INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Exploring maintainability and performance in Ballerina Microservices

ANDRÉ FILIPE RIBEIRO ALVES
junho de 2023

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Exploring maintainability and performance in Ballerina Microservices

André Filipe Ribeiro Alves

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Supervisor: Isabel de Fátima Silva Azevedo

Porto, June 30, 2023

Dedicatory

To my family, who gave me the strength to fulfill my objectives and to not give up.

To my friends, who stayed on my side during this journey and made it enjoyable.

To my supervisor, who supported and guided me in the right direction.

And finally, to myself, for being able to accomplish this work.

Statement of Integrity

I hereby declare having 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.

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

ISEP,June 2023

André Filipe Ribeiro Alves

Abstract

Microservices architecture currently is the industry norm for creating applications since it allows teams to focus on individual services related to specific business functionalities, reducing the overall application complexity and improving its maintainability. However, microservices architecture has liabilities regarding service integration, communication, governance, and data management. To solve these liabilities, the industry and academic community have focused on creating new frameworks and solutions. More recently, the focus changed to creating new programming languages focused on microservices.

This study aims to investigate the effects of language-oriented approaches in developing microservices. The work focuses on Ballerina, a programming language created to simplify the creation and integration of microservices. From the literature's analysis, Ballerina demonstrates the ability to be more beneficial than the most common implementations of microservices that use more common frameworks. To further investigate these statements, an experience based on the migration of existing microservices developed in Java with the use of the framework Spring Boot was conducted. This experience used a migration strategy created based on the language's specificities.

The resulting Ballerina microservice is compared with its original counterpart. The experience focused on analyzing both solutions in terms of maintainability and performance. Therefore, the Goals, Questions, Metrics (GQM) approach was used to obtain metrics for the mentioned quality attributes. From the obtained results, it was concluded that the Ballerina solution presents differences from the Spring Boot solution, being superior regarding maintainability and inferior in performance.

Keywords: Microservice Architecture, Ballerina, Language-Based Approach, Spring Boot

Resumo

A arquitetura baseada em microserviços atualmente é dentro da indústria considerada a norma para a criação de aplicações. A arquitetura baseada em microserviços apresenta diversas vantagens, como a possibilidade de distribuir as diferentes funcionalidades da aplicação por diversas equipas, possibilitando também o desenvolvimento dos seus serviços em linguagens de programação diferentes. Esta modularização traduz-se num acréscimo à independência das equipas, possibilitando a implementação dos seus próprios processos de trabalho como também reduz a complexidade total da aplicação, aumentando a sua manutenibilidade.

Contudo, a arquitetura baseada em microserviços, dado o aumento na sua utilização, deu origem a novas questões e preocupações para as equipas de desenvolvimento e para as organizações. As equipas de desenvolvimento encontraram dificuldades na integração e comunicação dos seus serviços, principalmente quando estes eram desenvolvidos em linguagens de programação bastante divergentes. Apesar da modularidade e desacoplamento dos serviços, começou a ser detetada uma grande dependência com metodologias de implantação do código, onde o sucesso de microserviços encontrava-se fortemente dependente do uso de contentores e sistemas de orquestração. As organizações identificaram um aumento nos seus custos devido ao aumento de processos, armazenamento em servidores e de software para suporte às diversas equipas.

Estas dificuldades advêm do facto de as linguagens de programação habitualmente usadas para a criação de microserviços não terem sido edificadas com o objetivo de suportar microserviços bem como não salvaguardam algumas das preocupações identificadas anteriormente. De forma a combater estas dificuldades, a indústria e a comunidade académica incidiram as suas investigações em novas *frameworks* e soluções. Recentemente, dado o novo foco na otimização para o "desenvolvimento", surgiram novas linguagens de programação orientadas a microserviços.

Este estudo tem como objetivo investigar os efeitos da utilização de linguagens de programação orientadas a microserviços e subsequentemente a análise do seu impacto relativamente a atributos de qualidade relevantes para os mesmos. O foco deste estudo incidirá na linguagem de programação Ballerina, uma linguagem de

programação criada com o intuito de facilitar a criação de serviços resilientes capazes de se integrarem e orquestrarem através de pontos de saída distribuídos. Foi realizada uma investigação à literatura existente sobre microserviços e sobre Ballerina, com especial atenção na literatura referente à utilização de Ballerina em microserviços.

Esta investigação objetivou a identificação dos atributos de qualidade relevantes, de onde se precisou a manutenibilidade e a performance como sendo os atributos a avaliar. Derivada da análise da literatura, quando confrontada com implementações de linguagens de programação mais tradicionais, Ballerina demonstra a capacidade de proporcionar benefícios aquando da sua utilização em microserviços. Ballerina apresenta vantagens tais como o suporte nativo para a utilização de contentores, suporte a *DevOps* e funcionalidades focadas na segurança e resiliência das aplicações.

De forma a aferir estas afirmações, foi efetuada uma experiência controlada baseada na migração de microserviços desenvolvidos na linguagem de programação Java com recurso à *framework* Spring Boot para a linguagem Ballerina. Para a realização desta experiência, primeiramente foi escolhido um projeto base do qual se realizaria a migração. O projeto a migrar teria de cumprir alguns requisitos, tais como: ser *open-source*, ter atividade recente e um número significativo de microserviços. Com base nestes requisitos, foi selecionado o projeto Lakeside Mutual, um projeto sobre uma companhia de seguros fictícia, desenvolvido com o propósito de demonstrar a utilização de padrões sobre APIs e design orientado ao domínio. Seguidamente, foi desenhada uma estratégia de migração com base nas especificidades da linguagem.

Dada a implementação da estratégia, foi edificado um serviço em Ballerina à semelhança do original implementado em Java, mantendo todos os seus atributos e funcionalidades. As soluções foram comparadas em termos de manutenibilidade e performance, em que para tal foi utilizada a abordagem Goal, Question, Metrics (GQM) de forma a obter métricas para os atributos de qualidade previamente mencionados. Em termos de manutenibilidade, foram analisados os números de linhas de código existentes e a complexidade calculada através dos níveis de indentação dado que Ballerina ainda não suporta ferramentas como o Sonarqube. Relativamente à performance, foi desenhado e implementado um plano de testes com um número variado de utilizadores virtuais que executavam um conjunto de pedidos REST.

Dos resultados obtidos, foi concluído que existem diferenças notáveis entre ambas as soluções. A solução em Ballerina apresenta melhores resultados do que a solução em Spring Boot relativamente à manutenibilidade. Contudo, relativamente à performance, a solução em Spring Boot demonstra obter melhores resultados.

Acknowledgement

Firstly, I want to thank my thesis' supervisor, Professor Isabel de Fátima Silva Azevedo, who supported me and for the availability and engagement when discussing the work for the dissertation.

Secondly, I would like to thank all my family and friends who always supported me during this dissertation and giving the strength to finish it.

Finally, I want to thank all the professors and colleagues who helped me along my journey at ISEP for helping me attain success in both my academic and professional career.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Abbreviations

AHP	Analytic Hierarchy Process.
BVM	Ballerina Virtual Machine.
CI	Consistency Index.
CLI	Command-line Interface.
CORS	Cross-Origin Resource Sharing.
CR	Consistency Ratio.
DDD	Domain Driven Design.
DEI	*Departamento de Engenharia Informática*.
DSL	Domain-Specific Language.
DSR	Design Science Research.
DSRM	Design Science Research Methodology.
DTO	Data Transfer Object.
ESB	Enterprise Service Bus.
FFE	Fuzzy Front End.
GPL	General-purpose programming language.
GQM	Goal Question Metric.
ISEP	*Instituto Superior de Engenharia do Porto*.
JDBC	Java Database Connectivity.
JPA	Java Persistence API.
JWT	JSON Web Token.
LOC	Lines of Code.

MEI	*Mestrado em Engenharia Informática*.
MOSIP	Modular Open Source Identity Platform.
NCD	New Concept Development.
NPD	New Product Development.
RI	Random Index.
SOA	Service-Oriented Architecture.
SRP	Single Responsibility Principle.
TCP	Transmission Control Protocol.
TMDEI	*Tese / Dissertação / Estágio Mestrado de Engenharia Informática*.

Chapter 1

Introduction

This dissertation illustrates a thesis project developed during the Master's Degree in Software Engineering at Instituto Superior de Engenharia do Porto (ISEP). The document covers all the crucial actions and information that led to its accomplishment.

This chapter contains the document's introduction. It presents the interpretation of the problem to be solved and the context of the dissertation. It also outlines the objectives necessary to resolve the mentioned problem. Next, it describes the chosen approach.

Lastly, this chapter also provides a summary of the document's structure.

1.1 Context

In today's society, applications have taken a significant role in daily life. Applications are becoming more complex, innovating, and updating to include various functionalities to satisfy the users' needs.

For many years, teams created most applications to serve all the needs of an individual business line. While they could be fully monolithic or separated by services, they encapsulated all the necessary business requirements and functionalities into a single unit. With innovation, they became overly complex and difficult to maintain (Indrasiri and Siriwardena 2018).

A new architecture emerged to solve these issues. It was the microservices architecture. This architecture focuses on creating a single microservice related to one business functionality. The solutions run their processes, and teams develop and deploy them independently. Microservices architecture allows each service to be developed in different programming languages since services communicate through their defined endpoints (Indrasiri and Siriwardena 2018).

As time passed, the microservices architecture propelled the creation of new programming languages more focused on its use cases and concerns. One of the examples is the Ballerina programming language, released in 2018 by WSO2.

Ballerina is a programming language and platform that "make it easy to create resilient services that integrate and orchestrate across distributed endpoints". It focuses on "baking integration concepts into a language, including a network-aware type system, sequence diagrammatic syntax, concurrency workers, being "DevOps ready", and environment awareness" (Jewell 2018).

Over the last few years, Ballerina has been going up on the programming languages popularity rankings, reaching its current ranking of 87th most popular programming language with around five years of production lifetime. RedMonk describes this increase in popularity as an exception to the rule where newly minted programming languages are not expected to cause impacts on the overall ranking (RedMonk 2022c). This exception emphasizes the increasing adoption of this programming language, making it relevant for possible studies on its use.

1.2 Problem

Microservices architecture caused a shift in the industry, but it also came with liabilities, such as the integration, communication, governance, and data management of different microservices written in various programming languages (Indrasiri and Siriwardena 2018).

These liabilities came from the creators of the most conventional programming languages not designing and creating their languages with microservices in mind, which resulted in the programmers needing to use more and more of their time to resolve issues that arose from this transition. As such, it led to the invention of frameworks and solutions to solve these issues (Guidi et al. 2017). More recently, it led to the development of new programming languages focused on microservices like Ballerina.

However, while Ballerina appears as a solution to the problems that are relevant in applications with multiple microservices, it also leads to the question of how applications that use it compare with other more conventional approaches in terms of some quality attributes, a question that remains unexplored.

1.3 Objectives

The main objective of this dissertation is to explore the effects of using a programming language such as Ballerina versus a more conventional programming language regarding maintainability and performance. This objective will be accomplished by following these steps:

- Study the Ballerina programming language.
- Create a research prototype with microservices architecture using the Ballerina programming language by migrating an open-source project.
- Explore the effects of using Ballerina in microservices in terms of maintainability and performance through the analysis of both solutions.

Maintainability and performance are the selected quality attributes to be evaluated in both solutions. Section 3.2 describes the analysis that led to choosing these quality attributes.

1.4 Research Questions

The investigation required a starting point, so the following research questions were established to carry out this research:

- What are the most relevant quality attributes related to microservices architecture?
- What are the effects of using Ballerina in microservices architecture-based applications regarding the identified quality attributes?

The first research question focuses on researching the most relevant quality attributes related to the microservices architecture. These quality attributes will be used as the main points of interest in analyzing the effects of the adoption of Ballerina. The second and last research question concentrates on the research of how developer teams can use Ballerina to create microservices and what are the impacts of using this language regarding the identified quality attributes.

1.5 Approach

This dissertation investigates the possible effects of integrating Ballerina in a project based on the microservices architecture, focusing on a group of quality attributes. Therefore, a research method that provides reliability and accomplishment is needed.

The approach that initially seemed to align best with this dissertation's context was the Design Science Research Methodology (DSRM) applied to information systems. However, considering the nature of the research, an alternative methodology inspired by DSRM principles was adopted instead.

DSRM is a framework that presents the necessary guidelines for successfully evaluating the present Design Science Research (DSR) in information systems. It includes the principles, practices, and procedures required to carry out a study and meet its objectives(Peffers et al. 2007).

It's important to note that the chosen approach does not have a specific name. The decision to deviate from DSRM emerged since DSRM predominantly focuses on delivering a new product, whereas this dissertation centers around the investigation and subsequent findings related to the migration of an already developed project.

The procedures carried out to ascertain this investigation's reliability, accomplishment, and optimization are comprised of six steps:

- Problem identification and motivation: this step comprises the definition of the targeted problem of the research and the justification of how the applied solution brings value in this context. In this document, section 1.2 details the problem, and the value of the proposed solution is in Chapter 4;
- Objectives for a solution: this step defines the objectives related to the problem and its constraints. This step requires a study and assessment of several available approaches and their assets and drawbacks. The objectives can be quantitative (where a solution can be more desirable over the current ones) or qualitative (how a certain artifact can be the solution to a given constraint). The objectives are defined in section 1.3;
- Design and development: this step represents the design and implementation of the selected solution. The design unfolds from analyzing the State of the Art, which details frameworks, patterns, techniques, and technologies that allow the development of a proof of concept that serves as a representation of the selected solution and achieves the specified objectives. This step is described in Chapters 5 and 6;
- Demonstration: this step describes and exhibits how effectively the selected solution resolves the existing problem. Chapter 7 will describe the experimentation and will show evidence of how well the implemented solution tackles the known problem.
- Evaluation: this step detects and measures the effectiveness of the implementation for the chosen solution, utilizing metrics and analyzing the obtained

results. Chapter 7 also describes this effectiveness, comparing the achieved results in the demonstration with the initial objectives.

- Communication: the final step communicates the delivery of the problem, its importance, the selected solution, and its effectiveness. As such, Chapter 8 will communicate the conclusions obtained through the work done in this research.

1.6 Document Structure

The current chapter represents the Introduction. This chapter presents the context and the interpretation of the problem to be solved and defines the objectives to achieve. It also describes the preferred approach to the problem and ends with the document structure.

The Background chapter introduces the microservices architecture, listing its main concepts and enumerating its benefits and liabilities. This chapter also describes the Ballerina programming language, starting with its history, then focusing on its key concepts, and ending the chapter with an industry use case of Ballerina.

The State of the Art chapter describes how developers can use Ballerina to create microservices, depicting the features that bring benefits to the microservices architecture-based applications that use this programming language. It also focuses on the performed analysis of quality attributes, where maintainability and performance were identified as the relevant quality attributes for this dissertation.

The following chapter represents the Value Analysis. This chapter describes the Innovation Process, including its theoretical specification and its implementation in the context of this dissertation. Also, this chapter represents the utilization of the Analytic Hierarchy Process (AHP), whose focus is to aid, based on multiple criteria, in selecting the approach to use for the creation of the proof of concept.

The chapter on Analysis and Design focuses on describing the analysis and creation of the design of the migration of a selected project using the approach chosen in the Value Analysis chapter.

The Implementation chapter will showcase the implemented solution. It details the transition of the base project to Ballerina, with demonstrations of how the code was migrated.

Next, the chapter Evaluation and Experimentation will validate the implemented solution based on the Goal Question Metric (GQM) approach and the performed tests on the implemented solution for maintainability and performance. The chapter

finishes with an overall summary of the conclusions obtained from the performed experiments.

The final chapter of the documents is the Conclusions. This chapter will start by addressing the achievements related to the defined objectives. Then, it describes the found limitations and outlines the possible future work. Finally, the chapter ends with the concluding considerations and contributions.

Chapter 2

Background

The following chapter provides some essential knowledge about microservices and the Ballerina programming language. First, microservices are introduced, mentioning their key concepts, benefits, and liabilities. Then, Ballerina's history and key features are detailed, along with an industry use case.

2.1 Microservices Architecture

For a long time, the monolithic architecture dictated how the software development process was designed and implemented. Applications built using this approach serve as a single unit. This unit is responsible for meeting all the needs and requirements of the business. Also, this unit is often written in a single programming language, uses a single shared database, and exists in a single repository and host machine (Madushan 2021).

However, monolithic architecture has many problems, especially with the maintenance of the applications, as they can grow into large, bulky systems. These problems led to the development of a new architectural approach called Service Oriented Architecture (SOA). The core idea of SOA is to create loosely coupled, reusable, and easily scalable units called services. Then, an application server hosts these services (Indrasiri and Siriwardena 2018).

SOA solved many of the problems that emerged from the use of monolithic applications. However, since it is also monolithic by nature, it could not accompany the evolutions of modern application development needs. The developers wanted to build more scalable and flexible systems that could take advantage of the various technologies available and operate more independently with fewer inter-application dependencies.These requirements led to the microservices architecture (Madushan 2021).

Microservices architecture is an architectural approach for quickly and securely creating software applications. This approach focuses on building a software system as a collection of self-contained and independent services that execute their processes and are developed, deployed, and scaled separately. Integrating all these services and other systems creates a software application like the one shown in Figure 2.1 (Indrasiri and Siriwardena 2018).

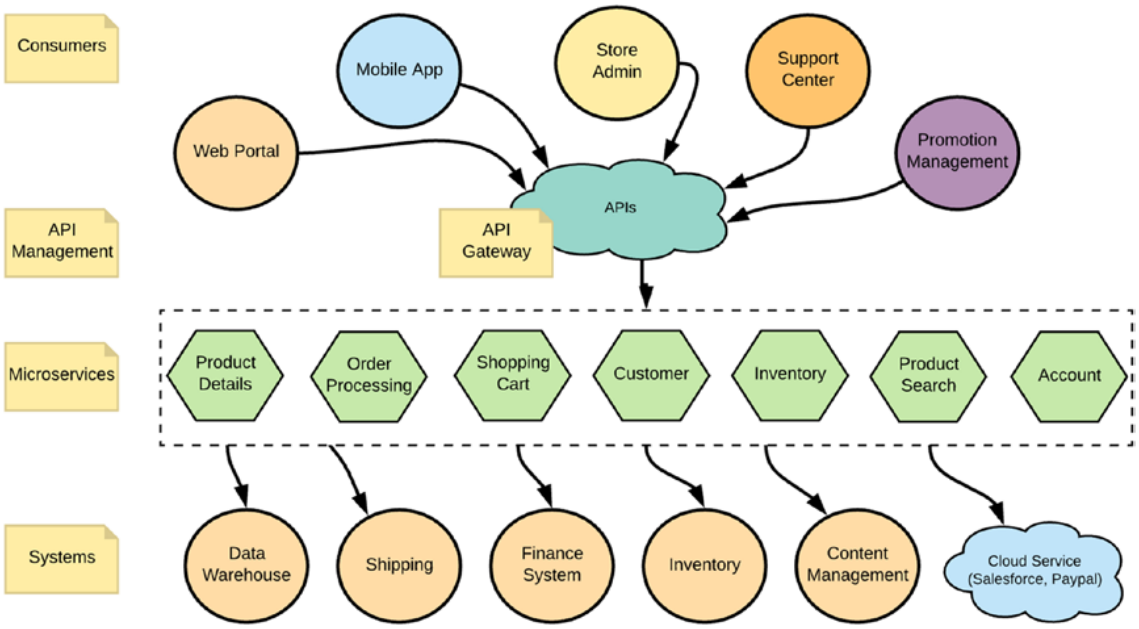

Figure 2.1: Example of application built using microservices architecture (Indrasiri and Siriwardena 2018)

2.1.1 Microservices Key Concepts

This section will briefly overview some of the main characteristics of the microservices architecture.

