INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

Evaluating Ballerina's impact on performance and elasticity in distributed applications: A comparison with Java

RICARDO JORGE GONÇALVES COELHO
setembro de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Evaluating Ballerina's impact on performance and elasticity in distributed applications: A comparison with Java

Ricardo Jorge Gonçalves Coelho

Dissertation

Master in Informatics Engineering - Software Engineering

Supervisor: Isabel Azevedo (IFP)

Porto, September 15, 2024

Acknowledgements

This section expresses my profound gratitude to all the people who played a significant role in the development of this project. To my friends and colleagues at ISEP, I am deeply thankful for the privilege of spending these years by your side. Your presence, both in the good and challenging times, has been instrumental in shaping my academic journey and personal growth.

However, I must highlight my closest group of friends because, without them, the completion of this project and my degree would not have been possible. My big thanks go to Nuno Rocha and Rui Soares for showing me that the sleepless nights and hours spent solving problems were worth it in the end, for supporting me throughout these years, and for helping me grow as a person.

To my family, I am eternally grateful for your unwavering support throughout my academic journey. My parents, my sister, and my girlfriend, your love and encouragement have been the bedrock of my success. Without you, this significant milestone in my life would have remained a distant dream.

I would also like to thank my supervisor at ISEP, Professor Isabel Azevedo, who was always willing to help and support me during this project. All this help and support has undoubtedly contributed to raising the quality of my work.

Declaration of Integrity

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

I have not plagiarized or applied any form of misuse of information or falsification of results throughout the process that led to its preparation.

Therefore, the work presented in this document is original, of my authorship, and has not been used previously for any other purpose.

I also declare that I am fully aware of P. PORTO's Code of Ethical Conduct.

ISEP, Porto, September 15, 2024

Ricardo Jorge Gonçalves Coelho

Resumo

Atualmente, existem inúmeras linguagens de programação, algumas mais versáteis e outras criadas com propósitos específicos [1]. Ballerina, uma nova linguagem de programação que surgiu recentemente, é, segundo a sua equipa de desenvolvimento, uma linguagem inovadora projetada especificamente para o desenvolvimento de aplicações na *cloud* [2].

Apesar de Ballerina surgir no mercado como uma solução para o desenvolvimento e integração de aplicações distribuídas [3], aspetos como a sua performance e elasticidade em comparação com linguagens mais tradicionais no desenvolvimento de aplicações, permanecem pouco explorados.

Assim, o seu impacto como linguagem e plataforma para o desenvolvimento dessas aplicações ainda é desconhecida, juntamente com as suas potencialidades e limitações. Além disso, numa das suas versões mais recentes foi anunciado que a linguagem suporta a criação de executáveis nativos GraalVM prometendo "melhorias de desempenho e redução nos tempos de inicialização" [4], aspetos esses que para além de serem bastante promissores ainda não foram examinados para se tirar uma conclusão concisa.

Tendo em conta estas incógnitas, analisou-se o impacto de Ballerina como linguagem e plataforma para o desenvolvimento de aplicações comparativamente com uma linguagem de programação mais tradicional, Java. Um projeto *open-source*, baseado numa arquitetura em microsserviços, foi migrado para Ballerina para depois se recolherem dados, quer das aplicações originais, quer das novas aplicações migradas. Os testes de performance e elasticidade determinaram que, com o uso de GraalVM e a compilação Ahead-Of-Time (AOT), as aplicações Ballerina conseguem melhorar os seus tempos de arranque em cerca de 90% e a sua performance em aspetos como tempos de resposta, rendimento e resiliência em gerir um maior número de utilizadores simultâneos. Para além disso, as aplicações Ballerina conseguiram superar os resultados das aplicações em Java em vários cenários, particularmente em condições de carga altas.

Com base nos resultados obtidos, concluiu-se que o uso de Ballerina traz várias vantagens em termos de performance e elasticidade, o que faz dela uma opção a ser considerada quer para o desenvolvimento ou integração de aplicações.

Palavras-chave: Ballerina, Java, GraalVM, Performance, Elasticidade, Aplicações distribuídas.

Abstract

Nowadays, there are many programming languages, some quite versatile and others developed for a specific purpose [1]. Ballerina, a new programming language recently launched, is, according to its development team, an innovative language designed specifically for the development of cloud applications [2].

Although Ballerina has emerged on the market as a solution for the development and integration of distributed applications [3], aspects such as its performance and elasticity compared to more traditional languages in application development remain slightly explored.

As such, its impact as a language and platform for developing these applications is still unknown, along with its potential and limitations. Furthermore, in one of its most recent versions, it was announced that the language supports the creation of native GraalVM executables, promising "performance improvements and a reduction in start-up times" [4], aspects which, apart from being very promising, have not yet been examined to draw a concise conclusion.

In response to these gaps, this study analysed the impact of Ballerina as a language and platform for developing applications compared to a more traditional programming language, Java. An open-source project, based on a microservices architecture, was migrated to Ballerina in order to collect data from both the original and newly migrated applications. Performance and elasticity tests determined that, with the use of GraalVM and Ahead-Of-Time (AOT) compilation, Ballerina applications were able to improve their start-up times by around 90% and their performance in aspects such as response times, throughput and resilience in managing greater number of simultaneous users. In addition, Ballerina applications were able to outperform Java applications in various scenarios, particularly under high-load conditions.

Based on the results obtained, it was concluded that the use of Ballerina brings several advantages in terms of performance and elasticity, which makes it an option to be considered for both application development and integration.

Keywords: Ballerina, Java, GraalVM, Performance, Elasticity, Distributed applications.

Table of contents

1 Introduction 1

1.1 Context 1

1.2 Problem description 2

1.3 Objective and research methodology 3

1.4 Ethical considerations 4

1.5 Document structure 4

2 Background 7

2.1 Key concepts in the Java Virtual Machine ecosystem 7

2.1.1 Java virtual machine 8

2.1.2 Java development kit 8

2.1.3 Java frameworks and libraries 8

2.2 Ballerina 8

2.2.1 History 8

2.2.2 Key features 9

2.2.3 Use cases 11

2.3 GraalVM 13

2.3.1 History 13

2.3.2 Key features 15

3 Literature review 17

3.1 Research questions 17

3.2 Data sources 17

3.3 Search terms 18

3.4 Eligibility criteria 19

3.5 Data collection process 19

3.6 Results 20

3.7 Discussion 20

3.7.1 RQ1: What features and improvements introduced in the Ballerina language affect the performance of distributed applications? 21

3.7.2 RQ2: What features and improvements introduced in the Ballerina language affect the elasticity of distributed applications? 24

3.8 Conclusion 24

4 Analysis and design 27

4.1 Project to migrate 27

4.1.1 Business context 27

4.1.2 Architecture 28

4.1.3 General workflow of the system 30

4.1.4 Domain model 30
4.2 Migration process 31
4.2.1 Selected components 31
4.2.2 Migration strategy 31
5 Implementation 33
5.1 API Gateway migration 33
5.2 User BO microservice migration 35
5.3 Adjustments 39
5.4 Generating native images 40
5.5 Docker deployment 40
5.6 Tests 41
6 Evaluation 43
6.1 Methodology 43
6.1.1 Performance 44
6.1.2 Elasticity 45
6.2 Experiments 45
6.2.1 Performance 45
6.2.2 Elasticity 50
6.3 Summary 51
7 Conclusion 53
7.1 Achievements and contributions 53
7.2 Difficulties 54
7.3 Threats to validity 54
7.4 Future work 55
7.5 Final considerations 55
References 57
Appendix A 63
Appendix B 65
Appendix C 71
Appendix D 73
Appendix E 77

List of Figures

Figure 1 - RedMonk ranking January 2024 2
Figure 2 - Key concepts about the Java Virtual Machine ecosystem 7
Figure 3 - Legacy solution vs Ballerina solution 12
Figure 4 - Differences between both approaches 14
Figure 5 - Search query 18
Figure 6 - PRISMA systematic 20
Figure 7 - Benchmark test results for selected Renaissance benchmarks 22
Figure 8 - Response time in milliseconds obtained from a custom benchmark 22
Figure 9 - Average test results for selected DaCapo benchmarks 23
Figure 10 - Current Implementation model 28
Figure 11 - Deployment diagram from PMS 30
Figure 12 - Docker resource metrics 41
Figure 13 - PMS postman collection 42
Figure 14 - GQM model structure 44
Figure 15 - Request to use the project 63
Figure 16 - Authorization from teacher 63
Figure 17 - Authorization from student 1 64
Figure 18 - Authorization from student 2 64
Figure 19 - Authorization from student 3 64
Figure 20 - Authorization from student 4 64
Figure 21 - General workflow of the system 71
Figure 22 - User aggregate 73
Figure 23 - Park aggregate 74
Figure 24 - Payments aggregate 75

List of tables

Table 1 - Data Sources 18
Table 2 - Search results 20
Table 3 - PMS components description 29
Table 4 – Load tests HTTP/1.1 results 46
Table 5 – Load tests HTTP/2 results 46
Table 6 - Comparison Ballerina AOT with Java AOT 47
Table 7 - Comparison Ballerina with Java 47
Table 8 - HTTP/1.1 Stress test results 50
Table 9 - HTTP/2 Stress test results 50
Table 10 - API Gateway average startup time 51
Table 11 - User BO average startup time 51
Table 12 - Saaty fundamental scale 66
Table 13 - Criteria comparison matrix 66
Table 14 - Criteria normalized comparison matrix and priority vector 66
Table 15 - Random consistency index 67
Table 16 - Consistency matrix comparison 67
Table 17 - AHP criteria weights 68
Table 18 - Projects rating by criteria 68
Table 19 - Project total scores 69
Table 20 - Load test results from API Gateway developed in Ballerina AOT (Configuration 1). 78
Table 21 - Load test results from API Gateway developed in Ballerina AOT (Configuration 2). 78
Table 22 - Load test results from API Gateway developed in Ballerina (Configuration 1) 79
Table 23 - Load test results from API Gateway developed in Ballerina (Configuration 2) 79
Table 24 - Load test results from API Gateway developed in Java AOT (Configuration 1) 80
Table 25 - Load test results from API Gateway developed in Java AOT (Configuration 2) 80
Table 26 - Load test results from API Gateway developed in Java (Configuration 1) 81
Table 27 - Load test results from API Gateway developed in Java (Configuration 2) 81

List of Code Snippets

Code Snippet 1 - Ballerina API Gateway Listener 34
Code Snippet 2 - Ballerina API Gateway Client 34
Code Snippet 3 - Ballerina API Gateway JWT validation 35
Code Snippet 4 - Ballerina User BO microservice Listener 36
Code Snippet 5 - Ballerina User BO microservice database connection 36
Code Snippet 6 - Ballerina User BO microservice Vehicle class 37
Code Snippet 7 - Ballerina User BO microservice VehicleOnCreation record 37
Code Snippet 8 - Ballerina User BO microservice getAllUserVehicles function 38
Code Snippet 9 - Ballerina User BO microservice login function 38
Code Snippet 10 - Spring API Gateway DNS resolver class 39
Code Snippet 11 - Ballerina application configuration inside PMS docker-compose 41
Code Snippet 12 - Resources allocation configuration 41
Code Snippet 13 - Hypothesis test structure 48

Abbreviations and Symbols

AHP	Analytic Hierarchy Process
API	Application Programming Interface
BO	Back Office
CE	Community Edition
CI	Consistency Index
CR	Consistency Ratio
CRUD	Create, Read, Update and Delete
DSL	Domain Specific Language
EE	Enterprise Edition
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
FO	Front Office
GQM	Goal Question Metric
ISEP	Instituto Superior de Engenharia do Porto
JDK	Java Development Kit
JPA	Java Persistence API
JVM	Java Virtual Machine
MEI	Master's Degree in Computer Engineering
MSA	Microservices Architecture Style
PMS	Park Management System
RFP	Request for Proposal
RI	Random consistency Index
SOA	Service Oriented Architecture

1 Introduction

This chapter was written for the Dissertation course in the second year of the Master in Informatics Engineering (MEI) program at Instituto Superior de Engenharia do Porto (ISEP). First, it briefly overviews the context, describes the problem, and outlines the main objectives and ethical considerations. Finally, it includes a description of the general structure of the document.

1.1 Context

Nowadays, there are many programming languages, some more popular than others. This is because some are quite versatile or have been developed for a specific purpose [1]. Over the years, RedMonk, an analysis company focused on programmers, has provided reports on the different aspects of the community. One of RedMonk's notable contributions is its biannual ranking of programming languages. These rankings are achieved based on their popularity on GitHub and StackOverflow [5]. Figure 1 demonstrates that JavaScript, Python, and Java are some of the most used languages in the development industry.

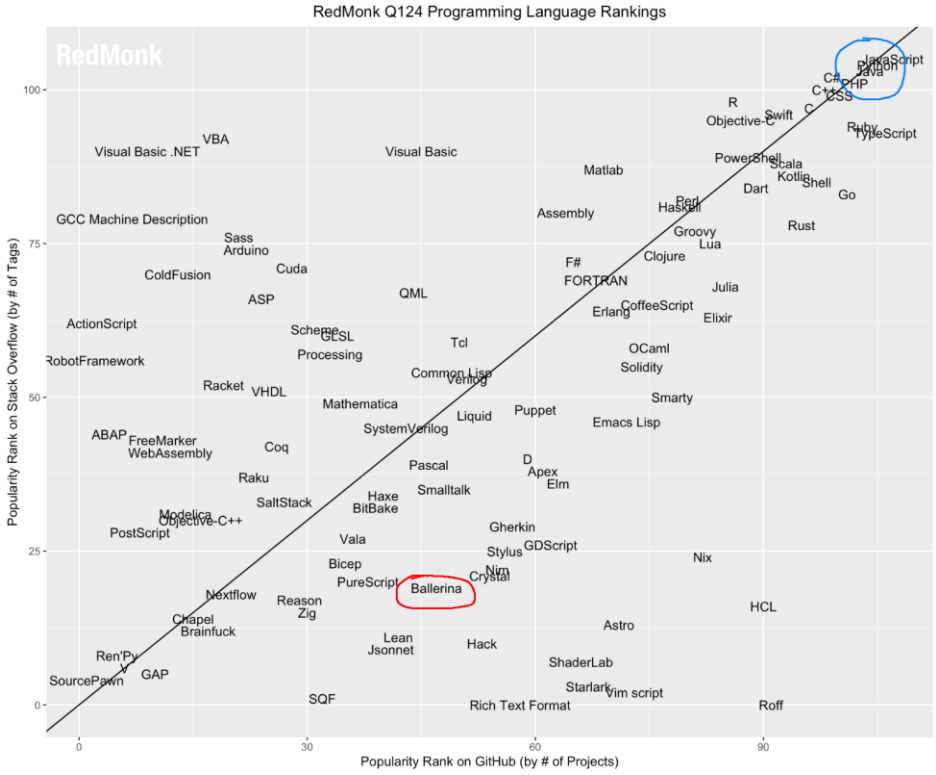


Figure 1 - RedMonk ranking January 2024
From [6]

By analysing Figure 1, it becomes clear that the Ballerina programming language, as it is relatively new to the market (with its first release in 2022), is not as widely used as other languages. Developed by WSO2, Ballerina is an innovative language designed for building cloud-era applications. As stated by the Ballerina Team [2], it includes unique features and design principles tailored for modern software development needs, such as a “network-aware type system, concurrency workers, being 'DevOps ready', and environment awareness” [7]. These characteristics position Ballerina as a particularly well-suited language for effectively “integrate, develop, deploy and manage distributed systems” [8].

1.2 Problem description

Choosing the correct programming language is crucial for the success of software projects, as it can impact development efficiency, system performance, and maintainability. Various studies have highlighted the importance of selecting appropriate technologies based on specific project requirements and constraints [9],[10]. Making informed decisions about technology, resources and design becomes complex without understanding the performance of each programming language and its purpose. This uncertainty makes it challenging for developers and organisations to adopt new languages confidently.

According to [11], successful technology selection involves considering factors such as performance, scalability, and maintainability. This study focuses on some of these aspects regarding Ballerina, particularly its performance and elasticity. In addition, a recent update of Ballerina (version 2201.7.0) introduces the official support for generating GraalVM native executables for its applications. According to the Ballerina Team, this update promises "performance improvements and reduced startup times" [4]. However, while GraalVM native executables are renowned for such advantages [12], no comprehensive studies have confirmed these promises for Ballerina.

By examining metrics such as response times, throughput, and startup times, this study aims to assess Ballerina's impact as a programming language when both developing and integrating applications. These insights will facilitate a comparative analysis with traditional languages, assisting in determining Ballerina's suitability for the development and integration of applications in distributed environments.

1.3 Objective and research methodology

The main objective of this study is to assess the impact of using Ballerina on performance and elasticity when developing distributed applications. Java, being one of the most established and widely used languages for distributed application development, serves as the baseline for comparison. To achieve this, components from a project based on a Microservices Architecture originally developed in Java were migrated to Ballerina to compare key performance and elasticity aspects between the two languages. Consequently, the research questions explored in this study are:

- What features and improvements introduced in the Ballerina language affect the performance of distributed applications?
- What features and improvements introduced in the Ballerina language affect the elasticity of distributed applications?

To address these research questions, a controlled experiment was conducted following a methodology inspired by the principles outlined by Briony June Oates in [13]. The research process involved the following tasks:

1. **Migration of some Java components to Ballerina (Pre-GraalVM native image support)**: Components from a Java open-source project were migrated to Ballerina (version 2201.6.0 or earlier, before GraalVM Native image support) to analyse the performance and elasticity outcomes.
2. **Upgrade Ballerina version with GraalVM native image support and Comparative Analysis**: The migrated components were then upgraded to Ballerina version 2201.7.0 or newer, utilizing GraalVM native images, and a comprehensive comparison was conducted against both the original Java implementation and the earlier Ballerina version.

Throughout the migration process, the primary objective was to maintain the original project's logic and dependencies as closely as possible to ensure that the comparison reflected typical development practices for both programming languages. Changes to the original Java project were intentionally minimized, and the aim was to preserve the existing architecture, logic, and dependencies, even if some could not be fully replicated in Ballerina. Only essential and blocking modifications were made to continue the experiment, ensuring a fair and practical comparison between Ballerina and Java in a development context.

1.4 Ethical considerations

Ethical considerations are essential in software engineering research as they directly influence the integrity and social impact of the work. This section addresses these considerations as they apply to the research and development of this work.

It is essential to highlight that this study was conducted in full compliance with the ethical standards and guidelines established by the Order of Engineers [14] and the Code of Ethics of the Institute of Electrical and Electronics Engineers [15]. The Declaration of Integrity included in this document is a requisite of the Code of Good Practices and Conduct of P. Porto [16], which was also wholly fulfilled.

In terms of permissions and approvals, as the project used in this study was developed in a previous course, it was necessary to seek permission from all group members and the responsible teacher before publishing and using it in the study, as shown in Appendix A. Ensuring all required acknowledgements and attributions to the original creators of the project. It's worth noting that all the software and information used was open-source and publicly available to everyone, fostering a sense of inclusivity and collaboration within the software engineering community.

1.5 Document structure

This document is structured into seven chapters: Introduction, Background, Literature review, Analysis and design, Implementation, Evaluation and Conclusion.

The current chapter, Introduction, aims to provide context, describe the problem, introduce the objectives and ethical considerations, and explain the document's structure.

