INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

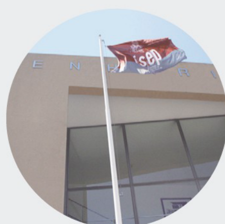
isep

Scaling Friendzone: From an MVP to Large Scale Production

LUÍS MANUEL DA SILVA NEVES
Setembro de 2024

POLITÉCNICO
DO PORTO

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA



Scaling Friendzone: From an MVP to Large Scale Production

LUIS MANUEL DA SILVA NEVES
September of 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Scaling Friendzone: From an MVP to Large Scale Production

Luís Manuel da Silva Neves

Master´s Degree Dissertation in Computer Engineering

Specialization in Software Engineering

Supervisor: Hélder Pinto

Porto, September 2024

Acknowledgments

I would like to thank my supervisor, Hélder Pinto, for his guidance in writing this thesis and to every professor who taught me something along the way, making this whole project possible.

To the ones who loved and supported me throughout the years

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

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

Therefore, the work presented in this document is original and authored by me, having not previously been used for any other end.

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

ISEP, Porto, 15 de setembro de 2024

Resumo

Esta tese acompanha a evolução da Friendzone, desde o Produto Mínimo Viável (MVP) até um ambiente de produção escalável, analisando estrategicamente os elementos essenciais para essa transição. São abordadas estratégias específicas de escalabilidade, como otimização de desempenho através de técnicas de caching e load balancing, visando distribuir a carga entre servidores e reduzir a latência. Adicionalmente, a tese analisa a migração de uma arquitetura monolítica para microserviços, com o objetivo de melhorar a modularidade e permitir uma gestão mais eficiente dos componentes à medida que a base de utilizadores cresce.

Testes rigorosos são conduzidos antes e depois da implementação das estratégias de escalabilidade e da migração para microserviços, incluindo avaliações de desempenho e testes de carga. Esses testes medem o impacto das melhorias na capacidade da Friendzone de lidar com um aumento significativo na atividade dos utilizadores. Estratégias de planeamento de recursos e medidas de fiabilidade, incluindo padrões de segurança, são implementadas para garantir a integridade da plataforma durante o crescimento.

Por fim, uma avaliação abrangente é realizada para analisar o impacto dessas mudanças na eficiência operacional da plataforma. A documentação das melhorias, processos e melhores práticas assegura que as lições aprendidas possam ser replicadas no futuro. A análise final conclui que a Friendzone evoluiu para um sistema robusto e escalável, capaz de atender a uma vasta base de utilizadores com eficiência e segurança.

Keywords: Estratégias de Escalabilidade, Otimização de Desempenho, Arquitetura de Sistema

Abstract

This thesis traces the evolution of Friendzone from its inception as a Minimum Viable Product (MVP) to a fully scalable production environment, strategically analyzing the essential elements for this transition. Initially, the research methodology establishes the criteria, sources, and key terms used to investigate Friendzone's progression. The 'State of the Art' section reviews existing testing practices, scaling methodologies, industry standards, and social media metrics, forming a foundation for further exploration.

The thesis critically examines Friendzone's technical architecture, pinpointing areas requiring enhancement to effectively manage large-scale demands. It outlines specific strategies for scalability, including performance optimization through caching and load balancing, aimed at distributing server load and minimizing latency. A key focus is the migration from a monolithic architecture to a microservices approach, enhancing modularity and facilitating more efficient management of components as the user base expands.

Rigorous testing, including performance evaluations and load testing, is conducted both before and after implementing scalability strategies and the transition to microservices. These tests measure the impact of improvements on Friendzone's capacity to handle a significant increase in user activity. Resource planning and reliability measures, such as security standards, are implemented to ensure platform integrity during growth.

Comprehensive documentation of the scalability enhancements, processes, and best practices is provided to ensure that the lessons learned can be replicated in future endeavors. The final evaluation analyzes the impact of these changes on the platform's operational efficiency and user experience. The analysis concludes that Friendzone has evolved into a robust, scalable system capable of efficiently and securely serving a large user base.