Business Capability Oriented

One of the fundamental ideas behind the microservices architecture is a service per business capability. Doing so will result in each service having a clear set of responsibilities for a specific business purpose. A given service should concentrate on performing one specific task exceptionally well (Newman 2021).

As such, the service's size depends on the scope and the business functionality. If a service is too small (has a small granularity, segregating a business functionality into small-scaled services) or too big (like a web service), it is not adequate for the microservices architecture. For identifying the scope and business functionalities of

a microservice, concepts like the Single Responsibility Principle (SRP) or Domain Driven Design (DDD) are used (Indrasiri and Siriwardena 2018).

Autonomous: Develop, Deploy, and Scale Independently

Unlike monolithic applications or web services, microservices are developed, deployed, and scaled independently, meaning they do not share execution runtimes between them. This independence comes from the use of containers, a technology that is increasingly used and provides service autonomy that is beneficial to the success of the microservices architecture (Newman 2021).

This autonomy makes the entire system more resilient by isolating failures within individual services. In addition, each service scales independently, so the other services that have more traffic can be managed separately without impacting other services (Indrasiri and Siriwardena 2018).

No Central ESB: Smart Endpoints and Dumb Pipes

Microservices architecture promotes the elimination of the use of the ESB. Instead of using an ESB, microservices architecture introduces a new service integration style called *smart endpoints and dumb pipes*. This style defines the microservice as the home of all the business logic, including the inter-service communication logic, making them *smart endpoints*. All services are then connected using a simple messaging infrastructure devoid of any business logic, serving as a *dumb pipe*. With this style, all the developed microservices encompass the entire complexity of the ESB (Indrasiri and Siriwardena 2018).

Failure Tolerance

Due to the increasing number of services and inter-service network communications, microservices are more prone to failures. A microservice application is a collection of these services, so the malfunction of one or more services must not cause the entire application to fail. Therefore, the microservices developed must be fault-tolerant so that in the event of an unexpected failure, the impact is minimal (Indrasiri and Siriwardena 2018).

Another aspect to consider is the ability to observe and predict failures in microservices. If failures can be predicted and observed, services can recover faster. Also, the symptoms can be investigated to avoid them in the future. This capability comes

from a good observability infrastructure with monitoring, tracing, logging, and other practices (Indrasiri and Siriwardena 2018).

Decentralized Data Management

One of the characteristics of monolithic applications is the existence of a centralized database responsible for saving all the information needed to implement the various functionalities of a given application. If the microservices architecture continued to use this centralized database, the services would not be independent (Indrasiri and Siriwardena 2018).

Therefore, each microservice must have its private database and database schema that stores the data required for its particular business capability. The service can only access its corresponding data and no others. For scenarios where more than one database needs to be updated, the service must use the appropriate service APIs (Indrasiri and Siriwardena 2018).

Because the databases are independent, they do not have to use the same management systems or technologies. Each database can use the system and technology best suited to its particular business capability (Indrasiri and Siriwardena 2018).

Service Governance

Service Governance in microservices is interpreted as the capability of each entity/team to govern its domain and its processes however it wants. However, it is not that simple.

Governance can be divided into two key aspects (Indrasiri and Siriwardena 2018):

- Design-time governance: consists of allowing the owner of the service to decide how to design, develop, and run the service. The owner can select and use the best tools for their needs and is not locked into a particular technology platform. However, the organization should define some common standards that must be mandatory, such as a review process and a list of technologies to be used to avoid incompatibilities.
- Runtime governance: consists of the features applied on a runtime that enable better management of services. It is often implemented as a centralized component and includes concepts such as service definition, service discovery, service versioning, service observability, and others.

2.1.2 Benefits and liabilities

Microservices architecture has its benefits and liabilities. The benefits can be associated with any distributed system but are better with microservices because of how the boundaries of the services are defined. The weaknesses can also be associated with any distributed system since its use increases complexity compared to a monolithic application (Newman 2021).

Benefits

The benefits of the microservices architecture are the following (Indrasiri and Siriwardena 2018; Newman 2021):

- Agile and Rapid Development of Business Functionalities: Each service is autonomous. As such, its development can also be independent, focusing only on the needs of the service and ignoring the functionality of the entire system.
- Replaceability: Each service focuses on a specific functionality and has a limited scope and size, making it easier to build and replace with a better implementation.
- Failure Isolation and Predictability/Robustness: As previously mentioned, the failure of one service does not cause the entire system to fail. Thus, the system continues to function while the developers replace the failed service. Observability also helps predict these failures and speeds up the resolution process.
- Agile Deployment and Scalability: Using containers makes it very easy to deploy and scale a service. Because they are autonomous services, developers can use container technologies to enable agile deployment and scaling processes.
- Align with Organizational Structure: Microservices focus on business capabilities, and organizations are usually structured by business capabilities as well. Therefore, any team responsible for a business capability can also be responsible for the associated microservice with a simple and well-defined scope.
- Technology Heterogeneity: Each service can use different technologies, each of which is better suited to the task at hand. In addition, new technologies can be adopted quickly, as trying something new does not have much impact since each service should not affect much of the system.
- Composability: Each service provides functionality that it can use in different ways. Therefore, developers can create independent code modules that encompass functionalities to cover most use cases for the services. This approach

aims for higher reusability.

Liabilities

The liabilities of the microservices architecture are the following (Indrasiri and Siriwardena 2018; Newman 2021):

- Inter-Service Communication: Implementing inter-service communication can be more challenging than developing concrete services because of the complexity of linking microservices to create composite business functionalities.
- Service Governance: Managing a large number of services is a challenge. Without a good governance strategy, the identification of services and the detection of failures becomes a nightmare.
- Heavily Depends on Deployment Methodologies: The success of microservices is heavily dependent on the use of containers and their orchestration systems. If these do not exist, developers need to invest time and energy in creating the infrastructure for container usage.
- Developer Experience: As the number of services increases, the developer's experience suffers because they can no longer run the application locally. This limitation often leads to development in the cloud, which can result in less regular and reliable feedback. The feedback problem exists because each service will output its logs and information, thus complicating the system's overall feedback readability. The increase can also limit the services available to developers to do their work, leading to a concentration of knowledge and the loss of the "collective ownership" model, where any developer can work on any part of the project.
- Technology Overload: New technologies have emerged to allow the adoption of the microservices architecture. This increase in technologies becomes problematic when companies want to introduce these new and often foreign technologies as requirements. These requirements lead to an overload of challenges mainly related to the existing knowledge about these new technologies and the time to understand them. In most cases, there is no need to explore these technologies as they only add complexity to the overall project and do not bring any real benefits.
- Cost: Microservices architecture increases costs because, in most cases, it requires more processes, more storage, and more supporting software. Organizational changes also have a higher impact, as more things need to be

learned and used effectively when developing new features, causing slowdowns or requiring more people to work on the project.

- Reporting: With the microservices architecture, reporting on all data became more difficult because all available services are owners of some part of the data, not existing a centralized location to obtain it.
- Security: Data flows between different services using the available endpoints. If these endpoints are not well secured, only making the data available to the authorized users, the data is more vulnerable to being observed or manipulated.
- Testing: Tests depend heavily on the scope they want to check. The larger the scope, the harder it is to set up data, the longer the test runs, and the harder it is to determine the cause of the test's failure. End-to-end tests are at the end of the scale in terms of scope because they cover most functionality. In the microservices architecture, these tests are even more extensive because they must run across multiple processes that must be well configured for the test scenarios. These tests also need to be prepared for false positives due to environmental issues. As a result, these tests do not provide the same level of assurance in the microservices architecture as they do in monolithic applications.
- Latency: In the microservices architecture, the data needed for a particular functionality can be split across multiple services. This partitioning results in the need to serialize, transfer, and deserialize the data, potentially impacting the latency of the system.
- Data Consistency: Previously, data was stored and managed in a single database. With the microservices architecture, the database is broken down into smaller databases, each for a specific service. This change makes data consistency harder to manage and maintain, as it is more troublesome to coordinate state changes between distributed databases.

2.2 Ballerina

Ballerina is an open-source, statically typed, cloud-native programming language created and supported by WSO2. You can quickly develop microservices, API endpoints and integrations, and any other type of cloud application using Ballerina. In addition, Ballerina contains every general-purpose feature you'd anticipate from a modern programming language, making it also be a general-purpose programming language (GPL) (WSO2 2022b).

Ballerina's compiler-level built-in support for frequently used data types like JSON and XML makes it much easier to concentrate on processing structured data, interacting with network services, and managing concurrency. It also offers specialized linguistic constructions for receiving and sending network services (WSO2 2022b).

Ballerina has concurrency and network interaction ideas and syntax that closely resemble sequence diagrams, which makes it possible for any Ballerina function to have a bidirectional mapping between its textual syntax representation and its graphical sequence diagram representation (WSO2 2022b).

2.2.1 History

WSO2 is a company that specializes in API integration (Oram 2019). It provided ESB solutions for SOA-based services integration for over a decade. WSO2 ESB's core mediation engine is an adaptation of Apache Synapse from WSO2. Synapse grants an XML-based Domain-Specific Language (DSL) to describe message processing and transformation logic. In SOA, communication between services uses SOAP requests. XML message processing mainly employs Xpath. However, it might be challenging to perform sophisticated messaging and transformation scenarios with a DSL; therefore, standard programming languages such as Java are adopted.

For this reason, and because of the need to improve the ESB, the CEO, and founder of WSO2, Dr. Sanjiva Weerawarna, decided to develop a programming language called Ballerina. This programming language was to focus primarily on providing a more efficient method of writing switching logic and deviating from the Synapse DSL. Dr. Sanjiva Weerawarna designed the programming language to "bridge the gap between integration and general programming languages." (Madushan 2021), which became very apparent when using ESB and other programming languages in the complete integration of business processes.

With these capabilities, Ballerina also proved to be a good option for microservices implementation, as it enabled the implementation of business logic in an agile manner and has strong integration support that allows the use of dumb pipes and smart endpoints, one of the patterns of microservices architecture (Madushan 2021).

2.2.2 Key features

This section gives a brief overview of some of the main features of the Ballerina programming language.

Visual-oriented Language

WSO2 created Ballerina to facilitate the creation of integration solutions. In the industry, these integration solutions are explained using sequence diagrams. Sequence diagrams are visual representations that make the solution to a problem understandable to both stakeholders and developers.

Therefore, Ballerina also provides the ability to use sequence diagrams to create solutions. The Ballerina editor offers a side-by-side visual and textual perspective. Developers can switch between the views as they work, and when they make changes in one of the views, the editor immediately converts the changes to the other without loss. As such, Ballerina includes code generation with its visual support, thus allowing the possibility to create the bal files only using the graphical representation.

This editor allows the developer to create integration logic with a textual or visual syntax. Additionally, Ballerina introduces a new extension for sequence diagrams that permits them to describe transport-specific properties such as REST APIs, OpenAPI standards, HTTP, and JMS access specifications (Weerawarana et al. 2018). Figure 2.2 shows an example of a visual representation of a process.

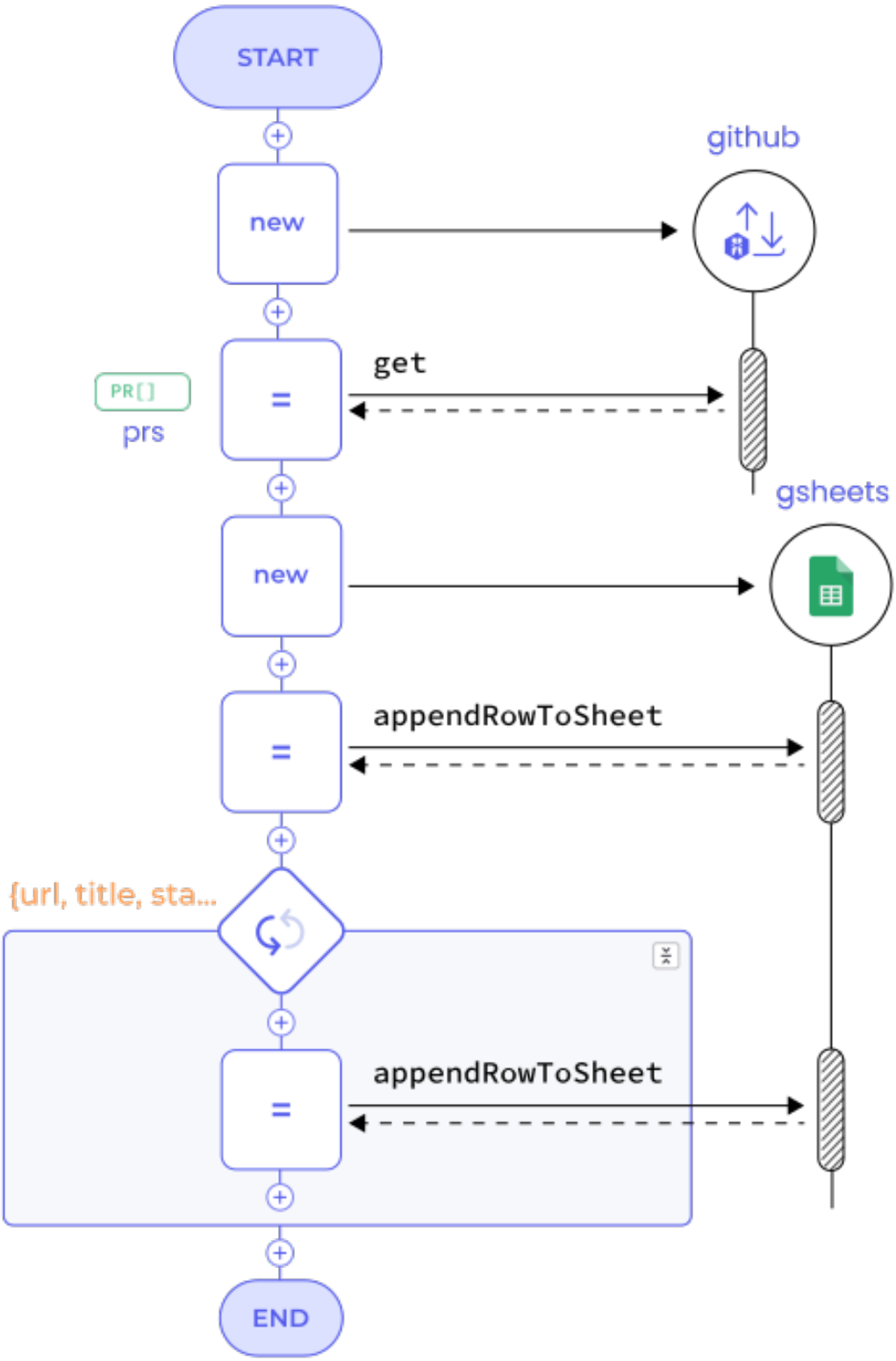

Figure 2.2: Ballerina Visual Representation Example (WSO2 2022c)

Type System

Ballerina provides a rich set of types that allow the natural implementation of integration scenarios. It includes simple types like Booleans and Integers, structured types like maps and arrays, and behavioral types like functions. Ballerina supports (Weerawarana et al. 2018):

- JSON and XML natively through the *json* and *xml* types. These types allow easy development of integration scenarios where developers need to apply transformations to the messages received in XML or JSON formats.
- Operations with SQL databases through the record and table types. The record type is a collection of fields. It also works as an SQL record. The table type is a collection of records. Each of them has an immutable key.
- Asynchronous operations through the *future* type. This type represents a construct responsible for storing the result of an action that is not finished and will return a value in the future.
- Data streaming through the *stream* type. A stream represents a sequence of values that is available over time and has a final value that indicates no more information to be obtained.
- Variable multi-typing through the use of union types. Union types are the junction of two base types represented by the type definitions with a pipe separating them. This type is helpful in situations where the variable can vary between the two defined types, like when a function can return a value of a string or can return an error.
- Value omission through the use of optional types. Optional types allow for variables to be of the desired type or Nil. An optional type uses a question mark after the required type definition. A union type of the desired type and the representation of Nil in Ballerina (*()*) can also represent an optional type.
- Type ignorance through the use of *any* type. The type *any* represents the union of the base types, except for *error*. To allow a variable to take any value, the developer can use the union of *any* and *error*. This union is helpful for cases where the type the developer wants to use is unknown at development time.

Thread Model

Ballerina presents itself as a parallel language that natively supports parallel executions. This is achieved through the use of workers. Workers are sets of Ballerina

instructions that have independent storage to store variables, can input arguments, and also output values. Workers can talk safely among themselves, can synchronize data, and support complex scenarios such as forks and joins. Workers can be run synchronously, running on the same thread of the function that invoked them, or can be run asynchronously, by having threads taken from the available thread pool assigned to them by the Ballerina Virtual Machine (BVM). When running asynchronously, the worker runs until it reaches a blocking instruction. At this point, BVM releases the designated thread and saves the worker's context in an appropriate callback function. When the blocking instruction returns a response and the callback is invoked, the callback function obtains the saved worker context and lets BVM go back to executing the worker by assigning it a new thread.

Besides parallelism, non-blocking I/O is supported by Ballerina with no additional programming effort required. I/O calls operate in a blocking way from the perspective of a programmer, meaning that the statement that follows the I/O call is only executed after the I/O call returns with a response. However, per the thread model mentioned above, whenever an I/O call is performed, BVM releases the underlying thread and saves the program state together with the next instruction pointer in a memory structure. The saved program state is continued by a new thread that BVM creates from its thread pool once the I/O call's outcome is available. Thus, to the developer, it seems that it is a blocking I/O call although threads are not blocked.

Testing framework

Ballerina provides a testing framework that allows for the creation of unit and integration tests. Ballerina provides a testing module that gives access to a variety of assertion functions that allow for the correct evaluation of the function's execution. This module also provides ways to configure data providers, mocking, and code coverage features to be used to test the Ballerina application. Tests can be done to functions or the services themselves, as well as to defined clients by using the mentioned mocking capabilities. Tests can also be run in groups, making that if changes are made in the application, only the affected part of the full application can be tested, being the developer the one who decides (WSO2 2022g).

Observability

Observability has three main pillars, each consisting of a method that can collect data that allows one to observe the system. These methods are logs, traces, and

metrics. Ballerina grants the implementation of all of these methods (Madushan 2021).

For logging, Ballerina has a built-in module that allows for the printing of logs to the standard output of the terminal. Four logging levels can be used (Madushan 2021):

- DEBUG logs: used to debug the code flow and log issues.
- INFO logs: used to log the normal flow of the code.
- WARN logs: used to log possible failures that can make the program fail. These normally are used to warn the user before the system fails.
- ERROR logs: Used to log errors. Usually, error logs indicate serious problems in the Ballerina application.