The Background section overviews essential concepts in the Java Virtual Machine ecosystem, and an introduction to Ballerina and GraalVM technologies.

The Literature review is an essential part of this study. It summarizes all the research steps and analyses studies about Ballerina and its features regarding performance and elasticity, providing a comprehensive understanding of the subject matter.

The Analysis and design chapter introduces an overview of the project and explains the migration process used.

The Implementation chapter details the changes made to the migrated components, demonstrating the meticulousness and accuracy of this approach.

The Evaluation chapter presents the methodology used, the results obtained along with their analysis, and a summary of the findings.

The Conclusion chapter outlines the achievements and contributions, difficulties, threats to validity, future work, and final considerations.

Moreover, the References and all the appendixes (Appendix A, Appendix B, Appendix C, Appendix D and Appendix E) that support the study are presented at the end.

2 Background

The following chapter provides some essential knowledge about key concepts in the Java Virtual Machine ecosystem, the Ballerina programming language, and, finally, GraalVM.

2.1 Key concepts in the Java Virtual Machine ecosystem

Java is a popular programming language used to build different types of applications. One of its main advantages is that Java applications can run on any device with the installed Java Virtual Machine (JVM) [17]. The following topics explore essential concepts in the Java Virtual Machine ecosystem.

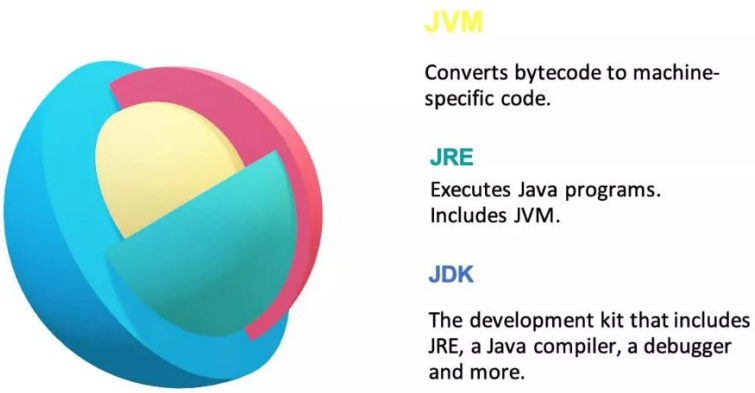


Figure 2 - Key concepts about the Java Virtual Machine ecosystem
From [18]

2.1.1 Java virtual machine

The Java virtual machine (JVM) makes Java and multi-language applications portable and executable. It achieves this by interpreting Java bytecode, managing memory through garbage collection, and enforcing security measures for safe execution on different devices [18].

2.1.2 Java development kit

The Java development kit (JDK) serves as a toolkit for Java developers. It includes a JRE (Java Runtime Environment) where the JVM is allocated, a compiler (Javac), an archiver (Jar), a documentation generator (Javadoc), and other utilities necessary to develop Java applications [18].

2.1.3 Java frameworks and libraries

Java frameworks and libraries provide pre-written code and patterns to solve common problems and speed up development. Frameworks like Spring, Hibernate, and Apache Struts enable the creation of robust and scalable applications, while libraries like JUnit, Log4J, and Apache Commons offer additional functionalities that can be reused [19]. These tools improve productivity, promote good programming practices, and make it easier to develop high-quality applications within the JVM ecosystem.

2.2 Ballerina

Ballerina, created by WSO2, is an open-source programming language designed for quickly developing and managing microservices, APIs, and cloud applications. It includes modern programming features expected from a recent language.

2.2.1 History

For over a decade, WSO2 has provided comprehensive, integrated solutions and Enterprise Service Bus (ESB) solutions for integrating Service Oriented Architecture (SOA) services [3]. WSO2 ESB leverages Apache Synapse as its mediation engine, using a Domain-Specific Language (DSL) based on XML for message handling and transformation logic. While SOAP requests form the backbone of messaging in SOA, XML manipulation relies on XPath. However, creating complex messages and transformation scenarios using a DSL can be difficult, often necessitating general programming languages like Java [3], [20].

This is where Ballerina steps in, addressing the need for a language that simplifies integration logic and overcomes the limitations of the Synapse DSL. The Chief Executive Officer and founder of WSO2 initiated the development of a new programming language called Ballerina, with its

initial release in 2022 [3]. The aim was to improve the efficiency of writing integration logic and move away from the limitations of the Synapse DSL. Ballerina was designed to "bridge the gap between integration and general programming languages" [3].

To point out essential milestones of the language, a timeline was created:

- **2017**: WSO2 publicly announce the Ballerina programming language.
- **2019**: Announcing Ballerina 1.0, showcasing its potential "to easily write software that just works" [21].
- **2020 and 2021**: WSO2 began releasing more versions of Ballerina, providing users with more tools, features and improvements.
- **2022**: Announcing the general availability of Ballerina 2201.0.0 (Swan Lake). With a focus on enterprise integration, this version was designed to create cloud-native applications with more features and platform tools that "handle network interactions, data, and concurrency straightforwardly and are easy to maintain" [22]. Later, other versions of the language appeared.
- **2023 and now**: Ballerina constantly received new versions. In version 2201.7.0, it introduced official support for generating GraalVM native executables, promising enhanced performance and reduced startup times. At the moment, Ballerina is in the 2201.9.0 version [23].

2.2.2 Key features

WSO2 designed Ballerina to surpass distributed application development challenges, providing a comprehensive language that integrates the best programming practices, network protocols, data formats, concurrency, and observability [8]. Some of Ballerina's key features that simplify the development of these applications are:

- **Built-in container support**: Ballerina makes writing business or integration logic easy and places it in containers with Docker's support. Instead of writing a separate Dockerfile, you can write containerisation instructions in the Ballerina settings. Kubernetes' artefacts can also be generated automatically with the Ballerina program. By compiling a Ballerina application, it automatically creates Docker and Kubernetes artefacts. This automation makes it easier to deploy any Ballerina program on Kubernetes or Docker [3].
- **DevOps support:** Ballerina offers an all-in-one solution with a comprehensive testing framework, a robust build tool, and an efficient packaging system. With its testing framework, developers can quickly write and execute test cases, ensuring the reliability and stability of their code before deployment. Besides that, with the Ballerina command-line interface (CLI), developers can create deployment artefacts and run automated tests before each deployment. When combined with other Ballerina build tools, it enables the automated deployment process by allowing the creation of deployment artefacts and checking out code from GitHub [3].

- **Security:** Ballerina provides native support for authentication and authorisation mechanisms, including basic web authorisation, JWT, and OAuth2 [3]. This language has encryption and built-in support for TLS and SSL, protecting sensitive data and allowing secure communication between applications. A "Taint checking" feature also helps identify vulnerabilities like SQL and HTML injections. Ballerina's security features show its commitment to providing a secure and reliable environment for microservices development [20].
- **Network interactions**: Ballerina's design optimises network interactions and API integrations. It comes with an HTTP module, which includes standard CRUD operations, enabling the creation of services and clients within Ballerina programs. Services serve as entry points for task resolution, while clients facilitate connections to external HTTP servers [20]. Moreover, Ballerina's integration with Ballerina Central extends its capabilities by providing connectors to diverse services such as Salesforce and Twilio [20]. By combining HTTP module features and external connectors, Ballerina becomes a powerful tool for developers to handle network interactions and integrate with various services easily.
- **Resiliency:** Ballerina helps make applications stronger by focusing on resiliency. It has circuit breaking, fail-over, load balancing, and retry mechanisms. Circuit breaking allows developers to create connectors with suspension policies. These policies stop the system from sending messages to unresponsive endpoints under certain suspension conditions. For instance, if too many requests fail within a certain period, circuit breaking can temporarily stop new requests to prevent service issues. Fail-over setups allow connections to switch smoothly to backup endpoints when there are failures, ensuring the service remains uninterrupted. Load balancing configurations use specific algorithms to distribute requests among designated endpoints. Using a Load balancer prevents backend services from becoming overloaded and optimises the use of resources. Ballerina also supports retry configurations for connections. Developers can define rules for retrying delivery if the initial request fails. These features help create more robust and reliable applications for different challenges and uncertainties [20].
- **Visual-oriented language:** The language uses a special graphical syntax to represent code as sequence diagrams, making it easier for developers to understand and communicate with each other.
- **Messaging protocol support:** Network protocols are essential for the functionality of cloud-native applications. Ballerina supports messaging formats like XML and JSON and protocols like HTTP and WebSocket. You can easily manipulate XML and JSON messages and send them through different protocols and connectors [3].
- **Concurrency system:** Ballerina offers advanced concurrency primitives and synchronisation mechanisms, allowing developers to write parallel and asynchronous code more intuitively. This feature is essential to create scalable applications that can efficiently handle multiple tasks or requests simultaneously. The language's model for managing threads, locks, and parallel execution paths can efficiently use resources and minimise the complexity usually involved in concurrent programming [20].

- **Observability:** Observability is how well we understand a system's internal states. We can achieve observability by monitoring, logging, and distributed tracing to see what's happening inside the system. In the case of Ballerina, these methods help external systems monitor metrics, analyse logs, and trace activity. Ballerina uses open standards to share observability info, making it compatible with different tools for collecting, analysing, and visualising data [20].
- **Syntax:** The purpose of programming languages is to give instructions to computers and help programmers understand them better. Ballerina syntax helps understand instructions since it focuses on providing expressive syntax for network-distributed systems [3].
- **Built-in libraries:** Ballerina's built-in libraries make it easier to develop applications. They provide ready-made functionality for everyday tasks like handling HTTP, working with databases, dealing with concurrency, ERP systems, and more. As mentioned, there is also Ballerina Central, which lets developers quickly find and download libraries from a centralised repository. With the libraries that Ballerina and the community provide, developers save time and effort in their application development process [3].

Ballerina uses the Java Virtual Machine to compile its applications, similar to what happens to Java applications. By running its applications on the JVM, Ballerina benefits from the JVM's mature ecosystem:

- **Java Integration**: Ballerina can directly call Java libraries and utilize Java classes within its applications, leveraging the extensive ecosystem of Java tools and frameworks.
- **IDE Support**: Developers can use popular Java IDEs like IntelliJ IDEA and Eclipse for Ballerina development, taking advantage of familiar tools for coding, building, and debugging.

These features ensure that developers can efficiently work with Ballerina while leveraging the robust ecosystem and performance optimizations available in Java [20].

2.2.3 Use cases

Since its launch, Ballerina has been applied in various scenarios. One such case study is the digital transformation of Fat Tuesday.

In 1984, situated in the vibrant surroundings of Bourbon Street in New Orleans, Fat Tuesday inaugurated its operations as an establishment specializing in superior frozen beverages. Over time, it expanded significantly, establishing a presence in renowned tourist destinations, vibrant entertainment hubs, and outdoor malls across the globe. However, Fat Tuesday's legacy data integration system posed significant challenges in operational efficiency and decision-making [24].

Fat Tuesday's data collection from its point-of-sale system used a file-based approach, which was time-consuming and prone to errors. The process involved manual intervention, increasing

the risk of data inaccuracies and obstructive real-time updates. This outdated method affected the authenticity and precision of information, leading to decisions based on old data. The need for manual handling and data processing, combined with the system's inability to capture subsequent real-time changes, resulted in an inefficient and error-prone system [24].

Fat Tuesday transitioned from its file-centric process to an API-driven approach to address these challenges, leveraging Ballerina to enhance data integration speed, reduce errors, and improve real-time data processing [24].

- **API-Driven Data Access:** Data is now directly accessed using REST APIs from the point-of-sale service, eliminating the need for centralized file storage and manual data handling. This real-time data processing ensures up-to-date information and efficient error handling.
- **Ballerina Integration:** Ballerina played a central role in this transformation. Its support for network protocols, data formats, and connectors simplified system integrations and eliminated the need for external tools. Ballerina's visual tooling, service design, and data mapping capabilities streamlined the development of the integration solution.
- **Real-Time Data Processing:** The Ballerina integration periodically fetches data via REST APIs, processes it, and stores the analysis results in databases. This automation removed the need for manual intervention and significantly improved data accuracy and timeliness.

Figure 3 compares the traditional legacy with the new Ballerina-implemented solution, highlighting the transition from manual operations to automation.

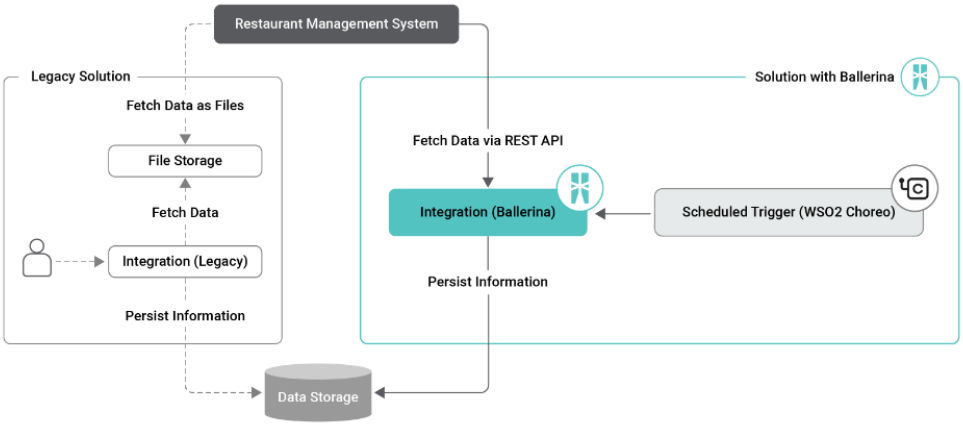


Figure 3 - Legacy solution vs Ballerina solution
From [24]

Benefits of Fat Tuesday by using Ballerina solution [24]:

- **Modern Cloud-Ready Integration**: Ballerina's lightweight, cloud-native features enabled seamless data automation, reducing manual errors and enhancing efficiency.

- **Enhanced System Interoperability**: Ballerina simplifies network interactions and API integrations with its diverse connectors, facilitating easy integration with various services, databases, and platforms. This improved the overall interoperability of Fat Tuesday's systems.
- **Improved Error Handling**: Ballerina's robust error management features, including error types, propagation, and structured error handling, ensured that services handled errors gracefully, enhancing system resiliency and user experience.
- **Robust Monitoring and Advanced Analytics**: Ballerina's inherent observability provided comprehensive insights into program execution and efficiency. This enhanced analytics capability gave Fat Tuesday actionable insights and proactive issue resolution.
- **Enhanced Development Productivity**: Ballerina's concise and efficient syntax and integrated visual tooling simplified the development process. This reduced the learning curve and supported long-term code readability, fostering innovation.
- **Operational Efficiency and Strategic Growth**: The API-centric model improved Fat Tuesday's operational capabilities, ensuring its store network operated efficiently and flexibly. This transformation supported Fat Tuesday's dynamic business needs and strategic growth.

In essence, Ballerina provided an efficient and intuitive integration for Fat Tuesday's digital transformation, improving data precision and reducing operational overhead. It enables the engineering team to focus on innovation in the future and promises further improvements and expansion for Fat Tuesday [24].

2.3 GraalVM

GraalVM, created by Oracle Labs, is a versatile Java Development Kit (JDK) that provides better performance and flexibility for JVM-based languages.

2.3.1 History

GraalVM originated from the Maxine Virtual Machine project at Sun Microsystems Laboratories, now part of Oracle Labs. The initial purpose of it was to create a JVM in Java to overcome the challenges of C++ development. The project later focused on building a high-performance compiler, resulting in the GraalVM compiler [25]. GraalVM is designed for executing programs within the JVM, using a Just-In-Time (JIT) compiler or the Ahead-Of-Time (AOT) compilation to create native images [26]. Both types of compilation have benefits and drawbacks, and when using GraalVM, developers may need to consider what kind of compilation they want to use. illustrates the differences between both compilations.

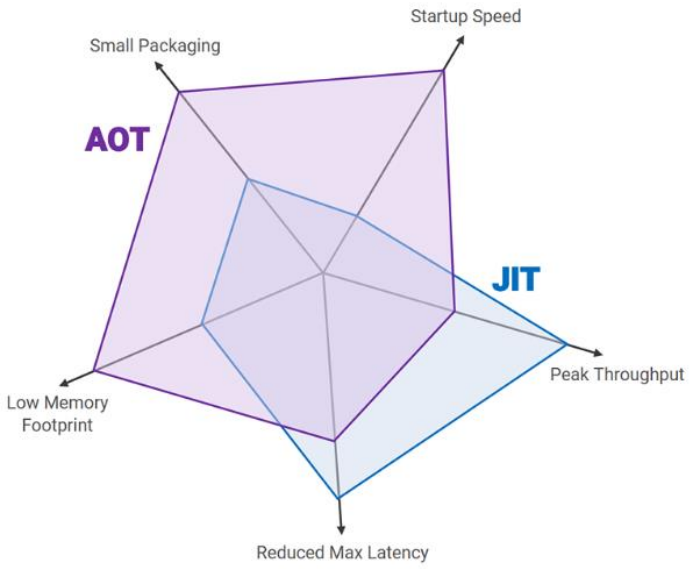


Figure 4 - Differences between both approaches
From [27].

By default, GraalVM is configured to use its JIT compilation as the top-tier compiler [28]. If the developer wants to use the AOT compilation, further steps are required to generate native images.

To understand what each compilation does, an explanation is provided below [29]:

- **JIT**: JIT compilation compiles code at runtime, translating bytecode into machine code as the program executes. This approach allows the compiler to optimize the code based on the actual execution profile, leading to potentially significant performance improvements over time. The main advantages of JIT compilation include adaptive optimization and dynamic profiling, which can result in highly optimized code execution. However, the initial startup time may be slower due to the overhead of dynamic compilation.
- **AOT**: AOT compilation, on the other hand, compiles code into native machine code before execution, producing a standalone executable. This method eliminates the need for a runtime compiler, leading to faster startup times and reduced memory consumption since the application does not require the JVM to execute. AOT compilation can benefit environments where startup time and resource usage are critical. However, it may lack some of the runtime optimizations that JIT compilation can achieve, as it does not have access to execution profiles during the compilation process.

2.3.2 Key features

This section briefly outlines some of the main features of GraalVM [30].

- **Low resource usage**: Native executables (AOT compilation) use only a fraction of the memory and CPU resources required by a JVM, which improves utilization and reduces costs.
- **Improved security**: Native executables contain only the classes, methods, and fields that your application needs, which reduces attack surface area.
- **Fast startup**: Native executables start almost instantly, requiring no warmup to run at peak performance.
- **Compact packaging**: Native executables are small and offer a range of linking options that make them easy to deploy in minimal container images.
- **Supported by frameworks**: Popular frameworks such as Spring Boot, Micronaut, Helidon, and Quarkus provide first-class support for GraalVM.
- **Supported by leading cloud platforms**: SDKs from leading cloud platforms such as AWS, Microsoft Azure, GCP, and Oracle Cloud Infrastructure integrate and support GraalVM.

GraalVM stands out as a significant advancement in the JVM ecosystem. It offers enhanced performance, flexibility, and the ability to create more efficient and diverse applications by integrating multiple programming languages within a single framework [31], [32].

3 Literature review

This literature review examines existing studies that explore the features and performance improvements of distributed applications using the Ballerina programming language. The objective of this chapter is to understand how Ballerina's capabilities impact these aspects. By analysing relevant studies, this chapter aims to provide insights and theories that can help predict the outcomes of the experimental work conducted in this study.