Keywords: Scalability Strategies, Performance Optimization, System Architecture

Index

Acknowledgments ..... *iii*

Statement of Integrity ..... *v*

Resumo ..... *vii*

Abstract ..... *ix*

Index ..... *xi*

Figures List ..... *xv*

Tables List ..... *xvii*

Code Snippets List ..... *xix*

Acronyms and Symbols ..... *xxi*

Acronyms List xxi

1. Introduction *1*

1.1. Context **1**

1.1.1. What is Friendzone? 1

1.1.2. Scaling Friendzone 1

1.2. Problem Statement **2**

1.3. Objectives **3**

1.4. Ethical Concerns **4**

1.5. Research Questions **4**

1.5.1. Research Hypotheses 5

1.6. Task Chronogram **7**

1.7. Report Structure **8**

2. Research Method *10*

2.1. Literature Review **10**

2.1.1. Inclusion and Exclusion Criteria 10

2.1.2. Information Sources 11

2.1.3. Research Terms 12

2.1.4. Selection Process ... 12

2.2. Research Method Outcome ... 13

2.3. Chapter Summary ... 15

3. State of the Art ... 16

3.1. Tests and Scaling Methods ... 16

3.1.1. Overview of Testing Methods ... 16

3.1.2. Current Industry Standards and Best Practices in Testing ... 18

3.1.3. Apache JMeter in Performance and Load Testing ... 19

3.1.4. Scaling Methods and Techniques ... 20

3.1.5. Tests and Scaling Methods Summary ... 25

3.2. Digital Marketing and Metric Analysis on Social Media ... 26

3.2.1. Basics of Digital Marketing on Social Media ... 26

3.2.2. Crucial Metrics in Social Media Analysis ... 26

3.2.3. Significance of Metric Analysis ... 27

3.2.4. Digital Marketing and Metric Analysis on Social Media Summary ... 27

3.3. Monetized Gamification ... 28

3.3.1. Understanding Monetized Gamification ... 28

3.3.2. Friendzone's Challenge Feature ... 28

3.3.3. Metrics for Monetized Gamification Analysis ... 28

3.3.4. Importance of Monetized Gamification in Social Networks ... 29

3.3.5. Monetized Gamification Summary ... 29

3.4. Chapter Summary ... 30

4. Solution Analysis, Design and Tests ... 31

4.1. Domain Model ... 31

4.2. Current Tech Stack ... 33

4.3. Design ... 33

4.3.1. Level 1 – Context ... 34

4.3.2. Level 2 – Container ... 35

4.3.3. Level 3 – Component ... 38

4.4. Testing the current system ... 39

4.4.1. Testing Environment ... 39

4.4.2. Apache JMeter ... 40

4.5. Improvement Proposal ... 49

4.5.1. Design ... 49

5. Implementation ... 56

5.1. New Tech Stack ... 56

5.2. New Codebase ... 57

5.2.1. Read and Write Service Configuration ... 58

5.2.2. Gateway ... 60

5.2.3. Eureka Load Balancer ... 64

5.2.4. RabbitMQ Broker ... 65

5.2.5. Redis Cache ... 68

5.2.6. Orchestrating with Docker-Compose ... 69

5.2.7. Deployed Result ... 74

5.3. Load Testing ... 75

5.3.1. 50 Concurrent Users ... 76

5.3.2. 1000 Concurrent Users ... 79

5.3.3. 5000 Concurrent Users ... 81

5.3.4. 10 000 Concurrent Users ... 84

6. Review, Impact and Answers ... 86

6.1. Improvement Impact and Review ... 86

6.1.1. Read Operations ... 86

6.1.2. Write Operations ... 89

6.2. Research Questions and Answers ... 92

6.2.1. Research and Test Based Answers ... 92

7. Conclusion ... 96

7.1. Results ... 96

7.2. Limitations and Solution Validity Threats ... 97

7.3. Future Work ... 97

7.4. Final Statement ... 98

References ... 100

Attachments ... 102

Figures List

Figure 1 - Chronogram Table | Image By Luís Neves ... 7
Figure 2 - Selection Process Diagram | Image By Luís Neves ... 12
Figure 3 - Friendzone Domain Model ... 31
Figure 4 - Level 1 Diagram ... 34
Figure 5 - Level 2 Logic View Diagram ... 35
Figure 6 - Level 2 Process Diagram ... 36
Figure 7 - Level 2 Physical Diagram ... 37
Figure 8 - Level 3 Logic Diagram ... 38
Figure 9 - Docker Container Available Resources ... 39
Figure 10 - 50 Concurrent Users Summary Report – Read Operation ... 40
Figure 11 - 50 Concurrent Users Response Time Graph | 100ms interval ... 41