Ballerina allows filtering logs by level and defining the output format through configurations in the Config.toml file. The Config.toml file is a file that contains the values needed by the configuration variables. A variety of Ballerina-provided packages use these variables. Therefore these need to be available for direct configuration. Additionally, developers can create new configuration variables if necessary (Madushan 2021).

To collect the logs, developers can use tools like Logstash, Filebeat, Elasticsearch, and Kibana. These tools allow the collection of the Ballerina logs for further processing or persistence in a database, for example (Madushan 2021).

For tracing, Ballerina supports OpenTelemetry. "OpenTelemetry is an open source observability framework that we can use to observe cloud native systems with traces. Traces can be used to track how a request flows through the system over different services.". With this support, developers can utilize implementations of OpenTelemetry like Jaeger to trace Ballerina applications (Madushan 2021). The Config.toml file must enable the tracing using the option *"tracingEnabled=true"* to use tracing. Also, the file must specify the tracer using the option *"tracingProvider"*.Using Jaeger, the *"tracingProvider"* option equals *"jaeger"*. The file also includes the configuration for Jaeger, with an example present in Figure 2.3. After configuring, the Ballerina program must be run using the command bal run and the flag *"–observability-included"* or by including the build options section with the configuration of *"observabilityIncluded=true"* in the Ballerina.toml file, which is responsible for containing metadata for the Ballerina packages. Finally, developers can use a docker container running Jaeger to see the traces of the Ballerina application (WSO2 2022e).

```
[ballerinax.jaeger]
agentHostname="localhost"
agentPort=55680
samplerType="const"
samplerParam=1.0
reporterFlushInterval=2000
reporterBufferSize=1000
```

Figure 2.3: Jaeger Configuration Example (WSO2 2022e)

For measuring metrics, Ballerina exposes an HTTP endpoint (/metrics) which exposes internal metrics that developers can use with tools like Prometheus to collect and Grafana to visualize, allowing the measurement of performance and the load of the program. To use this endpoint, developers must add the option *"metricsEnabled=true"* in the Config.toml file. It also must be specified which metrics reporter they will employ using the option *"metricsReporter"*. Using Prometheus, the *"metricsReporter"* option equals *"prometheus"* and in the file is also included the configuration for Prometheus, which an example can be seen in Figure 2.4.

```
[ballerinax.prometheus]
port=9797
host="0.0.0.0"
```

Figure 2.4: Prometheus Configuration Example (WSO2 2022e)

Just like Jaeger, after configuring, the Ballerina program must be run using the command bal run and the flag *"–observability-included"* or by including the build options section with the configuration of *"observabilityIncluded=true"* in the Ballerina.toml file. To use Prometheus, developers must create a configuration file named "prometheus.yml" with the specification of the service to be measured. This file will be specified when running Prometheus, whether locally or using a Docker container. An example of this configuration is represented in Figure 2.5.

```
global:
  scrape_interval:     15s
  evaluation_interval: 15s

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['a.b.c.d:9797']
```

Figure 2.5: prometheus.yml Example (WSO2 2022e)

While Prometheus provides some dashboards, developers can enhance the data visualization using Grafana. To use Grafana, developers can install it locally or run it on a Docker container. With Grafana running, the user specifies the data source, in this case, Prometheus, from where Grafana will get the measures. With the data source configured, the developer can create dashboards to see the information the way he wants (Madushan 2021). WSO2 already provides some dashboards specific to Ballerina, which they published on the Grafana website for public use (WSO2 2022e). Examples of these dashboards can be seen in Figures 2.6, 2.7, and 2.8.

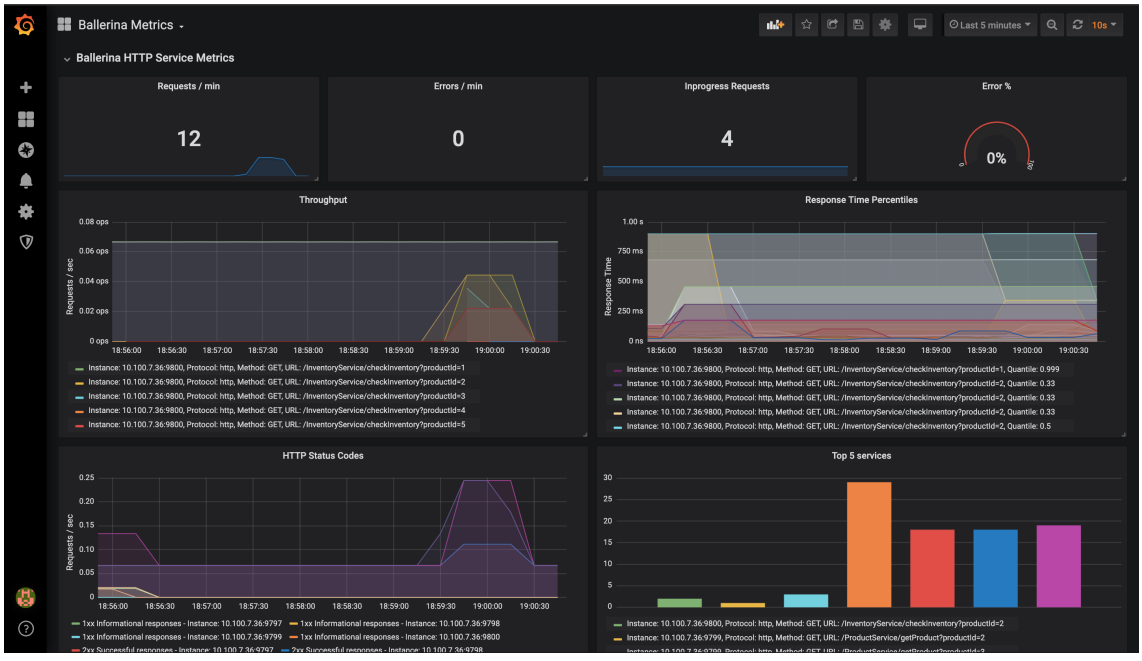

Figure 2.6: Grafana HTTP Service Metrics Dashboard (Labs 2022)

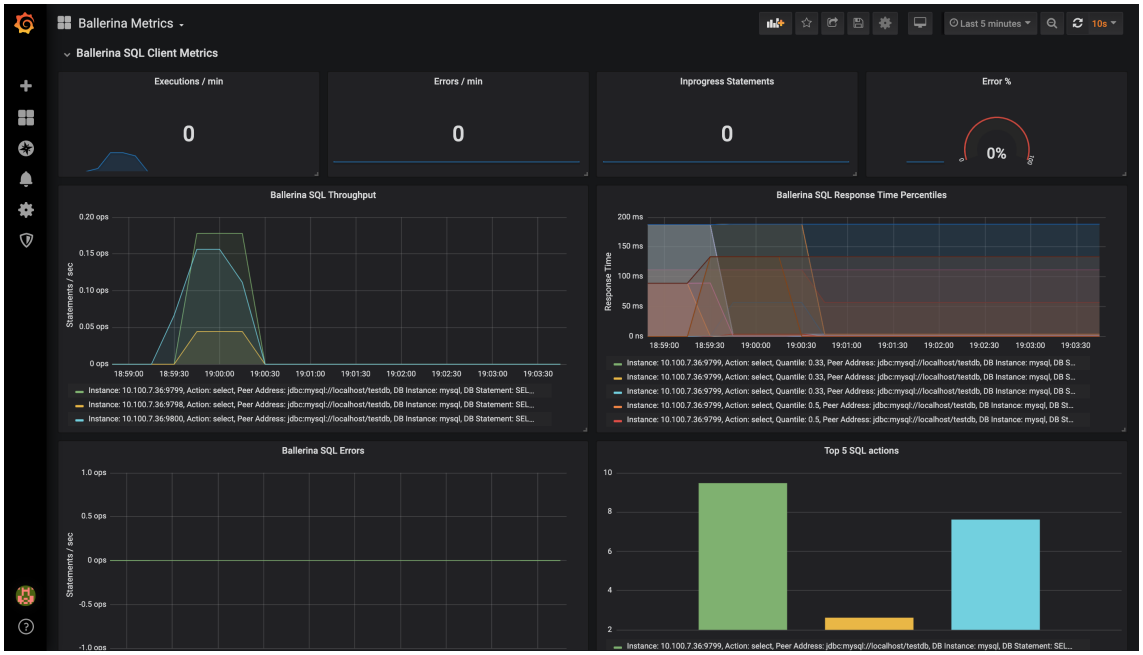

Figure 2.7: Grafana SQL Client Metrics Dashboard (Labs 2022)

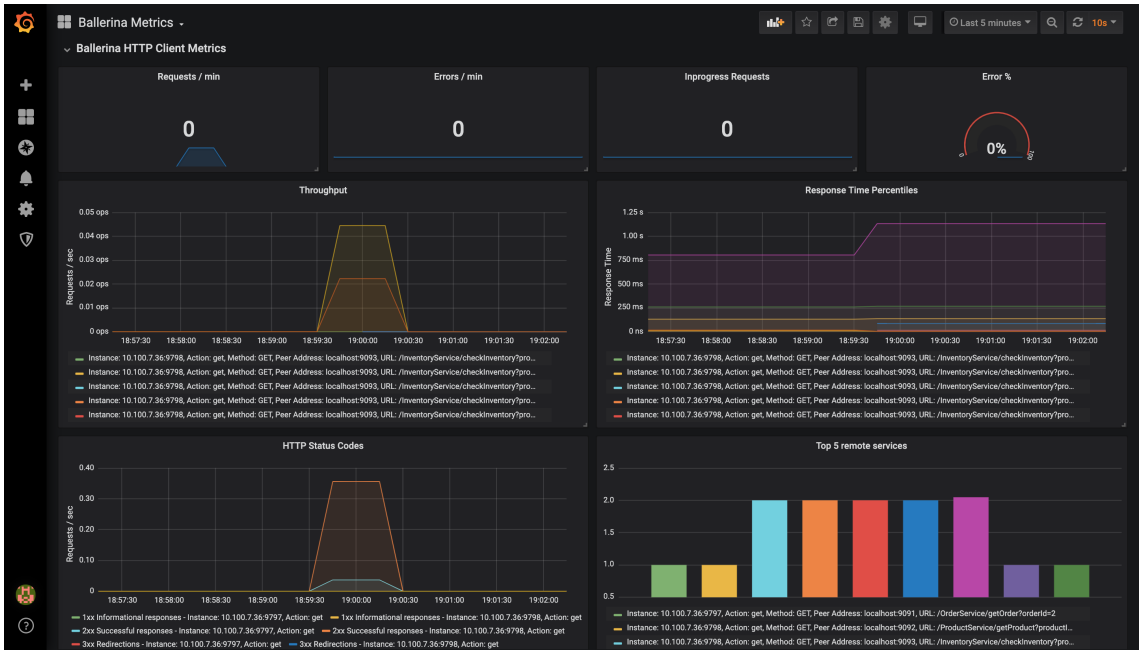

Figure 2.8: Grafana HTTP Client Metrics Dashboard (Labs 2022)

2.2.3 Industry Use Case

This section describes the example of the Modular Open Source Identity Platform (MOSIP). MOSIP is "a foundational identity platform that helps governments and

other organizations implement digital and foundational identity systems in a cost-effective way." (Ratnayake 2022). Countries are free to adopt MOSIP to construct their identity systems to meet common implementation challenges such as guaranteeing system uniqueness, interoperability, privacy by design, scalability, no vendor lock-in, and affordability. When a nation has a database of identities, that nation can use MOSIP for identity issuance, verification, and integration with numerous service providers that supply multiple goods and services to organizations and countries (Ratnayake 2022).

These identity systems have several operations, like registering new users, authenticating them for them to use other services, or updating the user's information, for example. These operations trigger changes in other services and applications. To communicate these changes, MOSIP decided that it needed to be able to use event-driven communication between different systems. To maintain the integration and deployment of its services, MOSIP also chose to use HTTP to transmit events instead of using a message broker like Kafka (Ratnayake 2022).

MOSIP used Ballerina to achieve their needs, integrating event-driven communication over HTTP. It adopted Ballerina mainly for two reasons: the language is open-source and allows customizations for specific requirements, like data persistence, security, and reliable data delivery. MOSIP achieved these customizations with the collaboration of the Ballerina team. MOSIP utilized the WebSubHub package of Ballerina. This package provides APIs that allow the usage of WebSubHub, which is an implementation of WebSub, an "open protocol for distributed pub/sub communication" (Ratnayake 2022) which allows for data to be published and subscribed in real-time and in a secure way. MOSIP currently uses this implementation for event-based communication between their internal systems, where an ID repository and an identity authentication service exist. MOSIP also utilizes this implementation for communication between printing partners, where the information to be printed in ID cards is transmitted (Ratnayake 2022).

Figure 2.9 illustrates the implementation of the MOSIP WebSubHub. The hub is the main component, responsible for talking with the subscribers and publishers, maintaining data persistence, resuming message delivery whenever a subscriber recovers from downtime, authenticating and authorizing hub operations, and scaling based on the number of subscribers. It also exists an event consolidator to remove duplicate events of the system and an implementation of a Kafka message broker responsible for enabling data persistence and managing subscriptions (Ratnayake 2022).

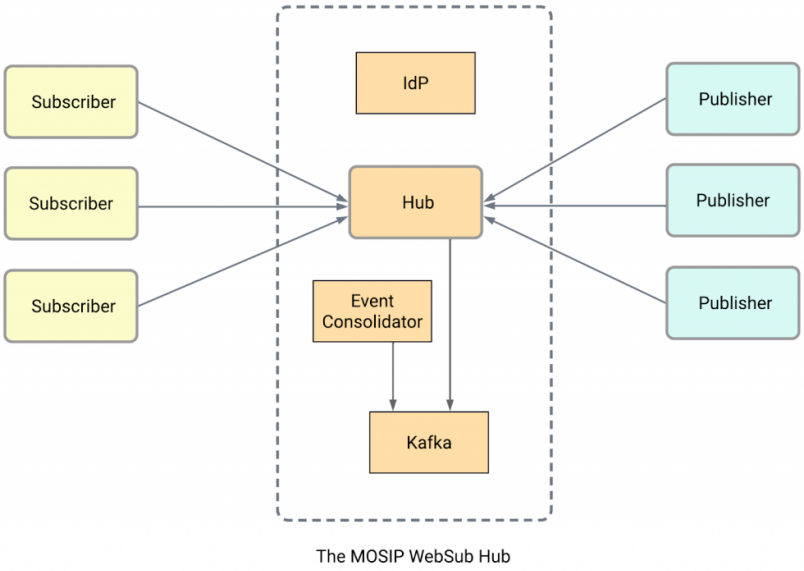

Figure 2.9: MOSIP's WebSubHub Implementation (Ratnayake 2022)

Chapter 3

State of the Art

The following chapter details the results obtained from the research done for this dissertation. This chapter divides itself into two sections. Firstly, the chapter focuses on the Ballerina features recognized as beneficial and that support the use of microservices architecture. Next, it presents the conducted research to identify the relevant quality attributes to the microservices architecture and this dissertation.

3.1 Ballerina Microservices

This section presents how implementing Ballerina can change the use of microservices. Also, it describes some of the features of Ballerina that bring advantages or help with some liabilities of the use of microservices.

3.1.1 Built-in Container Support

Ballerina supports the use of Docker containers. Ballerina has a built-in Docker image that contains the Ballerina runtime, which can be used to create containers by using the image and including the Ballerina executable. In addition, Ballerina can generate the required artifacts itself. Ballerina provides a code-to-cloud feature that allows the creation of a Docker image from Ballerina code. To use this feature, the developer defines the container image, tags, and repository in a configuration file called Cloud.toml. This file overrides the default values of the code-to-cloud feature. To create the artifacts, the developer builds the project using the bal build command with the additional configuration –cloud=docker to indicate to the Ballerina compiler that Docker is the selected cloud option. The result is an image stored in the developer's local Docker repository and a Dockerfile found in `<project_home>/target/docker/<package_name>/Dockerfile` (Madushan 2021).

Ballerina also supports the use of Kubernetes container orchestration. Like the creation of Docker artifacts, the code-to-cloud function also supports Kubernetes

artifact creation, found in the <project_home>/target/kubernetes/<package_name>/<package_name>.yaml file. When created, this file contains default Kubernetes instructions and values for deploying a Ballerina program. If the developer needs to override these values, he can define the new values in the Cloud.toml file mentioned earlier. To create the Kubernetes artifacts, the developer uses the same command as for Docker, with the same additional configuration, but with the value for cloud as k8s (Madushan 2021).

3.1.2 Network Awareness

Ballerina comes with an HTTP module. This module comes with all the usual operations like GET and POST, but it also allows the creation of services or clients. Services are collections of entry points that resolve tasks using resource functions declared on the Ballerina program. Clients are endpoints used to connect and interact with external HTTP servers (Fernando and Warusawithana 2020).

Ballerina also has access to connectors to services such as Salesforce or Twilio through modules available in Ballerina Central. Ballerina Central is a hub of available packages of reusable Ballerina code (Oram 2019; WSO2 2022a).

3.1.3 DevOps Support

Ballerina provides a testing framework, a build tool, and a packaging system to facilitate automated deployment. The testing framework allows the creation of test cases for Ballerina programs which the developer can automate (Madushan 2021).

The Ballerina command-line interface (CLI) allows developers to create deployment artifacts and run automated tests before each deployment. When used in conjunction with other Ballerina build tools, the CLI permits the creation of deployment artifacts and code checkout from GitHub as part of the automated deployment process (Madushan 2021).

For package versioning, Ballerina adheres to semantic versioning (SemVer). Semantic versioning is a frequently used versioning specification where the version number is generated in the format <major>.<minor>.<patch>, meaning, for example, that version 1.2.13 has a major version of one, a minor version of two, and a patch version of thirteen (Madushan 2021).

Ballerina manages dependencies using a tight versioning mechanism with strict dependency management policies to reduce conflicts between package versions. When

using a library, the developer must be particular about the version, unlike in other programming languages (Madushan 2021).

3.1.4 Security

Ballerina offers modules that allow for the straightforward use of familiar forms of authorization like basic web authorization, JSON Web Token (JWT), and OAuth2. In addition, Ballerina also offers an encryption module and a feature called taint checking (Oram 2019).

Taint checking consists of tracing data flow through each function and determining whether the data has malicious content. This feature is beneficial for common dangers like SQL injection, which consists of inserting malicious database commands in the data that goes to a database, or HTML injection, where the content uploaded to a public website contains harmful code (Oram 2019).

The data received from outside the program is marked as unsafe, meaning it is tainted. To use that data, it must be untainted, meaning it must go through a check. These checks can be diverse. For example, the check can match the data against a regular expression to check if it contains SQL commands or HTML tags that can be harmful. These checks must exist in sensitive functions, responsible for running the checks, issuing errors, and fixing or rejecting the data (Oram 2019).

