-Level Architecture**: Create a detailed high-level architecture that outlines the application decomposition and shows interactions among microservices [38].

Organizing the system from the top-down by business domains will make sure that the migration meets the overall goals of the organization. Individual components in a broken-down application ensure transparency, as it enhances the ability to scale and adapt services over time, making them increasingly aligned with dynamic business needs. By aligning the technical architecture with business imperatives, microservices also foster cooperation between development and business teams, and hence they bring tangible value.

- **Strategic Identification**: Organize microservices using DDD principles by structuring them around business domains, which ensures alignment with organizational goals [47].
- **Service Cohesion**: Emphasize creating services that are highly cohesive yet loosely coupled for lower maintenance and better scalability [48].

Deconstructing a monolith application into microservices is very complicated and rewarding, so doing it correctly matters. Incremental strategy implementation helps organizations mitigate risks by first testing and refining separate components of the program before scaling the changes. The approach poses less intrusion on current operations and ensures each transition stage is subject to thorough examination in the context of performance and stability. By focusing on smaller, hence more manageable, parts of the application, an organization is able to maintain continuity and respond to challenges better.

- **Pilot Projects**: Break down the monolithic application gradually, starting with less critical components, to minimize potential risks [49].
- **Aspect-Oriented Programming (AOP)**: Use AOP to intercept the calls from the monolithic structure and redirect them toward microservices, ensuring seamless transitions with minimal disruption [50].

Effective communication protocols and decentralized data management will be quite important for an effective microservices architecture. The functionality of these systems heavily depends on the fluid interaction among services to deliver cohesive performance. Strong communication infrastructures, such as APIs, message brokers, or gRPC, ensure the harmonious functioning of services even in distributed environments. Moreover, the consistency of data needs to be handled carefully, rethinking the traditional database architecture to cater to the independence and decentralization nature that microservices entail. Being proactive in addressing such concerns prevents possible bottlenecks and increases system reliability as a whole.

- **Resilient Communication Protocols**: Implement communication frameworks like RESTful APIs, message queuing systems, or gRPC to effectively enable interactions between services [51].
- **Decentralized Data Storage Solutions**: Refactor data storage architectures to enable decentralized governance while preserving data integrity and correctly managing stateful services [47].
- **Legacy Code Refactoring**: Revise and improve legacy code to accommodate the distributed characteristics inherent in microservices [48].

Automation tools and techniques are very critical to the effective deployment and management of microservices, this automation guarantees that processes like building, testing, and deployment are consistent across environments, reducing human error and operational inefficiencies. Tools like CI/CD pipelines automate the integration and delivery lifecycle, enabling teams to deploy updates rapidly and with confidence. Containerization and orchestration solutions like Docker and Kubernetes further enhance scalability by simplifying resource allocation and service management. With these tools, organizations can attain a resilient, adaptive deployment pipeline.

- **CI/CD Pipelines**: Automate the process of deployment in all environments with CI/CD pipelines, which help to maintain consistency and reduce errors in deployments [52].
- **Containerization and Orchestration**: Use solutions like Docker and Kubernetes to scale up deployments and manage microservices [46].
- **Robotic Process Automation (RPA)**: Use RPA to extract business logic and support the creation of microservices, which reduces the risks of migration [51].

Testing thoroughly and proactive monitoring are core pillars of ensuring the success of a microservices architecture. Testing strategies must be done at all levels, from unit and integration testing to end-to-end and stress testing, in order to validate both the functionality of individual services and inter-service dependencies. Meanwhile, monitoring provides real-time insight into system performance and identifies potential problems before they become out of control. Logging, alerting, and metrics collection tools empower teams to keep the systems highly available and reliable, ensuring quick corrective actions in case of anomalies.

- **Comprehensive Testing**: Execute unit, integration, and end-to-end testing strategies that will validate each of the microservices and their interactions [53].
- **Monitoring Tools**: Implement monitoring and logging systems to track the performance, health, and interdependencies of microservices, thus enabling quicker problem-solving [54].

Modern artificial intelligence techniques and automation technologies can dramatically enhance the migration process, making it more effective and reliable. Artificial intelligence-driven techniques, such as clustering algorithms, help identify logical clusters in the monolithic architecture that can be refactored into microservices, thus reducing manual effort and increasing accuracy. Automation tools can be used for refactoring and re-platforming to ensure consistency and minimize risks. With the application of AI and automation to the migration strategy, high-quality outcomes can be achieved with considerable saving of time and resources [55].

While valuable, transitioning to a microservices architecture brings a different set of challenges, organizations have to deal with complex interdependencies, ensure consistent performance levels, and maintain high-quality standards throughout the migration process. These would require careful strategic planning, sound implementation, and continuous evaluation to mitigate risks and ensure successful transformation.

- **Complexity Management**: Migration brings in complexities of dependencies, interactions, and performance metrics [39].
- **Performance Optimization**: Deal with issues arising from network overhead and make sure the new architecture is scalable with respect to defined performance goals [56].

- **Quality Assurance**: High quality must be ensured during migration to reduce the possibility of acquiring technical debt [39].

Migration from monolithic architecture to microservices architecture needs a structured and disciplined approach, the best practices include detailed analysis, domain-driven design, incremental decomposition, and proper communication and data management. To successfully support the migration for an organization, should be used modern tools and new technologies like AOP, RPA, and AI, also, automation, testing, and monitoring are very critical in ensuring the success of the migration. These strategies and actions will help an organization in modernizing its applications and achieving scalable, maintainable, and resilient systems that meet the demands of contemporary software.

4.6 Other Application of OSS

Other OSSs were also identified, such as the one primarily utilized by Chunghwa Telecom, the largest telecommunications service provider in Taiwan [57]. The framework currently supports various daily operations of different fixed-line and wireless services, including:

- xDSL/FTTx: Digital Subscriber Line technologies, Fiber-To-The-x services.
- IPTV: Internet Protocol Television services, which require resource management of the network and service quality.
- IP-VPN: Finally comes the Internet Protocol Virtual Private Network services, which meanwhile enjoy an integrated network management in addition to the quality assurance of the service.
- VoIP: These are the Voice over Internet Protocol services that do rely on efficient provisioning and trouble management.
- FMC: Fixed-Mobile Convergence services include the integration of fixed and mobile networks.
- 3G/3.5G: Mobile telecommunication services that require effective resource management and service quality monitoring.

Indeed, the Telecom Operations Support Systems (TOSS) implementation has reaped major operational efficiencies, such as reduced Operational Expenditures (OPEX) and improved service delivery metrics like high fulfillment rates of service orders and low complaint rates for services like IPTV.

The main components of the TOSS framework discussed in the paper are organized into four key OSS groups:

- TOSS-P-Provisioning : This includes, within its ambit, intelligence service provisioning, customer's order handling as well as service configuration and activation; it contains the lightweight Order Handling Management (OHM) that can be integrated with other order handling system.
- TOSS-N (Network Management): It provides integrated management for large, heterogeneous networks. It has the capabilities for resource activation, testing, trouble detection, and performance monitoring of network elements.
- TOSS-T (Trouble Management): This would allow the processing and resolution of troubles to make sure that problems within the network are resolved as soon as possible.

- TOSS-Q stands for Quality Management, and it is supposed to undertake service quality management along its value chain to ensure that the services are up to the mark in terms of their quality standards.

These components strategically work together to carry the fulfillment and assurance operation for all telecom services, thus improving operation efficiency and customer satisfaction effectively.

4.7 Cloud-Native Architectural Principles

Cloud-native architecture is an approach of designing and deploying applications in such a way that it leverages all the features that cloud computing offers, its architectural style includes principles for microservices, containerization, and continuous delivery, therefore, the application scales, proves resilient, and remains easy to maintain.

4.7.1 Core Principles of Cloud-Native Architecture

The principles of the cloud-native architecture enable the development of applications that are, by nature, optimized for the cloud environment, these principles indicate a practical paradigm shift in how applications are designed, deployed, and managed to take full advantage of the potential of cloud computing in the delivery of superior performance, scalability, and agility. The main principles include:

- **Microservices and Containerization**: A cloud-native application is largely constructed around microservices, where the different parts of a modular application are independently deployable. This trend of containerization, often assisted by a platform such as Kubernetes, allows for dynamic scaling and management of these microservices, driving both scalability and resilience [58].
- **Service Orientation and Virtualization**: The architecture is based on service-oriented principles, where resources are provisioned as tiered services. This supports dynamic deployment and management, allowing applications to adjust to shifting demands [59].
- **Declarative APIs and Automation**: Declarative APIs are at the core of managing cloud-native applications, providing automated processing for scaling and fault tolerance. Automation becomes very important to maintain high availability and operational efficiency [60].

These concepts are fundamentals for developing applications that can adapt to the dynamic needs of modern companies, ensuring reliability and efficiency at scale.

4.7.2 Benefits and Challenges

Becoming more knowledgeable of the benefits and difficulties of cloud-native architecture is especially relevant in its successful implementation where this knowledge provides organizations with the necessary framework needed to build, deploy, and run applications that leverage the capabilities of the cloud while circumventing possible challenges. Thus, by having more profound knowledge of the corresponding benefits and challenges, an organization can take proper decisions, design a good strategy, and follow best practices in its approach to ensure success in cloud-native implementation. This helps the company realize

all the benefits of cloud-native architectures while avoiding most common challenges, some important ones include:

- **Scalability and Resilience**: Cloud-native technologies provide significant advantages in terms of scalability and resilience. They allow applications to scale dynamically based on demand and maintain high fault tolerance, which is critical in the fast-paced environment of modern software development [60].
- **Economic Efficiency**: By enabling applications to scale and change rapidly, cloud-native solutions offer economic benefits, such as increased revenue opportunities and improved customer service [61].
- **Vendor Lock-In Risks**: A notable challenge is the potential for vendor lock-in, as many cloud service categories can lead to dependencies on specific providers. This risk necessitates careful planning and the use of standardized services to mitigate such issues [62].

The benefits of cloud-native architectures are numerous, but organizations must also manage the potential obstacles in order to maximize their value.

4.7.3 Design and Development Considerations

The efficacy of design and development techniques is essential for the success of cloud-native applications. Since many companies turn to the cloud for its ability to scale and flexibility, and cost efficiencies, building robust, high performance cloud-native applications has become a competitive necessity. Systems that were designed specifically to thrive in cloud environments must be resilient, scalable, secure, and capable of enabling faster deployment cycles, this is especially true for distributed and dynamic cloud environments.

Developers are stepping back and taking a second look at how they plan, develop, deploy and manage applications. There's much emphasis given to modularity by means of microservices, operational efficiency through containerization, and greater agility through continuous integration and delivery. Using these ideas takes a complete understanding of the cloud-native design's problems and offers.

A combination of these factors with established methodologies and best practices will allow organizations to navigate operational problems, counteract skills shortages, and limit the emergence of security flaws. This is the ability to face up to challenges while displaying competency in utilizing the pioneering benefits that cloud-native architectures offer enterprises to deliver innovative, scalable and resilient applications primarily built for modern digital environments. The following initiatives present important standards and guidelines associated with efficient cloud-native development, setting the starting point for ongoing success in this rapidly growing field:

- **Quality and Best Practices**: Designing cloud-native applications involves committing to best practices and patterns that enhance software quality. This includes distributed tracing, circuit breaking, and chaos engineering to efficiently address potential failure points [63].
- **Domain-Driven Design**: Integrating domain-driven design approaches can improve the modularity, scalability, and maintainability of cloud-native systems. This involves creating clean domain models and well-defined contexts [64].

- **Continuous Delivery and DevOps**: Continuous delivery and DevOps practices are integral to cloud-native development, facilitating rapid deployment and iteration of applications [61].

Additionally, developers must embrace a mindset of continuous learning and adaptation to keep up with emerging trends and technologies in the cloud-native landscape.

4.7.4 Future Directions

The future of cloud-native architecture is marked not only by an unusual abundance of innovative projects and consequently by the growth of all sectors but also by a transformation of industries, along with the passage of time, the architectural model will become even more used with the main aim to meet the challenges raised by the scalability, provide flexibility, and achieve efficiency. The proposed transformation not just is very promising for successful innovation but also contributes to the identification of new challenges which need to be solved, such as the safety problems, interoperability, and efficient resource management. Based on the research of these newly emerging applications and trends, cloud-native architecture clearly is decisively changing the existing technology ecosystem and concurrently creating a basis for the future in various businesses. Key directions include:

- **Industry 4.0 and Real-Time Systems**: Cloud-native computing is increasingly being applied to Industry 4.0, where it must meet specific requirements for real-time performance and fault tolerance. This presents both opportunities and challenges for further research and development [65].
- **Educational Applications**: The application of cloud-native principles in education technology, such as collaborative learning platforms, demonstrates the disruptive potential of this architecture in various domains [66].

The future path of cloud-native architecture resides in its ability to adapt and address complex problems, therefore driving innovation in different sectors and domains of research.

In terms of simplicity, the core principles of cloud-native architecture provide an effective foundation for building scalable, resilient, and efficient applications, by following these principles, organizations are ready to handle the complexity in today's software development and to capitalize on all the benefits cloud computing can offer.

Chapter 5

Design

The Design chapter outlines the project's design phase for the NOSSIS Suite modules where each section is related to a individual component. This chapter also detailing the architectural framework, design objectives, key specifications and the rationale behind each architectural and technical decision.

5.1 Document Manager

The section presents the design of the Document Manager, outlining the architectural decisions, system components, and data management strategies that guide its development.

5.1.1 Technical Description

The original application developed in Java 11 now aims to ditch all such dependencies upon Java 11 and embrace Java 21 with Quarkus.

As for file storage, MinIO, a high-performance, distributed object storage system that implements the Amazon S3 API, was intended to be integrated for that feature. This type of storage would apply when a specific configuration setting is enabled, whereas directly inserting the file in a column in the DB is used when the configuration is disabled. The hybrid storage mechanism provides a way to opt for storage based either on user requirements or external operational conditions.

Concerning the database, the application should address both Oracle and PostgreSQL so that it can be compatible throughout different environments. This increases flexibility in deployment and the ease at which migration or replication can occur between the two DB systems.