Figure 12 - 50 Concurrent Users Summary Report – Write Operation ... 41
Figure 13 - 50 Concurrent Users Response Time Graph – Write Operation ... 42
Figure 14 - 1000 Concurrent Users Summary Report - Read Operation ... 43
Figure 15 - 1000 Concurrent Users Response Time Graph - Read Operation ... 43
Figure 16 - 1000 Concurrent Users Summary Report - Write Operations ... 44
Figure 17 - 1000 Concurrent Users Response Time Graph - Write Operations ... 44
Figure 18 - 5000 Concurrent Users Summary Report ... 46
Figure 19 - 5000 Concurrent Users Response Time Graph | 100ms interval ... 46
Figure 20 - 5000 Concurrent Users Summary Report - Write Operations ... 47
Figure 21 - 5000 Concurrent Users Response Time Graph - Write Operations ... 47
Figure 22 - 10 000 Concurrent Users Summary Report ... 48
Figure 23 - 10 000 Concurrent Users Response Time Graph | 100 ms Interval ... 48
Figure 24 - Level 2 Logic View Diagram - Improved ... 50
Figure 25 - Level 2 Process View Diagram | Read Operations - Improved ... 51
Figure 26 - Level 2 Process View Diagram | Write Operations - Improved ... 52
Figure 27 - Level 2 Physical View - Improved ... 54
Figure 28 - Level 2 Physical View - Improved ... 54
Figure 29 - Eureka Registered Servicesredis: ... 74
Figure 30 - 50 Concurrent Users Summary Report | Read Operations | Upgraded ... 76
Figure 31 - 50 Concurrent Users Response Time Graph | Read Operations | Upgraded ... 76
Figure 32 - 50 Concurrent Users Summary Report | Write Operations | Upgraded ... 77
Figure 33 - 50 Concurrent Users Response Time Graph | Write Operations | Upgraded ... 78
Figure 34 - 1000 Concurrent Users Summary Report | Read Operations | Upgraded ... 79

Figure 35 - 1000 Concurrent Users Response Time Graph | Read Operations | UpgradedFigure 79
Figure 36 - 1000 Concurrent Users Summary Report | Write Operations | Upgraded 80
Figure 37 - 1000 Concurrent Users Response Time Graph | Write Operations | Upgraded 80
Figure 38 - 5000 Concurrent Users Summary Report | Read Operations | Upgraded 81
Figure 39 - 5000 Concurrent Users Response Time Graph | Read Operations | Upgraded 82
Figure 41 - 5000 Concurrent Users Response Time Graph | Write Operations | Upgraded 83
Figure 40 - 5000 Concurrent Users Summary Report| Write Operations | Upgraded 83

Tables List

Table 1 - Research Questions and Metrics 5
Table 2 - Inclusion and Exclusion Criteria 10
Table 3 - Research Method Result.................... 13
Table 4 - Read Operations Comparison Table 87
Table 5 - Write Operations Comparison Table 89

Code Snippets List

Code Snippet 1 - Challenge Service Read Main Class 58
Code Snippet 2 - Challenge Service Read Pom.XML Main Dependencies 59
Code Snippet 3 - Challenge Service Read application.yml file 60
Code Snippet 4 - Gateway Main Class 61
Code Snippet 5 - Gateway Application.yml file 63
Code Snippet 6 - Eureka Main Class 64
Code Snippet 7 - Eureka Application.yml file 65
Code Snippet 8 - RabbitMQ Queue and Exchange Setup 66
Code Snippet 9 - RabbitMQ Binding with Routing Keys 66
Code Snippet 10 - RabbitMQ Listener 67
Code Snippet 11 - Challenge Service Write - Create Challenge Method 67
Code Snippet 12 - Redis Configuration Class 68
Code Snippet 13 - Challenge Method with Caching Enabled 68
Code Snippet 14 - Docker-Compose Eureka Configuration 69
Code Snippet 15 - Docker-Compose Gateway Configuration 69
Code Snippet 16 - Docker-Compose Challenge Service Read Configuration 70
Code Snippet 17 - Docker-Compose Challenge Write Service Configuration 71
Code Snippet 18 - Docker-Compose Challenge Write Database Configuration 71
Code Snippet 19 - Docker-Compose Challenge Read Database Configuration 72
Code Snippet 20 - Docker-Compose RabbitMQ Configuration 73
Code Snippet 21 – Docker-Compose Redis Cache Configuration 74