3.1.5 Resiliency

Ballerina provides resiliency features to make the developed applications more robust. Ballerina offers circuit breaking, fail-over, load-balancing, and retry. Circuit breaking enables programmers to link connectors to suspension policies such that, if the suspension requirements are satisfied, connectors cease sending messages to non-responsive endpoints. For instance, if more than five percent (5%) of requests fail within thirty seconds, an HTTP connection can be set to halt new requests for five minutes. Similar to this, connections may be coupled with fail-over setups to choose backup endpoints in the event of an endpoint failure. Load balancing configuration provides a set of endpoints and a load-balancing algorithm so that requests are dispersed across the defined endpoints according to the given algorithm. This is useful to prevent the overloading of back-end services. Last but not least, retry configurations for connections can be defined to make the connector deliver the request again if it fails (Weerawarana et al. 2018).

3.2 Quality Attributes

As explained in the section 1.3, the main objective of this document is to analyze the effects of using the Ballerina programming language versus more traditional programming languages when creating microservices in terms of quality attributes. Therefore, this section describes the discussion and search realized regarding the quality attributes chosen to evaluate both solutions.

The quality of a product depends on the extent to which it meets the needs of stakeholders and creates value for them. These needs are usually represented through a quality model. The quality model determines product quality in terms of several quality attributes and sub-attributes that are considered when evaluating product quality (ISO/IEC JTC 1 2022b).

The ISO/IEC 25000, also known as System and Software Quality Requirements and Evaluation (SQuaRE), consists of a series of quality standards that is still in development and whose objective is to guide developers in the creation of software that attends to a variety of quality requirements and characteristics (ISO/IEC JTC 1 2022a). In this series, the ISO 25010 stands out as a standard created to specify a framework for software quality evaluation. This framework refers to a quality division model that consists of the quality attributes that evaluators need to consider in their software evaluation processes. This model is divided into eight quality attributes (Figure 3.1), each with sub-attributes (ISO/IEC JTC 1 2022b).

Figure 3.1: ISO 25010 (ISO/IEC JTC 1 2022b)

This dissertation analyzes some studies regarding microservices to identify the quality attributes that have the most interest when comparing solutions with traditional programming languages and the Ballerina programming language. In these studies, one of the most noticed affirmations is that the microservices architecture improves many quality attributes. However, these studies also acknowledge that

the microservices architecture still neglects several relevant quality attributes or that insufficient research has been conducted on them.

According to a study carried out by Li et al., there are a variety of quality attributes, like maintainability, reusability, or scalability, that developers need to prioritize when working with the microservices architecture. As such, the authors researched to identify the most mentioned quality attributes in works related to microservices architecture. This study showed that exist six quality attributes that stand above the rest. These quality attributes are scalability, performance, availability, monitorability, security, and testability. However, this study also highlights the need for more empirical research on other quality attributes that are also important, such as maintainability (Li et al. 2021).

In another article, Bushong et al. conducted a study to identify approaches and techniques that allow a better analysis of systems that use microservices since designing these services continues to be challenging. This study analyzed the evolution of microservices systems. From this analysis, the authors identified that the approaches target various challenges. One of the targeted challenges is the existence of some hard-to-analyze quality attributes. The study pinpointed that engineers have difficulties analyzing quality attributes like maintainability, security, and performance in microservices systems (Bushong et al. 2021).

Two studies on patterns related to microservices architecture identified a group of quality attributes influenced by the use of these patterns in the design or development of microservices. The first study found that the quality attributes most mentioned in works were maintainability, reliability, security, performance, compatibility, and portability. Maintainability was the most referenced quality attribute. The security attribute surprised the authors because it is not directly related to microservices architecture (Jose A. Valdivia, Limon, and Cortes-Verdin 2019). The second study, done one year after by the same authors with other researchers, and using a multivocal literature review, enforces more of the affirmations of the first study, identifying more works that give the same conclusions (J. A. Valdivia et al. 2020).

According to an industry study on the challenges of microservices, maintainability is a crucial quality trait and one of the primary reasons for the adoption of microservices. Also, issues related to maintainability are related to most of the concerns of developers from various companies. The developers are concerned about code management, particularly when making changes that may break the entire system. Furthermore, developers spend most of their time thinking about ways to prevent and deal with these breaking changes, stating that they only make them when necessary

(Wang, Kadiyala, and Rubin 2021).

Following the analysis, the quality attributes that bring more relevance and interest to study are maintainability and performance efficiency. Therefore, the dissertation will concentrate on these two quality attributes.

3.2.1 Maintainability

Maintainability reflects the degree of efficacy and efficiency with which developers can alter a system or product to upgrade, fix, or adapt it to the changes in the environment and user needs. It encompasses sub-attributes like (ISO/IEC JTC 1 2022b):

- Modularity - The degree to which a system or computer program is composed of distinct parts, such that changing one part barely affects the rest.
- Reusability - The extent to which a resource can be reused in several systems or to create additional resources.
- Analysability - The degree of efficacy and efficiency with which it is feasible to evaluate the effects of a planned change to one or more of a product's components, to diagnose a product for flaws or failure causes, or to identify pieces that need to be modified.
- Modifiability - The extent to which a system or product can be effectively and efficiently updated without introducing flaws or lowering the quality of the final product.
- Testability - The efficiency and effectiveness with which test criteria for a system, product, or component can be developed and used to verify that those criteria have been satisfied.

3.2.2 Performance Efficiency

Performance efficiency represents the level of performance of a system concerning the resources it uses in a specific set of conditions. This quality attribute includes sub-attributes like (ISO/IEC JTC 1 2022b):

- Time behaviour - The degree to which a product or system satisfies specifications when performing its functions regarding response, processing, and throughput rates.
- Resource utilization - The extent to which the types and quantities of resources employed by a system or product to carry out its functions comply with specifications.

- Capacity - The extent to which a product or system parameter's upper bounds comply with specifications.

Chapter 4

Value Analysis

This chapter focuses on Value Analysis. Value analysis is a systematic assessment method performed on product designs to compare how well they serve the functions that customers demand at the lowest possible cost while maintaining the required performance and dependability (Rich and Holweg 2000).

This chapter opens with a section on business processes and innovation. Next, the chapter focuses on applying the New Concept Development (NCD) model with its phases. Firstly, it identifies and analyzes the opportunity. Then, it generates some ideas. Finally, one of the ideas will be selected using the Analytic Hierarchy Process (AHP).

4.1 Innovation Process

When describing the innovation process, Koen et al. refers that the process is divided into three main parts: the Fuzzy Front End (FFE), the New Product Development (NPD), and the Commercialization. Figure 4.1 shows a diagram with this process (Koen et al. 2001).

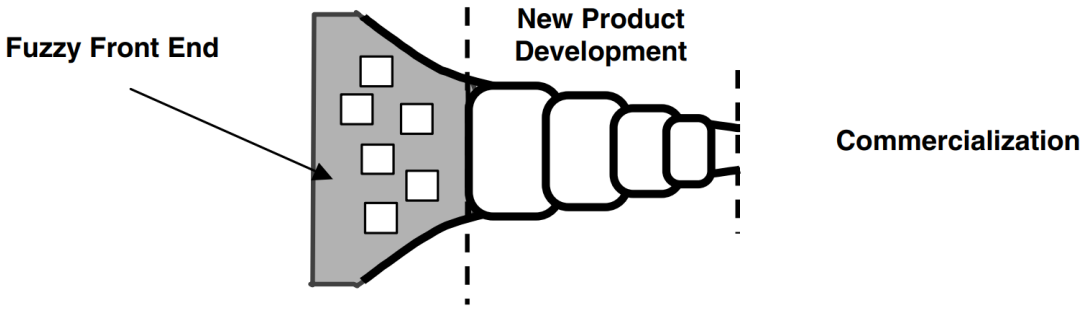

Figure 4.1: Innovation Process diagram (Koen et al. 2001)

The *Fuzzy Front End* represents the phase where activities that are less predictable and less structured but more innovative, profitable, risky, and chaotic are executed.

This phase is seen as the experimental part of the process since it is highly ambiguous and unpredictable. However, it is also seen as one of the best opportunities for improvements in the whole innovation process because the choices made in this phase have a high impact on the subsequent ones, meaning that increasing the attention given to this phase increases the overall value and quality of the generated outputs (Koen et al. 2001).

The *New Product Development* is the next phase of the innovation process. This phase consists of a group of formal and structured activities with specific goals, well-defined dates, and goal-oriented plans (Koen et al. 2001).

The *Commercialization* is the last phase of the innovation process. This phase symbolizes the innovation process reaching its end, with the culmination of the previous steps. This phase aims to take the generated outputs that form a final product and commercialize it (Koen et al. 2001).

With the *Fuzzy Front End* being one of the best phases to improve, Koen et al. wanted to compare the FFE practices between companies. However, since there were no uniform definitions or terms, Koen et al. could not make the comparison. Without a common language and terminology, it is impossible to create new knowledge and distinguish the different stages of the process. Even when using the same terms, if both parties mean opposite things, knowledge transfer becomes more challenging and ineffective (Koen et al. 2001).

Because of this shortcoming, Koen et al. developed a new model, the New Concept Development (NCD), intending to create a common terminology that could be universally understood to achieve an overall better understanding of the FFE (Koen et al. 2001).

4.1.1 New Concept Development

The NCD model shown in Figure 4.2 consists of three key parts (Koen et al. 2001):

- *Engine*: This part represents the organization's leadership, culture, and business strategy that drives the five essential factors that are controllable by the corporation.
- *Inner spoke*: This part represents the FFE's five controlled activity elements, which consist of opportunity discovery, opportunity analysis, idea generation and enrichment, idea selection, and concept definition.
- *Influencing Factors*: This part represents the organizational capabilities, the factors of the outside world like distribution channels, law, government policy,

customers, competitors, the political and economic climate, and the internal and external enabling sciences that may be involved in the process. These factors are mostly uncontrollable by the corporation and influence the entire innovation process.

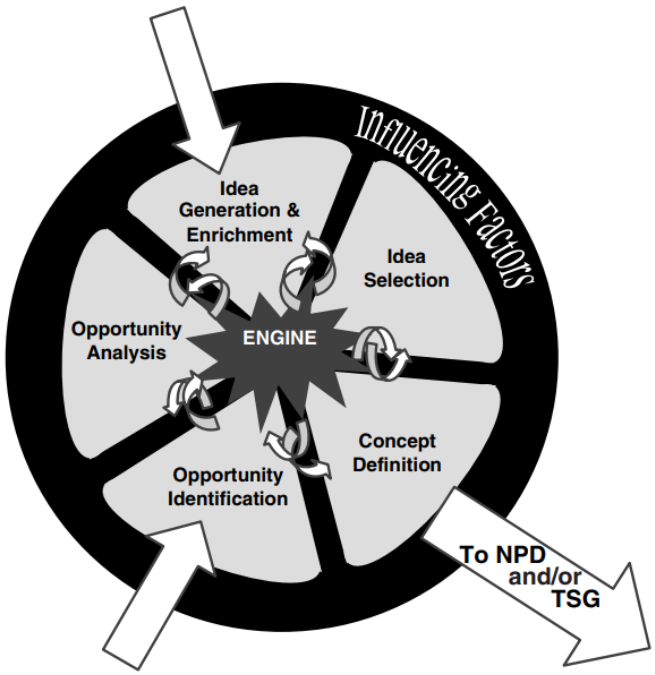

Figure 4.2: *NCD* model (Koen et al. 2001)

The NCD model has a circular shape, which hints at the idea that ideas flow between all five elements. The two arrows pointing to the *Opportunity Identification* and *Idea Generation & Enrichment* indicate that these are the most common starting points for projects and ideas (Koen et al. 2001).

4.1.2 Opportunity Identification

Opportunity Identification is the element where an organization identifies which opportunities are most relevant to them (Koen et al. 2001).

As stated in section 1.2, the microservices architecture came to revolutionize the industry of software development. However, it also came with its challenges. While developers were able to solve these challenges with the most common programming languages, the next evolution seems to be toward the utilization of programming languages whose purpose is to facilitate the solving of the challenges presented by microservices, like the Ballerina programming language. The creators of Ballerina

describe it as a language of easy use and understanding for developers and stakeholders (WSO2 2022b). However, after studying the market and verifying some of the current solutions that adhered to Ballerina, those involved in this dissertation have deemed the necessity to investigate the advantages Ballerina offers. Ballerina is also a language that has been increasing in popularity (this popularity will be depicted in section 4.1.3).

As such, this research focuses on the need to better understand Ballerina and its advantages by comparing how applications that use Ballerina fare against applications that use common programming languages.

4.1.3 Opportunity Analysis

Opportunity Analysis is the element where an opportunity is analyzed. The opportunity identified in the Opportunity Identification element is checked to see if its development is worthwhile pursuing (Koen et al. 2001).

Over the years, the ranking of programming languages has been changing. Since 2011, RedMonk has been constructing a ranking of programming languages, comparing their popularity in Github and StackOverflow. RedMonk is an analyst firm created to help developers and organizations understand themselves and help them with technical decisions, crunching data to make recommendations. Software giants like Google, LinkedIn, and Twitter have used their analyzes (RedMonk 2022a).

In the rankings of RedMonk, languages like Java, Python, and C# have established themselves as some of the most popular for developers and organizations to use in their projects. In more recent years, while the top ranks do not change much, the lower ones are constantly updating. In 2021, Ballerina entered the rankings, marking its presence as one of the languages that were rising in popularity, as can be seen in Figure 4.3 (RedMonk 2021). Since entering the rankings, its popularity has been growing, as can be seen in Figures 4.4 and 4.5, with the last one corresponding to their current ranking analysis (RedMonk 2022b,c).

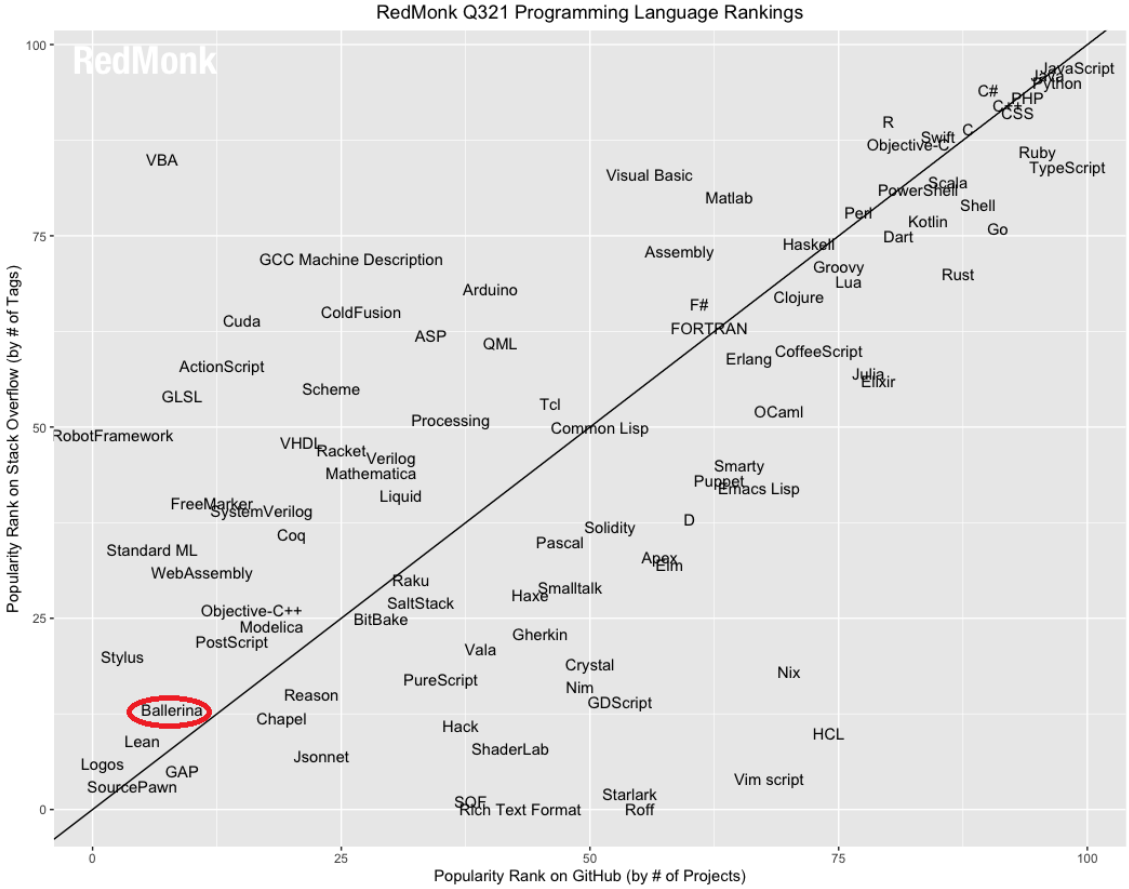

Figure 4.3: Ranking June 2021 (RedMonk 2021)

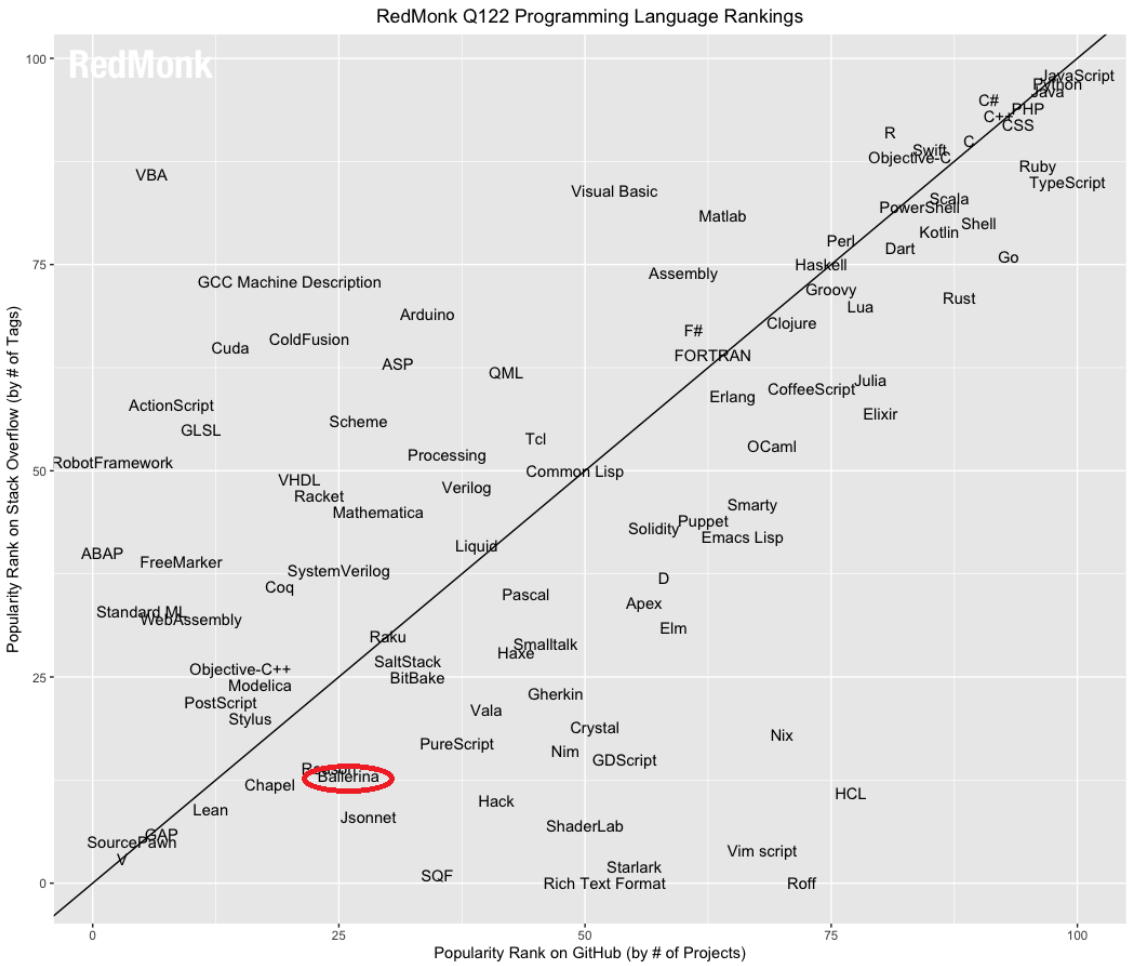

Figure 4.4: Ranking January 2022 (RedMonk 2022b)