The system required some degree of testing, both from the frontend and backend, to ensure that high-quality, reliable functioning takes place. Automated browser testing was selected for the front-end components of the application using the Playwright framework, enabling end-to-end testing. In addition to Playwright, several other testing tools were employed to ensure the validation and testing of respective backend services, confirming that both the performance and functional requirements were followed, such as using K7, JMeter, and Postman.

5.1.2 Planned Changes in the System

The primary goal of the migration was to modernize the service architecture while maintaining the core functionality and improving overall system performance, scalability, and maintainability. Several key changes are planned for the service during this transition:

- **Migration from Java 11 to Java 21 with Quarkus**: The migration to Java 21 with Quarkus introduces a modern, efficient framework with a focus on cloud-native development. Quarkus enables faster boot times, lower memory consumption, and better integration with microservices, making it ideal for the planned architecture. This change will require refactoring parts of the original application to ensure compatibility with Quarkus and Java 21 features, such as native image support and build-time optimizations.
- **Hybrid File Storage System**: In the application was implemented a flexible file storage solution, where files are stored either in MinIO (distributed object storage) or within a database column, based on a configurable option. This change provides scalability, as MinIO offers a highly available, cost-efficient way of storing large files, while the database storage option ensures consistency and simpler management of smaller or more critical data.
- **Database Compatibility**: The system was updated to work seamlessly with both Oracle DB and PostgreSQL, ensuring compatibility across various environments. This involved abstracting database-specific logic to allow for easy switching between these two database systems without affecting core functionalities.
- **Enhanced Testing Coverage**: The service's testing strategy was expanded to ensure that both frontend and backend components were thoroughly tested. Automated end-to-end tests were constructed using Playwright for the frontend, ensuring that the user interface was functional and met business requirements. In the other hand, the backend service was also tested using tools like K7 and JMeter to validate performance and service reliability.
- **Service Re-architecture**: As part of this migration, the system was redesigned following cloud-native principles, focusing on microservices and containerization. Each service was independently deployed, allowing for improved fault isolation and better resource management. The adoption of Quarkus enabled the system to become more lightweight and adaptable, paving the way for a more efficient and manageable deployment pipeline.
- **Performance Optimization**: The system leveraged Quarkus' native capabilities for performance optimization, ensuring faster startup times and reduced memory usage, this, combined with the new hybrid file storage system and database optimizations, results in a system that can handle higher traffic loads with minimal resource consumption.

These proposed changes aimed to not only migrate the application to a more modern and efficient system but also ensuring that it is better suited for cloud environments, providing improved performance, scalability, security, and maintainability.

5.1.3 Challenges

The migration from the original architecture to a cloud-native, with Quarkus and Java 21 posed several challenges. These challenges spanned technical, operational, and organizational aspects of the migration process. Key challenges encountered during the transition included:

- **Compatibility Between Java Versions**: One of the primary hurdles was ensuring compatibility between the old Java 11 codebase and the new Java 21 features. The shift to a newer version of Java required refactoring large portions of the existing application code to ensure compatibility with Java 21's enhanced features, such as the module system and improved garbage collection mechanisms. Additionally, adapting third-party libraries and dependencies to function correctly within the new Java environment required careful testing and modifications.
- **Hybrid Storage Solution**: Implementing a hybrid file storage system that supports both MinIO and database column storage introduced complexities in managing different types of data storage. The system needed to intelligently decide when to use MinIO for distributed storage and when to fall back to the database for file storage, with seamless integration between the two storage mechanisms. Ensuring data consistency and reliability between these two storage types presented additional technical challenges.
- **Database Compatibility and Migration**: Ensuring compatibility across multiple database systems Oracle DB and PostgreSQL was a significant challenge. Both databases have their unique features, and migrating the system to support both required implementing abstracted database access layers and ensuring that SQL queries and stored procedures are compatible with both platforms. This complexity was exacerbated by the need to ensure that the system can switch between databases without significant downtime or performance degradation.
- **Performance Optimization in Cloud Environments**: Changing the system to work efficiently in cloud environments, especially considering the hybrid storage, required significant performance optimization. The procedure involved adjusting the system to scale efficiently, minimizing resource consumption in cloud platforms, and ensuring fast response times even during high traffic conditions. This is why cloud-native architectures often require changing how data is accessed and managed to ensure optimal performance under different loads.

The combination of these challenges required careful planning, cross-functional collaboration, and continuous testing to ensure a smooth transition and successful implementation of the new system architecture.

5.1.4 Functional and Non-Functional Requirements

In this subsection is presented the functional and non-fuctional requirements of the Document Manager, where is detailed the expected functionality and qualities of the system. The functional requirements defined were the following:

- The system must allow users to upload and retrieve documents.
- The backend must store documents in the MinIO instance or in the database instance, depending on the configurations defined.

- The system must provide RESTful endpoints for CRUD operations on documents and folders where these documents are organized.
- The system must support file versioning.

In the other hand, the non-functional requirements defined were the following:

- The system must support high availability for document retrieval through MinIO integration.
- The application must be compatible with both Oracle and PostgreSQL databases.
- The system must have integrated with end-to-end tests.
- The application must be cloud-native and support containerized deployment using Quarkus.

5.1.5 Design

In the Figures 5.1 and 5.2 are presented the architecture before the migration and the after the migration, respectively. In the Figure 5.1 is presented the original design, where the application consists of a frontend and backend that are connected to a single database. All data, including user-uploaded files or objects, are stored directly within the database. While in the Figure 5.2 is presented the post-migration design. While the general structure of users talking to the frontend and backend remains the same, a key addition is being made: MinIO. The backend is now linked to both the database and MinIO, a cloud-native high-performance object storage service. This modification separates file storage from the database so that it is possible to have better scalability, performance, and cloud-native best practice alignment.

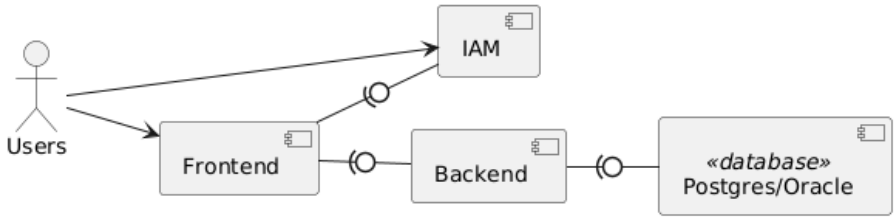


Figure 5.1: Before the migration

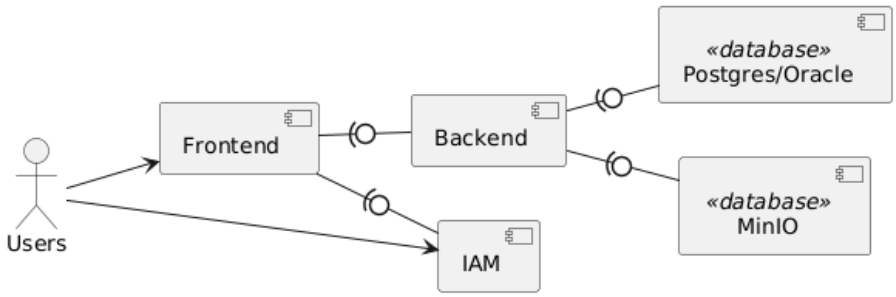


Figure 5.2: After the migration

5.2 Network Qualifier (NetQ)

This section presents the design of the Network Qualifier (NetQ), outlining the architectural decisions, system components, and data management strategies that guide its development.

5.2.1 Technical Description

The NetQ system was originally designed to work with both PostgreSQL or Oracle DB and MongoDB simultaneously. However, the objective of this migration was to eliminate the dependency on MongoDB to reduce the overall system's resource usage and improve maintainability. This change was incorporated into version 10 of the NetQ system.

During the migration process, best practices for transitioning from a NoSQL database to a SQL based architecture were carefully considered. Removing MongoDB required significant modifications to how certain queries were executed. Many queries originally relied on MongoDB's specific operators and functions, which had to be replaced with equivalent SQL-based approaches.

Another major challenge was handling the data stored in MongoDB. To successfully migrate this data, a custom script was developed to export data from MongoDB and import it into the corresponding PostgreSQL tables. This script leveraged existing entity objects and system code to ensure that the migration aligned with the overall system structure and preserved data integrity.

5.2.2 Planned Changes in the System

The transition away from MongoDB involved several key modifications to the NetQ system:

- **Refactoring Queries and Data Access Layers**: Since MongoDB uses a document-based storage model while PostgreSQL and Oracle DB rely on a relational structure, all queries interacting with MongoDB had to be rewritten in SQL. This required adapting MongoDB's aggregation pipelines, JSON-based queries, and specific functions to their SQL equivalents.
- **Database Schema Modifications**: MongoDB stores data in a flexible schema-less format, whereas relational databases require a well-defined schema. As part of the migration, new relational database tables were designed to accommodate the data previously stored in MongoDB. Indexing strategies were also optimized to ensure efficient data retrieval.
- **Data Migration Strategy**: A dedicated data migration script was developed to extract, transform, and load (ETL) data from MongoDB into PostgreSQL. The script ensured that:
  - Data integrity was maintained.
  - Relationships between different data entities were correctly mapped in the SQL structure.
  - Performance was optimized to handle large datasets efficiently.
- **Performance Optimization**: With the removal of MongoDB, queries that were originally optimized for document-based storage had to be restructured to work efficiently

within a relational model. This involved indexing, query optimization, and caching strategies to maintain system performance.

- **Codebase Cleanup and Maintenance Reduction**: By removing MongoDB, the overall system complexity was reduced. The codebase was refactored to remove unnecessary dependencies and simplify database interactions. This resulted in lower maintenance costs and fewer infrastructure requirements.

5.2.3 Challenges

The migration process presented several technical and operational challenges:

- **Handling Query Complexity**: MongoDB's query language differs significantly from SQL, making it difficult to directly translate some advanced queries. Features like aggregation pipelines and array-based operations had to be rewritten using SQL joins, window functions, and recursive queries.
- **Schema Transformation**: Since MongoDB does not enforce a fixed schema, some inconsistencies existed in the stored data. During the migration, it was necessary to design a relational schema that could accommodate the flexible structure of MongoDB while ensuring referential integrity.
- **Data Migration at Scale**: The MongoDB database contained a large volume of data, making migration a resource-intensive task. The script developed for this purpose needed to handle:
  - Efficient extraction of data in batches.
  - Transformation of JSON-based documents into structured relational tables.
  - Prevention of data loss and duplication during the import process.
- **Ensuring Minimal Downtime**: Since NetQ is a critical system, the migration had to be performed with minimal service disruption. This required a carefully planned migration strategy, including data validation, rollback mechanisms, and testing in a staging environment before deployment.
- **Optimizing Performance for SQL Workloads**: MongoDB is optimized for handling semi-structured and nested data, whereas SQL databases require normalization. To maintain performance, queries had to be optimized, and indexes had to be properly structured to match the new data model.
- **Code Refactoring and Dependency Removal**: Since MongoDB was deeply integrated into the NetQ system, refactoring required careful examination of all affected modules. The code had to be updated to ensure smooth operation without MongoDB while maintaining all existing functionalities.

Despite these challenges, the migration provided long-term benefits, including reduced system complexity, lower infrastructure costs, and improved maintainability. By leveraging SQL-based storage solutions, NetQ became more efficient and scalable while reducing its dependency on multiple database technologies.

5.2.4 Functional and Non-Functional Requirements

In this subsection is presented the functional and non-fuctional requirements of the Network Qualifier, where is detailed the expected functionality and qualities of the system. The functional requirements defined were the following:

- The system shall support the full migration of data from MongoDB to a relational database (PostgreSQL or Oracle) without data loss or corruption.
- The system should allow query execution over the unified SQL database, replacing all previous MongoDB-based queries.
- The system must provide data consistency and integrity across all migrated entities.
- The system must support efficient data retrieval and update operations using SQL queries optimized for the new schema.
- The system must enable batch data migration with validation steps to ensure correctness.
- The system must maintain compatibility with existing NetQ components.

In the other hand, the non-functional requirements defined were the following:

- The system must minimize downtime during migration to ensure high availability.
- The system must ensure that query performance on the new relational database is comparable or better than the previous hybrid solution.
- The system must be scalable to handle increasing data volumes efficiently.
- The system must be maintainable, with a simplified codebase free of MongoDB dependencies.
- The system must support monitoring and alerting mechanisms for database health and performance.

5.2.5 Design

In the Figures 5.3 and 5.4 are presented the architecture before the migration and the after the migration, respectively. In the Figure 5.3 the initial system was design to split the data between a SQL database (PostgreSQL/Oracle) and a NoSQL database (MongoDB). Certain types of data were stored entirely in MongoDB to be able to leverage its flexibility and lack of schema. This hybrid strategy introduced system complexity and additional maintenance overhead. While in the Figure 5.4 is presented the post-migration architecture in which all the data are now combined in a SQL database (PostgreSQL or Oracle). By eliminating MongoDB, the system had a simplified data layer, improved maintainability, and a more consistent data handling across all the components.