Acronyms and Symbols

Acronyms List

AI	Artificial Intelligence
CDN	Content Delivery Network
CQRS	Command Query Responsibility Segregation
ICT	Information and Communications Technology
IEEE	Institute of Electrical and Electronics Engineers
KPI	Key Performance Indicator
LLM	Large Language Model
MaSST	Metrics and Standards for Software Testing
MDPI	Multidisciplinary Digital Publishing Institute
MVP	Minimum Viable Product
PoC	*Proof of Concept*
SUT	Software Under Test

1. Introduction

This chapter provides context to the theme, which will be further explored in the following chapters of the state of the art, presenting the justification and motivation that lead to the realization of this research study. It also frames the dissertation within the scope of the academic program, highlighting the key courses that contributed to its development, such as "Integração de Sistemas" (INSIS), "Software Architecture" (ARQSOFT), and "Quality in Software Engineering" (QESOFT). These courses laid the foundational knowledge in system integration, architectural design, and quality assurance that were crucial for addressing the research objectives. Next, the chapter characterizes and refers to the objects and problem of the theme, addressing the objectives to achieve. Finally, it presents the temporal organization of the work developed and a description of the structure of this document.

1.1. Context

1.1.1. What is Friendzone?

Friendzone is a social network that revolves around challenges between its users. In this platform, users can challenge each other to tasks or activities. Here's how it works:

A user can challenge someone for an amount of tokens, called FZ, (e.g. “I challenge you to do a cartwheel in public for 100 FZs”), and if the challenged user accepts, he has 24 hours to respond to it. By completing the challenges, the challenged user receives the agreed amount of tokens, and the challenge becomes a piece of content on the platform.

Aside from challenges, Friendzone offers regular social networking features like stories, direct messages, likes and comments. There's also an option for 'Hidden Challenges,' visible only to those involved, adding a bit of mystery.

1.1.2. Scaling Friendzone

Scaling software systems is a complex task that requires careful consideration of various aspects. "Performance and scalability testing and measurements of cloud-based software services are necessary for future optimizations and growth of cloud computing. Scalability, elasticity, and efficiency are interrelated aspects of cloud-based software services’ performance requirements" (Al-Said Ahmad & Andras, 2019). This emphasizes the challenges faced when scaling software, including server downtime, slow response times, database bottlenecks, and unpredictable traffic spikes. This evolution is reflected in the development of the Friendzone application, which began as a PoC and has been carefully refined into a functional MVP. Currently,

Friendzone operates with approximately 50 users, but as it prepares for significant growth, the primary challenge lies in scaling the platform to support the next round of testing, which will have up to 10,000 concurrent users.

The thesis presented here is an in-depth exploration of the challenges involved in this transitional phase, focusing on the strategies, methodologies, and innovative solutions necessary to evolve Friendzone from a functional minimum viable product into a refined system optimized for large-scale production. The study rigorously evaluates the impact of the implemented changes by testing them with the current user base and simulating the anticipated load of 10,000 concurrent users, assessing both their effectiveness and their transformative potential on the platform's overall functionality and performance.

1.2. Problem Statement

In the industry of software development, the shift from an MVP to a scalable application is a crucial milestone. In fact, "The trouble with scaling any system is that, once we actually start down the path of growing it, inevitably run into some hidden complexities" (Eklund et al., 2014). This observation highlights the often-overlooked challenges of scaling a system, such as dealing with architectural limitations, centralization issues, and the need for efficient communication processes.

For Friendzone, these challenges are particularly pressing. Currently, the platform operates with approximately 50 active users, but the company anticipates rapid growth, aiming to support up to 10,000 concurrent users in the near future. This growth is driven by increasing user demand and the platform's expanding reputation and marketing efforts. However, the central problem that Friendzone faces is ensuring that its current architecture can scale to meet this demand without compromising on performance, reliability, or user experience.