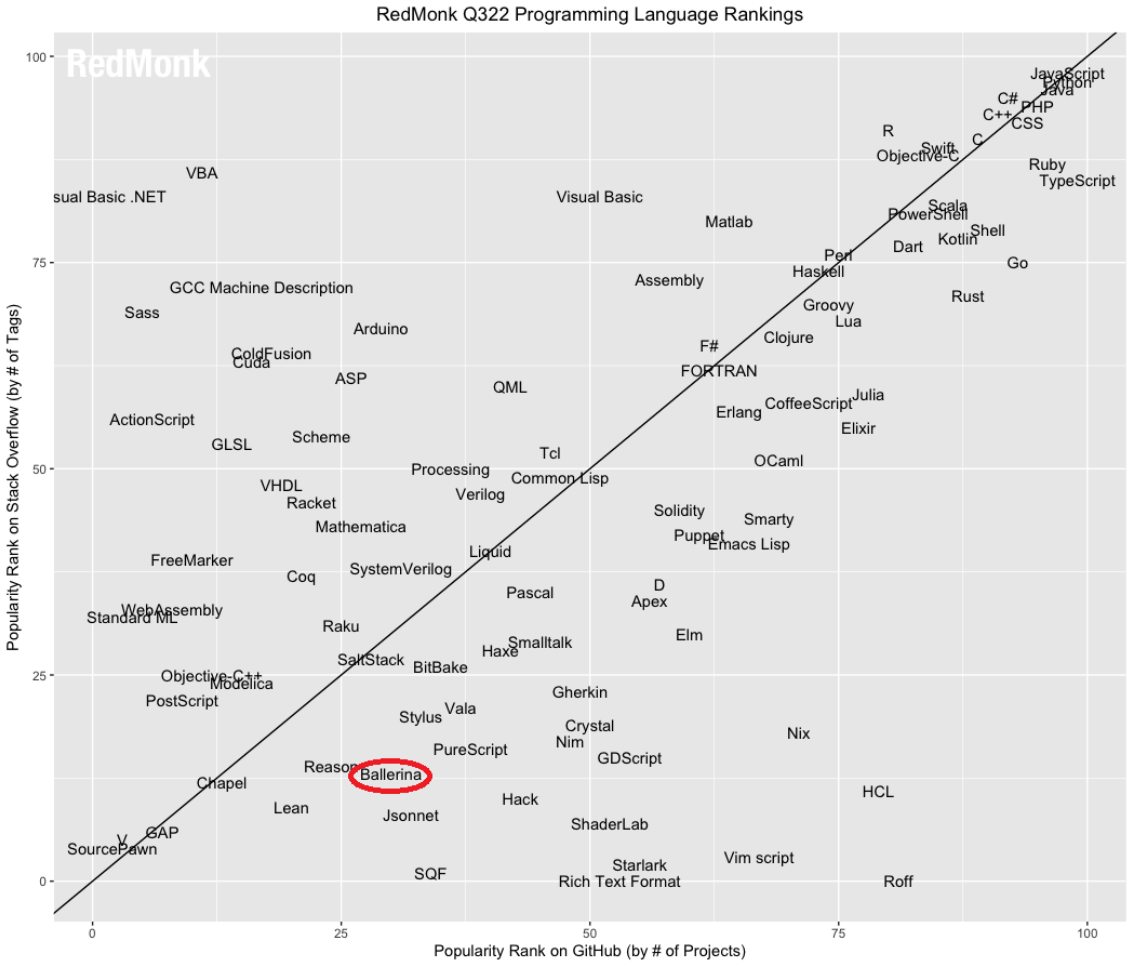

Figure 4.5: Ranking June 2022 (RedMonk 2022c)

4.1.4 Idea Generation & Enrichment

The element responsible for the emergence, development, and maturation of a concrete idea is Idea Generation & Enrichment. This creative process of generating ideas is based on building, tearing down, combining, reshaping, modifying, and upgrading ideas. An idea can go through many cycles, being reviewed, studied, discussed, and developed concerning other aspects of the *NCD* model. Keeping contact with customers and collaborating with other companies and organizations frequently enhances idea generation (Koen et al. 2001).

Section 1.4 defined the research questions this dissertation intends to answer. As such, to respond to the questions, it is necessary to compare two implementations of an application, one being in Ballerina, with the same domain between them. Thus, based on this idea, the following alternatives were created:

- Alternative 1: Development of an application using microservices architecture

in a common programming language and migration of the application to Ballerina.

- Alternative 2: Migration of some of the microservices of an existing microservices architecture-based application developed using a common programming language to Ballerina.
- Alternative 3: Total migration of an existing microservices architecture-based application developed using a common programming language to Ballerina.

4.1.5 Idea Selection

Idea Selection is the element where an idea created from the Idea Generation & Enrichment is selected to be implemented. Since businesses must choose the idea that will maximize their earnings, this is one of the more difficult decisions they must make. Picking the right idea is crucial to an enterprise's health and success. However, it does not exist a single process that guarantees a good selection. Most idea selection processes need many runs on the previously mentioned elements, with new insights from influencing factors and further instructions from the engine (Koen et al. 2001).

Analytic Hierarchy Process (AHP)

The *Analytic Hierarchy Process (AHP)* was used to choose one of the options mentioned in the preceding section. The AHP is one of the main methods developed regarding Multi-criteria decision analysis and was created by professor Thomas L. Saaty in 1980. This method allows the use of qualitative or quantitative criteria in the evaluation process and works by dividing the problem into hierarchical levels to facilitate their evaluations (Nicola 2022).

To start using the AHP, firstly, it is necessary to create a Hierarchical Decision Tree, which presents the objective, the decision criteria, and the alternatives (Nicola 2022). Regarding alternatives, the ones to be used are the ones mentioned in the section 4.1.4. In terms of criteria, the following were chosen:

- Time: The time criteria will determine which alternative is the best to implement, keeping in mind the time constraints of this dissertation.
- Adequacy: The adequacy criteria will determine which alternative is more appropriate to answer the research questions previously defined.
- Simplicity: The simplicity criteria will determine which alternative is simpler to implement, given the possible number of implementations to develop.

Figure 4.6 contains the representation of the Hierarchical Decision Tree.

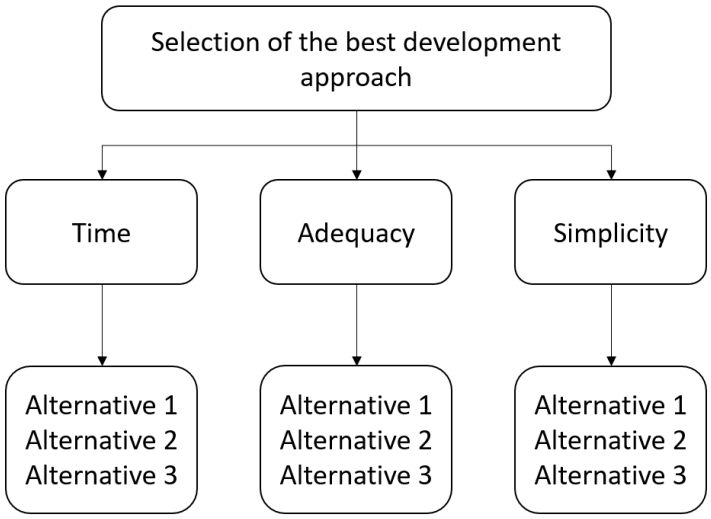

Figure 4.6: Hierarchical Decision Tree

After creating the decision tree, the next phase of the AHP is to create a comparison matrix to assess the importance of the criteria. Each criterion is going to be assigned a priority level, following the Saaty fundamental scale, represented in Table 4.1 (Nicola 2022).

Table 4.1: Saaty fundamental scale (Saaty 1990)

Intensity of importance	Definition	Explanation
1	Equal importance	Two activities contribute equally to the objective
3	Weak importance of one over another	Experience and judgment slightly favour one activity over another
5	Essential or strong importance	Experience and judgment strongly favour one activity over another
7	Demonstrated importance	An activity is strong favored and its dominance is demonstrated in practice
9	Absolute importance	The evidence favoring one activity over another is of the highest possible order of affirmation
2, 4, 6, 8	Intermediate values between the two adjacent judgments	When compromise is need

Based on the previously defined criteria, the following comparison matrix (Table 4.2) was created.

Table 4.2: Comparison Matrix between Criteria

	Time	**Adequacy**	**Simplicity**
Time	1.000	0.250	0.500
Adequacy	4.000	1.000	4.000
Simplicity	2.000	0.250	1.000
Sum	7.000	1.500	5.500

Following the criteria comparison matrix, the third phase of the AHP consists of normalizing all the values of the previous matrix and calculating the priority vector. Each value is divided by the total of the column in which it appears to normalize the values. Each line of normalized values' arithmetic average is utilized to calculate the priority vector (Table 4.3) (Nicola 2022).

Table 4.3: Normalized Comparison Matrix and Relative Priority Vector

	Time	**Adequacy**	**Simplicity**	**Priority Vector**
Time	0.143	0.167	0.091	0.133
Adequacy	0.571	0.667	0.727	0.655
Simplicity	0.286	0.167	0.182	0.211

With the normalized comparison matrix created, the next phase is applied. In this phase, it is calculated the Consistency Ratio (CR), which ensures that the consistency of the obtained priority vector is satisfactory. To achieve this, the value of the CR must be less than 0.1 (Nicola 2022).

The Consistency Ratio (CR) is calculated last in the next phase to assess the consistency of the judgments. If the value of CR is less than 0.1, the judgments are regarded as reliable. To calculate the CR, the Consistency Index (CI) is divided by the Random Index (RI) (Nicola 2022). The formula to calculate the CR is the following :

$$CR = \frac{CI}{RI} \tag{4.1}$$

Table 4.4 provides the value of the RI. The value to select depends on the number of criteria utilized in the analysis. Since three criteria are used, the chosen RI value is 0.58.

Table 4.4: Random Consistency Index (Adapted from Nicola 2022)

Nº of criteria	1	2	**3**	4	5	6	7	8	9	10	11	12	13	14	15
Index value	0.00	0.00	**0.58**	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

To obtain the value of the CI, the following formula is used:

$$CI = \frac{\lambda_{max} - n}{n - 1} \tag{4.2}$$

To obtain the λ_{max} it is used the formula:

$$Ax = \lambda_{max} x \tag{4.3}$$

The A corresponds to the normalized comparison matrix and x corresponds to the priority vector (which values are calculated in Table 4.3). Multiplying A with x we obtain the values in Table 4.5.

Table 4.5: Consistency Matrix Comparison

Time	0.402
Adequacy	2.035
Simplicity	0.642

With the values from Table 4.5, the following formula can be used to infer the value of the λ_{max}:

$$\lambda_{max} = \frac{0.402/0.133 + 2.035/0.655 + 0.642/0.211}{3} \approx 3.054 \tag{4.4}$$

Substituting all the values in the formula of the CI, we obtain the following value:

$$CI = \frac{3.054 - 3}{3 - 1} \approx 0.027 \tag{4.5}$$

With both the CI and the RI, the value of the CR can be calculated. Replacing the values in the formula, it is obtained an approximate value of 0.047, which is less than 0.1, which means it is possible to infer that the priority values are regarded as reliable.

$$CR = \frac{CI}{RI} = \frac{0,027}{0,58} \approx 0.047 \tag{4.6}$$

The next step is to create comparison matrices for each criterion after confirming

the consistency. These matrices will measure the significance of the choices concerning each criterion. The procedure consists of repeating the construction of the comparison matrices between each criterion and all alternatives, repeating the development of the normalizing matrix, and repeating the computation of the priority vector to create these matrices (Nicola 2022).

Starting with the comparison matrices between criteria and alternatives, the Tables 4.6, 4.7 and 4.8 show the comparison using the fundamental scale (Table 4.1). To create the matrices, the headers will be shortened to the values Alt 1, Alt 2, and Alt 3.

Table 4.6: Comparison Matrix between Time in Alternatives

	Time		
	Alt 1	**Alt 2**	**Alt 3**
Alt 1	1.000	0.200	0.333
Alt 2	5.000	1.000	3.000
Alt 3	3.000	0.333	1.000
Sum	9.000	1.533	4.333

Table 4.7: Comparison Matrix between Adequacy in Alternatives

	Adequacy		
	Alt 1	**Alt 2**	**Alt 3**
Alt 1	1.000	2.000	1.000
Alt 2	0.500	1.000	0.333
Alt 3	1.000	3.000	1.000
Sum	2.500	6.000	2.333

Table 4.8: Comparison Matrix between Simplicity in Alternatives

	Simplicity		
	Alt 1	**Alt 2**	**Alt 3**
Alt 1	1.000	0.143	0.250
Alt 2	7.000	1.000	3.000
Alt 3	4.000	0.333	1.000
Sum	12.000	1.476	4.250

Next, the normalizing matrices are calculated along with the local priority vectors for each criterion. The final results are displayed in Tables 4.9, 4.10, and 4.11.

Table 4.9: Normalized Comparison Matrix and Local Priority for the comparison between Alternatives regarding Time Criteria

	Time			
	Alt 1	**Alt 2**	**Alt 3**	Local Priority
Alt 1	0.111	0.130	0.077	0.106
Alt 2	0.555	0.652	0.692	0.633
Alt 3	0.333	0.217	0.231	0.260

Table 4.10: Normalized Comparison Matrix and Local Priority for the comparison between Alternatives regarding Adequacy Criteria

	Adequacy			
	Alt 1	**Alt 2**	**Alt 3**	Local Priority
Alt 1	0.400	0.333	0.429	0.387
Alt 2	0.200	0.167	0.143	0.170
Alt 3	0.400	0.500	0.429	0.443

Table 4.11: Normalized Comparison Matrix and Local Priority for the comparison between Alternatives regarding Simplicity Criteria

	Simplicity			
	Alt 1	**Alt 2**	**Alt 3**	Local Priority
Alt 1	0.083	0.097	0.059	0.080
Alt 2	0.583	0.677	0.706	0.656
Alt 3	0.333	0.226	0.235	0.265

With all the matrices created and with the obtained local priority vectors, the final phase is reached. In this phase, it is calculated the composite priority for the alternatives, through the merging of the local priority vectors (Tables 4.6, 4.7 and 4.8), creating a matrix that is multiplied with the priority vector of the criteria (Table 4.3) (Nicola 2022). The results of this multiplication are found in Table 4.12.

Table 4.12: Criteria/Alternatives Classification Matrix and Composite Priority

	Time	**Adequacy**	**Simplicity**	**Composite Priority**
Alt 1	0.106	0.387	0.080	0.285
Alt 2	0.633	0.170	0.656	0.334
Alt 3	0.260	0.443	0.265	0.381

Table 4.12 verifies that Alternative 3: the total migration of an existing microservices architecture-based application developed using a common programming language to Ballerina is considered the best alternative to implement with a value of approximately 0.381. However, it is noteworthy that the result of Alternative 2, with a rough value of 0.334, is very near the highest value, with an approximate difference of only 0.047. The reason behind this difference resides in the fact that the criterion with the highest weight is Adequacy (with an approximate value of 0.655), and for this criterion, Alternative 3 outweighs the other two alternatives (with a value of 0.443).

Analysis Retification

Due to the original projects complexity and time related issues, the result obtained from the application of the AHP method, which resulted in the total migration of an existing microservices architecture-based application developed using a common programming language to Ballerina, is not viable. Time was not the most relevant criteria during the application of the AHP, which resulted in the selection of an alternative which was not viable for the time available for the project. For the scope of this project, the alternative that was effectively implemented was Alternative 2, the Migration of some of the microservices of an existing microservices architecture-based application developed using a common programming language to Ballerina, which was possible for the time-frame available for the project.

Chapter 5

Analysis and Design

This chapter details the analysis and design of the developed implementation. As mentioned in section 1.3, the solution started from an open-source microservices architecture-based project that uses a conventional programming language. After, this project was migrated to the Ballerina programming language to help discover the effects of using Ballerina regarding maintainability and performance. Therefore, the chapter starts by explaining the design of the initial project, mentioning the business context and the defined architecture. Then, it describes the process that will be used for the migration into Ballerina.

5.1 Project to Migrate

As mentioned in section 1.3, a project will be migrated to analyze the effects of adopting the Ballerina programming language. When selecting the project, a few criteria were taken into account. As such, the project must have:

- The code available to everyone (Open-source).
- Recent activity, with commits less than a year ago.
- A significant number of microservices, to facilitate the investigation regarding Ballerina service integration.
- Documentation that should include its architecture and how it is used.
- Tests that prove its credibility.
- Characteristics like API Gateway and/or service discovery implemented.

The selected project was Lakeside Mutual, a made-up insurance company project on GitHub. This project is a sample application to show off API patterns and domain-driven design using microservices (Stocker 2021). The author selected this project because of its good documentation, dimension, and containing an implementation of service discovery.

The characteristics of the project are the following:

- Created on February 26th of 2021.
- Has commits dated to the last three months.
- Eight microservices developed in Java (Spring Boot) and Node.
- Three front-end applications using React and Vue.js.
- Tests for each of the microservices.
- Documentation of the architecture and patterns used.
- Documentation of the APIs (Springdoc Open API)

The project is also under the second version of the Eclipse Public License. Therefore, all code and documentation created regarding this project must be available as open-source code under the same license. The license is available on the project's repository (Stocker 2021).

5.1.1 Business Context

Lakeside Mutual has tasked one of its IT teams to extend a customer application with a self-service capability. The team identified the need for customer and policy data on demand in their first analysis. This data exists across several back-end systems that do not have suitable Web APIs or message channels to provide such data (Zimmermann et al. 2022).

The application's newest version will have to support various self-service features, such as allowing the customers of Lakeside Mutual to update their contact information without needing to contact an agent of the company (Zimmermann et al. 2022).

The team also gathered requirements in terms of quality (performance, availability, maintainability). For example, the contact update should take no more than two seconds in eighty percent of the cases. The application must handle around ten thousand users, which is what the company expects in terms of customers who will use these new functionalities. Also, about ten percent of these users will use the application concurrently (Zimmermann et al. 2022).

The company also has usability concerns. Lakeside Mutual might need to use more expensive channels, contradicting the development of these new functionalities, if the new application fails to meet the customers' self-service requirements effectively. Also, the application must be reliable and capable of being used during extended periods (working hours) and through all days, making the application available all the time for the users to take care of their insurance contracts (Zimmermann et al. 2022).