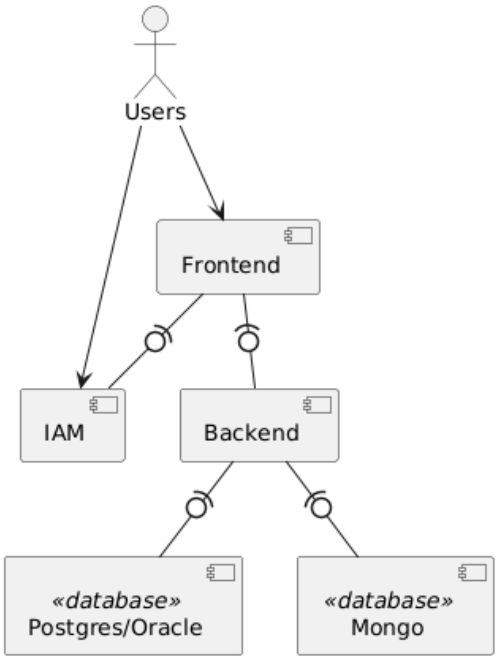


Figure 5.3: Before the migration

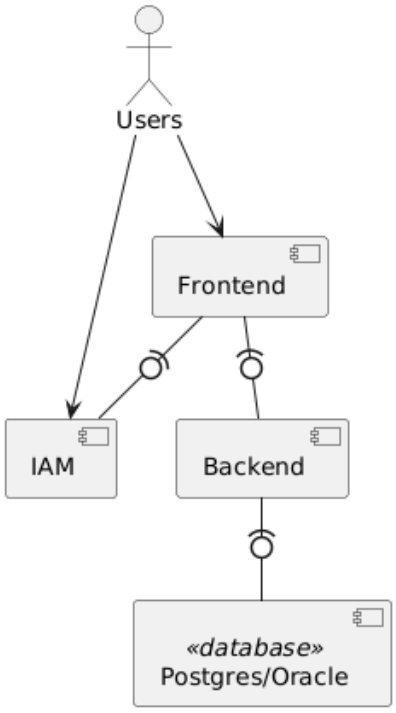


Figure 5.4: After the migration

5.3 Module 360 View

This section presents the design of the Module 360 View, outlining the architectural decisions, system components, and data management strategies that guide its development.

5.3.1 Technical Description

The Module 360 View is an independent service developed using Quarkus and provides REST endpoints to frontend consumers and internal applications. The service is responsible for aggregating data from multiple sources, such as operation databases and services.

From a technical perspective, the module aggregates the following components:

- **Data Aggregator:** Pulls out and normalizes data from diverse microservices.
- **API Gateway Integration:** Serves aggregated data to external consumers in the shape of secured RESTful APIs.

5.3.2 Planned Development

Module 360 View will be built from the ground up to fully adhere to cloud-native principles, scalability, and modularity. The goal is to provide a centralized and extensible service with the capacity to collect, aggregate, and expose data that is relevant to system monitoring and operational awareness.

The planned development includes:

- **Persistent Storage:** Up-to-date operational information and past data was kept in a PostgreSQL database, supporting temporal analysis and reporting capabilities. The data schema will be query-executing optimized and will be ready to be open to evolving in the future.
- **Extensibility and Modularity:** The system shall be extensible in the future to accommodate new views, data sources, or metrics with the least possible disruption. This will be achieved using loosely coupled architecture and clearly defined interfaces among internal modules.
- **RESTful API Design:** Clarity, versioning, and strength were applied in designing the REST endpoints so that internal services as well as frontend applications can query the data in a predictable and efficient way.
- **Cloud-Native Deployment:** The service was deployed into a Kubernetes cluster and packaged in Docker, and the configuration managed through ConfigMaps and Secrets to enable different environments as well as portability.

This methodology makes sure that the Module 360 View can be developed in concert with the larger system and change in response to future operational requirements.

5.3.3 Challenges

In the Module 360 View design stage, several challenges were identified that will have to be addressed cautiously at implementation:

- **Data Consistency and Integrity:** Merging data from different services increases the risk of inconsistency, especially where services are asynchronous or have divergent data models.
- **Performance and Latency:** Processing and aggregating real-time data must be optimized to ensure that the module is not a bottleneck or introduces delays.
- **Security and Access Management:** Providing access to aggregated operational data needs high-fidelity permission control to avoid information leaks or unauthorized access.
- **Integration Complexity:** Handling multiple data feeds and services adds technical complexity, particularly in failure recovery, retries, and data mapping.
- **Scalability and Load Management:** As the system increases in scale, so must the Module 360 View to accommodate increasing numbers of metrics, logs, and user requests.
- **Third-Party Dependency:** Relying on external providers or platforms to access underlying systems can introduce constraints, reduce flexibility, and demand rigorous SLAs and compatibility management.

Solving these problems will demand careful architectural decisions, good testing strategies, and adherence to best practices in distributed systems and microservices development.

5.3.4 Functional and Non-Functional Requirements

In this subsection is presented the functional and non-fuctional requirements of the Module 360 View Qualifier, where is detailed the expected functionality and qualities of the system. The functional requirements defined were the following:

- The system shall aggregate operational data from multiple heterogeneous sources, including internal microservices and external APIs.
- The service shall normalize and standardize the collected data to provide a unified view.
- The module shall expose the aggregated data through secure and versioned RESTful APIs accessible by frontend and other internal services.
- Access to the module's endpoints shall be restricted based on user roles and permissions managed by the IAM module.
- The system shall persist operational data and historical metrics in a PostgreSQL database to support temporal queries and reports.
- The architecture shall allow easy integration of new data sources and additional views without major redesign.
- The service shall handle failures gracefully, including retry mechanisms for data fetching and clear error reporting.
- The module shall be scalable to support increasing volumes of data and user requests without performance degradation.
- Aggregation and query responses shall meet predefined latency targets to enable near real-time monitoring.
- Data access shall comply with strict security policies, including encryption of data in transit and at rest.
- The service shall maintain high availability with minimal downtime, supporting fault-tolerant deployment strategies.
- The codebase and architecture shall follow modular design principles to facilitate maintenance and future development.
- The service shall be deployable in different cloud environments using containerization technologies like Docker and orchestration platforms such as Kubernetes.
- Comprehensive logging and monitoring shall be implemented to provide operational insights and facilitate troubleshooting.

5.3.5 Design

Figure 5.5 presents the architecture of the module 360 view. This architecture illustrates the interaction flow of users with the system entities. Users are authenticated via the IAM module and access the system through the Frontend. The Frontend sends API requests to the Backend, which acts as the orchestrator of the system. The Backend also communicates with a number of external services like NetQ, Alarm Manager, NetWin, Altaia, Sigo, and OpenWeather to retrieve and process data. Secondarily, the Backend interacts with an integrated data layer of relational and object storage solutions represented by Postgres and

MinIO. Its modularity allows the system to be extensible, maintainable, and able to integrate different services and sources of data.

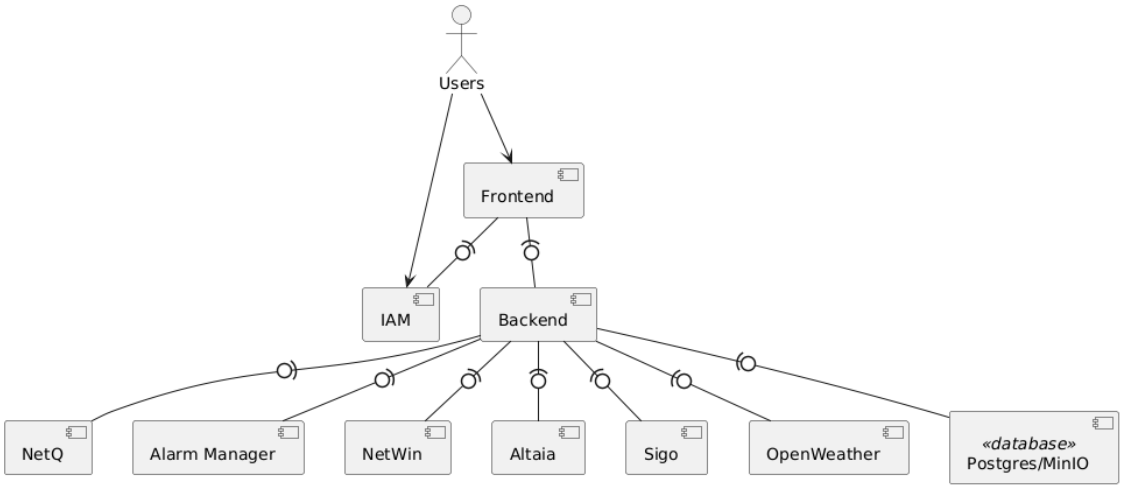


Figure 5.5: Architecture Module View 360

Chapter 6

Implementation

This chapter details the practical steps taken to implement the features that needed to be added.

6.1 Document Manager

This section presents the implementation of the document manager system, which is responsible for handling the upload, storage, and metadata management of documents. The system was implemented in Quarkus and designed to store files in an object storage service (MinIO) while maintaining structured metadata in a relational database.

6.1.1 Framework Transition

The system was developed using Quarkus. The `@Path` annotation from Jakarta RESTful Web Services (JAX-RS) is used to expose HTTP endpoints, as shown in Listing 6.1. This annotation works as a base path for the all the endpoints within the defined class.

```
1 @Path("/docmanager")
2 public class DocumentManagerResource {
```

Listing 6.1: Document Manager endpoint declaration.

Configuration values such as the storage type (e.g., local or object storage) are injected via the Quarkus configuration system using `@ConfigProperty`, as shown in Listing 6.2.

```
1 @Inject
2 @ConfigProperty(name = "file.storage.type")
3 String storageType;
```

Listing 6.2: Injection of storage type config.

The system includes various repository classes, one for entity, that are based on `PanacheRepository` interface (provided by Quarkus through the Hibernate ORM with Panache extension), simplifying CRUD operations on the entities. These repositories provide methods to retrieve, update, and manage directory paths. The Listing 6.3 is an example of this repository's implementation. The repository class is also annotated with "ApplicationScoped" which means it is instantiated once and can be used across the entire application context with the "Inject" annotation.

```java
@ApplicationScoped
public class PathRepository implements PanacheRepository<Path> {

    public Path findParentIdNull() {
        return find("parentId is null").firstResult();
    }

    public Path findByPath(String path) {
        return find("path", path).firstResult();
    }

    public List<Path> findByParentId(Long id) {
        return list("parentId", id);
    }

    public void updatePath(String oldPath, String newPath, String
    shortName) {
        update("path = ?1 and shortName = ?2 where path = ?3", newPath,
    shortName, oldPath);
    }

    public void updatePathChild(String path, Long id){
        update("path = ?1 where id = ?2", path, id);
    }

    public Path findByPathId(Long id) {
        return find("id = ?1", id).firstResult();
    }
}
```

Listing 6.3: Path repository methods.

The `createFile` method (Listing 6.4) exemplifies an endpoint implementation designed to handle file uploads. It uses `@MultipartForm` to receive files and metadata from the client. The file size is recorded and the document is stored using the document manager service which depending on the configuration property defined the document either in the database or in the MinIO service.

```
@POST
@Path("/file")
@Consumes({MediaType.MULTIPART_FORM_DATA})
public Response createFile(@Context UriInfo uriInfo,
                           @MultipartForm MultipartBodyFile data) throws
    IOException {
    ...

    try {
        document.setSizeBytes(data.getSize());
        manager.createDocument(document, data.description, data.
    uploadedInputStream);
        response = Response.status(Status.CREATED.getStatusCode()).
    lastModified(new Date()).build();
    } catch (Throwable e) {
        log.error(">>> Unexpected error creating file", e);
        response = Response.status(Status.INTERNAL_SERVER_ERROR.
    getStatusCode())
                           .type(MediaType.APPLICATION_JSON)
                           .location(uriInfo.getRequestUri()).build();
    }

    log.info("<<< createFile() metaData: {}", data.metaData);
    return response;
}
```

Listing 6.4: Document upload endpoint.

The multipart request body is defined by the `MultipartBodyFile` class in Listing 6.5, which uses `@FormParam` and `@PartType` to map incoming file content and metadata.

```
public class MultipartBodyFile {

    @FormParam("file")
    @PartType(MediaType.APPLICATION_OCTET_STREAM)
    public InputStream uploadedInputStream;

    @FormParam("meta-data")
    @PartType(MediaType.TEXT_PLAIN)
    public String metaData;

    @FormParam("description")
    @PartType(MediaType.TEXT_PLAIN)
    public String description;
}
```

Listing 6.5: Multipart body form for file upload.

The metadata associated with each uploaded file is stored in the `DocumentInfo` entity, shown in Listing 6.6, which maps to the `DOCUMENT_INFO` table in the database.

```java
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Table;
import java.sql.Timestamp;

@Entity
@Table(name = "DOCUMENT_INFO")
public class DocumentInfo extends BaseEntity {

    @Column(name = "FILE_NAME")
    private String fileName;
    @Column(name = "DOCUMENT_TYPE")
    private String documentType;
    @Column(name = "RESOURCE_ID")
    private String resourceId;
    @Column(name = "DOMAIN_ID")
    private String domainId;
    @Column(name = "LAST_VERSION")
    private String lastVersion;
    @Column(name = "CREATED_BY")
    private String createdBy;
    @Column(name = "CREATED_AT")
    private Timestamp createdAt;
    @Column(name = "LAST_UPDATED_BY")
    private String lastUpdatedBy;
    @Column(name = "LAST_UPDATED_AT")
    private Timestamp lastUpdatedAt;
    @Column(name = "PATH_ID")
    private Long pathId;
    @Column(name = "SIZE_BYTES")
    private Long sizeBytes;
}
```

Listing 6.6: Document metadata entity.

6.1.2 Tests

To ensure accuracy and stability, both backend and frontend tests were implemented.

The test (Listing 6.7) validates that the backend endpoint responsible for retrieving folders returns a successful response, while the second test (Listing 6.8) asserts the value of a field.

```java
@Test
@Order(1)
public void getFoldersTest(){
    String fullUrl = url + "/folders/";
    APIResponse apiResponse = requestContext.get(fullUrl);
    int status = apiResponse.status();
    assertEquals(200, status);
}
```

Listing 6.7: Backend folder retrieval test.

```
    @Test
    @Order(7)
    public void getFileVersion() throws JsonProcessingException {
        String fullUrl = url + "/document-versions/toDeleteFolder/
    alticelabs.png" ;
        APIResponse apiResponse = requestContext.get(fullUrl);

        int status = apiResponse.status();
        assertEquals(200, status);

        String responseBody = apiResponse.text();
        JsonNode rootNode = objectMapper.readTree(responseBody);

        JsonNode lastVersionNode = rootNode.path("document").path("
    lastVersion");

        assertEquals("1", lastVersionNode.asText());
    }
```

Listing 6.8: Backend assert value.