The urgency of this challenge is compounded by the fact that Friendzone operates under certain financial constraints. While there are partnerships in place that can help ensure server availability to a certain extent, these resources are not unlimited. This makes the need for an efficient and scalable architecture even more critical, as it must make the most of available resources while preparing for the increased load.

If Friendzone fails to address these architectural and scalability challenges now, the consequences could be severe. The platform risks not only a degradation in performance, leading to slow response times and potential downtime but also a possible decline in user satisfaction. In a competitive market, such issues could result in a loss of users for better-performing alternatives. Additionally, the inability to scale effectively could stifle the platform's growth, leading to a stagnation in user base expansion and a missed opportunity to capitalize on the growing demand.

Therefore, the core problem that this thesis addresses is how to architecturally evolve Friendzone from its current MVP state into a robust, scalable platform capable of handling 10,000 concurrent users. This evolution

must be accomplished within the constraints of limited financial resources and without compromising the user experience. The study will focus on identifying the necessary architectural enhancements, strategic optimizations, and scalability solutions that will enable Friendzone to meet its growth targets while maintaining operational efficiency and ensuring long-term success.

1.3. Objectives

The goal of this study is to find the best ways to scale Friendzone effectively. To do this, we'll take a close look at the technical details that are important for making the platform efficient and scalable. The objectives below will guide this work, focusing on practical strategies to help Friendzone grow while also being mindful of time and resources:

1. **Evaluate Current Architecture:**
    - Review the current monolithic architecture to identify areas that need improvement for large-scale operations. Plan the transition from a monolithic structure to a microservices-based architecture to enhance scalability.
2. **Test Current State of the application:**
    - Conduct a series of tests to benchmark the current performance of the application, establishing a baseline for comparison after scaling efforts.
3. **Develop Scalability Strategies:**
    - Create specific strategies to enable Friendzone to handle a significant increase in users and activities. This includes planning for the migration to microservices where it provides the most benefit.
4. **Performance Optimization:**
    - Identify and address performance bottlenecks to ensure that the application can maintain fast response times and operate efficiently under increased load.
5. **Performance Testing:**
    - Run comprehensive load tests to evaluate how the application performs under different user loads. Compare these results with the initial benchmarks to measure improvements.
6. **Ensure Reliability:**
    - Implement strategies to ensure the application remains reliable and available as it scales, particularly in a distributed microservices environment.
7. **Documentation and Best Practices:**
    - Keep thorough documentation of the improvements made and the best practices developed during the transition to a scalable architecture.

8. **Assess Impact:**

    - Re-run the initial tests and compare results to assess how the changes have impacted user experience and operational efficiency.

9. **Scaling Techniques Analysis:**

    - Evaluate different scaling techniques, including the shift to microservices, to determine the most effective methods for Friendzone's specific needs.

1.4. Ethical Concerns

In addressing ethical concerns, this project ensures that all phases adhere to the highest standards of ethical practice. The research process aligns with the Code of Good Practices and Conduct of P. Porto, specifically articles six, eight, and ten (IPP, 2020), and the software engineering process follows the ACM/IEEE-CS Software Engineering Code of Ethics (Gotterbarn, Miller, & Rogerson). These guidelines emphasize the importance of privacy, confidentiality, and integrity throughout the research and development processes.

Given the sensitive nature of user data in social networking platforms, particular emphasis is placed on privacy and confidentiality measures within the system's design and implementation. The company employs data anonymization techniques and strict access controls to protect user information, ensuring that personal data is handled securely and only accessible to authorized personnel. Additionally, data encryption is applied to safeguard information during transmission and storage, thereby minimizing the risk of unauthorized access.

As an employee of the company, I am committed to conducting all evaluations impartially and objectively. Measures are in place to prevent conflicts of interest, ensuring that the analysis and results presented in this dissertation are free from bias. This impartial approach is crucial to maintaining the integrity of the research findings and contributes to the reliability and trustworthiness of the platform as a whole. All findings and performance evaluations are reported transparently, aligning with the project's ethical commitment to fairness and responsibility toward all stakeholders involved.