The review approach involves posing critical questions that guide the exploration, leading to a comprehensive understanding of the topic and enabling the formulation of informed opinions. The chapter outlines the data sources, including terms, keywords, and eligibility criteria used in the research process. Finally, it details the data collection process and presents answers to the research questions based on the retrieved data.

3.1 Research questions

Considering the problem description and the objective, as mentioned previously in section 1.3, the research questions for this study are:

- **RQ1**: What features and improvements introduced in the Ballerina language affect the performance of distributed applications?
- **RQ2**: What features and improvements introduced in the Ballerina language affect the elasticity of distributed applications?

3.2 Data sources

Identifying the data sources that will be used is a crucial step in finding relevant studies on the topic. As such, Table 1 presents the data sources used.

Table 1 - Data Sources

Identifier	Database	URL
DS1	Google Scholar	https://scholar.google.com/
DS2	ACM Digital Library	https://dl.acm.org/
DS3	B-ON	https://www.b-on.pt/
DS4	IEEE Xplore	https://ieeexplore.ieee.org/Xplore/home.jsp

These data sources contain many publications, such as online books, conferences, newspapers, articles, and academic studies. These data sources offer a wide range of publications and set themselves apart by including a considerable number of peer-reviewed articles carefully selected for their exceptional relevance in information technology.

3.3 Search terms

In this section, the most relevant keywords for the problem described are Ballerina programming language, GraalVM, HTTP/2, Performance, Elasticity, and Optimization.

A query was created using these keywords from the data sources indicated above. The query is presented in Figure 5.

```
(
   "Ballerina programming" OR "Ballerina programming language"
   (
     (
       "GraalVM" OR "Graal"
     ) AND
     "Benchmark AND
     "JDK"
   ) OR
   (
     (
       "HTTP/2" OR "HTTP 2.0"
     ) AND
     "Microservices"
   )
 ) AND
 (
   "Performance" OR
   "Elasticity" OR
   "Start-up time" OR "Start up time" OR "Startup time" OR
   "Optimization" OR
   "JVM"
 )
```

Figure 5 - Search query

By combining the terms with the **OR** and **AND** operators, the search only returns publications on performance or elasticity related to Ballerina, GraalVM or HTTP/2. An analysis will then be carried out on these publications to obtain information to answer the research questions.

3.4 Eligibility criteria

Regarding the inclusion criteria for the articles, they are:

- **IC1**. The source explores the use of Ballerina, GraalVM or HTTP/2.
- **IC2**. The source needs to address objective metrics (performance or elasticity).

The exclusion criteria, on the other hand, are:

- **EC1**. The source does not provide practical scenarios/applications of the technology.
- **EC2**. The source is not published in English.

3.5 Data collection process

The author followed the PRISMA systematic review process [33] to obtain the relevant articles. This process consists of three steps:

- **Identification**: Articles that follow the eligibility criteria.
- **Screening**: Divided into two phases. In the first phase, all retrieved articles are categorised as "Relevant" or "Irrelevant." In the second phase, all the remaining articles are carefully analysed to see if they are relevant to our research questions and discarded if not.
- **Inclusion**: Total relevant articles where information should be extracted to answer the research questions.

These guidelines allow readers and researchers to evaluate the quality of their research and draw better conclusions. Figure 6 represents the PRISMA systematic for all the articles retrieved with previously defined search terms.

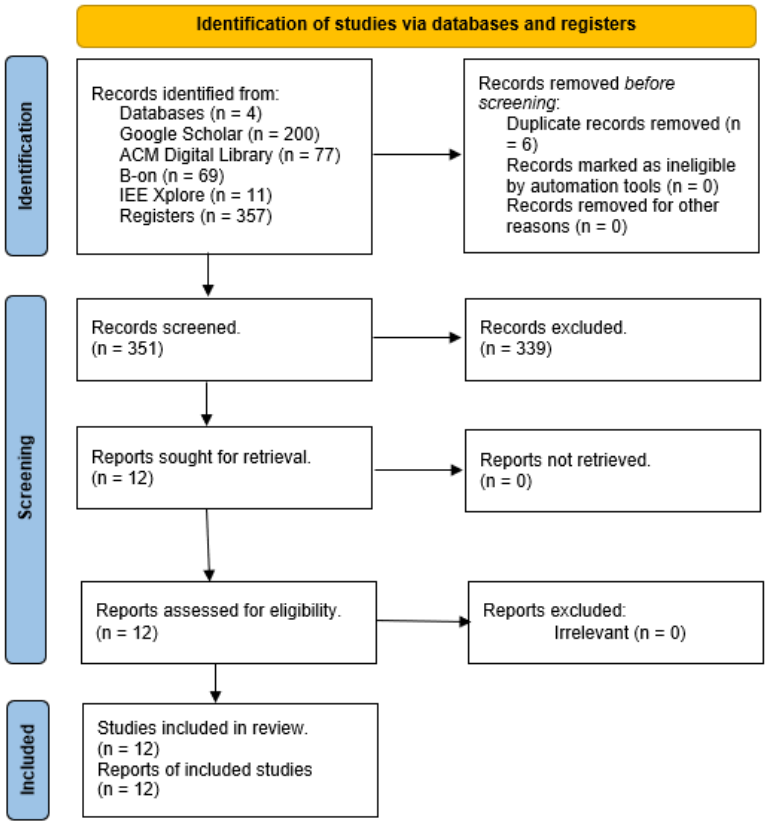


Figure 6 - PRISMA systematic

3.6 Results

Table 2 contains all analysed studies to answer the research questions.

Table 2 - Search results

Research Questions	Studies
RQ1	[25][26][34][35][36][37]
RQ2	[38][39]

3.7 Discussion

As Ballerina is a new and emerging language, there are very few studies comparing it with other languages. As it becomes more popular and integrates with tools like GraalVM, we can expect to see more studies appearing, especially in projects that use a microservice architecture or distributed applications.

3.7.1 RQ1: What features and improvements introduced in the Ballerina language affect the performance of distributed applications?

Ballerina offers several features that can significantly enhance performance. As mentioned in the Background section, the concurrency system with its asynchronous programming model is well-equipped to handle multiple simultaneous operations efficiently, consequently optimising Ballerina's response times and overall performance [20]. Furthermore, Ballerina supports the HTTP/2 protocol, which can further boost performance by reducing latency through features such as multiplexing, header compression, and server push, all of which surpass the capabilities of HTTP/1.1 [40].

Another feature of Ballerina is GraalVM, a technology that is often discussed for its improvements in performance and other aspects in Java, Javascript and other JVM-based languages [25], [26], [35], [36]. To confirm these claims and understand how GraalVM performs, it is crucial to rely on credible comparative studies about GraalVM and other Java Development Kits (JDKs). These studies offer insights into the efficiency and potential advantages of GraalVM, providing a solid foundation for the decision-making process.

GraalVM comprises two editions: the Community Edition (CE) and the Enterprise Edition (EE). The CE version is freely available and delivers the core functionalities of GraalVM, while the EE edition is a paid version offering additional features, enterprise-level support, and enhancements. Comparative studies, which comprehensively evaluate the performance of both editions relative to other Java Development Kits and frameworks, are the key to understanding GraalVM's potential benefits. By exploring these studies, the claims made about GraalVM's performance can be confirmed and thoroughly comprehend its potential benefits.

For instance, a study conducted by M. Šipek, D. Muharemagić, B. Mihaljević, and A. Radovan aimed to compare the performance of a Java application implemented in different frameworks and JDKs [25]. Their approach involved using the *Renaissance* benchmark suite, which includes various modern workloads and programming paradigms for the JVM. These workloads are designed to test multiple JVM components such as compilers, garbage collectors, profilers, and analysers [41]. The tests compared GraalVM CE with OpenJDK 8 and 13, and although the average response times were similar overall, there was a notable difference in the case of the *movie-lens*. GraalVM proved superior, achieving a result of about 31% better than OpenJDK 13 [25], as seen in Figure 7.

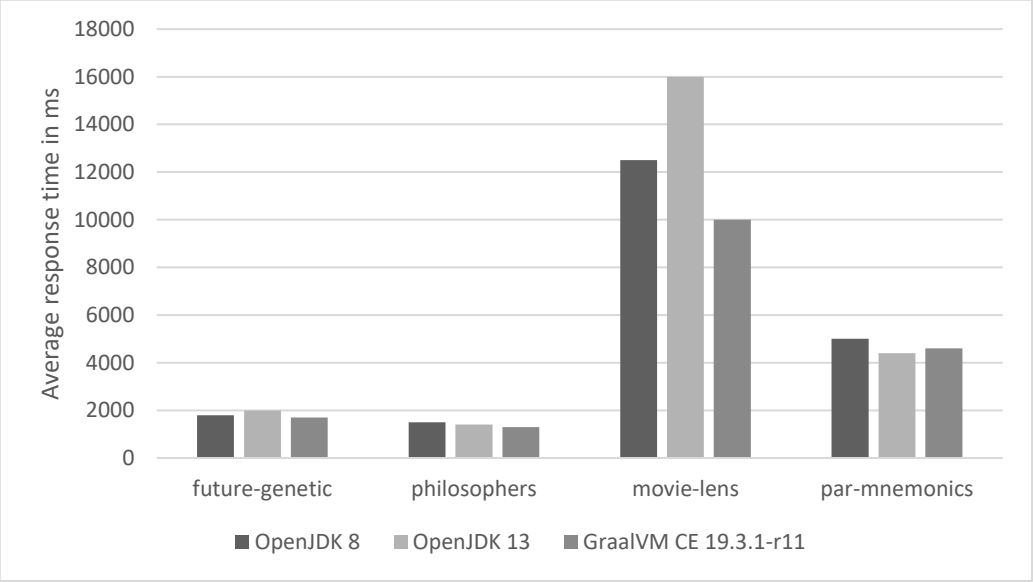


Figure 7 - Benchmark test results for selected Renaissance benchmarks
From [25].

In addition to comparing the JDKs, response times were also tested with various frameworks and JDKs. The results from Quarkus and GraalVM have demonstrated a significant performance advantage over Spring Boot, surpassing it by an impressive 81.74% [25]. This performance advantage was also observed when comparing Quarkus with OpenJDK 13, as shown in Figure 8.

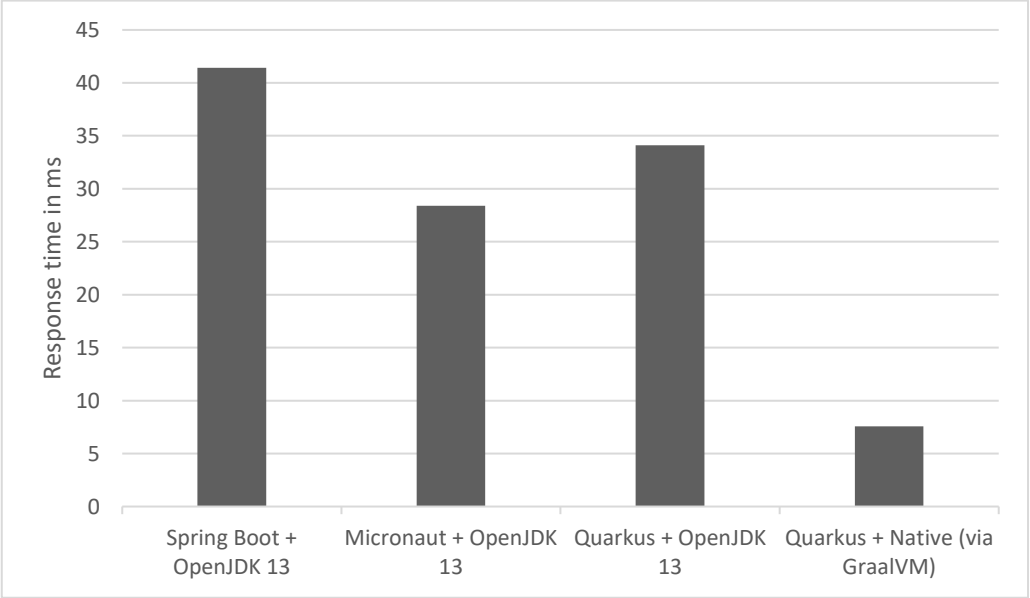


Figure 8 - Response time in milliseconds obtained from a custom benchmark
From [25].

At the end of those tests, the study's authors said that GraalVM's performance without deployment surpasses its competitors without doubt [25].

In another study, M. Šipek, B. Mihaljević, and A. Radovan used a distinct benchmark called *DaCapo* [26]. This particular benchmark evaluates the performance of JVMs and compilers by compiling Java programs that represent applications and workloads [42].

In this study, the comparisons were explicitly made between GraalVM EE, GraalVM CE, OpenJDK 10, and OpenJDK 11. The results underscore that GraalVM Enterprise Edition (EE) outperforms the others in four of six benchmarks, falling short only in *xalan* benchmark, where OpenJDK 10 takes a slight lead. Moreover, GraalVM excels in the *h2* benchmark by displaying a remarkable 19.8% improvement in performance. GraalVM Community Edition (CE) closely aligns with OpenJDK 11 regarding performance, except in the *h2* benchmark, where it considerably fell behind OpenJDK 11 and GraalVM EE [26]. The results mentioned can be seen in Figure 9.

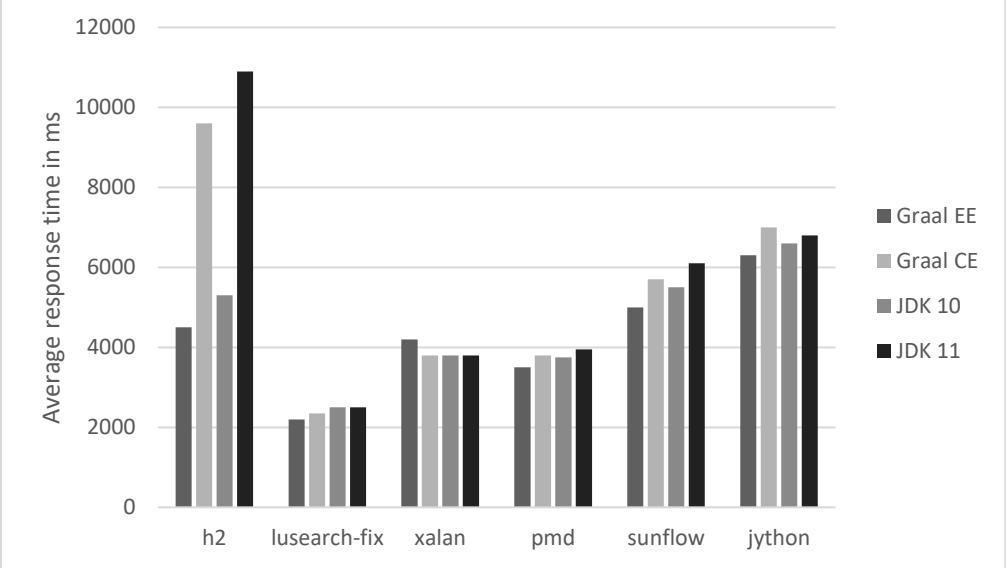


Figure 9 - Average test results for selected DaCapo benchmarks
From [26].

This study provides evidence of GraalVM's exceptional polyglot capabilities across various programming languages and confirms that both versions of GraalVM perform better than other commonly used JDKs most of the times [26].

In another study, Frederic Fong and Mustafa Read performed tests on two different machines comparing GraalVM (EE and CE) with other JDKs (8 and 11) [34]. The findings reveal that GraalVM EE version 11 not only outperformed in most projects on the first test machine, but also seven out of eleven projects on the second test machine for both JDK 8 and 11, solidifying its superiority [34].

In addition to these articles, three more studies prove that GraalVM improves performance [35], [36], [37]. According to the results and conclusions of these studies, GraalVM is a tool that can significantly improve the performance of applications, with its added value and the other features analysed previously, such as HTTP/2 and concurrency.

3.7.2 RQ2: What features and improvements introduced in the Ballerina language affect the elasticity of distributed applications?

Elasticity in microservices refers to the system's ability to adapt to changes in workload by dynamically adding or removing resources as needed [43]. This capacity to scale resources up or down based on demand ensures optimal performance and efficiency in a microservices architecture. It empowers the system to manage different levels of traffic and workload without encountering substantial downtime or performance degradation [39].

Ballerina's Resilience feature contributes to elasticity. It offers built-in resilience features, such as circuit breakers, retries, and timeouts, to maintain service stability under changing load conditions. This feature allows applications to recover from failures, quickly ensuring uninterrupted availability and responsiveness, contributing to its elasticity [20].

Another critical aspect that influences elasticity is the startup time of services. Faster startup times enable a system to quickly adapt to sudden changes in demand [39]. Ballerina's architecture and GraalVM's Ahead-of-Time (AOT) compilation can greatly enhance startup times. By converting bytecode into native machine code before execution, GraalVM's Ahead-of-Time (AOT) compilation delivers a significantly faster startup time and reduced memory usage [30]. This feature makes it an ideal solution for microservice environments prioritising rapid scaling and efficient resource utilisation.

A study by members of Oracle Labs in the USA published an article in 2022 that measured and compared startup times and memory footprint of "Hello World" projects and Java microservices using various frameworks [38]. After comparing GraalVM native image with JDK 8 and 12 from Java HotSpot VM, the study concludes that GraalVM, particularly its Enterprise Edition (EE), delivers substantial improvements in startup times compared to the other JDKs [38].

This reduction in startup times directly enhances the elasticity of distributed applications, enabling faster and more efficient scaling [39]. In summary, the features introduced in Ballerina with GraalVM's AOT compilation provide significant advantages regarding startup times and elasticity. They enable applications to respond to demand changes quickly, ensuring high availability and efficient resource utilisation.

3.8 Conclusion

This literature review examines the current features of Ballerina and focuses on its impact on the performance and elasticity of distributed applications. Although Ballerina is an emerging language with limited comparative studies, it shows promise with its compatibility with GraalVM and its ability to take advantage of GraalVM optimisations.

GraalVM has been the subject of extensive research, with numerous articles underscoring its performance enhancements in the JVM ecosystem. Comparative studies consistently affirm that GraalVM, particularly its Enterprise Edition, surpasses standard JDKs in various

benchmarks. This translates to superior application performance and optimized resource utilization, especially if used along with other Ballerina features like HTTP/2 and concurrency. These findings substantiate GraalVM's potential to elevate the performance of distributed applications.

In addition to performance, GraalVM's Ahead-of-Time (AOT) compilation demonstrates its efficacy in reducing startup times, a critical factor in achieving high elasticity in microservices environments. This, coupled with Ballerina's robust resilience features, underscores the potential benefits of adopting Ballerina and GraalVM in building efficient, scalable, and resilient microservices.

In conclusion, while more research is needed to fully understand the capabilities and limitations of Ballerina, the existing literature on GraalVM and other features of Ballerina provides a strong foundation for leveraging these technologies to improve the performance and elasticity of distributed applications. As Ballerina integrates with tools like GraalVM, future studies will likely provide more insights into the language and its role in the cloud development community.

4 Analysis and design

This chapter focuses on analysing and designing the project used for the experiment. It deals with a project that uses a microservices architecture, implemented in Java for the backend and React for the frontend. As mentioned in section 1.3, some project components were migrated to the Ballerina programming language to assess their impact on performance and elasticity. This chapter offers insights into the initial project design, architecture, and the migration process to Ballerina.