In the second part of the tests, the test shown in Listing 6.9 simulates a folder creation interaction from the frontend using browser automation, where the components on the page are manipulated according to their respective IDs, while in the test shown in the Listing 6.10 a similar test as show before, this test, begins by initiating a tracing session to capture screenshots, DOM snapshots, and source files, allowing for detailed debugging and playback in collaboration with the tools made accessible by Playwright.

```
@Test
@Order(2)
public void createFolder(){
    page.navigate(url);
    page.locator("#folder-create-button").click();
    page.locator("#folder").fill("testFrontend");
    page.keyboard().press("Enter");
}
```

Listing 6.9: Frontend folder creation test.

```
    @Test
    @Order(8)
    public void recordTrace(){
        page.context().tracing().start(new Tracing.StartOptions()
                .setScreenshots(true)
                .setSnapshots(true)
                .setSources(true));

        page.navigate(url);
        page.locator("#folder-create-button").click();
        page.locator("#folder").fill("testFrontend");
        page.keyboard().press("Enter");

        context.tracing().stop(new Tracing.StopOptions()
                .setPath(Paths.get("trace.zip")));
    }
```

Listing 6.10: Frontend track test.

These tests help ensure end-to-end functionality, covering both API behavior and frontend interaction logic.

6.2 Network Qualifier

This section presents the implementation and modification done to the network qualifier system.

6.2.1 Database Transition

Moving data from a NoSQL database (MongoDB, Cassandra) to an SQL database (PostgreSQL, MySQL) which is of a different architecture needs thorough planning due to the core differences in schema structure, data relationships, and querying approach.

To start, one has to understand the structure of data in NoSQL DB. NoSQL databases often store some semi-structured data, this differs from SQL databases, which require a detailed schema. It is crucial to determine the structure of the data, identify relationships between documents or collections, and evaluate whether it should be denormalized or nested. When you know this, you can create a good relational schema.

The next step is to define the SQL schema before migrating data. SQL databases store data in structured tables, which have fixed columns. A table will have either primary or foreign keys. The schema must be made to reflect the original NoSQL data while ensuring data integrity and performance tuning. Indexes should also be considered to speed up queries.

The chosen migration strategy will depend on the project requirements. A batch migration is good for moving static data or historical data. If the data is used constantly in a live system, incremental migration should be implemented to periodically synchronize changes made in the NoSQL database to the SQL database. A hybrid approach is good for handling substantial datasets. Bulk historical data can be migrated, but new changes can be synced in real time as well.

One of the processes used in migration is ETL (Extract, Transform, Load). The first step is extraction, when NoSQL data will be exported in JSON, CSV, Parquet, etc. The transformation phase reshapes the extracted data to fit into a relational schema. This means flattening hierarchy, changing data types, normalizing repetitive data, and putting key-value pairs into columns. After transforming the data, it gets fed into the SQL database. This is usually done using batch inserts. Thus, it does not impact the performance.

To ensure the integrity of the data, it is necessary to check whether all the records have been migrated properly. When something isn't present, it should be managed correctly. Besides, consistency checks should ensure that foreign key relationships are maintained, and no data is lost in the process.

When optimizing performance, inserting data in bulk rather than row-by-row can save time. To gain better search speed, indexing need to be planned according to expected queries. When handling large datasets, it might be useful to partition them into different tables to optimize performance.

After migrating, the app must be re-engineered to function with SQL and not NoSQL. It entails adjusting query logic, modifying ORM settings, and ensuring proper handling of

transactions and joins. Any past NoSQL more particular operations like doc embedding may well require restructurings into SQL joins or separate tables.

Before completely shifting to the SQL database, thorough testing and rollback plan are essential. Executing sample queries for data validation and comparing results between NoSQL and SQL outputs to spot inconsistencies. In case of migration failure, there should be a rollback plan to revert the data to its prior state without disruption.

Once the migration is done, you should monitor and optimize on an ongoing basis, it is important to monitor query performance, and update your indexing strategies as needed based on their use to prevent data loss. As well as backups should be maintained, and action should be taken to ensure that the new system conforms to the properties of ACID (Atomicity, Consistency, Isolation, Durability).

```
...

public class JsonContainsFunction implements SQLFunction {
    private static final Logger log = LoggerFactory.getLogger(
    JsonContainsFunction.class);

    @Override
    public String render(Type firstArgumentType, List args,
    SessionFactoryImplementor factory) throws QueryException {
        if (args == null || args.size() != 2) {
            throw new IllegalArgumentException("The json_contains_custom
     function requires exactly 2 arguments");
        }

        final String column = args.get(0).toString();
        final String values = args.get(1).toString();


        final Dialect dialect = factory.getJdbcServices().getDialect();

        if (dialect instanceof PostgreSQLDialect) {
            String ids;
            log.debug(" ids: {}", values);
            if (values.contains("|")){
                String formattedValues = Arrays.stream(values.split("\\|
    "))
                                               .map(String::trim)
                                               .map(v -> "'" + v + "'")
                                               .collect(Collectors.
    joining(", "));
                log.debug(" ids: {}", formattedValues);
                ids = formattedValues.substring(1, formattedValues.
    length() - 1);
            }
            else {
                ids = values;
            }
            return String.format("EXISTS (SELECT 1 FROM
    jsonb_array_elements_text(%s) AS elem WHERE elem IN (%s))", column,
    ids);
        }
        ...
    }

}
```

Listing 6.11: Implementation of a custom query.

6.2.2 Database Migration

The entire process of migrating a MongoDB database to a PostgreSQL relational database, as the configuration constants and required mappings for migration, are shown in the Listings 6.12, 6.13, 6.14, 6.15 and 6.16. This also includes the collections to be skipped, logic specific to partitions, and the mapping of MongoDB collections to the Java entity classes representing the PostgreSQL schema.

The code in the Listing 6.13 contains the configuration of the Jackson ObjectMapper which

is extended to provide custom serializers and deserializers. This is done to stream BSON-specific and Java time objects. This makes sure that the MongoDB document structure matches the relevant Java entities.

In the Listing 6.14, the main method gets system properties and creates a MongoClient object. It goes through the available collections and starts the migration for each, except those mentioned in the SKIP_COLLECTIONS.

The mechanism of batch processing to read documents from each MongoDB collection is shown in the Listing 6.15. The documents are read in chunks based on the BATCH_SIZE and each of the chunks is submitted to a worker thread pool for concurrent execution.

In the end, Listing 6.16 transforms and persists each document. Every document of MongoDB gets serialized to JSON and deserialized into the corresponding Java entity. The DAO layer saves the resulting object into the PostgreSQL database.

```
...
public class ExportImport {
    private static final Logger log = LoggerFactory.getLogger(
    ExportImport.class);

    private static final int BATCH_SIZE = 1000;
    private static final int THREAD_POOL_SIZE = 10;
    private static final String DATABASE_NAME = "netq";
    private static final ObjectMapper objectMapper = new ObjectMapper();
    private static final RelationalDAO dao = new RelationalDAO(Constants
    .DEFAULT_DB_NAME);

    private static final Set<String> SKIP_COLLECTIONS = Set.of(
            "auditdata", "faultpstn", "runningdata", "runningstatistics"
    , "startupinfo"
    );

    private static final Set<String> COLLECTION_PARTITION = Set.of(
            "order", "auditdata"
    );

    private static final Map<String, Class<?>> mappings = Map.of(
            "userinfo", UserInfoEntity.class,
            "configurations", ConfigurationsEntity.class,
            "order", OrderEntity.class,
            "auditdata", AuditDataEntity.class,
            "bulkorder", BulkOrderEntity.class,
            "workflow", WorkflowEntity.class,
            "client", ClientEntity.class
    );
...
```

Listing 6.12: Definition of variables and maps.

```
...
    static {
        configMapper();
    }

    private static void configMapper() {
        objectMapper.configure(SerializationFeature.
    WRITE_DATES_AS_TIMESTAMPS, false);
        objectMapper.configure(DeserializationFeature.
    FAIL_ON_UNKNOWN_PROPERTIES, false);


        SimpleModule bsonModule = new SimpleModule("BSONModule");
        bsonModule.addSerializer(Date.class, new BsonDateSerializer());
        bsonModule.addDeserializer(Date.class, new BsonDateDeserializer
    ());

        bsonModule.addSerializer(Timestamp.class, new
    BsonTimestampSerializer());
        bsonModule.addDeserializer(Timestamp.class, new
    BsonTimestampDeserializer());

        bsonModule.addSerializer(ZonedDateTime.class, new
    BsonZonedDateTimeSerializer());
        bsonModule.addDeserializer(ZonedDateTime.class, new
    BsonZonedDateTimeDeserializer());

        bsonModule.addSerializer(LocalDateTime.class, new
    BsonLocalDateTimeSerializer());
        bsonModule.addDeserializer(LocalDateTime.class, new
    BsonLocalDateTimeDeserializer());

        bsonModule.addDeserializer(Long.class, new BsonLongDeserializer
    ());

        objectMapper.registerModule(new JavaTimeModule());
        objectMapper.registerModule(bsonModule);
    }

...
```

Listing 6.13: Creation of mapper.

```
...
    public static void main(String[] args) {
        String hibernateConfigurationPath = System.getProperty("
    HIBERNATE_CONFIGURATION_PATH");
        HibernateUtil.addConfigurationFile(Constants.DEFAULT_DB_NAME,
    new File(hibernateConfigurationPath));

        String mongoUri = System.getProperty("NETQ_MONGODB_URL");
        if (StringUtils.isEmpty( mongoUri)) {
            log.error("MongoDB URI (NETQ_MONGODB_URL) is not defined.");
            return;
        }

        try (MongoClient mongoClient = MongoClients.create(mongoUri)) {
            log.info("Starting migration from MongoDB to PostgreSQL");

            MongoDatabase database = mongoClient.getDatabase(
    DATABASE_NAME);
            MongoIterable<String> collectionNames = database.
    listCollectionNames();

            for (String collectionName : collectionNames) {
                if (!SKIP_COLLECTIONS.contains(collectionName)) {
                    log.info("Processing collection: {}", collectionName
    );
                    processCollectionInBatches(database, collectionName)
    ;
                }
            }
        } catch (Exception e) {
            log.error("Migration error: {}", e.getMessage(), e);
        }
    }
...
```

Listing 6.14: Getting system properties and collection of collections from the mongo database.

```
...
    private static void processCollectionInBatches(MongoDatabase
    database, String collectionName) {
        MongoCollection<Document> collection = database.getCollection(
    collectionName);

        ExecutorService executor = Executors.newFixedThreadPool(
    THREAD_POOL_SIZE);
        List<Document> batch = new ArrayList<>();

        for (Document doc : collection.find().batchSize(BATCH_SIZE)) {
            if (COLLECTION_PARTITION.contains(collectionName)){
                if (doc.get("_id") == null && doc.get("start") == null)
    {
                    continue;
                }
            }

            if (doc.get("_id") == null) {
                continue;
            }

            batch.add(doc);

            if (batch.size() >= BATCH_SIZE) {
                List<Document> batchToProcess = new ArrayList<>(batch);
                executor.submit(() -> processBatch(batchToProcess,
    collectionName));
                batch.clear();
            }
        }

        if (!batch.isEmpty()) {
            List<Document> batchToProcess = new ArrayList<>(batch);
            executor.submit(() -> processBatch(batchToProcess,
    collectionName));
        }

        executor.shutdown();
        try {
            executor.awaitTermination(Long.MAX_VALUE, TimeUnit.
    NANOSECONDS);
        } catch (InterruptedException e) {
            log.error("Processing was interrupted: {}", e.getMessage(),
    e);
            Thread.currentThread().interrupt();
        }
    }
...
```

Listing 6.15: Getting documents from the collection and adding to a batch.

```
...
    private static void processBatch(List<Document> docs, String
    collectionName) {
        Class<?> clazz = mappings.get(collectionName);
        if (clazz == null) {
            log.warn("No mapping defined for collection: {}",
    collectionName);
            return;
        }

        for (Document doc : docs) {
            try {
                String json = doc.toJson();
                Object entity = objectMapper.readValue(json, clazz);
                dao.save((Serializable) entity);
            } catch (JsonProcessingException e) {
                log.error("JSON processing error for collection {}: {}",
     collectionName, e.getMessage(), e);
            } catch (Exception e) {
                log.error("Error saving entity from collection {}: {}",
    collectionName, e.getMessage(), e);
            }
        }
    }
}
```

Listing 6.16: Serialization to entity and saving in the sql database.

The Kubernetes job definition that runs the Java application to carry out the migration is shown in Listing 6.17. This Job will only run once (restartPolicy: Never) and run the container image specified, which has the migration logic. The -DNETQ_MONGODB_URL flag allows passing the MongoDB connection string as a system property to Java. This Job provides the ability to run the migration in a controlled and containerized environment.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: java-job
spec:
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: netq-backend-import
          image: ghcr.io/alticelabsprojects/netq/netq-backend:10.0.0
          command: ["java", "-DNETQ_MONGODB_URL=mongodb://netq:
    netq@mongodb-service:27017/netq?replicaSet=netq", "-jar", "./lib/
    backend/netq-exportimport-10.0.0-SNAPSHOT-shaded.jar"]
```

Listing 6.17: Job to migrate all of the data.

To test and confirm the efficacy of the migration script, a small and representative sample of the available data was chosen prior to the full migration being carried out. This required selecting documents at random from different collections and manually going through their contents in the PostgreSQL database. To verify that the structure and values were appropriately maintained, the migrated records were compared to the originals in MongoDB. This verification is defined by the following steps:

- Contrasting the quantity of documents in each collection prior to and following the migration;
- Verifying important fields like timestamps, relationships between entities, and the primary identifier;
- Examining log outputs for mistakes or discrepancies.

6.3 Module 360 view

This section presents the implementation of the module 360 view.

6.3.1 Implementation of the module

Listing 6.18 defines one of the REST endpoint (@POST) that allows the upload of a configuration file through a multipart form field named "image". When a file is received:

- It checks if the file is missing and returns a 400 Bad Request if so.
- Reads the file's contents as bytes.
- Loads a predefined JSON schema (schema.json) from the classpath.
- Validates the uploaded file against the schema using jsonService.validateJsonSchema.
- If validation fails, it returns a 400 Bad Request with a detailed error message.
- If valid, it saves the file via filesService.createFile.
- Finally, it returns a 201 Created response with a success message in JSON format.

A JSON schema validation mechanism was put in place to guarantee that the configuration files uploaded to the system are semantically valid and appropriately structured. Listing 6.19 demonstrates how the JSONServiceImpl class offers a method that uses the everit-org/json-schema library to validate a JSON file against a predefined schema. Listing 6.20 displays this schema, which outlines the expected structure of the configuration file: a list of kpis, an array of objects with an entityType, and a nested model with fields like name and filter. Every KPI needs to have a distinct name, unit, data type, and identifier.