1.5. Research Questions

This section presents the key research questions that will guide the research and analysis in this thesis. While the objectives have already been defined, these questions focus on the specific challenges of scaling the Friendzone platform. The purpose of addressing these questions is to identify effective strategies and techniques that will help ensure Friendzone can scale efficiently to meet the needs of its expanding user base. The answers will contribute to understanding scalable system design and provide practical insights for building a robust architecture for Friendzone and for replicating these techniques in any other system.

Table 1 - Research Questions and Metrics

Number	Questions	Metrics
1	What are the key challenges and considerations when designing a scalable architecture for social networking platforms with high user interaction and content generation?	Concurrent User Capacity
		Throughput
		Response Time
		System Availability (Uptime)
2	What are the best practices for implementing microservices architectures to support scalability, and how do these practices improve modularity and fault isolation compared to monolithic architectures?	Service Independence
		Inter-Service Latency
3	How can load balancing techniques be optimized to ensure both high availability and fault tolerance in social networking platforms with unpredictable traffic patterns?	Traffic Distribution Efficiency
		System Downtime Due to Overload
		Latency Variability
		Server Utilization Rate
4	How does caching improve the scalability and responsiveness of user-driven interactions in high-traffic social networking platforms?	Cache Hit Ratio
		Latency Reduction Due to Caching
		Backend Load Reduction
		Time to Live (TTL) Optimization

1.5.1. Research Hypotheses

In this section, to evaluate the functionalities and the behavior of the system, research hypotheses are created for each question described previously, so that the metrics defined can match the criteria and decide based on the evidence on the plausibility of the null hypothesis.

Q1: What are the key challenges and considerations when designing a scalable architecture for social networking platforms with high user interaction and content generation?

- **H0**: The current architecture and design of social networking platforms can handle high user interaction and content generation without requiring specific scaling considerations or architectural changes.
- **H1**: Designing a scalable architecture for social networking platforms with high user interaction and content generation requires significant changes, including specific scalability mechanisms such as load balancing, caching, microservices, and fault tolerance to manage system load efficiently.

Q2: What are the best practices for implementing microservices architectures to support scalability, and how do these practices improve modularity and fault isolation compared to monolithic architectures?

- **H0**: Implementing microservices architectures does not provide significant improvements in scalability, modularity, or fault isolation compared to monolithic architectures.
- **H1**: Implementing microservices architectures significantly improves scalability, modularity, and fault isolation compared to monolithic architectures through best practices such as service independence, API-based communication, and fault-tolerant design.

Q3: How can load balancing techniques be optimized to ensure both high availability and fault tolerance in social networking platforms with unpredictable traffic patterns?

- **H0**: Optimizing load balancing techniques does not significantly improve high availability and fault tolerance in social networking platforms with unpredictable traffic patterns.
- **H1**: Optimizing load balancing techniques significantly improves high availability and fault tolerance in social networking platforms, especially in environments with unpredictable traffic patterns.

Q4: How does caching improve the scalability and responsiveness of user-driven interactions in high-traffic social networking platforms?

- **H0**: Caching does not significantly improve the scalability and responsiveness of user-driven interactions in high-traffic social networking platforms.
- **H1**: Caching significantly improves the scalability and responsiveness of user-driven interactions in high-traffic social networking platforms by reducing database load and lowering response times.

1.6. Task Chronogram

The current thesis chronogram functions as a roadmap, outlining tasks and their deadlines. It aids in planning and monitoring progress. This visual tool helps identify potential delays and manage resources efficiently. By following this schedule, the thesis stays on track, ensuring timely completion. In essence, the chronogram is a vital aid for organizing and managing the thesis project.

As seen below in figure 1, an outline of the schedule is presented, from the start of the thesis conception, January 2024, to its completion, September 2024.