To obtain information about the steps that led the author to choose the Park Management System (PMS) project, Appendix B describes how and what process was used to evaluate projects against specific defined criteria.

4.1 Project to migrate

The characteristics of the Park Management System (PMS) project are the following:

- Recently development changes, with the last commit made in January 2024.
- Three backend microservices developed in Java.
- An API Gateway which secures and redirects all the requests made also in Java.
- A frontend application using React and four console applications representing the simulation of barriers and displays.
- It has an extensive documentation of the project, including architecture, patterns used, domain model, implementation model and much more.
- It is an open-source project and is under the MIT license.

4.1.1 Business context

The Parking Management System is a solution designed to improve the customer experience in parking lots. It aims to revolutionize how customers interact with parking services, namely

those provided by the fictitious "Park20", a prominent Portuguese company in the parking sector since 2015.

Park20 is working on making the customer experience better at their parking facilities. They want to use a new solution to improve how things work and make it easier for people to park their vehicles. Park20 has started a Request for Proposals (RFP) to achieve this vision of finding a suitable information system to manage different park operations effectively.

Park20's next-generation parking facilities aim to provide a better experience for registered users. Instead of traditional parking arrangements, the focus is on understanding and meeting the needs of individual customers. By promoting user registration, Park20 wants to make it easier for customers to enter and exit the park without dealing with tickets and payments. One of the critical features of the Park Management System is the introduction of PARKY "coins" a unique reward system designed to incentivize customer loyalty. Registered customers earn PARKY coins with each parking visit, anniversary, or friend invitation, which can be used to offset future parking expenses. This reward system promotes customer loyalty and enhances the overall parking experience by adding value. Receiving personal messages makes the customer feel welcomed and helps to improve their satisfaction, so every time a customer enters or leaves the park, the company wants a custom message to be displayed for them.

4.1.2 Architecture

As the Park Management System was a prototype and mentioned by the development team, only some of the features and components designed were implemented. A current implementation model was used, it includes all the developed components and can be found inside the PMS wiki repository [44]. Figure 10 presents the current implementation model of the Park Management System.

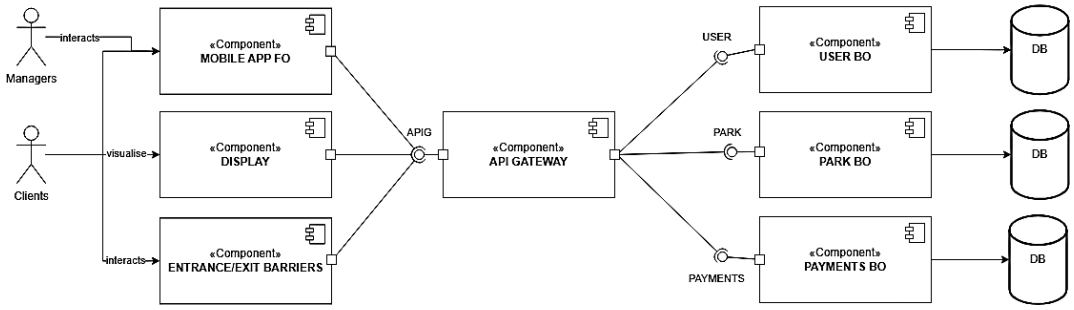


Figure 10 - Current Implementation model
From [44].

Table 3 includes a short description of the components presented in Figure 10, which helps to understand their purpose within the PMS.

Table 3 - PMS components description

Component name	Description
USER BO	Java application that manages all the user's personal data. It provides an HTTP resource API for the frontend applications and has asynchronous communication with the others backend components.
PARK BO	Java application that manages all the parks, barriers and displays information. It provides an HTTP resource API for the frontend applications and has asynchronous communication with the others backend components.
PAYMENTS BO	Java application that provides information and calculates all the payments based on the park price table and the time that the customer was inside the park. It provides an HTTP resource API for the frontend applications and has asynchronous communication with the others backend components.
API GATEWAY	Java application that receives all the frontend application requests, redirects them into the respective BO components and validates the authorization header. It provides an HTTP resource API for the frontend applications, uses Spring Cloud and Netflix Eureka for service discovery.
MOBILE APP FO	React application used by all the system users. It provides client information such registered cars, personal details, virtual park coins (PARKY coins) and other information's. Managers receive important information about either the park or the customers they need to manage, depending on their role. Communicates with the backend applications through the API Gateway.
DISPLAY	Console application that simulates a display for cars entering or leaving the park, providing important information such as the amount the client will be credited upon leaving the park. Communicates with the backend applications through the API Gateway.
ENTRANCE/EXIT BARRIERS	Console application that simulates a barrier for cars that are approaching the barrier, it reads the license plate of the car. Communicates with the backend applications through the API Gateway.

A deployment diagram was also created to illustrate the Park Management System deployment architecture. This diagram offers a detailed view of the current system's physical deployment, highlighting which components are deployed into Docker. Figure 11 represents the deployment diagram from the system.

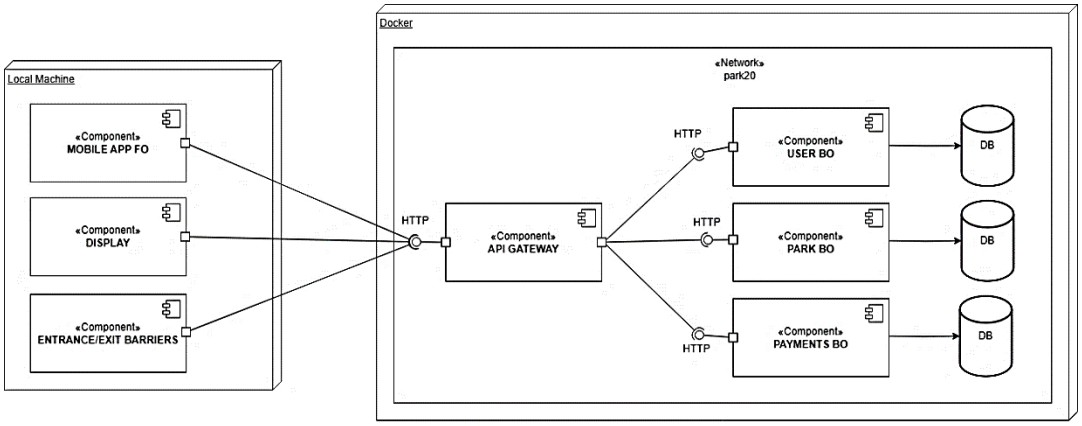


Figure 11 - Deployment diagram from PMS

4.1.3 General workflow of the system

The project wiki repository [44] also has a general workflow that shows the main steps that the system follows when a user interacts with it, which can be seen in Appendix C.

The overall workflow includes several phases, such as the initial phase in which the user searches for nearby parks, the park entry phase, the time spent inside the park, the user's exit from the park, and, finally, the post-park phase. Each phase is vital to the overall user experience and aligns with the Park20 company's vision.

4.1.4 Domain model

Regarding the project's domain model, as seen in the implementation model, the system consists of three main microservices: **Users**, **Parks**, and **Payments**. These microservices correspond to the three aggregate roots designed in the domain model. Each of these aggregates plays an essential role in guaranteeing the efficient management and operation of parks within the system.

1. **Users:** The user's aggregate root encapsulates all the essential information about the users registered in the system. This aggregate includes park clients, managers, and staff members. Each user has its identifier, and the user can have various roles and permissions within the system. Depending on which role you have, different interfaces will appear in the mobile app. For instance, a customer manager has access to other interfaces than those of regular customers.

2. **Parks:** Parks represent the physical spaces managed within the system. Each park is uniquely identified and contains relevant details such as location, operating hours, capacity limits, price table, etc. Park managers are responsible for overseeing the management of parks. They ensure that the parks run smoothly, are well-maintained, and follow regulations.

3. **Payments:** The Payments aggregate root handles financial transactions within the system. It calculates the fee for each visit to the park, including payments with discounts such as PARKY coins. Payment details are securely processed and recorded, ensuring transparency and accountability in all financial transactions.

Each aggregate root can be seen in Appendix D, and the whole domain model is published in the PMS wiki repository [44] along with other documentation files that can be useful for understanding more about the project.

4.2 Migration process

The migration process is essential for achieving the initial objective defined. It covers two main topics: the components migrated, the reasons behind it, and the migration strategy used.

4.2.1 Selected components

The selection process focused on components based on complexity, business importance, and potential impact on the system since the migration process aims to address Ballerina's effects on certain aspects compared to Java.

The decision was to exclude the frontend component from the migration process because Ballerina focuses more on backend development, which involves handling server-side logic and data processing. When analysing the implementation model, it became clear that the API Gateway is essential in the system's architecture. It acts as a central point for communication between different components and guarantees the security of incoming requests.

In addition to the API Gateway, another component was chosen for the migration. This component provides an analysis of the communication of Ballerina applications and the connection to a database. Despite the option to migrate any backend service, the User microservice (USER BO) was chosen due to its logic, which makes it an ideal candidate for the objective defined.

4.2.2 Migration strategy

A strategy was defined to ensure a smooth transition of technologies in a migration process. The API Gateway was the first component to be addressed due to its importance as a central entry point and security layer. A modular approach was used, migrating components by endpoint and their associated logic, such as the transition from domain Java classes to Ballerina classes and registries. This gradual approach minimised disruption and future risks, avoiding many errors that could arise during the component testing phase at the end of development.

5 Implementation

This chapter describes the experiment's implementation process using the Ballerina programming language. It represents the Ballerina implementation of the API Gateway and the User BO applications, following the strategy described previously in 4.2.2.

5.1 API Gateway migration

The API Gateway is an entry point for all client requests, routing them to the appropriate microservice. In the original implementation, the Spring Cloud Gateway handled requests and their routing. In Ballerina, there is a library called **ballerina/http** [45] designed for handling HTTP requests and responses. This module provides two network connectors, *Listeners* and *Clients*, allowing developers to create APIs and Web services.

A Listener is responsible for handling incoming HTTP requests on a specified port. It acts as the server-side component that processes requests from *Clients*. As seen in Code Snippet 1, the port specified was 9090, and an annotation was added to specify the version of HTTP requests that are allowed, as well as the keystore containing the certificate and private key for sending requests with HTTPS.

```
listener http:Listener apiListener = new(9090, {
    httpVersion: "2.0",
    secureSocket: {
        key: {
            path: "localhost.p12",
            password: "localhost"
        }
    }
});

isolated service / on apiListener {

    function init() {
        // Initialization logic
        log:printInfo("Service is ready to handle requests.");
    }

    ...
 resource isolated function post users/[string... path](http:Caller     caller,
http:Request req) returns error? {
     …
    }

}
```

Code Snippet 1 - Ballerina API Gateway Listener

As mentioned before, a *Client* is used to send HTTP requests to other services. It acts as the client-side component that initiates requests. In this case, Code Snippet 2 shows a *Client* called **userServiceClient** that is created to route the request received to the User BO microservice.

```
resource isolated function post users/[string... path](http:Caller     caller,
http:Request req) returns error? {
    …

    // Forward request to user microservice
    http:Client userServiceClient = check new ("https://" + mcsUsers + ":" +
mcsUsersPort, secureSocket = {enable: false});
    anydata response = check userServiceClient->post(fullPath, req);
    return caller->respond(response);
}
```

Code Snippet 2 - Ballerina API Gateway Client

The validation of the JWT tokens used in the system was also a feature of the original API Gateway. The **ballerina/jwt** [46] module was used to reproduce that in Ballerina. This module provides a framework for authentication/authorization and generation/validation of JWTs, ensuring that only authorized operations can be performed. Inside every resource of the API Gateway, a function named **validateJWT** is called. Code Snippet 3 represents the function that contains all the steps necessary to validate the JWT token used in requests, starting by verifying if the route requires authentication, followed by validating the token with its signature secret.

```
isolated function validateJWT(http:Request req, string path) returns boolean {
    // Check if path requires authentication
    if isUnauthenticatedRoute(path, req.method) {
        log:printInfo("No authentication required for path: " + path);
        return true;
    }

    // Extract the Authorization
    var authResult = req.get("Authorization");
    if authResult is string {
        if authResult.startsWith("Bearer ") {
            string token = authResult.substring("Bearer ".length());
            jwt:ValidatorConfig validatorConfig = getJwtValidatorConfig();

            // Validate the JWT
          jwt:Payload|error validationResult = jwt:validate(token, validatorConfig);

            if validationResult is jwt:Payload {
                log:printInfo("JWT validation succeeded");

                userId = validationResult["sub"].toString();
                role = validationResult["role"].toString();

                return true; // Validation successful
            } else {
               log:printError("JWT validation failed", 'error = validationResult);
            }
        } else {
            log:printError("Authorization  does not contain Bearer token");
        }
    } else {
        log:printError("Authorization  not found");
    }
    return false; // Default to validation failed
}
```

Code Snippet 3 - Ballerina API Gateway JWT validation

The library **ballerina/log** [47] was also used to trace and gather information while the application ran. This library is vital for monitoring and debugging services since logs were used in incoming requests, routing decisions, JWT token validation, and errors. Some examples of logs can also be seen in Code Snippet 3.

5.2 User BO microservice migration

This section describes the migration of the User BO microservice from Java to Ballerina. This service is mainly divided into two files: **users_service.bal** and **main.bal**.

The **users_service.bal** file contains the service responsible for listening to incoming requests and providing the appropriate responses. This file defines the HTTP service endpoints and handles the interaction with *Clients*. Similar to what was done in the API Gateway migration, the library **ballerina/http** [45] is used to create a *Listener* that handles requests from *Clients*. The Code Snippet 4 demonstrates the creation of the HTTP *Listener* in Ballerina.

```
01 import ballerina/http;
02 import ballerina/log;
03 import ballerina/lang.'int as langint;
04
05 listener http:Listener apiListener = new(9092, {
06     httpVersion: "2.0",
07     secureSocket: {
08         key: {
09             path: "localhost.p12",
10             password: "localhost"
11         }
12     }
13 });
14
15 isolated service /users on apiListener{
16     function init() {
17         // Initialization logic
18         log:printInfo("User service is ready to handle requests.");
19     }
20
21     resource isolated function post createUser(http:Caller caller, http:Request
request) returns error?{
22         http:Response response = new;
23         …
24     }
25  …
26 };
```

Code Snippet 4 - Ballerina User BO microservice Listener

The **main.bal** file contains all the Ballerina classes, records, and functions related to the user business logic. This includes definitions for the domain objects and the implementation of core functionalities such as fetching user data, creating new users, and others.

Firstly, all the necessary configurations were defined, including the database connection. A library called **ballerinax/postgresql** [48] was used to create a database connection in Ballerina. This library provides the functionality required to access and manipulate data stored in a PostgreSQL database, similar to the **ballerina/http** [45] library. This creates a connection to the desired database from an object named *Client*, as seen in Code Snippet 5.

```
01 import ballerina/regex;
02 import ballerina/sql;
03 import ballerina/jwt;
04 import ballerinax/postgresql;
05 import ballerinax/postgresql.driver as _;
06
07 // DATABASE CONFIGS -----------------------------------------
08
09 final postgresql:Client userDBClient = check new ("db_users_bo_mcs", "postgres",
"postgrespw", "postgres", 5510);
```

Code Snippet 5 - Ballerina User BO microservice database connection

Regarding the domain classes, they were transformed into Ballerina objects, which use the *class* type. Code Snippet 6 shows an example of a Ballerina object used in the service migration.

```
01 public class Vehicle {
02     string licensePlate;
03     VehicleType vehicleType;
04     VehicleEnergySource vehicleEnergySource;
05 
06     public isolated function init(string licensePlate, VehicleType vehicleType,
VehicleEnergySource vehicleEnergySource) {
07         self.licensePlate = licensePlate;
08         self.vehicleType = vehicleType;
09         self.vehicleEnergySource = vehicleEnergySource;
10     }
11 
12     public isolated function toJson() returns json {
13         return {
14             "licensePlateNumber": self.licensePlate,
15             "vehicleType": self.vehicleType,
16             "vehicleEnergySource": self.vehicleEnergySource
17         };
18     }
19 };
```

Code Snippet 6 - Ballerina User BO microservice Vehicle class

In addition to the objects, records were also created to facilitate database operations and convert payload requests from JSON to *records*. For example, the Vehicle object originated the VehicleOnCreation *record*, represented in Code Snippet 7.

```
01 public type VehicleOnCreation record {
02     string licensePlateNumber;
03     VehicleType vehicleType;
04     VehicleEnergySource vehicleEnergySource;
05 };
```

Code Snippet 7 - Ballerina User BO microservice VehicleOnCreation record

The original implementation used the *JpaRepository* interface to save entities inside the database, leveraging the Java Persistence API (JPA) to manage the database schema and its entity relationships. However, in Ballerina, it's not possible to use JPA. As such, it was necessary to implement all essential methods for managing data using SQL commands. In Code Snippet 8, a simple query can be seen.

```
01 public isolated function getAllUserVehicles(int userId) returns json|error? {
02     stream<VehicleRecord, error?> vehicleStream = userDBClient->query(`select v.*
from app_user_vehicles as auv join vehicle as v on auv.vehicles_license_plate_number
= v.license_plate_number where auv.user_user_id = ${userId};`, VehicleRecord);
03
04     string rawData = "";
05     check from VehicleRecord vehicle in vehicleStream
06         do {
07             Vehicle vehicleDto = new Vehicle(vehicle.license_plate_number,
mapIntToVehicleType(vehicle.vehicle_type),
mapIntToVehicleEnergySource(vehicle.vehicle_energy_source));
08             rawData = rawData + vehicleDto.toJson().toString() + ",";
09         };
10     check vehicleStream.close();
11
12     rawData = rawData.substring(0, rawData.length() - 1);
13     rawData = "[" + rawData + "]";
14
15     json vehiclesJson = check rawData.fromJsonString();
16     return vehiclesJson;
17 };
```

Code Snippet 8 - Ballerina User BO microservice getAllUserVehicles function

Another functionality from the original User BO microservice was the user login endpoint, which generated JWT tokens for authentication. This functionality was replicated in Ballerina using the **ballerina/jwt** [46] module. An *IssuerConfig* was created, including the expiration time, custom claims, signature algorithm and key, and other fields. This config was later used in the **jwt:issue** function to generate the JWT that the user can use in the system. Code Snippet 9 displays the login function used for the login endpoint.

```
01 public isolated function login(UserCredentials UserCredentials) returns json|error?
{
02      UserRecord queryResult = check userDBClient->queryRow(`select * from app_user
where email = ${UserCredentials.email};`);
03     UserRecord user = check queryResult.cloneWithType(UserRecord);
04
05     jwt:IssuerConfig issuerConfig = {
06         username: user.user_id.toString(),
07         expTime: 6000,
08         customClaims: {"role": mapIntToRole(user.role)},
09         signatureConfig: {
10             algorithm: "HS256",
11             config: secretKey
12         }
13     };
14
15     if (unhashedPassword != UserCredentials.password) {
16         return error("Invalid credentials");
17     }
18
19     string token = check jwt:issue(issuerConfig);
20     return new AccessToken(token, user.first_name).toJson();
21 };
```

Code Snippet 9 - Ballerina User BO microservice login function

5.3 Adjustments

During the implementation phase, various adjustments were necessary to ensure the successful migration and proper functioning of the services in Ballerina. Below are the fundamental changes and their implications:

First, it was necessary to modify the URL structures so that path variables were always placed at the end of the request URL. This adjustment was crucial for Ballerina to parse and handle these requests correctly. For instance, if the original request URL was structured with the path variable in the middle, it had to be rearranged to ensure the path variable appeared at the end. Otherwise, the endpoint would not be found.