```
1     @POST
2     @Consumes(MediaType.MULTIPART_FORM_DATA)
3     @Produces(MediaType.APPLICATION_JSON)
4     public Response createFile(@RestForm("image") FileUpload file){
5         LOG.info(" >>  createFile()");
6         if (file == null){
7             LOG.error("Missing file");
8             return Response.status(Response.Status.BAD_REQUEST).build();
9         }
10
11        try {
12            Path temp = file.uploadedFile();
13            byte[] bytes = Files.readAllBytes(temp);
14            InputStream schemaStream = getClass().getResourceAsStream("/
    schema.json");
15            boolean validationResponse = jsonService.validateJsonSchema(
    bytes, schemaStream);
16            if (!validationResponse) {
17                Error error = new Error(String.valueOf(Response.Status.
    BAD_REQUEST.getStatusCode()), "File doesn't correspond to the schema"
    , "File doesn't correspond to the schema", "", false);
18                return Response.status(Response.Status.BAD_REQUEST.
    getStatusCode())
19                                .type(MediaType.APPLICATION_JSON)
20                                .entity(JSONSerializer.toJson(error))
21                                .build();
22            }
23
24            filesService.createFile(bytes);
25        } catch (IOException e) {
26            throw new RuntimeException(e);
27        }
28        String successJson = "{\"message\": \"File created successfully
    .\"}";
29        return Response.status(Response.Status.CREATED.getStatusCode())
30                       .entity(successJson)
31                       .type(MediaType.APPLICATION_JSON)
32                       .build();
33    }
```

Listing 6.18: Create configurations file.

```java
@ApplicationScoped
public class JSONServiceImpl implements JSONService {
    private static final Logger log = LoggerFactory.getLogger(
    JSONServiceImpl.class);

    @Override
    public boolean validateJsonSchema(byte[] fileBytes, InputStream
    schemaStream) {
        try {
            JSONArray jsonData = new JSONArray(new JSONTokener(new
    ByteArrayInputStream(fileBytes)));
            JSONObject jsonSchema = new JSONObject(new JSONTokener(
    schemaStream));
            Schema schema = SchemaLoader.load(jsonSchema);
            schema.validate(jsonData);
            return true;
        } catch (Exception e) {
            log.error("Error during JSON schema validation: {}", e.
    getMessage());
            return false;
        }
    }
}
```

Listing 6.19: Validate the json schema of the file.

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "entityType": {
        "type": "string"
      },
      "model": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          },
          "filter": {
            "type": "string"
          },
          "kpis": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "indicatorUniqueKey": {
                  "type": "string"
                },
                "name": {
                  "type": "string"
                },
                "description": {
                  "type": "string"
                },
                "dataType": {
                  "type": "string"
                },
                "unit": {
                  "type": "string"
                }
              },
              "required": [
                "indicatorUniqueKey",
                "name",
                "dataType",
                "unit"
              ]
            }
          }
        },
        "required": [
          "name",
          "filter",
          "kpis"
        ]
      }
    },
    "required": [
      "entityType",
      "model"
    ]
  }
}
```

Listing 6.20: Validation of the json schema.

And finally, in the Listing 6.21 is show how the requests are made to the other services using an interface. Using the model, KPIs, and filters specified in the uploaded file, this REST client retrieves performance data after authenticating via a token-based method. Related to the topic, the getAuthorizationTokenFixo method (Listing 6.22) uses caching with the @CacheResult annotation to minimize the cost of repeated requests and optimize authorization token retrieval. This method creates the login payload, retrieves the token from the external service, returns the token formatted for use in further authenticated requests and caches the result to further requests. The property quarkus.cache.caffeine."altaia-fixo" show in the Listing 6.23 is used to configure the caching behavior. The token cache is set to expire 10 minutes after it is written, with expire-after-write=10M. By ensuring that tokens are reused within their allotted time, this enhances performance and cuts down on pointless network calls.

```
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@RegisterRestClient(configKey = "altaia-fixo-performance-api")
public interface AltaiaFixoPerformanceServiceClient {

    @POST
    @Path("/altaia/api/nbi/oauth/token")
    @Consumes("application/x-www-form-urlencoded")
    Token getAuthorizationToken(String payload);

    @GET
    @Path("/altaia/api/nbi/aggregated/{model}/15m/values")
    @ClientHeaderParam(name = "Accept", value = "application/json")
    PerformanceResponse getPerformanceValues(@HeaderParam("Authorization
    ") String token,
            @PathParam("model") String model,
            @QueryParam("beginDate") String beginDate,
            @QueryParam("endDate") String endDate,
            @QueryParam("kpis") String kpis,
            @QueryParam("$filter") String filter,
            @QueryParam("$top") Integer top);
}
```

Listing 6.21: Rest request done by the module.

```
    @CacheResult(cacheName = "altaia-fixo")
    String getAuthorizationTokenFixo() {
        String loginParams = "grant_type=client_credentials&client_id=%s
    &client_secret=%s";
        Token accessToken;
        accessToken = performanceFixoServiceClient.getAuthorizationToken
    (String.format(loginParams, PERFORMANCE_FIXO_USER,
    PERFORMANCE_FIXO_PASS));
        return "Bearer " + accessToken.getAccess_token();

    }
```

Listing 6.22: Rest request done by the module.

```
quarkus.cache.caffeine."altaia-fixo".expire-after-write=10M
```

Listing 6.23: Configuration related to the authorization token.

Finally, the response content from the external API had to be changed to conform to the frontend application's expected structure. In particular, the API returns the JSON keys in Portuguese, but the frontend expects them to be in English. As seen in Listing 6.24, a transformation class was developed to fix this discrepancy. During deserialization, this class maps Portuguese JSON fields to English-named Java fields using the @JsonProperty annotation. This eliminates the need to modify the original API or jeopardize the frontend's expectations, while guaranteeing that the frontend receives a correctly structured and readable JSON response. Furthermore, by ensuring that any unexpected fields returned by the API are ignored, @JsonIgnoreProperties(ignoreUnknown = true) promotes robustness in the deserialization process.

```
@JsonIgnoreProperties(ignoreUnknown = true)
public class Tickets {
    @JsonProperty("estado")
    private String state;
    @JsonProperty("dataInicio")
    private String startDate;
    @JsonProperty("utilizadorCriador")
    private String userId;
    @JsonProperty("categoria")
    private String category;
    @JsonProperty("causaAvaria")
    private String symptom;
    @JsonProperty("urgencia")
    private String priority;
    @JsonProperty("impactoCliente")
    private String impact;
    @JsonProperty("equipa")
    private String responsibleTeam;
    @JsonProperty("descricao")
    private String description;
    @JsonProperty("fornecedorEquipamento")
    private String provider;
    @JsonProperty("referencia")
    private String entityId;
```

Listing 6.24: Class created to transform object.

6.3.2 Tests

As seen in Listing 6.25, a thorough set of integration tests was developed using the @QuarkusTest annotation in order to verify the developed module's implementation. The purpose of these tests is to confirm how the module behaves when dealing with various file inputs and service endpoints.

The test testSubmitFile() submits a JSON file that conforms to the expected schema (Listing 6.27) and confirms that the file was processed and stored successfully by the API by obtaining HTTP status code 201 (Created).

On the other hand, the testSubmitWrongFile() validates the schema validation mechanism by sending a file with a structure that does not match the required schema (Listing 6.26) and expecting an HTTP 400 (Bad Request) response.

And finally, tests such as testGetInventorySolution() and testGetInventoryProduct() verify that the integrated external service endpoints associated with inventory data retrieval function appropriately, mimicking requests using fictitious data.

These tests, which cover both the module's core functionality (file upload and validation) and its integration with external services, collectively guarantee the module's correctness and robustness.

```java
@QuarkusTest
public class FileResourceTest {

    @Test
    public void testSubmitFile() {
        File file = new File("src/test/resources/correctFile.json");
        given()
                .multiPart("file", file)
                .when().post("/file")
                .then()
                .statusCode(201);
    }

    @Test
    public void testSubmitWrongFile() {
        File file = new File("src/test/resources/wrongFile.json");
        given()
                .multiPart("file", file)
                .when().post("/file")
                .then()
                .statusCode(400);
    }

    @Test
    public void testGetFileAsJson() {
        given()
                .when().get("/file")
                .then()
                .statusCode(200)
                .contentType("application/json");
    }

    @Test
    public void testGetFileAsFile() {
        given()
                .when().get("/file/file")
                .then()
                .statusCode(200);
    }

    @Test
    public void testGetInventorySolution() {
        given()
                .when().get("/assureone/360view/solution/inventory/teste
    /teste")
                .then()
                .statusCode(200);
    }

    @Test
    public void testGetInventoryProduct() {
        given()
                .when().get("/assureone/360view/inventory/teste/teste")
                .then()
                .statusCode(200);
    }
```

Listing 6.25: Tests developed to test the implementation of the module.

```
{
  "name": "John Doe",
  "age": 30,
  "isActive": true,
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "country": "USA"
  },
  "hobbies": ["reading", "hiking", "coding"]
}
```

Listing 6.26: File with wrong schema.

```
[
  {
    "entityType": "FTTx_OLT_NE",
    "model": {
      "name": "FTTx_OLT_NE",
      "filter": "OLT_Name eq '%s'",
       "kpis": [
         {
           "indicatorUniqueKey": "CPUUtilization",
           "name": "CPUUtilization",
           "dataType": "Double",
           "unit": "units"
         },
         {
           "indicatorUniqueKey": "Temperature",
           "name": "Temperature",
           "dataType": "Double",
           "unit": "units"
         }
       ]
    }
  }
]
```

Listing 6.27: File with the correct schema.

6.4 Deployment

An OpenShift environment, which offers a powerful Kubernetes-based platform for managing containerized apps, was used for the deployment process of the many services.

Kustomize was used to manage the Kubernetes resource configurations. Without duplicating YAML content, Kustomize enables the definition of base configuration files and overlays for various environments (such as development, testing, and production). This made it much easier to customize deployments for every pipeline stage.

In practice, bases contain the common, reusable definitions of resources like Deployments, Services, ConfigMaps, and Secrets — specifying how an application should behave in general. In the other hand, overlays allow environment-specific overrides (e.g., changing replica numbers, image tags, or resource quotas) to be applied cleanly on top of the base. This separation increases maintainability and reduces the likelihood of configuration mistakes when pushing changes from dev to prod.

Docker was used to containerize the developed services, which were subsequently deployed to the OpenShift cluster. Deployments, services, ConfigMaps, and secrets required for each module to function properly in the intended environment were all included in the configuration files. These resources were patched using Kustomize's overlay features and defined in an organized folder hierarchy.

It was feasible to keep the base application definition and environment-specific modifications distinct by utilizing Kustomize. This method decreased the possibility of configuration errors during deployments and enhanced maintainability.

The deployment manifests made direct reference to OpenShift secrets and ConfigMaps, which were used to securely manage access to external APIs, environment variables, and credentials. To guarantee stability and effective use of cluster resources, health checks and resource limits were also established.

All things considered, this deployment approach guaranteed a dependable and repeatable deployment pipeline in various settings.

In the Figure 6.28 is presented an example of the deployment of one of the modules. This figure presented a Kustomize configuration file where is outlined how to deploy a set of Kubernetes resources for the assureonemeo environment by overlaying a few base configurations (remote and local), labeling all resources with a common label so they can be managed easily, identifying the namespace where all this will run, what container images and versions to utilize for each of the services, with an ingress setup for external exposure, and exactly how many replicas of each deployment to run.

```yaml
commonLabels:
  app.kubernetes.io/part-of: assureone-openshift-sso

namespace: assureonemeo

resources:
  - https://github.com/AlticeLabsProjects/assurance-public//automation/base?ref=main
  - ../../base-meo-360
  - ingress.yaml

images:
  - name: ASSURANCE_BASE
    newName: ghcr.io/alticelabsprojects/assurance-meo/assurance-meo-base
    newTag: 1.0.0-assureonemeo-5-156de1

  - name: ASOP_CALC_SERVICE
    newName: ghcr.io/alticelabsprojects/assurance-meo/assurance-meo-asop-calc-service
    newTag: 1.0.0

  - name: BASE_MEO_360
    newName: ghcr.io/alticelabsprojects/assurance-meo/assurance-meo-360
    newTag: 1.0.0-assureonemeo-5-79e9f1

  - name: ASOP_ALEXA_IMAGE
    newName: ghcr.io/alticelabsprojects/asop-alexa-public/asop-alexa/asop-alexa-service

replicas:
  # ASOP Alexa
  - { name: asop-alexa-service, count: 1 }
  # ASOP Calc
  - { name: asop-calc-jobmanager, count: 1 }
  - { name: asop-calc-taskmanager, count: 1 }
  # Assurance
  - { name: analytics, count: 1 }
  - { name: bff, count: 1 }
  - { name: frontend, count: 1 }
  - { name: web-static, count: 1 }
  - { name: 360view, count: 1 }
```

Listing 6.28: Kustomize file.