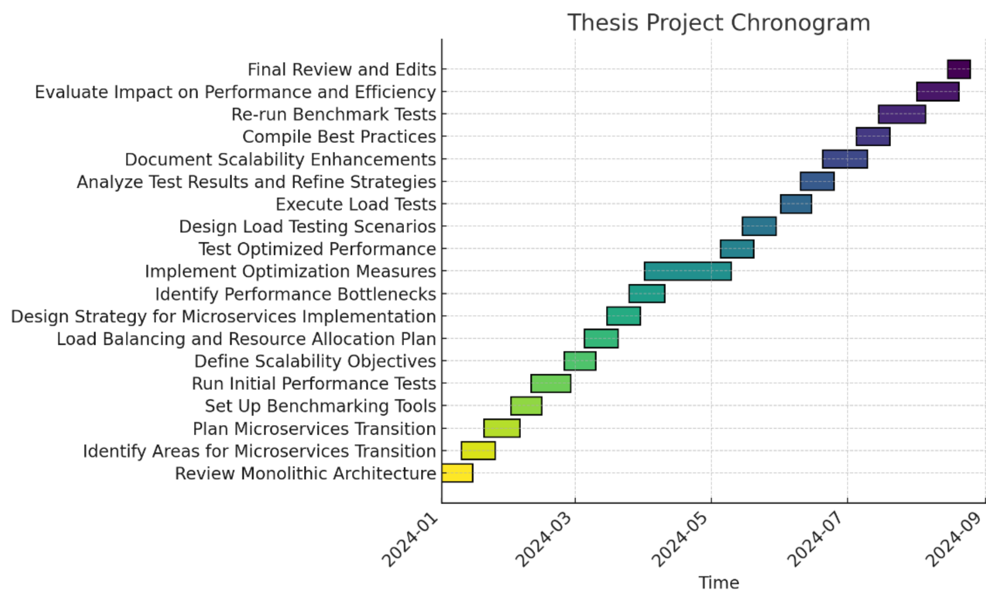


Figure 1 - Chronogram Table | Image By Luís Neves

1.7. Report Structure

The following report is divided into 7 main topics, which are the following:

1. **Introduction**: This chapter sets the stage for the research. It provides an overview of the Friendzone platform, discusses the challenges and objectives associated with scaling Friendzone, outlines the research method to be used as well as the research questions to be answered, and presents a chronogram of the research timeline. The chapter concludes with an explanation of the document's structure.

2. **Methodology**: This chapter describes the research methodology in detail. It outlines the research model, provides information about the sample used in the research, discusses the characteristics of the data, explains how the data will be analyzed, and outlines the ethical procedures that will be followed during the research.

3. **State of the Art**: This chapter reviews the current landscape of software architecture and scalability solutions relevant to the Friendzone platform. It focuses on key strategies such as transitioning from monolithic to microservices architectures, utilizing load balancing, caching, and CDNs to enhance performance and reliability. Additionally, it covers advanced patterns like CQRS and the use of message brokers for efficient service communication. The importance of real-time monitoring, performance metrics, and thorough evaluation of scaling techniques through testing is also discussed, laying the groundwork for the design and implementation phases of the project.

4. **Solution Analysis and Design:** This chapter focuses on the current state of Friendzone's system, providing a detailed overview of the existing architecture and its design. It includes a comprehensive assessment of the current monolithic API, highlighting its structure and the technologies used in its implementation. The chapter also presents the results of initial performance tests, establishing a benchmark for the current system's capabilities. By analyzing the existing design and performance, this section lays the foundation for identifying the necessary improvements and transitioning to a more scalable architecture.

5. **Implementation:** This chapter details the implementation of the proposed solution. It covers the step-by-step process of migrating from the platform to a microservices architecture, implementing load balancing, optimizing performance, and integrating new technologies. The chapter includes code snippets, architectural diagrams, and other technical documentation to support the explanation. Finally, it also tests the new implementation to have a benchmark for comparison with the previous ones.

6. **Review, Impact and Answers:** In this chapter, the effectiveness of the implementation is reviewed. It evaluates how well the implemented solution meets the initial objectives, including scalability, performance, and reliability. The chapter also analyzes the impact of these changes on the platform, both in terms of technical performance and user experience. This section includes performance benchmarks and a comparison of pre- and post-implementation states.

7. **Conclusion**: Summarizes the thesis at its current development state, giving an overview of the whole document and an end statement to close the topics.

8. **References**: This chapter lists all the sources of information used in the research. It ensures that all the information provided in the document is properly credited and can be traced back to its original source.

Each chapter is designed to provide a comprehensive understanding of the topic, and they collectively contribute to the overall objective of the research. The document concludes with a reflection on the initial objectives and final results, discussing the limitations, future challenges, and outlining future work. This structure ensures a logical flow of information, making the document easy to follow and understand. Furthermore, at the end of the report, some helpful and insightful documents are added for further support.