Another issue encountered was with requests to the root path (/), which did not work as expected. To address this problem, requests at the root path were changed in the original, and Ballerina services to work around that problem and prevent any potential issues.

Moreover, a specific configuration change was necessary for the Spring Cloud settings. The property *spring.cloud.refresh.enabled=false* was added because of Eureka's compatibility with Spring Native, as mentioned in the spring documentation [49]. This configuration change was essential to prevent any runtime issues related to service discovery and registration in the native image executables.

In addition to the other changes, a DNS resolution issue was encountered when generating a native image of the original Spring API Gateway. After some research, the problem was fixed by adding a custom configuration class to the Spring API Gateway. This class, **HttpClientResolverFixConfig**, customizes the HTTP Client to use the **DefaultAddressResolverGroup**, resolving the DNS error. The class mentioned above is represented in Code Snippet 10.

```
01 package com.labdsof.gateway;
02
03 import org.springframework.cloud.gateway.config.HttpClientCustomizer;
04 import org.springframework.context.annotation.Bean;
05 import org.springframework.context.annotation.Configuration;
06
07 import io.netty.resolver.DefaultAddressResolverGroup;
08 import reactor.netty.http.client.HttpClient;
09
10 @Configuration
11 public class HttpClientResolverFixConfig {
12
13     @Bean
14     public HttpClientCustomizer httpClientResolverCustomizer() {
15         return new HttpClientCustomizer() {
16
17             @Override
18             public HttpClient customize(HttpClient httpClient) {
19                 return httpClient.resolver(DefaultAddressResolverGroup.INSTANCE);
20             }
21         };
22     }
23
24 }
```

Code Snippet 10 - Spring API Gateway DNS resolver class

These adjustments were crucial for ensuring the smooth operation, integration and analysis of the migrated services, which were compiled using GraalVM default and AOT compilation methods.

5.4 Generating native images

Creating native images is an essential step of this experiment since GraalVM AOT compilation gives better results for the objective aspects of this analysis.

A series of essential steps were followed to generate a native image for a Ballerina service. Initially, it was vital to ensure the installation and proper configuration of the GraalVM on the machine. This involved setting the **GRAALVM_HOME** environment variable and updating the system **PATH** to incorporate GraalVM binaries. Then, to generate the native image, a Ballerina build command with the *--graalvm* flag option was used. The full Ballerina command should be something like *bal build --graalvm*.

The approach taken for Java involved similar steps to guarantee the successful generation of a native image. Firstly, just like with Ballerina services, it was essential to ensure that GraalVM was properly installed and configured. Following this, the GraalVM Native Maven Plugin was added to the project's **pom.xml**, providing support for building and testing GraalVM native images. The native image was built by executing the Maven command *mvn -Pnative native:compile*.

5.5 Docker deployment

Ensuring a consistent environment for implementing the applications developed for the experiment was a key consideration. Initially, the PMS project used Docker to containerise the applications, so as well as including the applications developed in the PMS docker-compose, a configuration was added to each container to limit and reserve minimum resources for them.

The following steps outline the deployment process for the developed Ballerina applications:

1. **Generate Artefacts**: When running the build command adding the *--cloud=docker* flag would generate Docker artefacts.
2. **Build the Docker Image**: The previous command creates a Docker image for the Ballerina service. However, the Ballerina service was generated by the already PMS Docker compose.
3. **Docker Compose Configuration**: The Docker Compose file was updated to include the new Ballerina applications, ensuring they integrate with the existing PMS system components inside the park20 network.

The Code Snippet 11 represents how a Ballerina component was deployed into the park20 network.

```yaml
ballgateway:
  <<: *default-resources
  build:
    context: ./bal_api_gateway/target/docker/bal_api_gateway
    dockerfile: Dockerfile
  environment:
    eureka.client.service-url.defaultZone: http://host.docker.internal:8761/eureka
    eureka.instance.ip-address: host.docker.internal
  ports:
    - "9090:9090"
  networks:
    - park20
```

Code Snippet 11 - Ballerina application configuration inside PMS docker-compose

The resource allocation within Docker containers was managed to achieve uniform distribution and minimize performance variations. Its settings were defined using a configuration variable (*default-resources*) within the *docker-compose.yml* file. This variable enforced limits and reservations for CPU and memory usage across the service containers. Code Snippet 12 represents how these configurations were defined.

```yaml
x-common-resources: &default-resources
  deploy:
    resources:
      limits:
        cpus: '1'
        memory: 512M
      reservations:
        cpus: '0.5'
        memory: 256M
```

Code Snippet 12 - Resources allocation configuration

CPU usage and memory utilization/limits can then be consulted within Docker to validate whether the resource allocation was ensured. Figure 12 illustrates these metrics, confirming that each service is attached to its allocated resources.

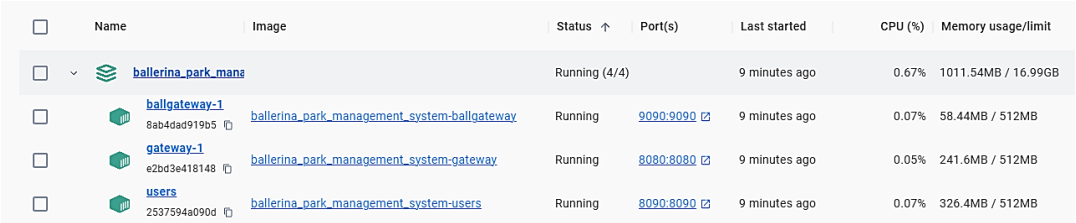


Figure 12 - Docker resource metrics

5.6 Tests

Each Ballerina application was migrated and subjected to integration tests. This process ensured that the implementation was functional and that everything worked as expected.

The testing strategy focused on validating the integration of the migrated components within the PMS project. It leveraged the existing Postman collection within the PMS project to do that. This collection, with its coverage of all microservice endpoints, ensured the correctness of each application. Its simple structure, with predefined login requests and folders containing the endpoints of the respective microservice, facilitates the tests on the system. Figure 13 provides a visual representation of the structure of the Postman collection that was used.

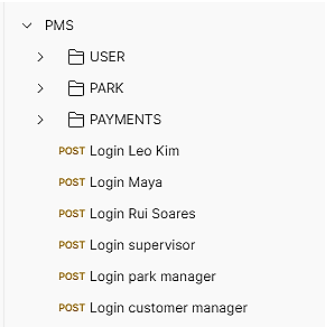


Figure 13 - PMS postman collection
From [44]

All the API Gateway endpoints and the respective endpoints of the USER BO microservice were tested. At the end of both application migrations, the integration of both applications was also tested to ensure that they worked together as intended.

After the successful implementation of the applications in Docker, the Postman collection was run again to check its functionality in the respective containers. The results indicated that the migration was successful, with the Ballerina services working as expected and integrating into the existing PMS system.

6 Evaluation

This chapter presents the methodology used to analyse and evaluate Ballerina according to the implementation developed for the Park20 company's park system. The aim is to assess Ballerina's impact on performance and elasticity aspects for distributed applications and compare its results with those of Java. To this end, the Goal Question Metric (GQM) method was used to establish the metrics for evaluating the performance and elasticity of the applications.

6.1 Methodology

The GQM approach is a goal-oriented method that helps define and interpret software metrics. It consists of three levels [50]:

- **Conceptual level (Goal)**: This level defines the purpose of the measurement, the object to be measured, the issue of interest, and the perspective of the measurement.
- **Operational level (Question)**: This level breaks down the goal into specific questions that must be answered to achieve it.
- **Quantitative level (Metric)**: This level identifies the data that needs to be collected to answer the questions and achieve the goal.

The GQM model begins by defining a goal, which is then broken down into questions and refined into metrics. The same metric can be used for different questions under the same goal. Once the model is defined, strategies for collecting data for the metrics must be determined [50]. Figure 14 represents the structure of the GQM model followed.

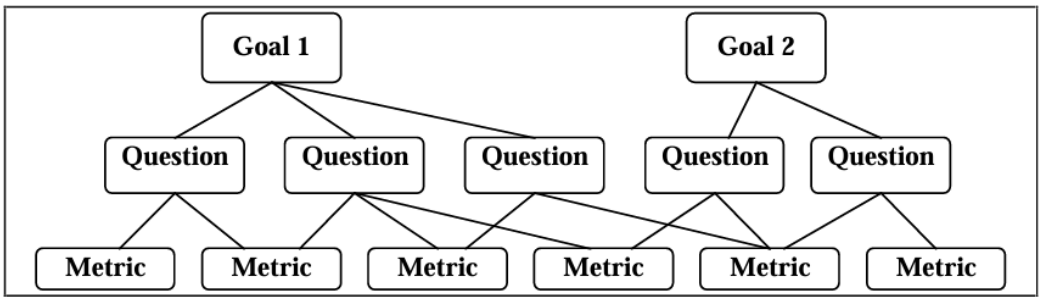


Figure 14 - GQM model structure
From [50]

In this study, the GQM structure is applied to assess the impact of using Ballerina on performance and elasticity when developing distributed applications. These aspects are compared to an established language, in this case, Java. The questions are derived from the research questions outlined in section 1.3, and the relevant metrics are used to answer these questions. Therefore, the structure is:

Goal: Assess the impact of Ballerina on performance and elasticity when developing distributed applications.

Questions:

- **Q1**: What is the impact on the performance of distributed applications developed with Ballerina compared to Java?
- **Q2**: What is the impact on the elasticity of distributed applications developed with Ballerina compared to Java?

Metrics:

- **M1**: Response time
- **M2**: Throughput
- **M3**: Startup time

Metrics data can be obtained using tools or through manual analysis. The following sections describe the methods used to gather the necessary data and explain the meaning behind each metric.

6.1.1 Performance

To explore the performance metrics stated in the GQM structure, the primary focus was on response time and throughput.

Response time is the system's duration to respond to a client request. This metric is crucial for understanding how quickly a cloud application can process requests, directly impacting overall performance and user experience. To measure response time, Grafana k6 was employed to

simulate client requests and capture the system's response times across various load scenarios, involving different numbers of simultaneous users.

Throughput refers to the number of requests the system can process per second, reflecting its ability to manage and scale with demand. Higher throughput indicates better performance and greater scalability. Throughput was also measured using Grafana k6 during the load tests, providing insights into the system's capacity under different stress levels.

6.1.2 Elasticity

In terms of elasticity, the metric explored was the startup time.

Startup time measures how quickly a service becomes fully operational and ready to handle requests after it has been initiated. This metric is crucial for assessing the flexibility of microservices and distributed applications, particularly in scalable environments where services may need to start and stop frequently. Shorter startup times contribute to better system scalability and responsiveness.

The startup time was measured by deploying the services in a controlled environment and recording the duration until each service was ready to handle requests. This was achieved through a script for Ballerina applications, while for Java applications, the startup time was logged directly by the system.

6.2 Experiments

This section presents the results obtained from the experiments comparing Ballerina with Java applications. As mentioned in the previous chapter, the experiments were conducted in a controlled environment to minimize external variables that could influence the results. Specifically, the system was deployed using Docker with allocated resources to ensure consistency. The machine used to conduct the experiments has the following specifications:

- **Operating System**: Windows 11 Pro
- **Processor**: Intel Core i7-13700K 3.40GHz
- **Memory**: 32GB RAM

6.2.1 Performance

Grafana k6 was the primary load-testing tool to evaluate the application's performance. k6 is an open-source tool designed to load test functional behaviour and measure performance by simulating multiple users sending requests to the application and recording performance metrics [51]. To test the performance of the applications with k6, Javascript scripts had to be created. These scripts consisted of sending various requests via the API gateway to the user's

microservice, explicitly targeting the "Get all users" endpoint and retrieving complete information about the user, including associated vehicles, parking history, roles and other data.

The performance tests consisted of various scenarios and different types of tests, load tests, stress tests and hypothesis tests to analyse whether there were significant differences between both technologies and the two types of compilation.

6.2.1.1 Load Testing

The load tests were the first set of experiments conducted, focusing on a representative scenario with 10 simultaneous users sending requests, which was sufficient to highlight differences between the technologies and compilation methods. Initially, a comparison between the HTTP/1.1 and HTTP/2 protocols was performed to evaluate Ballerina's performance across these two HTTP versions compared to Java. Table 4 and Table 5 present the load test results for AOT compilation under both protocols.

The results demonstrate that HTTP/2 consistently outperforms HTTP/1.1 in all configurations, with Ballerina's results surpassing those of Java. This performance improvement is evident across various tests and setups, validating the superior efficiency of HTTP/2 over the older version and Ballerina over Java.

Table 4 – Load tests HTTP/1.1 results

API Gateway Technology	User BO Technology	Average Response Time (ms)	Max (ms)	Median (ms)	Min (ms)	Throughput (req/s)
Ballerina AOT	**Ballerina AOT**	**7.53**	**17.39**	**5.54**	**2.60**	**9.86**
Ballerina AOT	Java AOT	17.74	32.16	16.89	11.67	9.71
Java AOT	Ballerina AOT	9.14	15.10	8.16	2.59	9.85
Java AOT	Java AOT	18.65	46.18	18.61	13.66	9.70

Table 5 – Load tests HTTP/2 results

API Gateway Technology	User BO Technology	Average Response Time (ms)	Max (ms)	Median (ms)	Min (ms)	Throughput (req/s)
Ballerina AOT	**Ballerina AOT**	**7.06**	**15.10**	**6.36**	**2.62**	**9.91**
Ballerina AOT	Java AOT	8.57	20.86	7.21	3.20	9.89
Java AOT	Ballerina AOT	16.83	26.58	16.23	11.12	9.80
Java AOT	Java AOT	17.57	31.95	16.01	10.18	9.77

Given the superior performance of HTTP/2, the subsequent comparisons will focus exclusively on this protocol to streamline even more the presentation of results. For a more complete analysis, Appendix E contains additional scenarios with both HTTP protocol versions, analysed previously, and more users simultaneously.

The next comparison was the Ballerina AOT with Java AOT, Table 6 compares the performance of Ballerina and Java using AOT compilation for the API Gateway and User BO Microservice. Ballerina AOT demonstrated a clear performance advantage: when both the API Gateway and the User BO Microservice were implemented with Ballerina AOT, the average response time was 7.06 ms, significantly lower than the 17.98 ms observed when both components were

implemented in Java AOT. Maximum response times also favoured Ballerina AOT, with a peak of 15.10 ms compared to Java AOT's 53.96 ms. Even in mixed configurations, when Ballerina AOT was used on User BO, where most of the logic is implemented, it showed better performance with an average response time of 8.40 ms compared to Java AOT with 16.83 ms.

Table 6 - Comparison Ballerina AOT with Java AOT

API Gateway Technology	User BO Technology	Average Response Time (ms)	Max (ms)	Median (ms)	Min (ms)	Throughput (req/s)
Ballerina AOT	**Ballerina AOT**	**7.06**	**15.10**	**6.36**	**2.62**	**9.91**
Ballerina AOT	Java AOT	16.83	26.58	16.23	11.12	9.80
Java AOT	Ballerina AOT	8.40	22.65	8.70	2.06	9.87
Java AOT	Java AOT	17.98	53.96	14.11	11.43	9.74

Having established the advantages of Ballerina AOT over Java AOT, the next comparison is between Ballerina and Java with the default compilation, as shown in Table 7. It presents results from Ballerina and Java that further support the trend observed with AOT compilation. When both the API Gateway and User BO microservice were implemented in Ballerina, the average response time was 9.46 ms, which is significantly lower than the 20.38 ms recorded with all components using Java. This performance gap illustrates Ballerina's efficiency, even without the optimizations provided by AOT compilation. Additionally, Ballerina's maximum response time of 16.13 ms was markedly lower than Java's 38.75 ms, further highlighting Ballerina's reliability and consistency in handling requests.

Table 7 - Comparison Ballerina with Java

API Gateway Technology	User BO Technology	Average Response Time (ms)	Max (ms)	Median (ms)	Min (ms)	Throughput (req/s)
Ballerina	**Ballerina**	**9.46**	**16.13**	**9.04**	**6.42**	**9.83**
Ballerina	Java	19.48	51.53	15.08	11.29	9.75
Java	Ballerina	10.21	25.00	8.85	4.16	9.79
Java	Java	20.38	38.75	20.39	12.40	9.70

Another important comparison is between Ballerina default and AOT compilations, which underscores the performance benefits of AOT compilation. When looking again at Table 6 and Table 7, Ballerina AOT achieved an average response time of 7.06 ms, compared to 9.46 ms for the default compilation. This reduction in response time indicates that AOT optimizes Ballerina's performance and makes it even more efficient under similar conditions.

This reduction in response times not only demonstrates the effectiveness of AOT compilation but also aligns with the earlier findings discussed in section 3.7.1, which highlighted Ballerina's optimization capabilities in various scenarios by using GraalVM AOT compilation.

In general, the load testing results reinforce Ballerina's strengths, particularly when using AOT compilation. Ballerina consistently demonstrated lower response times, better throughput, and more stable performance than Java, regardless of the specific configuration and HTTP protocol. These findings underscore Ballerina's suitability for high-performance scenarios, especially in

environments where low latency and reliable response times are critical. The observed improvements with AOT compilation further validate Ballerina's efficiency, making a positive impact on distributed applications, particularly when integrated into diverse technological ecosystems.

6.2.1.2 Hypothesis tests

To statistically validate the performance differences observed in the load tests, hypothesis tests were conducted using the R programming language. These tests aimed to determine whether the differences between Ballerina and Java, as well as between their respective compilation methods, were statistically significant or simply due to random variation.

The testing process began with assessing the distribution of the performance data to determine the appropriate statistical tests. Given the sample sizes, the *Lilliefors*[1] test was used to check for normality since the number of samples exceeded 30. Otherwise, for smaller samples, the *Shapiro-Wilk*[2] test would be applied. Once the distribution was assessed, the next step was to evaluate the symmetry of the distributions, which informed whether parametric or non-parametric hypothesis tests would be used. If the data was symmetrically distributed and followed a normal distribution, a parametric test like the t-test[3] would be used. Otherwise, a non-parametric alternative, such as the Wilcoxon signed-rank[4] test, was employed.

Four key hypothesis tests were performed to compare the performance results:

- **HTTP/1.1** protocol vs. **HTTP/2** protocol.
- **Ballerina AOT** compilation vs. **Java AOT** compilation.
- **Ballerina** default compilation vs. **Java** default compilation.
- **Ballerina AOT** compilation vs. **Ballerina** default compilation.