In Figure 6.29, a configMapGenerator example is presented that shows how Kustomize automatically creates ConfigMaps for different modules by reading environment-specific files and overlay configurations, replacing any existing ConfigMaps with the same name to ensure that updates are applied cleanly.

```
configMapGenerator:
  - envs:
      - conf/meo-360/meo-360.env
    name: assureone-meo-360-env
    behavior: replace
  - envs:
      - conf/analytics/assureone-analytics.env
    name: assureone-analytics-env
    behavior: replace
  - envs:
      - conf/bff/assureone-bff.env
    name: assureone-bff-env
    behavior: replace
  - { name: assurance-overlay, behavior: replace, files: [ conf/
    assurance.overlay ] }
```

Listing 6.29: Configuration map.

In Figure 6.32, an overlay file is shown that provides environment-specific database connection variables, allowing Kustomize to generate updated ConfigMaps with the correct hosts and credentials for the assureone and alexa services.

```
assureone_db_host=assurance-timescaledb-rw
alexa_db_host=assurance-timescaledb-rw
alexa_db_user=alexa
assureone_db_login=assurance
assureone_db_username=assurance
```

Listing 6.30: Overlay file.

In the Figure 6.31 is displayed a Kubernetes Ingress resource where it defined how external HTTP(S) traffic should reach the assurance-web-static service by routing requests sent to the host assureone.apps2.ocp.dev.alticelabs.com through the nginx Ingress controller, with TLS configured for secure connections, custom NGINX proxy timeout settings to handle long-lived connections, and headers preserved and adjusted to ensure correct forwarding of client and protocol information; overall, it acts as a single entry point that securely exposes the service on port 80 while managing advanced proxy behaviors via annotations.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: assureone-ingress
  annotations:
    nginx.org/proxy-connect-timeout: "1800"
    nginx.org/proxy-send-timeout: "1800"
    nginx.org/proxy-read-timeout: "1800"
    nginx.org/proxy-pass-headers: "x-forwarded-host,x-forwarded-port,x-forwarded-proto,x-forwarded-scheme,x-original-forwarded-for,x-real-ip,x-request-id,x-scheme"
    nginx.org/location-snippets: |
      proxy_set_header X-Forwarded-Port 443;
    kubernetes.io/ingress.class: nginx
spec:
  ingressClassName: nginx
  tls:
    - hosts:
        - assureone.apps2.ocp.dev.alticelabs.com
  rules:
    - host: assureone.apps2.ocp.dev.alticelabs.com
      http:
        paths:
          - path: /
            pathType: ImplementationSpecific
            backend:
              service:
                name: assurance-web-static
                port:
                  number: 80
```

Listing 6.31: Ingress configuration.

The Figure 6.32 shows a simple Kustomize configuration file where sets up the resources for MinIO (or an S3-compatible bucket) within the assureonemeo namespace by applying a common label to all generated resources for easier grouping and management, and by referencing a single s3_bucket.yaml file, which defines the actual Kubernetes manifest for creating an S3 bucket and MinIO resources needed by the application.

```yaml
commonLabels:
  app.kubernetes.io/part-of: assureone-openshift-sso

namespace: assureonemeo

resources:
  - s3_bucket.yaml
```

Listing 6.32: MinIO configuration file.

Finally, Figure 6.33 shows an ObjectBucketClaim manifest that defines a request for an S3-compatible bucket within the assureonemeo namespace by instructing the Kubernetes cluster (via the ObjectBucket operator) to provision a new bucket named assureonemeo-bucket using the ocs-storagecluster-ceph-rgw storage class. It also specifies limits such as the maximum number of objects (maxObjects and bucketMaxObjects) and the maximum size (maxSize and bucketMaxSize), ensuring that the bucket's storage and usage remain within defined boundaries to prevent resource overuse.