5.1.2 Architecture

To provide the new features to their customers, the IT team of Lakeside Mutual created the following architecture, represented in Figure 5.1.

Table 5.1 contains a brief description of each of the components that constitute the architecture of the solution, which purpose is to provide the company's customers and employees the services they need.

Table 5.1: Service Descriptions (Stocker 2021)

Service Name	Description
Customer Core	The Customer Core backend is a Spring Boot application that manages the personal data about individual customers. It provides this data to the other backend services through an HTTP resource API.
Customer Self-Service Backend	The Customer Self-Service backend is a Spring Boot application that provides an HTTP resource API for the Customer Self-Service frontend.
Customer Self-Service Frontend	The Customer Self-Service frontend is a React application that allows users to register themselves, view their current insurance policy and change their address.
Customer Management Backend	The Customer Management backend is a Spring Boot application that provides an HTTP resource API for the Customer Management frontend and the Customer Self-Service frontend. In addition, WebSockets are used to implement the chat feature to deliver chat messages in realtime between the call-center agent using the Customer Management frontend and the Customer logged into the Self-Service frontend.

Service Name	Description
Customer Management Frontend	The Customer Management frontend is a React application that allows Customer-Service operators to interact with customers and help them resolve issues related to Lakeside Mutual's insurance products.
Policy Management Backend	The Policy Management backend is a Spring Boot application that provides an HTTP resource API for the Customer Self-Service frontend and the Policy Management frontend. It also sends a message (via ActiveMQ messaging) to the Risk Management Server whenever an insurance policy is created / updated.
Policy Management Frontend	The Policy Management frontend is a Vue.js application that allows Lakeside Mutual employees to view and manage the insurance policies of individual customers.
Risk Management Server	The Risk-Management server is a Node.js application that gathers data about customers / policies and can generate a customer data report on demand.
Risk Management Client	The Risk-Management client is a command-line tool built with Node.js. It allows the professionals of Lakeside Mutual to periodically download a customer data report which helps them during risk assessment.
Eureka Server	Eureka Server provides a service registry. It is a regular Spring Boot application to which all other Spring services can connect to access other services. For example, the Customer Self-Service Backend uses Eureka to connect to the Customer Core. Usage of Eureka is optional.

Service Name	Description
Spring Boot Admin	Spring Boot Admin is an open source software for managing and monitoring Spring Boot applications. It is a Spring Boot application too. Usage within the Lakeside Mutual services is optional and only included for convenience with all security disabled.

Currently, since Ballerina has its own service discovery implementation, it does not provide native support for the transition of Eureka Server to Ballerina (there are no available libraries for Eureka in Ballerina). As such, this service will not be considered for the scope of this work. Also, the Spring Boot Admin service is not something valuable in Ballerina, so it will also not be considered.

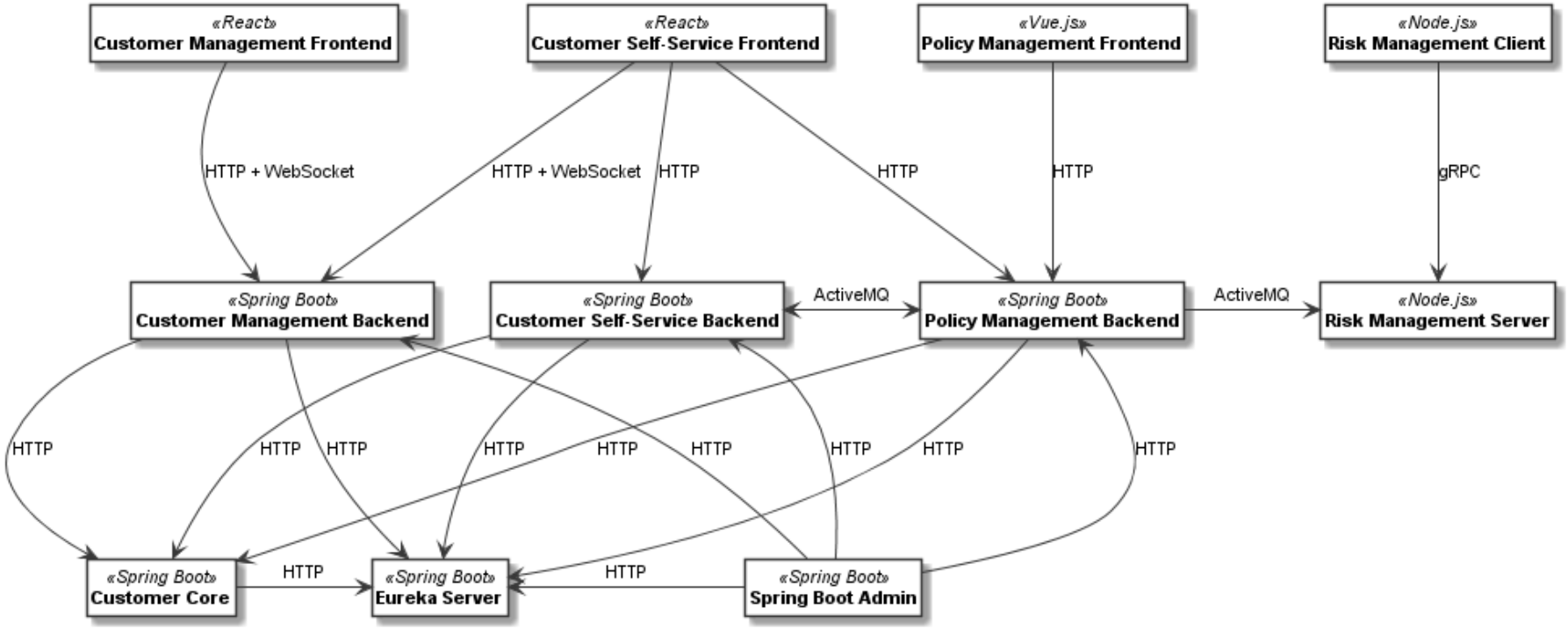

Figure 5.1: Service components at Lakeside Mutual and their relationships (Stocker 2021)

5.2 Migration Process

The migration to Ballerina will maintain the requirements of the base project. For migrating the solution to Ballerina, two main topics need to be addressed:

- Selected Service: The service that will be migrated, based on criteria like difficulty, importance, and interest.
- Strategy: The way the service will be converted. The service will be separated into different parts, with an explanation of how each of them will be migrated.

5.2.1 Selected Service

The first main decision is the selection of which service to migrate. Due to time and complexity constraints, only a single service will be migrated. The factors that will impact the choice of the service are the technology used to develop them and the aspects of those services that garner the most interest.

In terms of technology, the front-end services are excluded since Ballerina does not provide front-end capabilities, putting them outside of the scope of this work. For the back-end services, the Java Spring Boot services will be considered since these are more in number and contain the core of the project.

Regarding importance/interest, the services with some noteworthy characteristics besides the basic REST implementation will be the first focus of the migration.

With these factors in mind, the Customer Management Backend service was selected since it uses Spring Boot and it contains an implementation of WebSockets, a TCP-based protocol created for two-way communication between the client and the server (Madushan 2021), besides its REST implementation.

5.2.2 Strategy

The service consists of a domain, an infrastructure, and a group of interfaces for its main code. It also contain tests and documentation.

Firstly, the domain will be migrated. The Ballerina class definition, which is the closest representation of a Java object class definition, will be used for representing the entity and value object definitions. Additionally, as Ballerina interacts with its database implementations through records, a record definition is constructed for each object related to the database.

Next, the focus is the infrastructure. Ballerina's main file contains methods equivalent to each of the Java classes' methods, meaning all business logic exists in this

file. For instance, several functions are created from the Repository specifications to enable CRUD activities in the database.

After, the emphasis shifts to the interfaces. All the DTOs are converted into records. The remaining classes are changed into methods in either the main file or the service file, which contains the definition of the service endpoints. Configuration classes are also migrated, when possible, to service configurations in Ballerina, like the CORS configuration.

Chapter 6

Implementation

This chapter describes the implementation process of the solution using the Ballerina programming language. It describes the Ballerina implementation of the selected service, following the process described in the Analysis and Design (Chapter 5).

6.1 Service Migration

This section describes the technical details of the migration of the Customer Management Backend service into Ballerina. First, it starts by explaining the main structure of the service in Ballerina. Then, the following subsections focus on the different steps defined for the migration strategy created in the section 5.2.2. All the code presented in the following sections can be consulted on the public repository named Ballerina-Lakeside-Mutual-Customer-Management-Backend (Alves 2023).

6.1.1 Main structure

The service conversion is mainly into two files: the main.bal and the service.bal. The main.bal contains the definitions of the objects, records, and all the functions related to the business logic. The service.bal contains the services responsible for listening to requests and answering accordingly.

Some of the configurations are in the Config.toml file. The microservice port, authorization key, database username, and password are some examples of configurations in this file.

6.1.2 Domain

The domain classes were transformed into Ballerina objects, which use the "class" type. The functions of the objects utilize the "isolated" definition to provide concurrency-safe access to its attributes. Code Snippet 6.1 represent the Ballerina implementation

of the InteractionEntity, an example of a domain class.

```
public class InteractionEntity {
  private string? id;
  private string? date;
  private string? content;
  private boolean? sentByOperator;

  public isolated function init(string? id, string? date, string?
   content, boolean? sentByOperator){
    self.id = id;
    self.date = date;
    self.content = content;
    self.sentByOperator = sentByOperator;
  }

  public isolated function getId() returns string? {
    return self.id;
  }

  public isolated function setId(string? id) {
    self.id = id;
  }

  ...

  public isolated function toJson() returns json {
    return {
      "id": self.id,
      "date": self.date.toJson(),
      "content": self.content,
      "sentByOperator": self.sentByOperator
    };
  }
};
```

Code Snippet 6.1: Ballerina Implementation of InteractionEntity

A record was also made to enable accessibility to database operations. For example, the InteractionEntity object originated the InteractionEntityRecord, which can be seen in Code Snippet 6.2.

```
public type InteractionEntityRecord record {|
  string id?;
  string date?;
  string content?;
  boolean sent_by_operator?;
|};
```

Code Snippet 6.2: Ballerina Implementation of InteractionEntityRecord

The repository of the service is implemented in the base implementation using the JpaRepository interface, meaning the database implementation on the code uses the Java Persistence API (JPA) for managing the database schema. In Ballerina, it's not possible to use JPA.

As such, it was used the Java Database Connectivity (JDBC) module available for Ballerina, thus the necessity to implement all the essential methods necessary for managing data with the use of SQL commands. Code Snippet 6.3 represents the instation of the JDBC client necessary to access the database. Code Snippet 6.4 contains a representation of one of the methods developed in Ballerina relative to the implementation of a repository. The rest of the methods can be consulted on the projects GitHub repository (Alves 2023).

```
jdbc:Client jdbcClient = check new (
  url =  datasource,
  user = username, password = password,
  options = {
    properties: {"connectionTimeout": "300000"}
  },
  connectionPool = {
    maxOpenConnections: 10000
  }
);
```

Code Snippet 6.3: Ballerina Implementation of the JDBC Client

```
1  public function getInteractionLog(string customerId) returns
       InteractionLogAggregateRoot|error {
2    InteractionLogAggregateRootRecord rec = check jdbcClient->
       queryRow(`SELECT CUSTOMER_ID, USERNAME,
       LAST_ACKNOWLEDGED_INTERACTION_ID FROM INTERACTIONLOGS WHERE
       CUSTOMER_ID = ${customerId}`);
3    stream<InteractionEntityRecord, error?> interactionsEntries =
       jdbcClient->query(`SELECT ID,DATE,CONTENT,SENT_BY_OPERATOR FROM
       INTERACTIONS WHERE ID IN (SELECT INTERACTIONS_ID FROM
       INTERACTIONLOGS_INTERACTIONS WHERE
       INTERACTION_LOG_AGGREGATE_ROOT_CUSTOMER_ID = ${rec.customer_Id})
       `);
4    InteractionEntity[] interactions = [];
5    check from InteractionEntityRecord item2 in interactionsEntries
6      do {
7        interactions.push(new(item2.id,item2.date,item2.content,item2
       .sent_by_operator));
8      };
9    check interactionsEntries.close();
10   return new(rec.customer_Id,rec.username,rec.
       last_acknowledged_interaction_id,interactions);
11 }
```

Code Snippet 6.4: Ballerina Implementation of the InteractionLogRepository

The use of JDBC comes with an extra step regarding database initialization. The base project uses an H2 database that, by default, is dropped and created every time the service starts. By using JPA, the schema is rebuilt and automatically creates the tables. Since the JDBC Ballerina module provides no automation for table creation, the manual initialization of the tables necessary for the respective service is in the main.bal file. Code Snippet 6.5 shows the implementation of the initialization of the tables.

```
public function main() returns error?{
    log:printInfo("--- Customer Management backend started ---");
    log:printInfo("Start of main function.. ");

    if(ddl_auto == "drop_and_create"){
        _ = check jdbcClient->execute(`DROP TABLE IF EXISTS
    INTERACTIONLOGS_INTERACTIONS`);
        _ = check jdbcClient->execute(`DROP TABLE IF EXISTS
    INTERACTIONLOGS`);
        _ = check jdbcClient->execute(`DROP TABLE IF EXISTS
    INTERACTIONS`);
    }

    _ = check jdbcClient->execute(`CREATE TABLE IF NOT EXISTS
    INTERACTIONS(
        ID VARCHAR(255) NOT NULL,
        DATE TIMESTAMP,
        CONTENT VARCHAR(255),
        SENT_BY_OPERATOR BOOLEAN NOT NULL,
        PRIMARY KEY (ID)
    )`);

    _ = check jdbcClient->execute(`CREATE TABLE IF NOT EXISTS
    INTERACTIONLOGS(
        CUSTOMER_ID VARCHAR(255) NOT NULL,
        USERNAME VARCHAR(255),
        LAST_ACKNOWLEDGED_INTERACTION_ID  VARCHAR(255),
        PRIMARY KEY (CUSTOMER_ID)
    )`);

    _ = check jdbcClient->execute(`CREATE TABLE IF NOT EXISTS
    INTERACTIONLOGS_INTERACTIONS(
        INTERACTION_LOG_AGGREGATE_ROOT_CUSTOMER_ID VARCHAR(255) NOT
     NULL,
        INTERACTIONS_ID VARCHAR(255) NOT NULL,
        CONSTRAINT FKNRLR4POAGW2DTE8QMGEWNL9EU_INDEX_B FOREIGN KEY
    (INTERACTION_LOG_AGGREGATE_ROOT_CUSTOMER_ID) REFERENCES
    INTERACTIONLOGS(CUSTOMER_ID),
        CONSTRAINT UK_F9MORY4MPI8W7CI4IBSS33M11_INDEX_B UNIQUE (
    INTERACTIONS_ID)
    )`);

    log:printInfo("End of main function.. ");
}
```

Code Snippet 6.5: Ballerina Implementation of a repository initialization

The Data Transfer Objects (DTO) were mapped into records. The record contains each of the fields of the respective DTO. This record, however, varies from the original Java DTO in that it lacks associated methods like getters and setters, allowing each field to be changed by just replacing it with the new intended value. Additionally, whereas the original has a private access modifier in the Java implementation, the access to the record is public. Code Snippet 6.6 represents the Ballerina implementation of a DTO.

```
1 public type PaginatedCustomerResponseDto record {
2   string filter;
3   int 'limit;
4   int offset;
5   int size;
6   CustomerDto[] customers;
7 };
```

Code Snippet 6.6: Ballerina Implementation of a DTO

The exceptions were also mapped into records. Each record of an exception includes the type of the correspondent HTTP error. This inclusion allows this record to use the fields of the HTTP error. Code Snippet 6.7 shows the Ballerina implementation of an exception.

```
1 public type CustomerCoreNotAvailableException record {|
2   *http:BadGateway;
3 |};
```

Code Snippet 6.7: Ballerina Implementation of an exception

The REST controllers were all mapped into a single Ballerina REST service created with an HTTP listener. This HTTP listener was created by passing the port it must listen to and the version of the HTTP protocol it will use (lines 1 to 3 of the Code Snippet 6.8). All the operations of the REST controllers were converted to service resource functions, each with its request mapping. This functions also have a Ballerina HTTP Caller that saves the information of the endpoint requester and an HTTP Request that contains the request information. The resources can also have path and query parameters that allow sending information like, for example, ids or filters (line 7 of the Code Snippet 6.8). Each resource then responds with an HTTP Response that contains a status code and a body containing the information requested or a message to the caller. When the resources utilize functions that need concurrent access, it is utilized the "lock" statement, which allows for safe access to

mutable variables. Code Snippet 6.8 shows the equivalent of the REST controllers of a service migrated into a single HTTP service, with one of the available endpoints. The full service code can be seen on the project's GitHub repository (Alves 2023).

```
listener http:Listener httpListener = check new(port, {
  httpVersion: http:HTTP_1_1
});

service / on httpListener{

  resource function get customers(http:Caller caller, http:Request
   request, string filter = "", int 'limit = 10, int offset = 0)
   returns error? {
    http:Response response = new;
    lock {
      PaginatedCustomerResponseDto|error<
   CustomerCoreNotAvailableException> result = getCustomers(filter
   ,'limit,offset);
      if(result is PaginatedCustomerResponseDto){
        response.statusCode = 200;
        response.setJsonPayload(result.toJson());
      }else{
        return result;
      }
    }
    check caller->respond(response);
    return;
  }

...

}
```

Code Snippet 6.8: Ballerina Implementation of HTTP service

The REST controllers have security configurations where the Cross-Origin Resource Sharing (CORS) settings are defined. In Ballerina, this was implemented using the available ServiceConfig annotation from the HTTP module, which has a "cors" attribute that can be configured with several options like allowed methods, allowed origins, and other configurations like the base implementation. Code Snippet 6.9 demonstrates the CORS implementation in Ballerina.

```
@http:ServiceConfig {
  cors: {
    allowOrigins: ["*"],
    allowCredentials: true,
    allowMethods: ["GET", "POST", "PUT", "PATCH", "DELETE", "
   OPTIONS"],
    allowHeaders: ["authorization", "content-type", "x-auth-token"
   ],
    exposeHeaders: ["x-auth-token"]
  }
}
```

Code Snippet 6.9: Ballerina Implementation of CORS

The calls to the Customer Core service were transformed into methods that handle the response after calling the service using a Ballerina HTTP client. This HTTP client is the equivalent of the Feign Client used on the base project. The Customer Core service may return an error or a message mapped into a DTO, represented by a record in Ballerina. In Ballerina, it was used the "auth" attribute of the Ballerina HTTP client to implement the interceptor used by the client for setting the token needed for authorization. Code Snippet 6.10 the Ballerina implementation of the HTTP Client and one of calls to the Customer Core service to get information of the customers. The other calls can be consulted on the projects GitHub repository (Alves 2023).

```
http:Client coreClient = check new ("http://"+baseUrl,
    auth = {
        token: apiKeyValue
    }
);

public function getCustomer(CustomerId customerId) returns (()|
   CustomerDto|error<CustomerCoreNotAvailableException>) {
  string? customerIdString = customerId.getId();
  if(customerIdString is string){
    CustomersDto|error response = coreClient->get("/customers/" +
    customerIdString);
    if(response is CustomersDto){
      if(response.customers.length() == 0){
        return ();
      }else{
        return response.customers[0];
      }
    }else{
      log:printInfo(errorMessage,response);
      error<CustomerCoreNotAvailableException> err = error(
    errorMessage);
      return err;
    }
  }else{
    log:printInfo(errorMessage,customerIdString);
    error<CustomerCoreNotAvailableException> err = error(
    errorMessage);
    return err;
  }
}
```

Code Snippet 6.10: Ballerina Implementation of a call to Customer Core service

As stated in the description in Table 5.1, this service has a Websocket service implementation which is accountable for managing the chat feature and responsible for delivering chat messages in real time between a Lakeside Mutual agent and a customer.