Code Snippet 13 represents the structure followed for every hypothesis test performed.

```
01 # Normality test
02 lillie.test(ConjuntoAverageTime)
03 lillie.test(ConjuntoAverageTime)
04
05 # Asymmetric test
06 skewness(ConjuntoAverageTime)
07 skewness(ConjuntoAverageTime)
08
09 # Hypothesis test
10 valor_p <- wilcox.test(ConjuntoAverageTime,ConjuntoAverageTime,
paired=FALSE)$p.value
11 valor_p
```

Code Snippet 13 - Hypothesis test structure

[1] *Lilliefors* - https://en.wikipedia.org/wiki/Lilliefors_test
[2] *Shapiro-Wilk* - https://en.wikipedia.org/wiki/Shapiro%E2%80%93Wilk_test
[3] *t-test* - https://en.wikipedia.org/wiki/Student%27s_t-test
[4] *Wilcoxon signed-rank* - https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

The results of the hypothesis tests consistently rejected the null hypothesis (H0), which consists of assuming no significant differences between the compared group of samples. Instead, the tests supported the alternative hypothesis (H1), indicating that the observed differences in performance were statistically significant.

- **HTTP/1.1 vs. HTTP/2**: The tests confirmed that HTTP/2 provides significant performance improvements over HTTP/1.1, supporting the load testing results that highlighted the benefits of the newer protocol.
- **Ballerina AOT vs. Java AOT**: The test results showed significant differences in favour of Ballerina AOT, reinforcing the performance advantages observed in Table 6.
- **Ballerina vs. Java**: Similarly, the hypothesis test confirmed that Ballerina also outperforms Java in its default compilations, as detailed in Table 7.
- **Ballerina AOT vs. Ballerina**: The statistical analysis demonstrated that AOT compilation provides a performance boost for Ballerina, with significant differences in response times supporting the earlier conclusions from the load testing performance analysis.

The hypothesis tests provide statistical evidence that the performance differences observed between the four comparisons, are not due to random chance. The rejection of the null hypothesis across all tests underscores that Ballerina, particularly when using AOT compilation and HTTP/2, offers superior performance characteristics in terms of response times and reliability.

6.2.1.3 Stress Tests

In addition to the initial load tests, a stress test was conducted to evaluate the application's stability and resilience under extreme load conditions. Unlike load testing, which assesses performance under expected conditions, stress testing pushes the system beyond its normal operational capacity to identify its breaking point and observe how it handles failures.

The stress test gradually increased the number of virtual users sending requests to the system over time, aiming to determine the maximum number of simultaneous users the application could handle before encountering failures, such as request timeouts. The test continued to add users until the first instance of a timeout error, marking the system's maximum capacity under stress.

The results of the stress test are presented in Table 8 and Table 9, which detail the maximum number of simultaneous users the system could handle before experiencing failures under HTTP/1.1 and HTTP/2 protocols, respectively.

Table 8 - HTTP/1.1 Stress test results

Technology	Number of simultaneous users
Ballerina AOT	**9,579**
Ballerina	8,381
Java AOT	7,644
Java	6,356

Table 9 - HTTP/2 Stress test results

Technology	Number of simultaneous users
Ballerina AOT	**11,373**
Ballerina	9,172
Java AOT	8,984
Java	7,757

The stress test results indicate that Ballerina, especially with AOT compilation and when using the HTTP/2 protocol, significantly outperforms Java in handling concurrent users. Ballerina AOT managed up to 11,373 simultaneous users, compared to Java AOT's 8,984 users, demonstrating Ballerina's superior scalability and efficiency under heavy load conditions. These results, regardless of the HTTP protocol version used, confirm that Ballerina, particularly with AOT compilation, offers greater resilience and robustness, making it better equipped to handle high-demand scenarios than Java.

6.2.2 Elasticity

When analysing elasticity, the startup times for Ballerina and Java applications were measured to assess their responsiveness under scaling conditions. For Ballerina applications, a custom script recorded the time taken from application initiation to readiness for handling requests. This script logged the start time and then executed a Docker command to initiate the desired container. For Java services, the startup time was extracted from the logs indicating when the service was ready to accept requests. To ensure accuracy, each service was started ten times, and the average startup time was calculated.

The results, shown in Table 10 and Table 11, highlight that Ballerina demonstrates significantly quicker startup times, particularly when using Ahead-of-Time (AOT) compilation. For instance, the API Gateway service compiled with Ballerina AOT launched in an average of 66 ms, nearly a 90% reduction in startup time compared to Ballerina's default compilation. Although Ballerina's default compilation is slower than AOT, it still outperforms Java's standard compilation, reinforcing that Ballerina, especially with AOT, offers substantial advantages in startup speed. The same occurs at the User BO microservice, leading to a conclusion that Ballerina is highly suitable for environments where rapid scaling and deployment are critical, offering significant improvements over traditional Java implementations.

Table 10 - API Gateway average startup time

Technology	Average startup time (s)
Ballerina AOT	**0.066**
Ballerina	0.635
Java	1.408
Java AOT	0.088

Table 11 - User BO average startup time

Technology	Average startup time (s)
Ballerina AOT	**0.081**
Ballerina	0.818
Java	3.015
Java AOT	0.228

6.3 Summary

The evaluation findings presented in this chapter address the GQM questions related to performance and elasticity, offering insights into the capabilities of Ballerina compared to Java.

Performance: Ballerina demonstrated superior performance across various configurations, particularly when using Ahead-of-Time (AOT) compilation and HTTP/2. The results revealed that Ballerina achieved better response times and higher throughput under load, showcasing its efficiency in handling requests and maintaining stability even under heavy load conditions. Stress tests further affirmed Ballerina's resilience, with the application successfully managing a significantly higher number of simultaneous users compared to Java.

Elasticity: Ballerina exhibited a clear advantage in startup times, particularly with AOT compilation. For instance, the API Gateway application achieved a mean startup time of just 66 ms with Ballerina AOT. Even when using default compilation, Ballerina outperformed Java, demonstrating its effectiveness in environments requiring rapid deployment and scalability.

In summary, Ballerina, especially with AOT compilation and HTTP/2, provides superior performance and elasticity compared to Java. This positions Ballerina as a robust and viable option for modern and scalable cloud applications contributing with a positive impact when integrated into a distributed applications environment. For a more comprehensive view of the experiment and detailed results, refer to the migration PMS repository [52], which includes extensive data and insights supporting the analysis of both Ballerina and Java.

7 Conclusion

This chapter summarizes the study's achievements and contributions, difficulties, threats to validity, future work, and final considerations. It offers an overview of the findings and reflections gathered throughout this dissertation.

7.1 Achievements and contributions

This study successfully met its primary objectives, providing insights into the performance and elasticity of the Ballerina programming language within the context of distributed applications. Through a comprehensive analysis that included performance testing and elasticity evaluation, the study established a clear understanding of Ballerina's strengths and limitations compared to Java, a widely established language in this domain.

A significant achievement of this study was the successful migration of the original project components and the execution of performance tests using Grafana k6. These tests generated detailed data on Ballerina's performance under varying conditions, highlighting its capacity to handle high loads. Notably, Ballerina demonstrated performance advantages, particularly when utilizing GraalVM native image. The statistical analysis, including hypothesis testing, provided depth to these findings, confirming that the observed performance differences between Ballerina and Java were statistically significant.

The elasticity evaluation further underscored Ballerina's strengths, especially with Ahead-of-Time (AOT) compilation. The study found that applications developed in Ballerina, such as the API Gateway service, achieved a remarkable mean startup time of 66 ms with AOT, marking a significant reduction compared to default compilation methods. This finding positions Ballerina as a highly suitable option for environments requiring rapid deployment and scaling, aligning well with the needs of modern distributed systems.

Moreover, the results validated claims made by the Ballerina development team [23] regarding the performance benefits of AOT compilation, such as reduced startup times and enhanced overall performance. This validation enhances Ballerina's credibility as a viable option for developing distributed applications, providing developers and organizations with evidence-

based insights into adopting Ballerina as a competitive alternative to traditional programming languages like Java.

7.2 Difficulties

The migration and experimentation phases presented several challenges that required careful consideration and adjustments to the original implementation and Ballerina applications. Given that Ballerina is a relatively new programming language, specific difficulties were encountered.

One significant challenge was the lack of literature, and resources specifically focused on Ballerina. Most available information was centred around more established languages like Java, which made it difficult to find relevant guidance and examples. This shortage of resources steeped the learning curve and extended the development timeline.

During the implementation process, additional obstacles occurred due to unfamiliarity with Ballerina and the need to address uncommon edge cases. Problem-solving became more time-consuming, necessitating a deeper exploration of Ballerina's features and functionalities. At times, the available documentation was not comprehensive enough to address these issues effectively.

These challenges underscore the complexities involved in working with new technologies. They highlight the importance of detailed research, extensive experimentation, and adaptability when undertaking such migrations, especially when dealing with a relatively new and less-documented language like Ballerina.

7.3 Threats to validity

Several factors can affect the validity of the results obtained in this study. One important consideration is the selection of the project used for migration and testing. To ensure a fair comparison, the Ballerina applications were developed to reflect the steps and functions of the original Java implementations, preserving established development practices. However, this need may have introduced inefficiencies that don't fully explore Ballerina's potential optimisations. The choice to follow the original logic ensured comparability but may have limited the performance benefits observed in Ballerina.

It is also important to recognise that the author is not a Ballerina expert and learned the language independently for this study. Although a significant effort was made to minimise the impact of this learning curve, the results could have been different with more experience in Ballerina and its best practices. This factor could have influenced the efficiency of the implementation and, consequently, the performance results of the Ballerina services.

7.4 Future work

As a continuation of this project, a logical next step would be to migrate the entire PMS backend system to Ballerina. This comprehensive migration would provide deeper insights into Ballerina's full range of capabilities and limitations, allowing for a more detailed assessment of its suitability for large-scale distributed systems. Expanding the scope of the migration would also help identify additional performance and elasticity improvements or challenges that were not evident in this study.

Additionally, the results of this experiment are publicly available [52], providing resources for the Ballerina community, researchers, and developers. Future studies can leverage this data to compare Ballerina's performance and elasticity against other programming languages and frameworks, such as C# and Go. Such comparisons would contribute to a more informed, evidence-based approach to technology selection for specific use cases and operational requirements, guiding decision-makers in adopting the most suitable technologies for their needs.

7.5 Final considerations

This dissertation represents a significant milestone for the author, marking the beginning of an exploration into a new programming language, Ballerina. The journey was both challenging and rewarding, as the Ballerina community continues to grow and evolve. The author hopes that this work will inspire further research and experimentation, leading to more detailed evaluations and analyses within the broader academic and development communities.

The project presented several challenges, pushing the author to innovate and refine his research and dissertation management skills. This experience not only enlarged the author's understanding of distributed applications but also provided valuable insights that will undoubtedly benefit his future career. Learning Ballerina required continuous adaptation and problem-solving, which strengthened the author's technical expertise and ability to navigate new technologies.

In conclusion, this dissertation was a valuable experience for both personal and professional growth. It provided crucial insights into software development, particularly in the context of distributed applications and emerging technologies, and a strong foundation was built for future research and development attempts.

References

[1] D. Lu, J. Wu, Y. Sheng, P. Liu, and M. Yang, “Analysis of the popularity of programming languages in open source software communities,” in *2020 International Conference on Big Data and Social Sciences (ICBDSS)*, IEEE, Aug. 2020, pp. 111–114. doi: 10.1109/ICBDSS51270.2020.00033.

[2] Ballerina Team, “Language basics.” Accessed: Oct. 25, 2023. [Online]. Available: https://ballerina.io/learn/language-basics/

[3] D. Madushan, *Cloud Native Applications with Ballerina: A guide for programmers interested in developing cloud native applications using Ballerina Swan Lake*. Packt Publishing, 2021.

[4] Ballerina Team, “Announcing Ballerina 2201.7.0 (Swan Lake Update 7).” Accessed: Oct. 25, 2023. [Online]. Available: https://blog.ballerina.io/posts/2023-07-14-announcing-ballerina-2201.7.0-swan-lake-update-7/

[5] RedMonk Team, “About.” Accessed: Dec. 28, 2023. [Online]. Available: https://redmonk.com/about/

[6] S. O’Grady, “The RedMonk Programming Language Rankings: January 2024 – tecosystems.” Accessed: May 15, 2024. [Online]. Available: https://redmonk.com/sograd y/2024/03/08/language-rankings-1-24/

[7] T. Jewell, “Ballerina Microservices Programming Language: Introducing the Latest Release and ‘Ballerina Central.’” Accessed: Oct. 25, 2023. [Online]. Available: https://www.infoq.com/articles/ballerina-microservices-language-part-1/

[8] Ballerina Team, “Ballerina for integration,” 2023, Accessed: Jan. 01, 2024. [Online]. Available: https://ballerina.io/usecases/integration/

[9] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillère, “Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects,” in *2013 IEEE 37th Annual Computer Software and Applications Conference*, 2013, pp. 303–312. doi: 10.1109/COMPSAC.2013.55.

[10] P. Bhattacharya and I. Neamtiu, “Assessing programming language impact on development and maintenance: a study on c and c++,” in *Proceedings of the 33rd International Conference on Software Engineering*, in ICSE ’11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 171–180. doi: 10.1145/1985793.1985817.

[11] M. Ait Said, A. Ezzati, S. Mihi, and L. Belouaddane, “Microservices Adoption: An Industrial Inquiry into Factors Influencing Decisions and Implementation Strategies,”

International Journal of Computing and Digital Systems, vol. 15, pp. 2210–142, Mar. 2024, doi: http://dx.doi.org/10.12785/ijcds/1501100.

[12] C. Sharma, “GraalVM-Native Images & Microframeworks.,” May 2022, Accessed: Dec. 30, 2023. [Online]. Available: https://medium.com/codex/graalvm-native-images-microframeworks-9b3fab2e8f17

[13] B. Oates, *Researching Information Systems and Computing*. SAGE Publications Ltd, 2006.

[14] Ordem dos Engenheiros, “Código de Ética e Deontologia,” Nov. 2016. Accessed: Dec. 12, 2023. [Online]. Available: https://www.ordemengenheiros.pt/fotos/editor2/regulamentos/codigo_ed.pdf

[15] IEEE Board of Directors, “IEEE Code of Ethics.” Accessed: Jan. 01, 2024. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html

[16] Instituto Politécnico do Porto, “Regulamento do Código de Boas Práticas e de Conduta do Instituto Politécnico do Porto,” Oct. 2020. Accessed: Dec. 12, 2023. [Online]. Available: https://www.ipp.pt/comunidade/missao-equidade-diversidade-inclusao/DespachoP.PORTOP0402020CodigodeBoasPraticasedeConduta.pdf/view

[17] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language*, 3rd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[18] IBM Cloud Education, “JVM vs. JRE vs. JDK: What’s the Difference?” Accessed: Mar. 13, 2024. [Online]. Available: https://www.ibm.com/blog/jvm-vs-jre-vs-jdk/

[19] S. Roy, “The Difference Between a Framework and a Library | Baeldung on Computer Science.” Accessed: Mar. 13, 2024. [Online]. Available: https://www.baeldung.com/cs/framework-vs-library

[20] S. Weerawarana, C. Ekanayake, S. Perera, and F. Leymann, “Bringing Middleware to Everyday Programmers with Ballerina,” in *Business Process Management: 16th International Conference, BPM 2018, Sydney, NSW, Australia, September 9–14, 2018, Proceedings*, Berlin, Heidelberg: Springer-Verlag, 2018, pp. 12–27. doi: 10.1007/978-3-319-98648-7_2.

[21] “Announcing Ballerina 1.0 - The Ballerina programming language.” Accessed: Jun. 19, 2024. [Online]. Available: https://blog.ballerina.io/posts/2019-09-09-annoucing-1.0.0/

[22] Ballerina Team, “Announcing Ballerina 2201.0.0 (Swan Lake) - The Ballerina programming language.” Accessed: Jun. 19, 2024. [Online]. Available: https://blog.ballerina.io/posts/2022-02-01-announcing-ballerina-2201.0.0-swan-lake/

[23] Ballerina Team, “Announcing Ballerina 2201.7.0 (Swan Lake Update 7).” Accessed: Jan. 01, 2024. [Online]. Available: https://blog.ballerina.io/posts/2023-07-14-announcing-ballerina-2201.7.0-swan-lake-update-7/

[24] Ballerina Team, “How Ballerina empowered Fat Tuesday’s digital transformation - The Ballerina programming language.” Accessed: Jun. 20, 2024. [Online]. Available: https://ballerina.io/case-studies/fat-tuesday/

[25] M. Šipek, D. Muharemagić, B. Mihaljević, and A. Radovan, “Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus,” in *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2020, pp. 1746–1751. doi: 10.23919/MIPRO48935.2020.9245290.

[26] M. Šipek, B. Mihaljević, and A. Radovan, “Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM,” in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 1671–1676. doi: 10.23919/MIPRO.2019.8756917.

[27] Ballerina Team, “GraalVM executable overview,” 2023, Accessed: Dec. 30, 2023. [Online]. Available: https://ballerina.io/learn/graalvm-executable-overview/

[28] “Graal JIT Compiler Operations Manual.” Accessed: Jun. 20, 2024. [Online]. Available: https://www.graalvm.org/dev/reference-manual/compiler/operations/

[29] A. W. Wade, P. A. Kulkarni, and M. R. Jantz, “AOT vs. JIT: impact of profile data on code quality,” in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–10. doi: 10.1145/3078633.3081037.

[30] GraalVM Team, “Build faster, smaller, leaner applications.” Accessed: Dec. 29, 2023. [Online]. Available: https://www.graalvm.org/

[31] A. Yurenko and K. Silz, “Revolutionizing Java with GraalVM Native Image - InfoQ.” Accessed: Mar. 13, 2024. [Online]. Available: https://www.infoq.com/articles/native-java-graalvm/

[32] A. E. Eldeeb, “GraalVM: The future of JVM languages.” Accessed: Mar. 13, 2024. [Online]. Available: https://medium.com/@ahmedeeldeeb/graalvm-the-future-of-jvm-languages-350fa892ae45

[33] M. J. Page *et al.*, “The PRISMA 2020 statement: an updated guideline for reporting systematic reviews,” *BMJ*, vol. 372, 2021, doi: 10.1136/bmj.n71.

[34] F. Fong and M. Raed, “Performance comparison of GraalVM, Oracle JDK andOpenJDK for optimization of test suite execution time,” 2021.

[35] S. Di Meglio, L. L. L. Starace, and S. Di Martino, “Starting a New REST API project? A Performance Benchmark of Frameworks and Execution Environments,” 2023.

[36] R. Larsson, “Evaluation of GraalVM Performance for Java Programs,” pp. 0–30, 2020.

[37] T. Vergilio, L. Ha, and A.-L. Kor, “Comparative Performance and Energy Efficiency Analysis of JVM Variants and GraalVM in Java Applications,” *International Journal of Environmental Sustainability and Green Technologies*, vol. 14, pp. 1–32, Jan. 2023, doi: 10.4018/IJESGT.331401.

[38] C. Wimmer *et al.*, “Initialize once, start fast: application initialization at build time,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 0–184, Oct. 2019, doi: 10.1145/3360610.

[39] N. Ford, M. Richards, P. Sadalage, Z. Dehghani, and an O. M. Company. Safari, “Software Architecture : the Hard Parts : modern trade-off analyses for distributed architectures,” p. 450, Accessed: Jun. 06, 2024. [Online]. Available: https://www.oreilly.com/library/view/software-architecture-the/9781492086888/

[40] HTTP/2 Team, “HTTP/2 Frequently Asked Questions.” Accessed: Jun. 02, 2024. [Online]. Available: https://http2.github.io/faq/#what-about-non-browser-users-of-http

[41] Renaissance Team, “Renaissance Suite, a benchmark suite for the JVM.” Accessed: Jun. 02, 2024. [Online]. Available: https://renaissance.dev/

[42] S. M. Blackburn *et al.*, “The {DaCapo} Benchmarks: {J}ava Benchmarking Development and Analysis,” in *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, Portland, OR, USA: ACM Press, 2006, pp. 169–190. doi: http://doi.acm.org/10.1145/1167473.1167488.