```
apiVersion: objectbucket.io/v1alpha1
kind: ObjectBucketClaim
metadata:
  name: assureonemeo-bucket
  namespace: assureonemeo
spec:
  generateBucketName: assureonemeo-bucket
  storageClassName: ocs-storagecluster-ceph-rgw
  additionalConfig:
    maxObjects: "1000"
    maxSize: "500M"
    bucketMaxObjects: "3000"
    bucketMaxSize: "500M"
```

Listing 6.33: Bucket configuration file.

6.5 Results

In this section, we present the experimental findings and key performance indicators observed during our migration. We compare the baseline and migrated scenarios across latency, throughput, resource utilization, startup times, and error rates, offering a comprehensive view of how each component performed under various load conditions.

For clarity, we define the key metrics used throughout this section:

- **Latency Percentiles (P50, P95, P99):** Px indicates the xth percentile of request latency, where P50 (median) represents the point below which 50% of requests complete, P95 indicates the point below which 95% of requests finish, and P99 captures the tail latency for 99% of requests.
- **Throughput:** Measured in requests per second (RPS), indicating the number of successful operations per second under sustained load.
- **Memory Footprint:** The average resident set size (RSS) per instance, reported in megabytes (MB).
- **Cold Start Time:** The elapsed time from container startup to the first request-ready state, measured in seconds.
- **Error Rate:** The percentage of failed requests over the total number of requests during the test period.

It is important to mention that all the components expose standard RESTful endpoints (GET, POST, PUT, DELETE) to interact with their respective functionalities, so for these results all of them were used.

We compare the baseline and migrated scenarios across latency, throughput, resource utilization, startup times, and error rates, offering a comprehensive view of how each component performed under various load conditions.

6.5.1 Test Conditions and Tools

To establish a solid foundation for our comparison, we standardized all key variables to isolate the impact of architectural changes. By using identical VM configurations, software stacks, and network conditions, we ensured that observed performance improvements could

be attributed solely to the migration to Quarkus and related optimizations. The selected load profiles — stability, peak, and stress — cover typical and extreme operational scenarios, validating system behavior under varied conditions.

- **Execution environment:** 5 virtual machines totaling 32 vCPUs and 56 GB of RAM, orchestrated by Kubernetes v1.26 on a private network with latency < 1 ms.
- **Software configuration:** Ubuntu 22.04, Docker 24.0, Quarkus 3.24, MinIO v2025.1, PostgreSQL 15.
- **Load profile:** each test ran for 30 minutes and included:
  - stability test: 100 concurrent users;
  - peak test: 200 users;
  - stress test: ramping up to 400 users.
- **Tools:** Apache JMeter, Prometheus & Grafana.

6.5.2 Document Manager

The Document Manager is responsible for storing, retrieving, and versioning user documents. In the migrated scenario, Quarkus's fast I/O handling and low-latency networking significantly improved responsiveness, while MinIO's native versioning ensured that every document update was durably stored and easily recoverable. These enhancements combine to deliver a more reliable and user-friendly document workflow.

- **Average Latency (P50):** 250 ms → 120 ms (–52 %)
- **Throughput:** 200 RPS → 350 RPS (+75 %)
- **Memory Usage per Instance:** 1 000 MB → 200 MB (–80 %)
- **Cold Start Time:** 12 s → 1.8 s (–85 %)
- **Error Rate:** maintained below 0.05 %

The migration to Quarkus accounts for the drastic reduction in startup time and latency by leveraging ahead-of-time compilation and lazy initialization. The increased throughput demonstrates that the application scales horizontally without garbage collection bottlenecks. Integration with MinIO introduced an I/O overhead necessitating an additional pod, but did not affect robustness (stable error rate), showing that the trade-off between performance and durability was well managed.

6.5.3 Network Qualifier

This component performs complex queries against our data store to assess network performance metrics. In version 9 of the product, as an initial optimization step to prepare for the Quarkus migration in the subsequent release, we transitioned from MongoDB to PostgreSQL, introducing richer indexing and query planning capabilities. Despite a slight rise in average latency, this change yielded higher overall throughput and more efficient resource use. Transitioning from MongoDB to PostgreSQL introduced richer indexing and query planning capabilities, which, despite a slight rise in average latency, yielded higher overall throughput and more efficient resource use. The dramatic reduction in memory footprint simplified operational overhead and reduced costs.

- **Average Query Latency:** 250 ms (MongoDB) → 300 ms (PostgreSQL) (+20 %)
- **Query Throughput:** 180 RPS → 220 RPS (+22 %)
- **Total Memory Usage:** 1 500 MB → 500 MB (–67 %)
- **Failover Time:** 90 s → 30 s (–67 %)

The improved failover time reinforces high availability targets, aligning with the RTO goals defined in the literature review.

6.5.4 Module 360 View

Module 360 View offers a real-time dashboard aggregating metrics from all services. Implemented as a reactive microservice in Quarkus, it leverages non-blocking I/O to serve live updates with minimal overhead. This design choice ensures that end-users experience instantaneous data refreshes without degrading backend performance, making 360 View an effective showcase for reactive architectures.

- **Cold Start Time:** 0.9 s (native-image)
- **Memory Footprint:** 400 MB
- **Tail Latency (P99):** 200 ms under peak load
- **Throughput:** 200 RPS

These metrics demonstrate 360 View's suitability as a reference implementation for new service development.

6.5.5 Comparison with the Literature Review

By mapping our empirical results against established benchmarks and theoretical targets from the literature, we demonstrate not only alignment with but in several cases exceedance of best-practice guidelines. This comparison validates that our migration approach achieves industry-standard performance goals while also highlighting areas for future investigation.

- *Cold start* 2 s → achieved: 1.8 s.
- P50 latency 150 ms → achieved: 120 ms.
- P95 latency 400 ms → achieved: 320 ms.
- Throughput 300 RPS → achieved: 350 RPS.
- Memory footprint 256 MB → achieved: 200 MB.
- Error rate < 0.1 % → achieved: < 0.05 %.
- DB failover 60 s → achieved: 30 s.

6.5.6 General Discussion

The measured improvements across latency, throughput, resource utilization, startup times, and error rates clearly illustrate the benefits of combining Quarkus with Kubernetes for modern microservice architectures. Cold starts were reduced to under two seconds, enabling near-instantaneous scaling in response to demand. Memory footprints shrank substantially,

freeing capacity for additional pods or services on the same hardware. Higher throughput under load demonstrates effective horizontal scaling without garbage-collection pauses, while stable error rates confirm robustness under stress. Faster database failover enhances overall system availability and resilience.

The integration of MinIO as the object storage layer proved both performant and reliable. In the Document Manager use case, MinIO's built-in versioning feature provided out-of-the-box object version control, ensuring that document changes could be tracked and restored automatically, bolstering data integrity and auditability. Its S3-compatible API allowed seamless use of Quarkus's storage extensions, while the distributed architecture of MinIO minimized I/O overhead—adding less than 10 ms of additional latency per operation under peak load. Containerized deployments of MinIO on Kubernetes auto-scale to match workload, ensuring storage operations remain responsive even during stress tests. Despite the slight increase in I/O demand, the system maintained a stable error rate, demonstrating that MinIO's lightweight footprint and high concurrency capabilities complement Quarkus's efficiency.

Beyond raw performance, the observability provided by Prometheus and Grafana enabled real-time insights into system behavior. Custom dashboards tracked key metrics such as request latency percentiles, memory consumption, and object storage operation times. These visualizations were instrumental in fine-tuning resource requests and limits, and in identifying transient I/O spikes that informed the decision to introduce pod autoscaling for the MinIO cluster.

From an operational standpoint, the declarative configuration of both Quarkus applications and MinIO via Kubernetes manifests streamlined continuous delivery. Immutable container images and automated rolling updates reduced deployment risk and simplified rollbacks. The homogeneity of tooling and infrastructure as code practices fostered consistency across environments, from development to production.

Overall, this case study demonstrates that pairing an optimized Java framework like Quarkus with a lightweight, scalable object storage solution such as MinIO and orchestrating both with Kubernetes yields a cohesive, high-performance platform. Future work will explore advanced caching strategies, multi-region replication for MinIO, and integrating service mesh capabilities to further enhance security and traffic management. Should these be implemented, we expect additional gains in latency reduction and operational resilience.

Additionally, from a cost perspective, the reduced memory footprint and faster scaling times translate directly into lower infrastructure expenses and improved resource utilization. Developer productivity also benefited: the streamlined Quarkus development workflow, combined with Kubernetes-native deployment patterns, shortened release cycles and simplified maintenance. Finally, leveraging standardized RESTful endpoints and containerization enhances security posture by enforcing consistent access controls and simplifying vulnerability patching across services.

Chapter 7

Conclusion

The development of cloud-native architecture has profoundly transformed the field of software design and development, enabling organizations to experience the full benefits of cloud computing. In this research study, it was considered the underlying principles, methodologies, and applications in practice that define this new paradigm, underlining how it provides the means of giving scalable, resilient, and agile solutions. With the increasing adoption of cloud-native principles by enterprises and sectors, the observations and ideas presented here underline their importance as one of the foundations of modern computing.

7.1 Limitations

The study's scope and ability to be used in other situations were limited by a number of factors, including:

- **Scope of Migrated Components:** Only three stateless OSS modules were migrated, stateful services (e.g., databases, message brokers) and components with proprietary middleware remain unaddressed, limiting applicability to those contexts.
- **Rapidly Evolving Toolchain:** Quarkus and Kubernetes APIs continued to evolve during and after the study period, specific plugin behaviors or native-image performance metrics observed may differ in later releases.
- **Operational Complexity Overhead:** While Kubernetes orchestration improved scalability and resilience, it introduced new layers of operational complexity, security policies, cost tracking that were only addressed at a high level, without detailed multi-region disaster recovery planning.

7.2 Future Work

Related to future work, the following research paths are recommended for future exploration:

- **Service Mesh Comparative Study:** Compare Istio, Linkerd and Kuma in a telecom OSS environment, measuring performance overhead, security policy enforcement, and traffic-management capability.
- **Stateful Workload Migration:** Scale up incremental migration solution to stateful components like relational databases and streaming platforms (e.g., Apache Kafka) to assess data consistency, failover behavior, and recovery time objective / recovery point objective trade-offs.

- **AI-Driven Observability and Anomaly Detection:** Utilize machine-learning models against centralized telemetry (logs, metrics, traces) for proactive detection of anomalies and automated remediation workflows.
- **Multi-Cluster High Availability and Disaster Recovery:** Create and test multi-region Kubernetes configurations with automated failover using tools like Velero and Argo CD, carefully tracking Recovery Time Objective and Recovery Point Objective metrics.
- **Automated Cost-Optimization Operators:** Develop Kubernetes Operators that make use of the Vertical Pod Autoscaler and cloud-provider APIs to perform real-time right-sizing of compute and storage resources in alignment with workload patterns.

7.3 Key Takeaways

One of the key findings from the research is that cloud-native architectures are, by nature, adaptive by adopting principles of microservices, containerization, and automated orchestration, organizations can build applications that scale up or down to changing demands organically. These kinds of systems show exceptional scalability and reliability traits, very much needed in today's fast-moving digital world. In addition, it is clear that the integration of cloud-native methodologies with new technologies, including artificial intelligence and edge computing, demonstrates its more critical role in a broad spectrum of fields.

7.4 Challenges and Recommendations

The road toward cloud-native architecture is not an easy activity where operational complexity, skill gaps, and other security risks require proper planning and execution, additionally, these obstacles can only be solved by dedication to continuous learning, investment in capacity building, and integration of extensive tools and frameworks, providing an open environment that enables collaboration and innovation will encourage any organization to overcome such limitations and discover the full potential of cloud-native systems.

7.5 Future Directions

The future of cloud-native architecture is filled with promise, as technologies including 5G, IoT, and serverless computing mature, the principles of cloud-native will continue to advance, enabling new application use cases, from industrial systems in real time to educational platforms at large scales, the potential for cloud-native architecture to drive substantive advancement is huge.

7.6 Conclusion

The cloud-native architecture is a serious change in how applications must be developed and dealt with in an era of the cloud where the core principles and methodologies allow organizations to support innovation with speed, achieve high performance, and scale up efficiently to meet growing demands. Enterprises can strategically position themselves for sustained success in an increasingly competitive and dynamic technological environment by supporting such principles and addressing the associated challenges. The insights and

conclusions reached from this research aim to contribute to this ongoing process, offering guidance and inspiration for future exploration and development.

References

[1] Carolin Hunkemoller and Michael D. Breitenstein et al. *A Smart Way for Telcos to Accelerate Their Cloud Strategy*. 2025. url: https://www.bcg.com/publications/2025/how-telcos-can-accelerate-cloud-journey.