WebSocket is a TCP-based protocol that enables two-way communication between the client and the server. WebSocket establishes and maintains a TCP (Transmission Control Protocol) connection with the server, allowing the server to send messages to the client on any occasion. This two-way communication is essential when the server

needs to update the client application in real time. The majority of web browsers also support WebSocket communication on web pages (Madushan 2021).

In Ballerina, a WebSocket service was implemented, which listens on a WebSocket listener created with the HTTP listener utilized in the REST service. The HTTP listener, which was modified to include HTTP version 1.1, was used to allow the REST and WebSocket services to listen on the same port. The WebSocket service has a single resource function responsible for responding to the HTTP upgrade request with the definition of a WebSocket service or an appropriate error message. Depending on the endpoint path, the resource function response is different. Code Snippet 6.11 shows the implementation of Websocket in Ballerina.

```
listener websocket:Listener wsListener = new websocket:Listener(
    httpListener);

service /ws on wsListener{
  resource function get .(http:Request req) returns websocket:
    Service|websocket:UpgradeError{
     log:printInfo(req.rawPath);
     if(req.rawPath.includes("/chat/messages")){
       return new WsService();
     }else{
       return service object websocket:Service {
         remote function onMessage(websocket:Caller caller,json
    message) returns websocket:Error?|error?{
              log:printInfo(message.toJsonString());
         }
       };
     }
  }
}
```

Code Snippet 6.11: Ballerina Implementation of WebSocket

The Websocket service was defined as a service class that includes the Service class from the Websocket Ballerina module. The service overrides the onMessage method that treats the requests received by a WebSocket Caller. In the context of this service, the onMessage listens to a message from a customer that is converted into an interaction. This interaction is then associated with the customers' interaction log if it exists. If it does not exist, a new interaction log is created. When the interaction log is created/updated, a notification is broadcasted for WebSocket

clients listening on the "notification" endpoint, and a message is sent to the clients listening on the "message" endpoint. Code Snippet 6.12 shows the implementation of the WebSocket Service.

```
service class WsService {
  *websocket:Service;

  remote function onMessage(websocket:Caller caller,MessageDto
   message) returns websocket:Error?|error?{
    log:printInfo("Processing message from " + message.username);
    string clientUrl = "ws://localhost:"+port.toString()+"/ws/topic
   /messages";
    final string? customerId = message.customerId;
    final string id = uuid:createType1AsString();
    string date = time:utcToString(time:utcNow());
    final InteractionEntity interaction = new(id,date,message.
   content,message.sentByOperator);

    InteractionLogAggregateRoot|error optInteractionLog =
   getInteractionLog(message.customerId);
    InteractionLogAggregateRoot interactionLog;
    if(optInteractionLog is InteractionLogAggregateRoot){
      InteractionEntity[]? interactions = optInteractionLog.
   getInteractions();
      if(interactions is InteractionEntity[]){
        interactions.push(interaction);
        interactionLog = new (optInteractionLog.getCustomerId(),
   optInteractionLog.getUsername(),optInteractionLog.
   getLastAcknowledgedInteractionId(),interactions);
        _ = check updateInteractionLog(interactionLog);
      }
    }else{
      InteractionEntity[] interactions = [];
      interactions.push(interaction);
      interactionLog = new (customerId,message.username,(),
   interactions);
      _ = check addInteractionLog(interactionLog);
    }
    _ = check broadcastNotifications();
    websocket:Client wsClient = check new(clientUrl);
    MessageDto dto = {id:id, date:date, customerId:message.
   customerId, username:message.username, content:message.content,
   sentByOperator:message.sentByOperator};
    check wsClient->writeMessage(dto);
  }
}
```

Code Snippet 6.12: Ballerina Implementation of WebSocket Service

6.2 Test Implementation

This section describes the implementation of unit and integration tests that were created to verify the solution created in Ballerina. First, it will be given a brief description of how to generally implement tests in Ballerina. Then, each of the following subsections will focus on each of the types of test, giving a brief explanation of the tests and an example.

6.2.1 Test structure

The Ballerina Test Framework utilizes resources and configurations to test code under diverse circumstances. It adheres to a general, ordered framework (WSO2 2022f).

A subdirectory called "tests" must be created inside the module for unit tests that are related to it. This module is typically linked to a test suite. The tests in this subdirectory are all regarded as belonging to the same test suite. Any name is permissible for the test source files (WSO2 2022f).

The test functions are merely Ballerina functions marked as tests using a unique annotation. There is no constraint on the test function name. However, test functions must be defined with the "@test:Config" annotation. A module's defined functions, services, and global variables are reachable from the test files. Therefore, if a symbol has already been declared in the module, you cannot redefine it in the test files. However, symbols defined in test files won't be visible inside module source files (WSO2 2022f).

Any files or resources solely for testing should be kept in the "resources" subfolder inside the "tests" directory. An absolute or relative path can be used to access these resource files (WSO2 2022f).

Configurable variables can be used to give testing configurations. A file called Config.toml, which is stored in the testing directory, can be used to supply the values for configurable variables (WSO2 2022f).

For the Customer Management Backend, it was created two test files, the service_test.bal for the service.bal file and main_test.bal for the main.bal file. This files test the functions and services of their counterparts.

6.2.2 Unit tests

The unit tests are the most simple of the implemented tests. Therefore, as observed in Code Snippet 6.13, the unit tests are composed of the necessary elements, like object instances, to test the diverse functions available. However, to make the tests more focused on the function's code, stubs like the one in line 6 were used to remove the dependencies of related functions.

```
1 import ballerina/test;
2
3 @test:Config
4 function testGetNumberOfUnacknowledgedInteractions(){
5     InteractionEntity mockEntity = test:mock(InteractionEntity);
6     test:prepare(mockEntity).when("isSentByOperator").thenReturn(
    false);
7     InteractionLogAggregateRoot log = new
    InteractionLogAggregateRoot("bunlo9vk5f","test","",[mockEntity])
    ;
8     test:assertEquals(log.getNumberOfUnacknowledgedInteractions()
    ,1);
9 }
```

Code Snippet 6.13: Ballerina Implementation of Unit Test

6.2.3 Integration tests

Integration tests were created to test the communication between the Ballerina service with outside sources. In the case of this work, these tests verify the connections with the service database and the Customer Core service. JDBC and HTTP clients are responsible for establishing these connections, respectively. The responses obtained from these clients are mocked with stubs to reduce complexity. They function by specifying the expected result for distinct method executions with specific arguments. Code Snippets 6.14 and 6.15 show the implementation of a test for a function with HTTP and JDBC clients, respectively.

```
import ballerina/test;
import ballerina/http;

@test:BeforeSuite
function beforeSuite() {
    coreClient = test:mock(http:Client);
}

@test:Config {}
function testGetCustomer() {
    CustomerId id = new CustomerId("bunlo9vk5f");

    CustomerDto expected = {
        "customerId": "bunlo9vk5f",
        "firstname": "Ado",
        "lastname": "Kinnett",
        "birthday": "1975-06-13T23:00:00.000+00:00",
        "streetAddress": "2 Autumn Leaf Lane",
        "postalCode": "6500",
        "city": "Bellinzona",
        "email": "akinnetta@example.com",
        "phoneNumber": "055 222 4111",
        "moveHistory": []
    };
    CustomersDto mockDto = {customers: [expected]};

    test:prepare(coreClient).when("get").withArguments("/customers/
    bunlo9vk5f").thenReturn(mockDto);
    test:assertEquals(getCustomer(id),expected);
}
```

Code Snippet 6.14: Ballerina Implementation of HTTP Test

```
import ballerina/test;
import ballerinax/java.jdbc;
import ballerina/sql;

@test:BeforeSuite
function beforeSuite() {
    jdbcClient = test:mock(jdbc:Client);
}

@test:Config
function testGetInteractionLog(){
    InteractionLogAggregateRoot expected = new
    InteractionLogAggregateRoot("bunlo9vk5f","test","",[]);

    stream<InteractionEntityRecord, sql:Error?> mockEntities = new
    ();

    InteractionLogAggregateRootRecord mockLog = {
        "customer_Id": "bunlo9vk5f",
        "username": "test",
        "last_acknowledged_interaction_id": "",
        "interactions": []
    };

    test:prepare(jdbcClient).when("queryRow").thenReturn(mockLog);
    test:prepare(jdbcClient).when("query").thenReturn(mockEntities)
    ;
    InteractionLogAggregateRoot|error result= getInteractionLog("
    bunlo9vk5f");
    if(result is InteractionLogAggregateRoot){
        test:assertExactEquals(result.getCustomerId(),expected.
    getCustomerId());
        test:assertExactEquals(result.getUsername(),expected.
    getUsername());
        test:assertExactEquals(result.
    getLastAcknowledgedInteractionId(),expected.
    getLastAcknowledgedInteractionId());
        test:assertEquals(result.getInteractions().toString(),
    expected.getInteractions().toString());
    }
}
```

Code Snippet 6.15: Ballerina Implementation of Database Test

Furthermore, some integration tests for the REST and WebSocket services were created. In addition to the previous definition, these tests utilize a Client to call the service exposed method to simulate a request from an actual real client. The most relevant difference resides in the additional assert on the response status code to confirm that the service responded as intended. Code Snippets 6.17 and 6.16 are an example of the implementation of a test to the WebSocket and REST services.

```
1  import ballerina/test;
2  import ballerina/http;
3  import ballerina/websocket;
4
5  @test:Config
6  public function testWebsocket() returns error? {
7      websocket:Client wsTestClient = check new ("ws://localhost
       :8100/ws");
8      var resp = wsTestClient.getHttpResponse();
9      if resp is http:Response {
10       test:assertEquals(resp.statusCode,http:
       STATUS_SWITCHING_PROTOCOLS);
11     }else{
12         test:assertFail("This test failed due to not receiving an
       HTTP Response.");
13     }
14 }
```

Code Snippet 6.16: Ballerina Implementation of WebSocket Service Test

```
import ballerina/test;
import ballerina/http;

@test:BeforeSuite
function beforeSuiteRest() {
    coreClient = test:mock(http:Client);
}

http:Client testClient = check new ("http://localhost:8100");

@test:Config {}
function testGetCustomerRest() returns error?{
    CustomerDto customer = {
        "customerId": "bunlo9vk5f",
        "firstname": "Ado",
        "lastname": "Kinnett",
        "birthday": "1975-06-13T23:00:00.000+00:00",
        "streetAddress": "2 Autumn Leaf Lane",
        "postalCode": "6500",
        "city": "Bellinzona",
        "email": "akinnetta@example.com",
        "phoneNumber": "055 222 4111",
        "moveHistory": []
    };

    CustomersDto mockDto = {customers: [customer]};

    json expected = customer.toJson();

    test:prepare(coreClient).when("get").withArguments("/customers/
    bunlo9vk5f").thenReturn(mockDto);
    http:Response result = check testClient->get("/customers/
    bunlo9vk5f");
    test:assertEquals(result.statusCode, http:STATUS_OK);
    test:assertEquals(result.getTextPayload(), expected.toJsonString
    ());
}
```

Code Snippet 6.17: Ballerina Implementation of REST Service Test

Chapter 7

Evaluation and Experimentation

This chapter documents the performed evaluation and experimentation.

After having a better understanding of the project from the research of the Ballerina programming language and microservices, and based on the main objective specified in the section 1.3, the author identified the premise for this project. The hypothesis is: the solution implemented in Ballerina will have better effects on microservices architecture-based applications when compared to the solution implemented using a more conventional programming language regarding maintainability and performance. Maintainability and performance are the quality attributes identified in the State of the Art (Chapter 3) as the ones to be evaluated.

To prove the hypothesis, the author of this dissertation selected a project that used a common programming language. This project needed to fulfill some criteria like being open-sourced, the number of existing microservices, having documentation, and the application of some patterns like API gateway and service discovery. Then, the author refactored the project using the Ballerina programming language. After having both solutions, the author will use an approach to help him understand how Ballerina influences maintainability and performance. The selected approach for the evaluation is the Goal Question Metric (GQM) paradigm.

This dissertation will provide information for future developers to help them make informed decisions when selecting Ballerina as one of the programming languages for their projects.

7.1 Approach

The evaluation method used to analyze both solutions is Goal Question Metric (GQM). Victor Basili and David Weiss suggested the GQM paradigm as an analysis method to assist teams in determining how to quantify and assess challenging issues in software development (Ciceri et al. 2022).

The fundamental principle of GQM is straightforward: one must comprehend the reasons for measuring something to measure it effectively. Understanding the motives - the goals to achieve and the questions to answer to assess progress - allows the discovery and selection of optimal metrics. Teams are more likely to trust the metrics and use them as the basis for future decisions if they know the reasons behind them. GQM can transform goals from vague wishful statements into models that can be measured and verified (Ciceri et al. 2022).

The models have three levels (Basili, Caldiera, and Rombach 1994):

- Conceptual level (Goal): A goal is specified for an object (products, processes, or resources), for different reasons, concerning various quality models, from diverse points of view, and concerning a particular environment.
- Operational level (Question): A series of questions describes how a particular goal is achieved based on a characterizing model. The questions attempt to characterize the object of measurement with a chosen quality problem and determine its quality from the selected point of view.
- Quantitative level (Metric): A set of metrics are related to every question to answer it quantitatively. The metrics can be objective, meaning they only depend on the object of measurement, or subjective, depending on the target and the point of view.

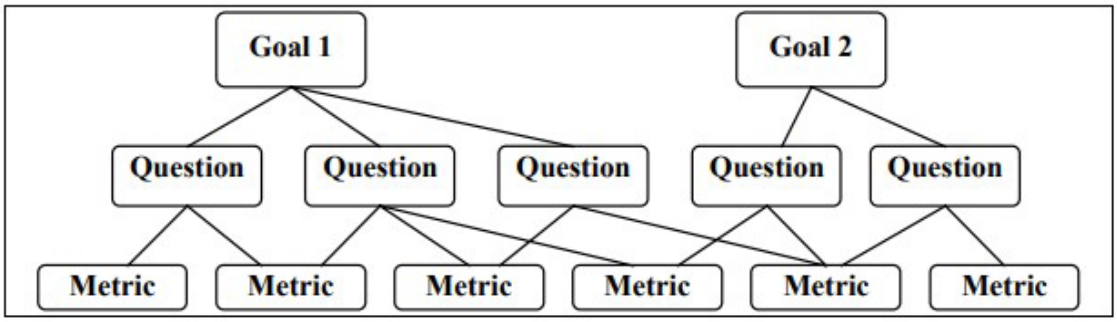

Figure 7.1: GQM Model Structure (Basili, Caldiera, and Rombach 1994)

A GQM model consists of a hierarchical structure, as shown in Figure 7.1. To create a model, first, it is identified a set of quality goals. From these goals, the evaluator specifies the questions. The following step is the definition of the measures (metrics) needed to answer the questions. After having the model, the evaluator needs to identify the data collection solutions that will gather data for the metrics.

As stated at the beginning of this chapter, the GQM will be used to identify the

metrics necessary to reach a defined goal. Figure 7.2 and Table 7.1 display the implemented GQM model with the definition of the goal, questions, and metrics related to the project. The goal is to discover the effects of adopting the Ballerina programming language for microservices. The obtained results will allow engineering teams to make informed decisions. The questions are derived from the research questions described in the section 1.4. Therefore, the question "What are the effects of using Ballerina in microservices architecture-based applications regarding the identified quality attributes?" was used as a reference target for the performance and maintainability, which were the quality attributes identified as the answer for the question "What are the most relevant quality attributes related to microservices architecture?". The metrics are related to one of the questions of each quality attribute. Each question has two metrics for which data will be gathered for both solutions, allowing for a comparison between them. This comparison will help reach an understanding of the effects of using Ballerina.

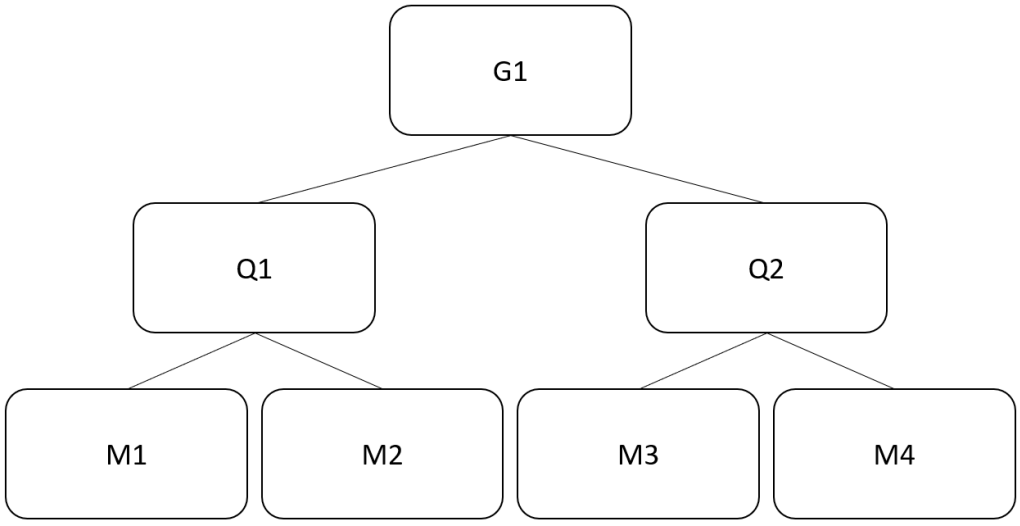

Figure 7.2: Implemented GQM Model

Table 7.1: GQM

G1: Discover the effect of adopting Ballerina for an application with a microservices architecture to allow an engineering team to make informed decisions	
Q1: What are the effects of using Ballerina in microservices architecture-based applications regarding maintainability?	Q2: What are the effects of using Ballerina in microservices architecture-based applications regarding performance?
M1: LOC M2: Indentation Debt Complexity	M3: Average Response Time M4: Throughput

The data for the metrics can be obtained by using some tools or by manually analyzing the solutions. Each of the following sections will describe the methods used to obtain the necessary data, along with the scales or values that are considered acceptable. The sections are divided by quality attributes and address all the decided metrics related to the said attribute.

7.1.1 Maintainability

For maintainability, the evaluated metric will be the lines of code (LOC) and the Indentation Debt of both solutions. At the moment of writing this document, Ballerina does not have support for any maintainability measuring tools. However, the team behind the language is studying the use of some tools like Sonarqube to support them in the future. This was confirmed by one of the instructors of the official Ballerina discord server, which can be joined from the official community page (WSO2 2022d).