[43] N. R. Herbst, S. Kounev, and R. Reussner, “Elasticity in Cloud Computing: What It Is, and What It Is Not,” in *10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA: USENIX Association, Jun. 2013, pp. 23–27. [Online]. Available: https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst

[44] Carvalho Daniel, Castro Ivo, Rocha Nuno, Coelho Ricardo, and Soares Rui, “Park_Management_System.” Accessed: Mar. 20, 2024. [Online]. Available: https://github.com/RicardoCoelho8/Park_Management_System

[45] Ballerina Team, “http - Ballerina Central.” Accessed: Jun. 27, 2024. [Online]. Available: https://central.ballerina.io/ballerina/http/2.11.2

[46] Ballerina Team, “jwt - Ballerina Central.” Accessed: Jun. 27, 2024. [Online]. Available: https://central.ballerina.io/ballerina/jwt/2.12.1

[47] Ballerina Team, “log - Ballerina Central.” Accessed: Jun. 27, 2024. [Online]. Available: https://central.ballerina.io/ballerina/log/2.9.0

[48] Ballerina Team, “postgresql - Ballerina Central.” Accessed: Jun. 27, 2024. [Online]. Available: https://central.ballerina.io/ballerinax/postgresql/1.12.0

[49] Spring Team, “Spring Cloud Netflix.” Accessed: Jun. 27, 2024. [Online]. Available: https://docs.spring.io/spring-cloud-netflix/docs/current/reference/html/#using-the-eurekaclient

[50] V. R. Basili, G. Caldiera, and H. D. Rombach, “The goal question metric approach,” *Encyclopedia of software engineering*, pp. 528–532, 1994, Accessed: Sep. 11, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:13884048

[51] Grafana Team, “k6 Documentation.” Accessed: Aug. 07, 2024. [Online]. Available: https://k6.io/docs/

[52] R. Coelho, “Ballerina_Park_Management_System.” Accessed: Jun. 06, 2024. [Online]. Available: https://github.com/RicardoCoelho8/Ballerina_Park_Management_System

[53] O. S. Vaidya and S. Kumar, “Analytic hierarchy process: An overview of applications,” *Eur J Oper Res*, vol. 169, no. 1, pp. 1–29, Feb. 2006, doi: 10.1016/J.EJOR.2004.04.028.

[54] T. L. Saaty, “How to make a decision: The analytic hierarchy process,” *Eur J Oper Res*, vol. 48, no. 1, pp. 9–26, Sep. 1990, doi: https://doi.org/10.1016/0377-2217(90)90057-I.

[55] M. I. Rahman, S. Panichella, and D. Taibi, “A curated Dataset of Microservices-Based Systems,” in *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution*, vol. 2520, CEUR-WS, 2019. Accessed: Mar. 31, 2024. [Online]. Available: https://github.com/davidetaibi/Microservices_Project_List?tab=readme-ov-file#References

[56] M. Raible and J. Long, “Beer catalog.” Accessed: May 31, 2024. [Online]. Available: https://github.com/oktadev/spring-boot-microservices-example

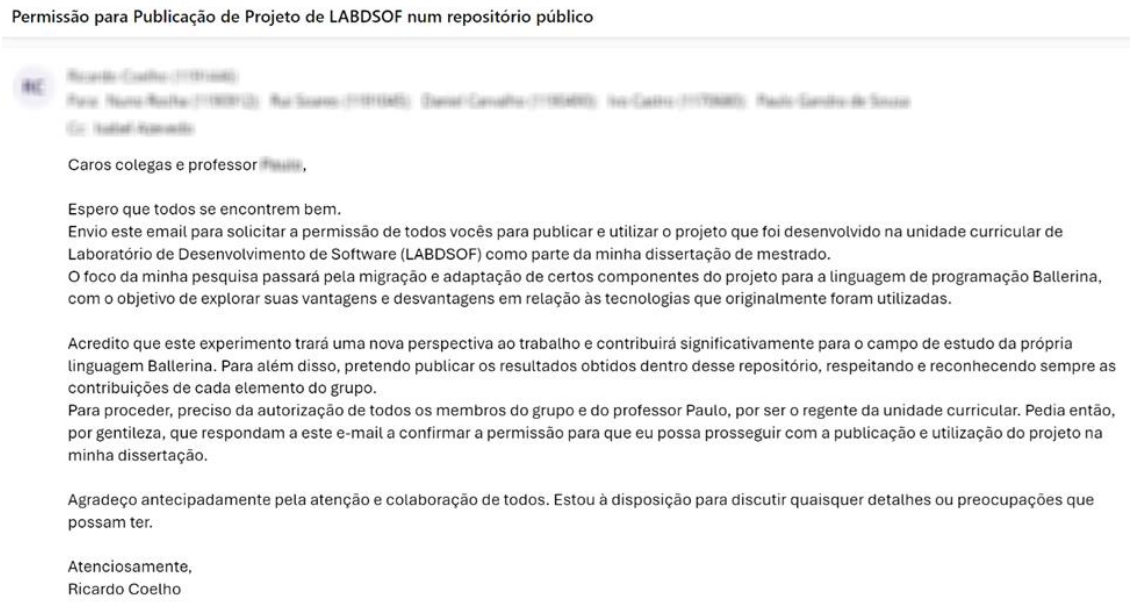
[57] F. Campos, “Blog post.” Accessed: May 31, 2024. [Online]. Available: https://github.com/fernandoabcampos/spring-netflix-oss-microservices

[58] B. Wilcock, “CQRS microservice application.” Accessed: Mar. 31, 2024. [Online]. Available: https://github.com/benwilcock/cqrs-microservice-sampler

[59] Ballerina Team, “The Ballerina programming language.” Accessed: Aug. 10, 2024. [Online]. Available: https://ballerina.io/

Appendix A

This appendix presents the authorisations of all the group members and the teacher responsible before the publication of the project used for this study, these authorisations were necessary to comply with ethical terms. The images below (Figure 15, Figure 16, Figure 17, Figure 18, Figure 19 and Figure 20) show the request and the respective responses from each group member and teacher:



Permissão para Publicação de Projeto de LABDSOF num repositório público

Caros colegas e professor Paulo,

Espero que todos se encontrem bem.
Envio este email para solicitar a permissão de todos vocês para publicar e utilizar o projeto que foi desenvolvido na unidade curricular de Laboratório de Desenvolvimento de Software (LABDSOF) como parte da minha dissertação de mestrado.
O foco da minha pesquisa passará pela migração e adaptação de certos componentes do projeto para a linguagem de programação Ballerina, com o objetivo de explorar suas vantagens e desvantagens em relação às tecnologias que originalmente foram utilizadas.

Acredito que este experimento trará uma nova perspectiva ao trabalho e contribuirá significativamente para o campo de estudo da própria linguagem Ballerina. Para além disso, pretendo publicar os resultados obtidos dentro desse repositório, respeitando e reconhecendo sempre as contribuições de cada elemento do grupo.
Para proceder, preciso da autorização de todos os membros do grupo e do professor Paulo, por ser o regente da unidade curricular. Pedia então, por gentileza, que respondam a este e-mail a confirmar a permissão para que eu possa prosseguir com a publicação e utilização do projeto na minha dissertação.

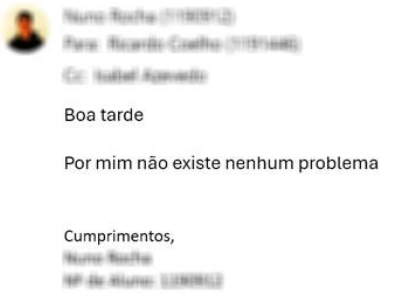
Agradeço antecipadamente pela atenção e colaboração de todos. Estou à disposição para discutir quaisquer detalhes ou preocupações que possam ter.

Atenciosamente,
Ricardo Coelho

Figure 15 - Request to use the project

Boa tarde
Da minha parte não há impedimento.
Boa sorte na tese.
Cumprimentos

Figure 16 - Authorization from teacher

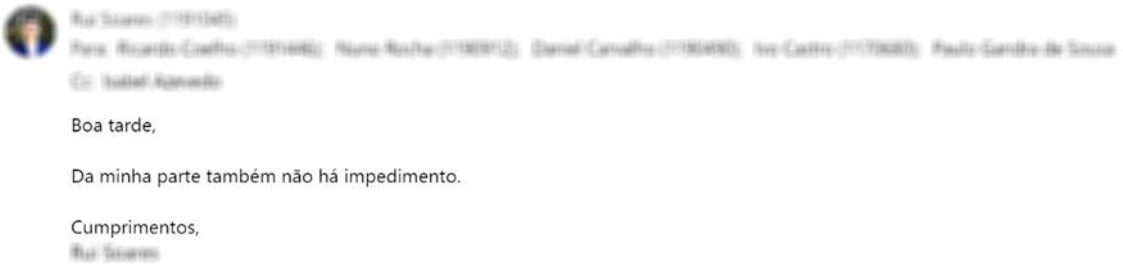


Boa tarde

Por mim não existe nenhum problema

Cumprimentos,

Figure 17 - Authorization from student 1



Boa tarde,

Da minha parte também não há impedimento.

Cumprimentos,

Figure 18 - Authorization from student 2

Boa tarde,

Do meu lado, tudo ok.

Cumprimentos,

Figure 19 - Authorization from student 3

Boa tarde,

Da minha parte não há impedimento.

Cumprimentos,

Figure 20 - Authorization from student 4

Appendix B

This appendix describes how the author selected the project used for the experiment. The Analytical hierarchy process (AHP) was used to evaluate projects carefully against specific criteria.

The AHP method is a powerful tool that simplifies complex decisions. It does this by breaking them down into parts like the goal, criteria, sub-criteria, and alternatives. These parts are then evaluated and ranked through pairwise comparisons using a rating scale and their priority weights. The method ensures reliable judgments through consistency checks and aggregates the final scores. This aggregation is key, allowing the method to rank the alternatives and ultimately lead to the best decision outcome [53].

AHP criteria evaluation

The first step in the AHP method is to define the criteria, and the following three criteria were chosen:

- **C1 - Complexity**: This criterion assesses how difficult and complex the project is. A project with the right level of complexity ensures that the experiment will be challenging enough to provide valuable insights into the capabilities of the Ballerina language compared to Java.
- **C2 - Decent documentation**: This criterion evaluates the project's documentation. Good documentation is essential for understanding the project's structure, dependencies, and features, making it easier to implement and migrate the project.
- **C3 - Project adequacy and familiarization**: This criterion assesses the project's suitability for the experimental objectives and the author's familiarity with it. A project that aligns closely with the experimental goals and is well-known to the author can be executed more effectively and efficiently.

Pairwise comparison and weight assignment

Now each criterion is going to be assigned a priority level, following the Saaty fundamental scale, represented in Table 12 [53].

Table 12 - Saaty fundamental scale
From [54]

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 favoured, and its dominance is demonstrated in practice
9	Absolute importance	The evidence favouring 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 has been created, as illustrated in Table 13.

Table 13 - Criteria comparison matrix

	C1	C2	C3
C1	1.000	0.333	0.200
C2	3.000	1.000	0.333
C3	5.000	3.000	1.000
Sum	9.000	4.333	1.533

The next phase of the Analytic Hierarchy Process (AHP) consists of normalizing the values from the previous matrix and calculating the priority vector. To normalize these values, each one is divided by the sum of its respective column, and then the arithmetic mean of the normalized values in each row is calculated to determine the priority vector, as shown in Table 14 [53].

Table 14 - Criteria normalized comparison matrix and priority vector

	C1	C2	C3	Priority Vector
C1	0.111	0.077	0.130	0.106
C2	0.333	0.231	0.217	0.260
C3	0.555	0.692	0.652	0.633

Once the normalized comparison matrix has been established, the next step is calculating the Consistency Ratio (CR). This measure guarantees a satisfactorily consistent priority vector. For the vector to be considered consistent, the CR value must not exceed 0.1 [53].

The Consistency Ratio (CR) is determined to assess the consistency of the judgments. A CR value below 0.1 signifies that the judgments can be considered reliable. The CR is computed by

dividing the Consistency Index (CI) by the Random Index (RI) [53]. The formula for calculating the CR is as follows:

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

Table 15 indicates the value of the Random Index (RI). The appropriate RI value is determined based on the number of criteria employed in the analysis. Given that three criteria are utilized, the selected RI value is 0.58.

Table 15 - Random consistency index
From [54]

Nº of criteria	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**…**
Index value	0.00	0.00	**0.58**	0.90	1.12	1.24	1.32	1.41	1.45	1.49	…

To calculate the CI value, the following formula is used:

$$CI = \frac{\lambda\max\ -\ \mathrm{n}}{\mathrm{n} - 1}$$

To obtain the λ_{max} value, another equation is used:

$$Ax = \lambda\max\ x$$

The *A* value corresponds to the comparison matrix, while *x* represents the priority vector since their values were previously calculated in Table 14. Now it is necessary to multiply the *A* matrix with *x* vector and obtain the values in Table 16.

Table 16 - Consistency matrix comparison

C1	0.319
C2	0.789
C3	1.943

With those values from Table 16 the λ_{max} can be calculated:

$$\lambda\max = \frac{\frac{0.319}{0.106} + \frac{0.789}{0.260} + \frac{1.943}{0.633}}{3} \approx 3.038$$

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

$$CI = \frac{3.038 - 3}{3 - 1} \approx 0.019$$

The value of the CR can be calculated using both the CI and the RI. By replacing the values in the formula, an approximate value of 0.033 is obtained, which is less than 0.1, which means that the priority values are reliable.

$$CR = \frac{\text{CI}}{\text{RI}} = \frac{0.019}{0.58} \approx 0.033$$

Since the values used are reliable and consistent. Table 17 displays the official criteria weights obtained from the AHP criteria weight process. These weights were subsequently utilized in the evaluation of each project, highlighting their importance.

Table 17 - AHP criteria weights

Criteria	Weight
C1	0.106
C2	0.260
C3	0.633

AHP project evaluation

A GitHub repository listing projects with a microservice architecture was consulted to find open-source projects for the experiment [55]. The first three Java projects in that repository were chosen. A fourth project was added since it was developed by a group that included the author. This relevant project has substantial Java implementation, which makes it a good candidate for the experiment.

The selected projects were:

- **P1 -** Beer catalog [56].
- **P2 -** Blog post [57].
- **P3 -** CQRS Microservice Application [58].
- **P4 -** Park Management System [44].

With the weights for all criteria now defined, the selected projects will undergo a rating process. Each project will be evaluated on a scale from 1 to 9 for each criterion, with 1 representing the worst value and 9 the best value. This process, as detailed in Table 18, will provide a comprehensive assessment of the projects, ensuring an informed evaluation.

Table 18 - Projects rating by criteria

	C1	C2	C3
P1	5	7	6
P2	7	8	7
P3	7	6	7
P4	8	8	8

AHP results

By analysing the data in Table 18 and applying the corresponding criteria weights, we can calculate each project's total score. The project with the highest score will be the most suitable for the experiment.

Table 19 - Project total scores

	Total score
P1	6.148
P2	7.253
P3	6.733
P4	**7.992**

As evidenced in Table 19, the **Park Management System** [44] project has emerged with the highest score of 7.992, positioning it as the ideal project choice for the experiment.

Appendix C

This appendix presents the general workflow of the PMS project in Figure 21.

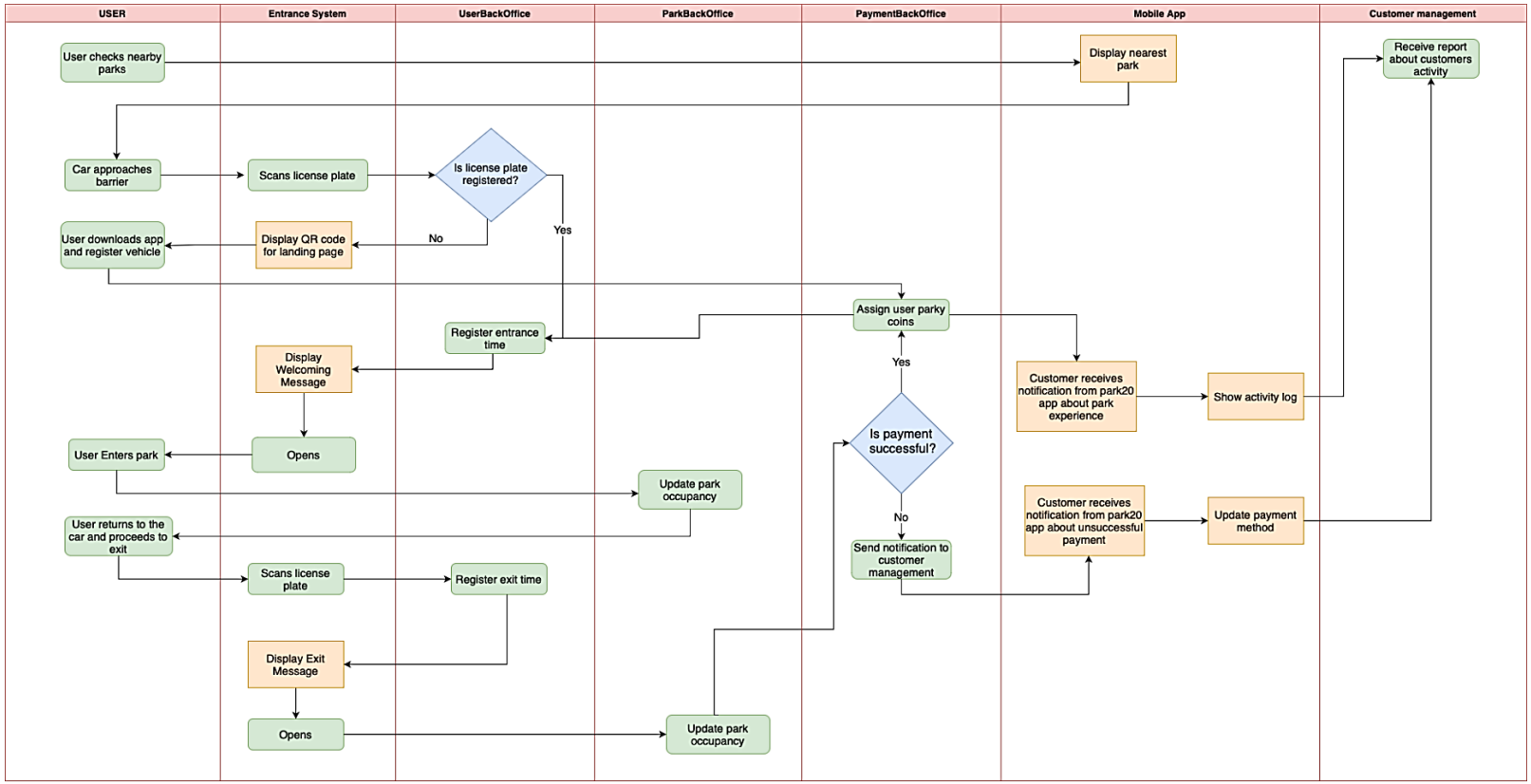


Figure 21 - General workflow of the system

Appendix D

In this appendix, the domain model is presented by aggregate root, since the complete domain model is too large to be presented.