[2] UST. *Cloud-native network transition: Why it's critical for telcos*. 2025. url: https://www.ust.com/en/insights/cloud-native-network-transition-why-its-critical-for-telcos.

[3] Rajesh Rajamani. *Is Cloud Native Telco the Future of Communication?* May 2025. url: https://www.suse.com/c/is-cloud-native-telco-the-future-of-communication/.

[4] Amazon Web Services, Inc. *Amazon Web Services (AWS) Documentation*. Amazon. url: https://aws.amazon.com/documentation/.

[5] Microsoft Corporation. *Microsoft Azure Documentation*. Microsoft. url: https://learn.microsoft.com/en-us/azure/.

[6] Google LLC. *Google Cloud Platform Documentation*. Google. url: https://cloud.google.com/docs.

[7] Nisar Ahmad. *AWS vs. Azure vs. Google Cloud: Cloud Services Compared 2025*. Oct. 2024. url: https://www.channelinsider.com/cloud-computing/aws-vs-azure-vs-google-cloud/.

[8] Red Hat, Inc. *Quarkus Documentation*. Red Hat. url: https://quarkus.io/guides/.

[9] Red Hat. *Why develop Java apps with Quarkus on Red Hat OpenShift?* June 2021. url: https://www.redhat.com/en/technologies/cloud-computing/openshift/quarkus.

[10] Faisal Tariq and Muhammad Khandaker et al. "A Speculative Study on 6G". In: *arXiv preprint arXiv:1902.06700* (2019). doi: 10.48550/arXiv.1902.06700. url: https://arxiv.org/abs/1902.06700.

[11] Bashair Althani and Souheil Khaddaj. "Systematic Review of Legacy System Migration". In: (2017). doi: 10.1109/DCABES.2017.41.

[12] Briony J. Oates. *Researching Information Systems and Computing*. London; Thousand Oaks, Calif.: SAGE Publications Ltd, Jan. 2006. isbn: 978-1-4129-0223-6.

[13] Instituto Politécnico do Porto. *Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto*. Online. Diário da República, 2.a série PARTE E Artigo 2.o. url: https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf.

[14] Donald Gotterbarn, Keith Miller, and Simon Rogerson. "Software engineering code of ethics". In: *Communications of the ACM* 40.11 (Nov. 1997), pp. 110–118. doi: 10.1145/265684.265699.

[15] Damian Eke and Bernd Stahl. "Ethics in the governance of data and digital technology: An analysis of European data regulations and policies". en. In: *Digit. Soc.* 3.1 (2024). url: https://link.springer.com/article/10.1007/s44206-024-00101-6.

[16] Altice Labs. *NOSSIS One*. May 2024. url: https://www.alticelabs.com/wp-content/uploads/2024/05/BR_NOSSISONE_ALB_EN.pdf.

[17] Altice Labs. *Altaia: End-to-End Assurance Solution*. Oct. 2021. url: https://www.alticelabs.com/wp-content/uploads/2021/10/PACALTAIA_BRP_ALB_ING.pdf.

[18] Brendan Burns and Brian Grant et al. "Borg, Omega, and Kubernetes". In: *Communications of the ACM* 59.5 (2016), pp. 50–57. doi: 10.1145/2890784. url: https://doi.org/10.1145/2890784.

[19] Andrew Jeffery, Heidi Howard, and Richard Mortier. "Rearchitecting Kubernetes for the Edge". In: *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. 2021, pp. 7–12. doi: 10.1145/3434770.3459730. url: https://doi.org/10.1145/3434770.3459730.

[20] Giovanni Toffetti and Alberto Martinez Ballesteros et al. "An Architecture for Self-Managing Microservices". In: *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*. 2015, pp. 19–24. doi: 10.1145/2747470.2747474. url: https://doi.org/10.1145/2747470.2747474.

[21] Ghofrane El Haj Ahmed, Felipe Gil-Castiñeira, and Enrique Costa-Montenegro. "KubCG: A Dynamic Kubernetes Scheduler for Heterogeneous Clusters". In: *Software: Practice and Experience* 51.2 (2021), pp. 213–234. doi: 10.1002/spe.2898. url: https://doi.org/10.1002/spe.2898.

[22] Zhiheng Zhong and Rajkumar Buyya. "A Cost-Efficient Container Orchestration Strategy in Kubernetes-Based Cloud Computing Infrastructures with Heterogeneous Resources". In: *ACM Transactions on Internet Technology* 20.2 (2020), pp. 1–24. doi: 10.1145/3378447. url: https://doi.org/10.1145/3378447.

[23] Matija Sipek, Branko Mihaljevic, and Aleksander Radovan. "Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM". In: *CoRR* abs/2112.14716 (2021). arXiv: 2112.14716. url: https://arxiv.org/abs/2112.14716.

[24] Alex Soto Bueno and Jason Porter. *Quarkus Cookbook: Kubernetes-Native Java Frameworks*. O'Reilly Media, Inc., 2020. isbn: 9781492062646. url: https://www.oreilly.com/library/view/quarkus-cookbook/9781492062646/.

[25] Lukasz Wyciślik, Lukasz Latusik, and Anna Malgorzata Kamińska. "A Comparative Assessment of JVM Frameworks to Develop Microservices". In: *Applied Sciences* 13.3 (2023). issn: 2076-3417. doi: 10.3390/app13031343. url: https://www.mdpi.com/2076-3417/13/3/1343.

[26] Spring.io. *Spring Framework Official Website*. https://spring.io/.

[27] Micronaut.io. *Micronaut Framework Official Website*. https://micronaut.io/.

[28] Sam Newman. *Monolith Decomposition Patterns*. 60 min presentation at NDC Oslo 2019, Oslo Spektrum, Oslo, Norway. June 2019. url: https://samnewman.io/talks/monolith-decomposition-patterns/.

[29] Sam Newman. *Building Microservices: Designing FineGrained Systems*. 2nd. O'Reilly Media, Inc., Oct. 2021. isbn: 978-1492034025. url: https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/.

[30] Sam Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Inc., Nov. 2019. isbn: 978-1492047841. url: https://www.oreilly.com/library/view/monolith-to-microservices/9781492047834/.

[31] Sam Newman. *Pattern: UI Composition*. Dec. 2019. url: https://samnewman.io/patterns/architectural/ui-composition/.

[32] Sam Newman. *Pattern: Branch By Abstraction*. Online; Sam Newman's website. Dec. 2019. url: https://samnewman.io/patterns/architectural/branch-by-abstraction/.

[33] Sam Newman. *Pattern: Backends For Frontends*. Online; Sam Newman's website. Nov. 2015. url: https://samnewman.io/patterns/architectural/bff/.

[34] Majid Zamiri and Ali Esmaeili. "Strategies, Methods, and Supports for Developing Skills within Learning Communities: A Systematic Review of the Literature". In: *Administrative Sciences* 14.9 (2024), p. 231. doi: 10.3390/admsci14090231. url: https://doi.org/10.3390/admsci14090231.

[35] Gopal Kapur. "Covering Project Skill Gaps". In: *PM Network* (2003). url: https://www.pmi.org/learning/library/covering-project-skill-gaps-5007.

[36] Mary Ellen Berger. "Contribution of Individual Project Participant Competencies to Project Success". In: *PMI® Research Conference: Defining the Future of Project Management* (2010). url: https://www.pmi.org/learning/library/individual-project-competencies-project-success-6457.

[37] Monash University Library. *Developing Research Questions*. url: https://www.monash.edu/library/help/assignments-research/developing-research-questions.

[38] Andrei Furda and C. Fidge et al. "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency". In: *IEEE Software* 35 (2018), pp. 63–72. doi: 10.1109/MS.2017.440134612.

[39] Vinicius L. Nogueira and Fernando S. Felizardo et al. "Insights on Microservice Architecture Through the Eyes of Industry Practitioners". In: *ArXiv* abs/2408.10434 (2024). doi: 10.48550/arXiv.2408.10434.

[40] G. Blinowski, Anna Ojdowska, and Adam Przybyłek. "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation". In: *IEEE Access* 10 (2022), pp. 20357–20374. doi: 10.1109/ACCESS.2022.3152803.

[41] Freddy Tapia and M. Mora et al. "From Monolithic Systems to Microservices: A Comparative Study of Performance". In: *Applied Sciences* (2020). doi: 10.3390/app10175797.

[42] Tolu Koleoso. *Microservices with Quarkus*. Apress, 2020. doi: 10.1007/978-1-4842-6032-6_3. url: https://doi.org/10.1007/978-1-4842-6032-6_3.

[43] Anderson Rossetto and Daniel Noetzold et al. "Enhancing Monitoring Performance: A Microservices Approach to Monitoring with Spyware Techniques and Prediction Models". In: *Sensors (Basel, Switzerland)* 24 (2024). doi: 10.3390/s24134212. url: https://doi.org/10.3390/s24134212.

[44] Tolu Koleoso. *Welcome to Quarkus*. Apress, 2020. doi: 10.1007/978-1-4842-6032-6_1. url: https://doi.org/10.1007/978-1-4842-6032-6_1.

[45] Zhongshan Ren and Wen Wang et al. "Migrating Web Applications from Monolithic Structure to Microservices Architecture". In: *Proceedings of the 10th Asia-Pacific Symposium on Internetware* (2018). doi: 10.1145/3275219.3275230.

[46] S. Dhomne, T. Upendar, and G. Soni. "A Comprehensive Study for Developing a Framework for Microservices Architecture Migration". In: *International Journal of Scientific Research in Engineering and Management* (2023). url: https://doi.org/10.55041/ijsrem27200.

[47] M. Seedat and Q. Abbas et al. "Transition Strategies from Monolithic to Microservices Architectures: A Domain-Driven Approach and Case Study". In: *VAWKUM Transactions on Computer Sciences* (2024). url: https://doi.org/10.21015/vtcs.v12i1.1808.

[48] J. Kazanavicius and D. Mazeika. "An Approach to Migrate from Legacy Monolithic Application into Microservice Architecture". In: (2023). url: https://doi.org/10.1109/eStream59056.2023.10135021.

[49] A. Kiani. and A. Sarwar. et al. "Catalysing Monolithic to Microservices Migration: A Strategic Approach Using Refactoring and Pilot Projects". In: (2024).

[50] A. Freire. and A. Sampaio et al. “Migrating Production Monolithic Systems to Microservices Using Aspect Oriented Programming”. In: *Software: Practice and Experience* 51 (2021). url: https://doi.org/10.1002/spe.2956.

[51] B. Bamberger and B. Körber. “Migrating Monoliths to Microservices Integrating Robotic Process Automation into the Migration Approach”. In: *Journal of Automation, Mobile Robotics and Intelligent Systems* 16 (2022). url: https://doi.org/10.14313/jamris/1-2022/8.

[52] M. Kyryk and O. Tymchenko et al. “Methods and process of service migration from monolithic architecture to microservices”. In: *2022 IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET)* (2022), pp. 553–558. doi: 10.1109/tcset55632.2022.9767055.

[53] M. Kaloudis. “Evolving Software Architectures from Monolithic Systems to Resilient Microservices: Best Practices, Challenges and Future Trends”. In: *International Journal of Advanced Computer Science and Applications* 15.9 (2024). url: https://doi.org/10.14569/ijacsa.2024.0150901.

[54] Daniel Yeri Kristiyanto and R. Sarno et al. “Comprehensive Framework for Transitioning Monolithic to Microservices in MVC Context”. In: *2024 3rd International Conference on Creative Communication and Innovative Technology (ICCIT)* (2024), pp. 1–7. doi: 10.1109/ICCIT62134.2024.10701144.

[55] A. M. Saucedo and Guillermo Rodríguez. “Migration of Monolithic Systems to Microservices using AI: A Systematic Mapping Study”. In: (2024), pp. 1–15. doi: 10.5753/cibse.2024.28435.

[56] Zhijun Ding and Yuehao Xu et al. “Microservice Extraction Based on a Comprehensive Evaluation of Logical Independence and Performance”. In: *IEEE Transactions on Software Engineering* 50 (2024), pp. 1244–1263. doi: 10.1109/TSE.2024.3380194.

[57] Yuan-Kai Chen and Chang-Ping Hsu et al. “TOSS: Telecom Operations Support Systems for Broadband Services”. In: *Journal of Information Processing Systems* 6.1 (Mar. 2010), pp. 1–13. doi: 10.3745/JIPS.2010.6.1.001.

[58] Dennis Gannon, R. Barga, and Neel Sundaresan. “Cloud-Native Applications”. In: *IEEE Cloud Comput.* 4 (2017), pp. 16–21. doi: 10.1109/MCC.2017.4250939.

[59] C. Pahl, Pooyan Jamshidi, and O. Zimmermann. “Architectural Principles for Cloud Software”. In: *ACM Transactions on Internet Technology (TOIT)* 18 (2018), pp. 1–23. doi: 10.1145/3104028.

[60] Oyekunle Claudius Oyeniran and Oluwole Temidayo Modupe et al. “A comprehensive review of leveraging cloud-native technologies for scalability and resilience in software development”. In: *International Journal of Science and Research Archive* (2024). doi: 10.30574/ijsra.2024.11.2.0432.

[61] Dhanyaa Bharadwaj and B. S. Premananda. “Transition of Cloud Computing from Traditional Applications to the Cloud Native Approach”. In: *2022 IEEE North Karnataka Subsection Flagship International Conference (NKCon)* (2022), pp. 1–8. doi: 10.1109/NKCon56289.2022.10126871.

[62] Nane Kratzke and R. Peinl. “ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects”. In: *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)* (2016), pp. 1–10. doi: 10.1109/EDOCW.2016.7584353.

[63] Robin Lichtenthaler, J. Fritzsch, and G. Wirtz. “Cloud-Native Architectural Characteristics and their Impacts on Software Quality: A Validation Survey”. In: *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)* (2023), pp. 9–18. doi: 10.1109/SOSE58276.2023.00008.

[64] Jordan Jordanov and Pavel Petrov. "Domain Driven Design Approaches in Cloud Native Service Architecture". In: *TEM Journal* (2023). doi: 10.18421/tem124-09.

[65] Guilherme Gil, Daniel Corujo, and P. Pedreiras. "Cloud Native Computing for Industry 4.0: Challenges and Opportunities". In: *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (2021), pp. 01–04. doi: 10.1109/ETFA45728.2021.9613386.

[66] D. M. K. Reddy and Shaik Ahmad Nawaz et al. "Design and Implementation of a Cloud Native Application for Collaborative Learning". In: *International Journal for Research in Applied Science and Engineering Technology* (2024). doi: 10.22214/ijraset.2024.59296.

Appendix A

Appendix

The Figure A.1 illustrates one of the sections of the Gantt chart for the project timeline.

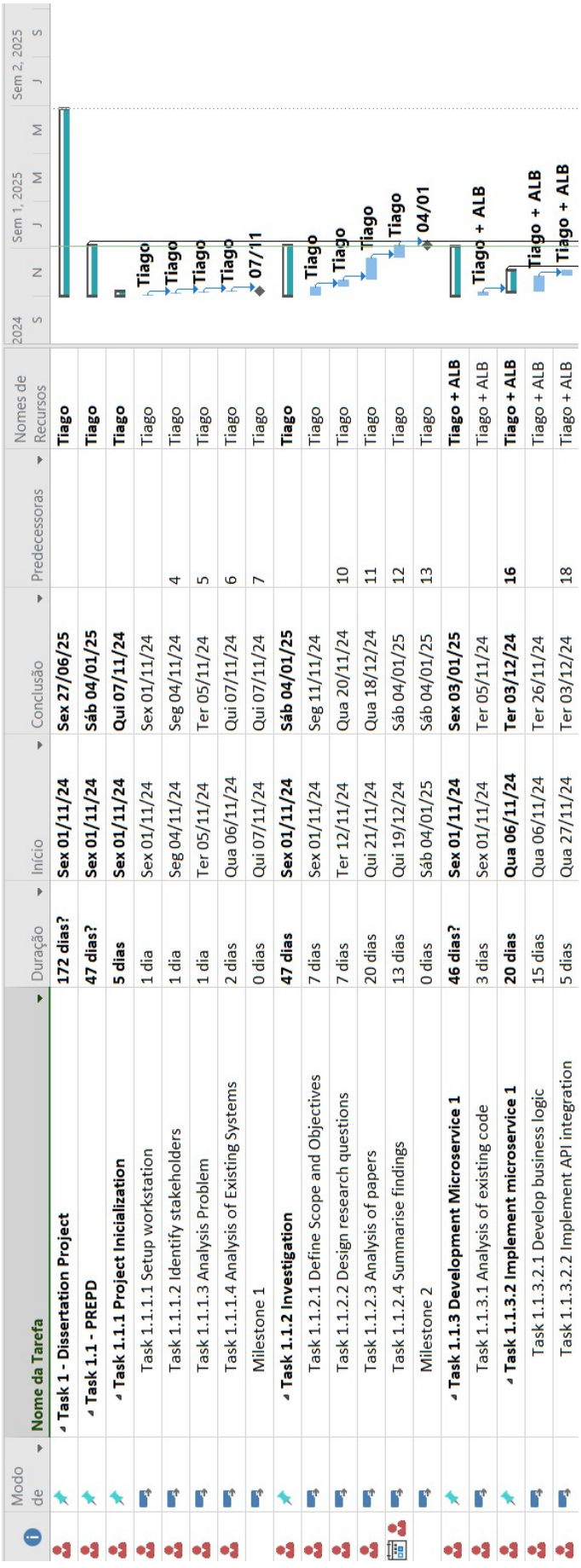
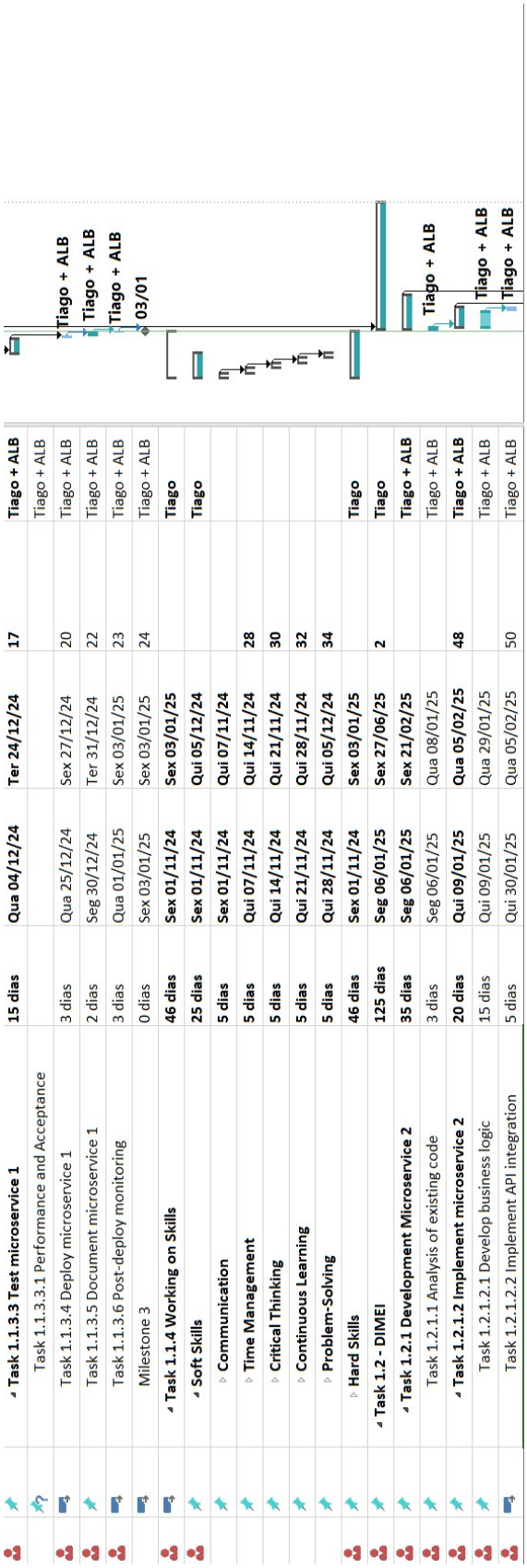


Figure A.1: Gantt chart part 1

The Figure A.2 illustrates one of the sections of the Gantt chart for the project timeline.



	Task Name	Duration	Start	Finish	Predecessors	Resource Names
	◢ **Task 1.1.3.3 Test microservice 1**	**15 dias**	**Qua 04/12/24**	**Ter 24/12/24**	**17**	**Tiago + ALB**
	Task 1.1.3.3.1 Performance and Acceptance					Tiago + ALB
	Task 1.1.3.4 Deploy microservice 1	3 dias	Qua 25/12/24	Sex 27/12/24	20	Tiago + ALB
	Task 1.1.3.5 Document microservice 1	2 dias	Seg 30/12/24	Ter 31/12/24	22	Tiago + ALB
	Task 1.1.3.6 Post-deploy monitoring	3 dias	Qua 01/01/25	Sex 03/01/25	23	Tiago + ALB
	Milestone 3	0 dias	Sex 03/01/25	Sex 03/01/25	24	Tiago + ALB
	◢ **Task 1.1.4 Working on Skills**	**46 dias**	**Sex 01/11/24**	**Sex 03/01/25**		**Tiago**
	◢ **Soft Skills**	**25 dias**	**Sex 01/11/24**	**Qui 05/12/24**		**Tiago**
	▷ **Communication**	**5 dias**	**Sex 01/11/24**	**Qui 07/11/24**		
	▷ **Time Management**	**5 dias**	**Qui 07/11/24**	**Qui 14/11/24**	**28**	
	▷ **Critical Thinking**	**5 dias**	**Qui 14/11/24**	**Qui 21/11/24**	**30**	
	▷ **Continuous Learning**	**5 dias**	**Qui 21/11/24**	**Qui 28/11/24**	**32**	
	▷ **Problem-Solving**	**5 dias**	**Qui 28/11/24**	**Qui 05/12/24**	**34**	
	▷ **Hard Skills**	**46 dias**	**Sex 01/11/24**	**Sex 03/01/25**		**Tiago**
	◢ **Task 1.2 - DIMEI**	**125 dias**	**Seg 06/01/25**	**Sex 27/06/25**	**2**	**Tiago**
	◢ **Task 1.2.1 Development Microservice 2**	**35 dias**	**Seg 06/01/25**	**Sex 21/02/25**		**Tiago + ALB**
	Task 1.2.1.1 Analysis of existing code	3 dias	Seg 06/01/25	Qua 08/01/25		Tiago + ALB
	◢ **Task 1.2.1.2 Implement microservice 2**	**20 dias**	**Qui 09/01/25**	**Qua 05/02/25**	**48**	**Tiago + ALB**
	Task 1.2.1.2.1 Develop business logic	15 dias	Qui 09/01/25	Qua 29/01/25		Tiago + ALB
	Task 1.2.1.2.2 Implement API integration	5 dias	Qui 30/01/25	Qua 05/02/25	50	Tiago + ALB

Figure A.2: Gantt chart part 2

The Figure A.3 illustrates one of the sections of the Gantt chart for the project timeline.

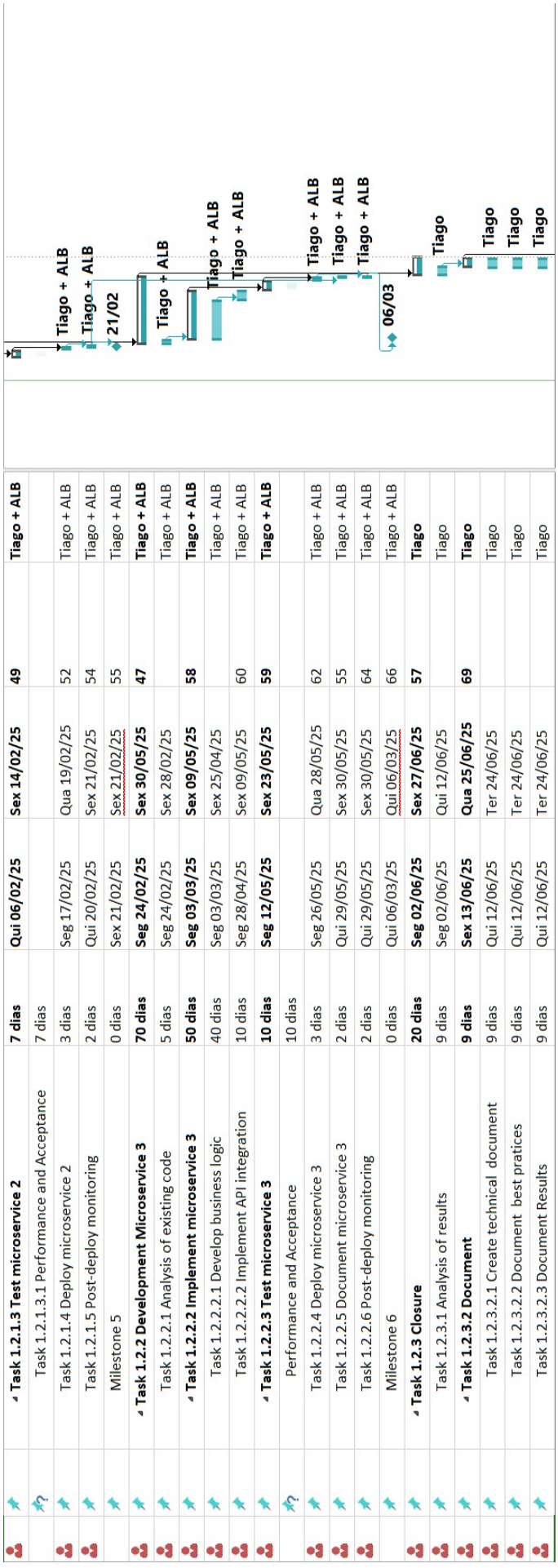


Figure A.3: Gantt chart part 3

The Figure A.4 illustrates one of the sections of the Gantt chart for the project timeline.

Figure A.4: Gantt chart part 4