LOC consists on counting every line of code that contains actual code, skipping empty lines and comment lines. It counts as a complexity metric, where the biger the size, the more complex it is (Ciceri et al. 2022). To obtain the values of LOC for both solutions it will be used the VS Code Counter extension (Ushiyama and Mayhé 2020).

Indentation Debt is a metric that focus on the maximum indentation levels in functions and methods. The more indentation a function or method has, the more complex it is. To know the overall complexity of a file or class, it can be used a weighted average of the complexities of all functions and/or methods of a class or

file. To calculate the complexity based on the indentation debt, it will be used a python script that reads the content of the files of each solution, calculates the number of indentations and then calculates the complexity (Markus 2018).

7.1.2 Performance

In terms of performance, what is going to be evaluated is the average response time and the throughput in both solutions.

The average response time is the mean average of a group of requests' response times. Response Time is the difference in milliseconds between the start and end of a request (Apache 2023).

Throughput corresponds to the number of requests divided by a time unit. The time to consider is the difference between the start of the first request to the end of the last (Apache 2023).

To obtain the necessary values, the tool used is JMeter since it provides dashboards that show these metrics, as well as allows the creation of tests where multiple requests can be performed concurrently (Apache 2023).

7.2 Experiments

7.2.1 Maintainability

Lines of code (LOC), as previously discussed, is used to measure maintainability. The results are shown in Table 7.2, obtained from using the VS Code Counter extension (Ushiyama and Mayhé 2020) on the Spring Boot and Ballerina services folders.

Table 7.2: LOC Metrics

Solution	**LOC**
Spring Boot	1150
Ballerina	677

As previously stated, a python script was used to calculate the indentation debt complexity. Before doing the calculations, it is required to ascertain the meaning of an indent for each project. By analyzing the code of both solutions, an indent was

characterized by a single tab. With the meaning of an indent identified, the python script can be applied. The steps for the script are the following:

- Get all the files of a solution;
- Read the content of each file and concatenate the data in a single structure;
- Calculate relevant information (number of lines, if a line is a comment or empty, indents per line)
- Filter out the comments and empty lines and aggregate the indents by files
- Calculate the complexity of a file based on the lines and indents
- Calculate the full complexity of the solution

By applying the python script, the results in Table 7.3 were obtained.

Table 7.3: Indentation Metrics

Solution	**Total Indentation**	**Min**	**Max**	**Complexity**
Spring Boot	1176	0	6	31.2
Ballerina	1061	0	6	5.84

Based on the examination of Table 7.2 the original Spring Boot-based solution has around 69.9% more LOC than the Ballerina-based version. From the analysis of Table 7.3 the Ballerina solution is 81,3% less complex than the Spring Boot-based solution. Since the LOC measurement and the indentation debt complexity for the Ballerina-based solution is lower than the one for Spring Boot, it is thought to have greater maintainability.

7.2.2 Performance

Apache JMeter (Apache 2023) is the tool used to automate the procedure for performance metrics. The scenarios were modified to fit the capabilities of the developer's system. The test plan for the service can be seen in Figure 7.3. In this test plan, the users execute all the HTTP requests of the Thread Group several times as defined in the configuration. The different configurations that were applied on the Ballerina and Spring Boot solutions are:

- 10 virtual users executing the plan 10 times;
- 100 virtual users executing the plan 10 times;
- 1000 virtual users executing the plan 10 times.

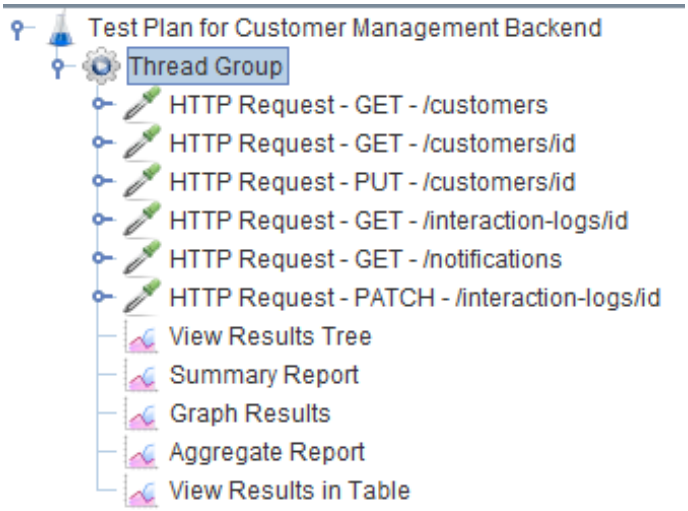

Figure 7.3: Apache JMeter test plan

Table 7.4 shows the performance metrics obtained for the different configurations for both services. The tests were done on the same computer, with only the necessary programs running. The computer has a 12th Gen Intel(R) Core(TM) i7-12700H CPU processor, 16GB of RAM and runs Windows 11 Pro.

Table 7.4: Response Time and Throughput Table Report for the Customer Management Backend

Solution	**Number of Virtual Users**	**Min.**	**Max.**	**Avg.**	**Throughput**
Ballerina	10	1	14	3	519.9
Ballerina	100	1	323	39	1289.8
Ballerina	1000	1	12360	721	1769.4
Spring Boot	10	1	25	4	522.6
Spring Boot	100	1	121	24	2360.3
Spring Boot	1000	1	700	396	2384.6

From a brief analysis of the results, for a low and medium number of virtual users both solutions perform almost the same in terms of response time. However, when the number of virtual users increases drastically, the Spring Boot solutions presents better results. In terms of throughput, the Spring Boot solution is able to transfer more data per second than the Ballerina solution.

7.3 Summary

Both solutions are very different in terms of the technologies they use. The service in Ballerina uses the modules available on the Ballerina Central while the Spring Boot service developed in Java uses the Spring Boot framework with its libraries. However, both solutions have the same functionalities available for use. Therefore, the tests done on those functionalities allow the comparison of both solutions using the same tests and metrics.

During the analysis of the literature in the State of the Art (Chapter 3), the qualitie attributes performance and maintainability were identified as the answer for the research question "What are the most relevant quality attributes related to microservices architecture?".

The evaluation findings presented in this chapter shed light on the previously developed research question: "What are the effects of using Ballerina in microservices architecture-based applications regarding the identified quality attributes?", from which derived the two questions of the GQM.

Performance-wise, it is stated that the Spring Boot-based solution exhibits better performance compared to Ballerina when a higher number of virtual users (1000) execute service requests. The results for this number of users have the most discrepancy, with the maximum response time in Spring Boot being almost 95% lower than the max response time in Ballerina, the average response time in Ballerina being around 80% higher and the throughput being around 25% lower. However, for a smaller number of virtual users, there were no noticeable differences. Therefore, it indicates that choosing Ballerina as the technology for microservices architecture may impact performance, especially when dealing with larger volumes of data.

In terms of maintainability, this chapter highlights that the Ballerina solution surpasses Spring Boot by having a lower line of code (LOC) and a lower indentation debt complexity. As such, it suggests that Ballerina offers better maintainability in the context of microservices architecture-based applications, as it requires less code to achieve similar functionality and has an overall lower complexity.

The evaluation findings demonstrate that both Ballerina and Spring Boot services are capable of meeting the crucial quality requirements for microservices architecture. As stated in section 5.1.1, the original solution had some requirements in terms of performance and maintainability. This requirements were achieved by both solutions, with both solutions being capable of handling 1000 users concurrently and having at least 80% of the requests answered in less than two seconds. However, there are

notable differences between the two services. Spring Boot outperforms Ballerina in terms of performance when handling a higher number of virtual users, while Ballerina excels in maintainability due to its lower LOC and indentation debt complexity compared to Spring Boot.

Chapter 8

Conclusion

This chapter details all of the findings made during this dissertation. Firstly, it details what has been achieved. Next, it describes the found limitations and the threats to validty. Finally, the chapter delineates the possible future work and ends with a section dedicated to the study, concluding considerations and contributions.

8.1 Achievements

This thesis provides information about the current state of the art of microservices-based systems and the evolution toward language-based solutions with the example of Ballerina. The thesis also analyzes the migration of an existing Spring Boot-based solution into a Ballerina solution, which is available on GitHub (Alves 2023). Additionally, it was performed a comparison between the solutions regarding maintainability and performance.

The outputs of the document are an assessment of the benefits and liabilities of using Ballerina in contrast to Spring Boot. The document also provides a possible strategy to convert existing microservices into Ballerina equivalents.

8.2 Difficulties

During the thesis development, there were some challenges related to the innovative factor of using language-based approaches, specifically with using Ballerina, which is a very recent programming language. Therefore, there were some difficulties in finding literature about the use of Ballerina when most of the focus is on using frameworks like Spring Boot.

There were also some difficulties with the implementation. Ballerina is an innovative programming language with user-friendly official documentation that provides

explanations and examples of utilizing the language to create several microservice-oriented use cases. However, due to the unfamiliarity with the programming language and the implementation of some edge cases not often used, some issues took more time to figure out how to solve and overcome.

8.3 Threats to Validity

Firstly, the experiments were performed in a local environment with limited resources and availability. Therefore, there was no possibility of doing tests on a larger scale. For the tests performed, the maximum number of virtual users used was 1000. If the tests were to be done in an actual production environment, the results would probably be different.

In addition, it is noteworthy that the developer is unfamiliar with Ballerina. Should the developer have had more knowledge of how to use and the good practices of Ballerina, the results could vary.

8.4 Future Work

For the context of this project, only a single service was migrated. In the future, it would be interesting to migrate more services to see the possible difficulties of implementing other functionalities like messaging and to discover the impacts on the orchestration of multiple Ballerina services.

The tests performed for the performance metrics were done to a finite number of virtual users that made a finite number of requests. It would be interesting to do a ramp up test were the number of users increases over a period of time while always making requests. It also would be interest to develop stress tests do discover at which point the solutions are not able to handle the load of requests they are receiving.

Additionally, the original service was created in Spring Boot. It would be noteworthy to migrate services of other frameworks or languages to Ballerina to investigate how Ballerina fares against other technologies regarding maintainability and performance.

8.5 Final Considerations

Firstly, the objective of the thesis has been achieved. As such, the author considers the thesis's overall development a success. The author had to interact with a language-based approach for developing microservices, which was a new experience since the author was unknowledgeable before the thesis execution. Language-based approaches for developing microservices are a concept that is becoming more and more interesting in the microservices community, and the author expects it to be further explored in the future by the research community, along with the expansion of this experience.

The project has challenged the author to think and discover how to conduct research and manage a thesis. Additionally, the project challenged the author to test and develop something unknown to him. The experience is of much value to the author since it provided him with valuable experience in microservices, which can be beneficial to his future career.

Bibliographic References

Alves, Andre (2023). *Ballerina-Lakeside-Mutual-Customer-Management-Backend*. url: https://github.com/AndreAlves1171068/Ballerina-Lakeside-Mutual-Customer-Management-Backend (visited on 06/04/2023).

Apache (2023). *Glossary*. url: https://jmeter.apache.org/usermanual/glossary.html (visited on 01/14/2023).

Basili, Victor R., Gianluigi Caldiera, and H. Dieter Rombach (1994). "The goal question metric approach". In: *Encyclopedia of software engineering*, pp. 528–532.

Bushong, Vincent et al. (Aug. 2021). "On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study". In: *Applied Sciences* 11 (17). issn: 2076-3417. doi: 10.3390/app11177856.

Ciceri, Christian et al. (May 2022). *Software Architecture Metrics*. Ed. by Melissa Duffield, Sarah Grey, and Katherine Tozer. 1st ed. O'Reilly Media, Inc., pp. 171–172. isbn: 9781098112233.

Fernando, Anjana and Lakmal Warusawithana (2020). *Beginning Ballerina Programming*. 1st ed. Apress. isbn: 978-1-4842-5138-6. doi: 10.1007/978-1-4842-5139-3. url: http://link.springer.com/10.1007/978-1-4842-5139-3.

Guidi, Claudio et al. (Apr. 2017). "Microservices: a Language-based Approach". In: *Computing Research Repository* abs/1704.08073. url: http://arxiv.org/abs/1704.08073.

Indrasiri, Kasun and Prabath Siriwardena (Dec. 2018). *Microservices for the Enterprise*. Ed. by Jonathan Gennick, Laura Berendson, and Jill Balzano. 1st ed. Apress. isbn: 978-1-4842-3857-8. doi: 10.1007/978-1-4842-3858-5. url: http://link.springer.com/10.1007/978-1-4842-3858-5.

ISO/IEC JTC 1 (2022a). *ISO 25000 Portal*. url: https://iso25000.com/index.php/en/ (visited on 01/29/2023).

– (2022b). *ISO/IEC 25010*. url: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (visited on 01/29/2023).

Jewell, Tyler (May 2018). *Ballerina Microservices Programming Language: Introducing the Latest Release and "Ballerina Central"*. url: https://www.infoq.com/articles/ballerina-microservices-language-part-1/.

Koen, Peter et al. (2001). "Providing Clarity and A Common Language to the "Fuzzy Front End"". In: *Research-Technology Management* 44.2, pp. 46–55. doi: 10.1080/08956308.2001.11671418. eprint: https://doi.org/10.1080/08956308.2001.11671418. url: https://doi.org/10.1080/08956308.2001.11671418.

Labs, Grafana (2022). *Ballerina Metrics*. url: https://grafana.com/grafana/dashboards/5841-ballerina-metrics/ (visited on 12/27/2022).

Li, Shanshan et al. (2021). "Understanding and addressing quality attributes of microservices architecture: A Systematic literature review". In: *Information and Software Technology* 131, p. 106449. issn: 0950-5849. doi: https://doi.org/10.1016/j.infsof.2020.106449. url: https://www.sciencedirect.com/science/article/pii/S0950584920301993.

Madushan, Dhanushka (Sept. 2021). *Cloud Native Applications with Ballerina*. Ed. by Rohit Singh et al. 1st ed. Packt Publishing Ltd. isbn: 9781800200630.

Markus (2018). *Calculating Indentation-based Complexity*. url: https://www.feststelltaste.de/calculating-indentation-based-complexity/ (visited on 06/26/2023).

Newman, Sam (July 2021). *Building Microservices*. Ed. by Melissa Duffield et al. 2nd ed. O'Reilly Media, Inc. isbn: 9781492034025.

Nicola, Susana (2022). *ANÁLISE DE VALOR*. url: https://moodle.isep.ipp.pt/pluginfile.php/240297/mod_resource/content/1/An%C3%A1lise_Valor_Aula_4_21NOV_2018_1hora_AHP.pdf (visited on 12/28/2022).

Oram, Andy (Aug. 2019). *Ballerina: A Language for Network-Distributed Applications*. Ed. by Ryan Shaw et al. 1st ed. O'Reilly Media, Inc. isbn: 9781492061151. url: https://wso2.com/wso2_resources/ballerina-a-language-for-network-distributed-applications.pdf.

Peffers, Ken et al. (2007). "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24.3, pp. 45–77. doi: 10.2753/MIS0742-1222240302. eprint: https://doi.org/10.2753/MIS0742-1222240302. url: https://doi.org/10.2753/MIS0742-1222240302 (visited on 11/19/2022).

Ratnayake, Dakshitha (June 2022). *How MOSIP Uses Ballerina WebSubHub for Event-Driven Integration*. url: https://thenewstack.io/how-mosip-uses-ballerina-websubhub-for-event-driven-integration/.

RedMonk (2021). *The RedMonk Programming Language Rankings: June 2021*. url: https://redmonk.com/sogrady/2021/08/05/language-rankings-6-21/ (visited on 12/28/2022).

RedMonk (2022a). *About*. url: https://redmonk.com/about/ (visited on 12/28/2022).

– (2022b). *The RedMonk Programming Language Rankings: January 2022*. url: https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/ (visited on 12/28/2022).

– (2022c). *The RedMonk Programming Language Rankings: June 2022*. url: https://redmonk.com/sogrady/2022/10/20/language-rankings-6-22/ (visited on 12/28/2022).

Rich, Nick and Matthias Holweg (2000). "Value analysis". In: *Value engineering*. (Visited on 02/12/2022).

Saaty, Thomas L. (1990). "How to make a decision: The analytic hierarchy process". In: *European Journal of Operational Research* 48.1. Desicion making by the analytic hierarchy process: Theory and applications, pp. 9–26. issn: 0377-2217. doi: https://doi.org/10.1016/0377-2217(90)90057-I. url: https://www.sciencedirect.com/science/article/pii/037722179090057I (visited on 12/28/2022).

Stocker, Mirko (2021). *Lakeside Mutual*. url: https://github.com/Microservice-API-Patterns/LakesideMutual (visited on 02/21/2023).

Ushiyama, Kentaro and Junior Mayhé (2020). *VScode Counter*. url: https://github.com/uctakeoff/vscode-counter (visited on 06/17/2023).

Valdivia, J. A. et al. (Dec. 2020). "Patterns Related to Microservice Architecture: a Multivocal Literature Review". In: *Programming and Computer Software* 46 (8), pp. 594–608. issn: 0361-7688. doi: 10.1134/S0361768820080253.

Valdivia, Jose A., Xavier Limon, and Karen Cortes-Verdin (Oct. 2019). "Quality attributes in patterns related to microservice architecture: a Systematic Literature Review". In: IEEE, pp. 181–190. isbn: 978-1-7281-2524-4. doi: 10.1109/CONISOFT.2019.00034.

Wang, Yingying, Harshavardhan Kadiyala, and Julia Rubin (July 2021). "Promises and challenges of microservices: an exploratory study". In: *Empirical Software Engineering* 26 (4). issn: 1382-3256. doi: 10.1007/s10664-020-09910-y.

Weerawarana, Sanjiva et al. (Sept. 2018). "Bringing Middleware to Everyday Programmers with Ballerina". In: ed. by Mathias Weske et al. 1st ed. Springer Cham, pp. 12–27. doi: 10.1007/978-3-319-98648-7_2. url: http://link.springer.com/10.1007/978-3-319-98648-7_2.

WSO2 (2022a). *Ballerina Central*. url: https://central.ballerina.io (visited on 12/26/2022).

– (2022b). *Ballerina GitHub*. url: https://github.com/ballerina-platform (visited on 12/19/2022).
– (2022c). *Ballerina.io*. url: https://ballerina.io (visited on 12/27/2022).
– (2022d). *Community*. url: https://ballerina.io/community/ (visited on 06/17/2023).
– (2022e). *Observe Ballerina programs*. url: https://ballerina.io/learn/observe-ballerina-programs/ (visited on 12/26/2022).
– (2022f). *Structure tests*. url: https://ballerina.io/learn/test-ballerina-code/structure-tests/ (visited on 06/17/2023).
– (2022g). *Test Ballerina Code*. url: https://ballerina.io/learn/test-ballerina-code/test-a-simple-function/ (visited on 12/29/2022).

Zimmermann, Olaf et al. (Nov. 2022). "Lakeside Mutual Case Study". In: *Patterns for API-Design: Simplifying Integration with Loosely Coupled Message Exchanges*. 1st ed. Vol. 1. Addison-Wesley, pp. 85–99. isbn: 9780137670093.