Figure 22 represents the User aggregate from the domain model of the PMS. As mentioned in 4.1.4, this aggregate contains all the essential information about the users.

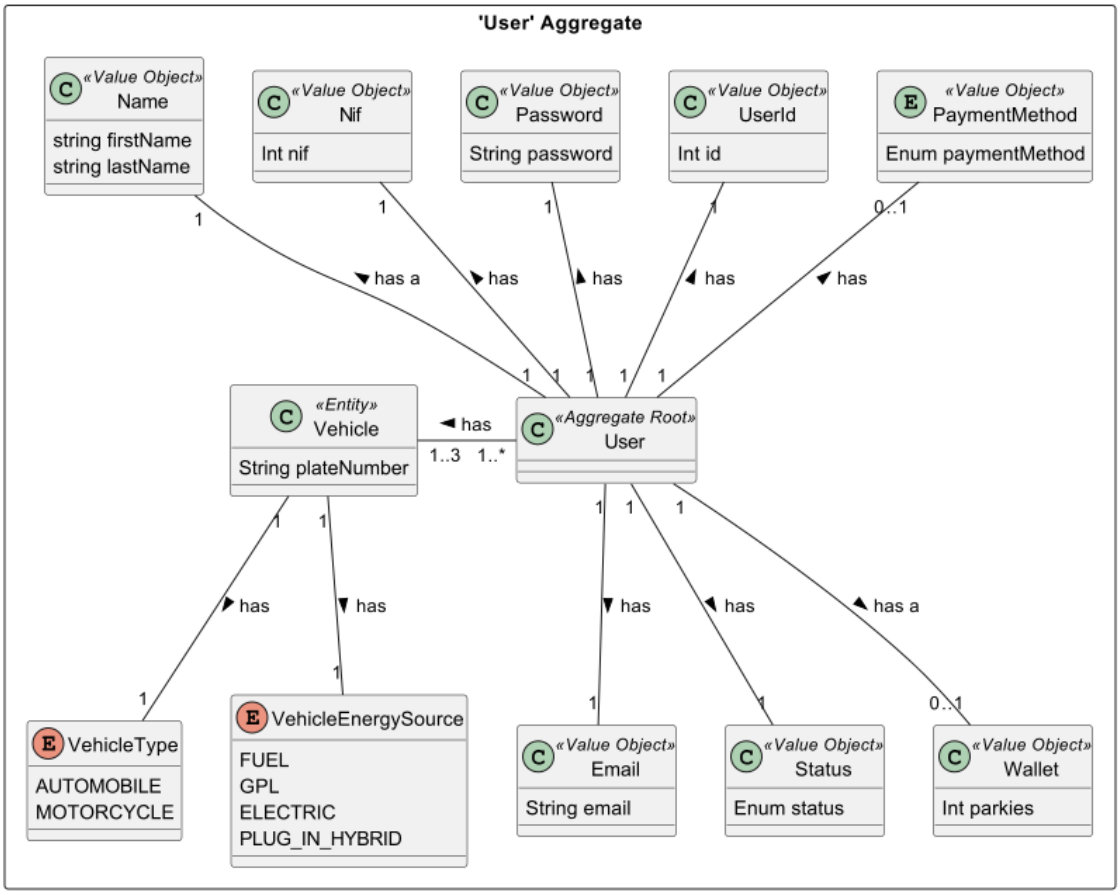


Figure 22 - User aggregate

Figure 23 shows the Park aggregate root, similar to what happens in the User aggregate. It contains all the information about the park20 company's parks.

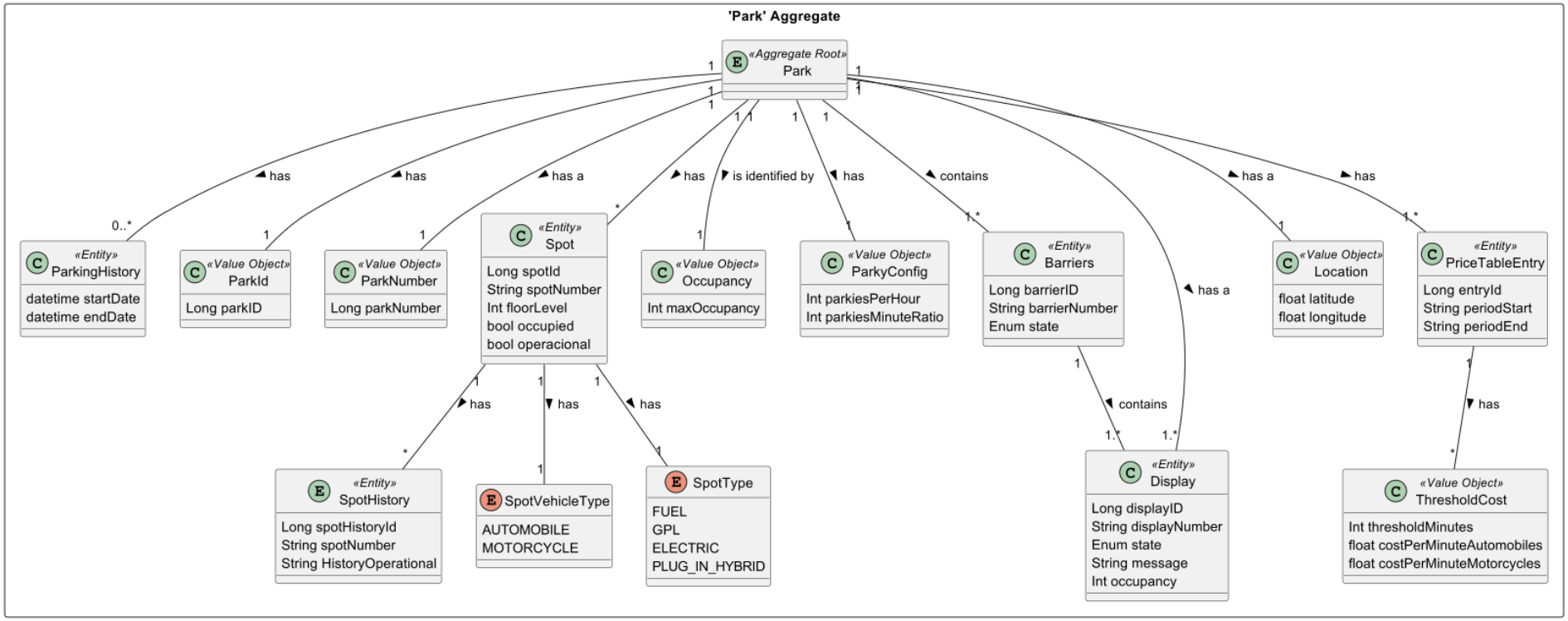


Figure 23 - Park aggregate

Furthermore, Figure 24 shows another aggregate root, Payments. This root handles all payment transactions within the system.

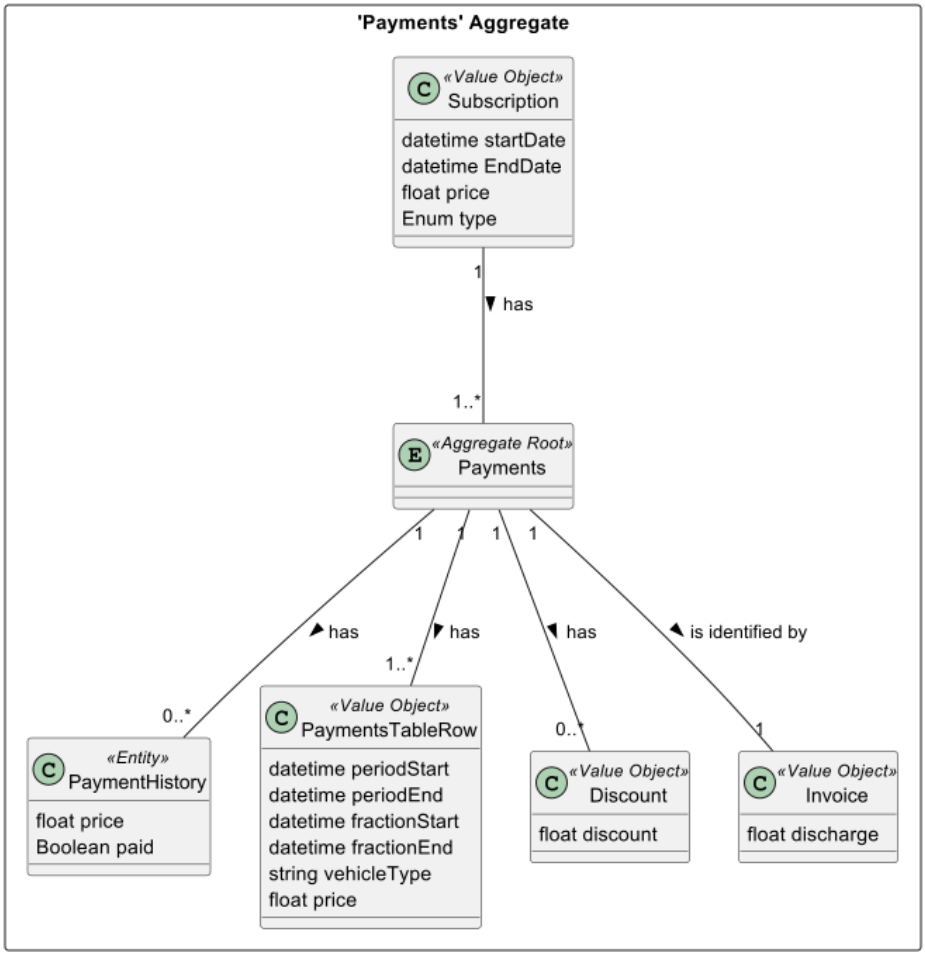


Figure 24 - Payments aggregate

Appendix E

This appendix presents the results of the load test experiments in eight tables, organized by technology (Ballerina or Java) and compilation method (Default or AOT). Each table corresponds to a specific combination of technology and compilation method used in the API Gateway service.

Within each table, there is a column indicating the technology used for the User BO microservice. The tables also differentiate test configurations: a configuration value of 1 represents tests conducted with 10 virtual users, while a configuration value of 2 indicates tests conducted with 100 virtual users. Additionally, the tables include the HTTP protocol used and provide all relevant performance metrics collected during the tests, ensuring an error rate of 0%.

The first two tables, Table 20 and Table 21 present the performance results retrieved from the API Gateway developed in Ballerina with AOT compilation.

Table 20 - Load test results from API Gateway developed in Ballerina AOT (Configuration 1)

User BO Technology	HTTP Protocol	Average (ms)	Max (ms)	Median (ms)	Min (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Ballerina AOT	1.1	7.53	17.39	5.54	2.60	14.21	15.68	9.86
Ballerina AOT	**2**	**7.06**	**15.10**	**6.36**	**2.62**	**10.95**	**12.82**	**9.91**
Ballerina	1.1	9.63	21.91	10.43	3.11	13.76	18.35	9.84
Ballerina	2	8.57	20.86	7.21	3.20	15.35	16.99	9.89
Java AOT	1.1	17.74	32.16	16.89	11.67	22.55	26.74	9.71
Java AOT	2	16.83	26.58	16.23	11.12	21.98	22.65	9.80
Java	1.1	19.47	48.72	16.23	12.38	25.93	36.37	9.72
Java	2	17.57	31.95	16.01	10.18	24.47	27.14	9.77

Table 21 - Load test results from API Gateway developed in Ballerina AOT (Configuration 2)

User BO Technology	HTTP Protocol	Average (ms)	Max (ms)	Median (ms)	Min (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Ballerina AOT	1.1	11.98	47.84	9.62	2.61	23.21	26.79	98.82
Ballerina AOT	**2**	**11.28**	**45.66**	**9.31**	**2.06**	**21.58**	**23.58**	**98.90**
Ballerina	1.1	13.30	39.55	12.06	3.10	22.92	28.24	98.28
Ballerina	2	12.43	37.59	11.19	3.09	22.12	24.94	98.29
Java AOT	1.1	37.31	397.75	29.11	10.36	54.83	70.40	95.23
Java AOT	2	36.19	384.36	26.78	10.35	50.90	65.62	96.74
Java	1.1	22.29	156.76	16.38	10.57	33.00	37.66	96.87
Java	2	20.25	138.57	16.08	9.69	28.26	36.05	97.20

Table 22 and Table 23 present the results from the API gateway developed in Ballerina with its default compilation.

Table 22 - Load test results from API Gateway developed in Ballerina (Configuration 1)

User BO Technology	HTTP Protocol	Average (ms)	Max (ms)	Median (ms)	Min (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Ballerina AOT	1.1	9.02	14.63	8.76	4.16	12.81	13.44	9.85
Ballerina AOT	**2**	**8.56**	**13.82**	**7.84**	**4.98**	**12.61**	**13.31**	**9.87**
Ballerina	1.1	10.38	16.55	10.07	6.87	14.49	15.17	9.80
Ballerina	2	9.46	16.13	9.04	6.42	11.84	12.81	9.83
Java AOT	1.1	19.61	31.69	18.04	13.51	28.09	29.46	9.72
Java AOT	2	17.27	48.73	15.97	13.19	19.07	20.27	9.78
Java	1.1	20.75	54.13	16.22	12.07	42.22	46.43	9.68
Java	2	19.48	51.53	15.08	11.29	43.90	46.77	9.75

Table 23 - Load test results from API Gateway developed in Ballerina (Configuration 2)

User BO Technology	HTTP Protocol	Average (ms)	Max (ms)	Median (ms)	Min (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Ballerina AOT	1.1	13.37	35.39	12.17	3.09	22.92	26.94	98.80
Ballerina AOT	**2**	**12.76**	**40.43**	**11.85**	**3.12**	**22.00**	**25.55**	**98.85**
Ballerina	1.1	14.88	38.60	14.12	4.49	23.03	24.98	98.04
Ballerina	2	13.68	39.79	12.98	2.80	21.38	25.37	98.16
Java AOT	1.1	39.56	385.78	30.73	10.49	52.88	70.14	93.83
Java AOT	2	38.68	400.90	27.12	10.72	58.74	68.67	96.06
Java	1.1	26.66	185.45	21.04	12.17	38.71	49.65	96.64
Java	2	24.84	132.41	23.09	9.85	35.35	38.61	96.71

On Table 24 and Table 25 is presented the results from the original API Gateway developed in Java with AOT compilation.

Table 24 - Load test results from API Gateway developed in Java AOT (Configuration 1)

User BO Technology	HTTP Protocol	Average (ms)	Max (ms)	Median (ms)	Min (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Ballerina AOT	1.1	9.14	15.10	8.16	2.59	14.55	14.62	9.85
Ballerina AOT	**2**	**8.40**	**22.65**	**8.70**	**2.06**	**12.85**	**14.25**	**9.87**
Ballerina	1.1	10.09	17.24	9.69	3.14	14.27	15.93	9.80
Ballerina	2	9.25	18.44	9.00	3.64	13.35	16.04	9.82
Java AOT	1.1	18.65	46.18	18.61	13.66	21.25	22.83	9.70
Java AOT	2	17.98	53.96	14.11	11.43	29.05	35.07	9.74
Java	1.1	20.50	43.37	19.09	11.01	28.70	42.79	9.68
Java	2	18.63	41.83	16.50	9.85	28.77	31.98	9.73

Table 25 - Load test results from API Gateway developed in Java AOT (Configuration 2)

User BO Technology	HTTP Protocol	Average (ms)	Max (ms)	Median (ms)	Min (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Ballerina AOT	1.1	12.41	42.98	10.35	1.55	24.44	28.02	62.33
Ballerina AOT	**2**	**11.57**	**48.12**	**8.02**	**2.11**	**24.74**	**31.00**	**70.91**
Ballerina	1.1	13.70	79.02	11.66	3.11	20.65	27.14	70.31
Ballerina	2	12.58	35.00	10.72	2.60	23.32	28.69	74.13
Java AOT	1.1	39.53	348.10	32.56	9.27	63.31	77.18	94.36
Java AOT	2	37.04	369.60	27.97	9.89	57.52	74.26	94.82
Java	1.1	26.26	147.60	22.48	10.91	35.26	43.81	96.75
Java	2	23.56	93.51	22.21	10.81	33.37	41.41	97.03

Finally, Table 26 and Table 27 present the results from the original API gateway developed java in with its default compilation.

Table 26 - Load test results from API Gateway developed in Java (Configuration 1)

User BO Technology	HTTP Protocol	Average (ms)	Max (ms)	Median (ms)	Min (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Ballerina AOT	1.1	9.86	33.56	9.34	4.76	14.37	23.27	9.82
Ballerina AOT	**2**	**9.00**	**18.69**	**8.77**	**4.49**	**13.29**	**14.79**	**9.86**
Ballerina	1.1	11.16	20.18	11.10	4.35	17.49	18.64	9.78
Ballerina	2	10.21	25.00	8.85	4.16	16.78	20.52	9.79
Java AOT	1.1	20.57	55.49	16.36	11.81	31.84	44.82	9.67
Java AOT	2	19.56	52.26	16.64	11.51	29.04	35.59	9.72
Java	1.1	21.84	35.83	21.22	12.67	28.13	34.02	9.67
Java	2	20.38	38.75	20.39	12.40	25.49	29.35	9.70

Table 27 - Load test results from API Gateway developed in Java (Configuration 2)

User BO Technology	HTTP Protocol	Average (ms)	Max (ms)	Median (ms)	Min (ms)	90% Line (ms)	95% Line (ms)	Throughput (req/s)
Ballerina AOT	1.1	14.03	54.50	11.75	2.97	24.15	39.11	59.66
Ballerina AOT	**2**	**13.30**	**53.44**	**12.67**	**2.58**	**20.98**	**23.57**	**67.62**
Ballerina	1.1	15.59	66.02	13.51	3.10	23.45	34.11	56.23
Ballerina	2	15.40	57.68	14.07	4.37	20.35	29.92	64.29
Java AOT	1.1	40.76	471.07	31.33	9.86	63.16	92.52	93.76
Java AOT	2	40.64	342.63	33.59	8.27	64.89	71.64	94.19
Java	1.1	27.71	140.91	23.24	11.88	42.36	54.81	96.62
Java	2	26.92	128.82	24.18	12.26	35.56	39.71	96.71

By analysing the results presented in Table 20, Table 21, Table 22, Table 23, Table 24, Table 25, Table 26 and Table 27. A bold highlight is present in each table, indicating the best result for the tested configuration. Overall, the results clearly show that Ballerina generally outperforms Java across all evaluated configurations.

Specifically:

- **Under low-load conditions (10 simultaneous users)**: Ballerina consistently exhibits superior performance in both response times and throughput when compared to Java.
- **Under high-load conditions (100 simultaneous users)**: Ballerina maintains better response times than Java, even though Java shows higher throughput when the Gateway API is implemented in Java. This suggests that while Java might handle raw throughput more effectively in certain scenarios, Ballerina's efficiency in managing response times remains better, particularly in heterogeneous environments where services are implemented using various technologies, which explains why Ballerina is known as a language optimised for integration [59].

These findings indicate that Ballerina is particularly well-suited for high-load scenarios involving communication across services built with different technologies, where maintaining low response times is critical.