2. Research Method

The research methodology employed in this thesis is a rigorous and systematic approach that prioritizes academic integrity. This method involves a literature review, which serves as a crucial component of the research process. Furthermore, it should be noted that the full research was made with the consideration of the P. Porto Code of Good Practices and Conduct (Diário da República, 2020).

2.1. Literature Review

Literature review is a critical component of academic research, providing a comprehensive overview of current knowledge on a particular topic. "A literature review is a survey of scholarly sources on a specific topic. It provides an overview of current knowledge, allowing you to identify relevant theories, methods, and gaps in the existing research that you can later apply to your paper, thesis, or dissertation topic" (Wee & Banister, 2016). This emphasizes the role of a literature review in identifying relevant theories, methods, and gaps in existing research. Furthermore, "The literature review represents the most important step of the research process in qualitative, quantitative, and mixed research studies. As noted by Boote and Beile (2005), 'A thorough, sophisticated literature review is the foundation and inspiration for substantial, useful research. The complex nature of education research demands such thorough, sophisticated reviews'" (Onwuegbuzie et al., 2012). This underscores the importance of a literature review in positioning the research in relation to other researchers and theorists, and demonstrating how the current study addresses a gap or contributes to a broader debate.

2.1.1. Inclusion and Exclusion Criteria

In conducting the literature review, stringent inclusion and exclusion criteria were adopted to maintain high standards. This iterative process of exclusion based on specific criteria ensured that irrelevant studies were systematically filtered out, as shown in table 1 below.

Table 2 - Inclusion and Exclusion Criteria

Criteria	Justification
Inclusion	Articles that mention patterns, principles, tools and technologies to implement when scaling an application.

	Articles that reflect practical cases of scaling an application.
	Articles that present all sorts of testing, fault tolerance and scalability for applications.
	Articles that mention scaling metrics for applications.
	Books, scientific articles, and technical reports.
Exclusion	Commercial publications.
	Articles that are not written in English, except for the ethics and good practices document by the Portuguese Republic.
	Articles not related to Software Engineering.
	Articles that do not show evidence of the author's perspective.
	Articles older than 2012, except for articles related to the ethics in Software Engineering

2.1.2. Information Sources

The Institute of Electrical and Electronics Engineers (IEEE), MDPI, Springer Link for gathering books, Cornell University's archive, named Arxiv, National Institute of Standards and Technology, Journal of Cloud Computing and Google Scholar were the adopted sources of information.

2.1.3. Research Terms

The following terms were used to obtain the plethora of papers this thesis uses for its state-of-the-art research. The terms are the following:

- Microservices + Scaling techniques
- CQRS + Scaling Techniques
- Message Brokers + Kafka OR RabbitMQ
- Scaling Techniques + Testing
- Scalability Testing
- Caching + Redis
- Increase Reliability + API
- Scaling Software
- Scalability Metrics

2.1.4. Selection Process

Throughout the process of reviewing and selecting the most appropriate papers to include in the current thesis, a systematic approach was taken into consideration to ensure that none of the inclusion or exclusion criteria were broken. The following diagram demonstrates the initial number of papers, blog posts and magazine articles that were gathered from an initial standpoint and later strained and filtered in order to obtain the final list of papers to be utilized in this thesis.

Figure 2 - Selection Process Diagram | Image By Luís Neves

Finally, the selection process yielded 37 papers that fit the criteria. However, the number of papers that were used came down to 34, which will be enumerated in the following topic.

2.2. Research Method Outcome

In conclusion, the research method and the literature review process adopted in this thesis are designed to ensure a thorough and rigorous exploration of the topic, contributing to the body of knowledge in a meaningful and impactful way. The used articles are the following:

Table 3 - Research Method Result

ID	Title	Author(s)	Year	Source
1	A Comparative Assessment of JVM Frameworks to Develop Microservices	W. Lukasz, L. Lukasz & K. Anna	2022	MDPI
2	A Study on Software Testing Standard Using ISO/IEC/IEEE 29119-2	P. Cristiano, P. Rui & M. Gonçalo	2013	SpringerLink Book
3	An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation	S. Max, N. Sarah, E. Aryaz & T. Frank	2023	Cornell University Arxiv
4	Improvement and Implementation of a High Performance CQRS Architecture	L. Zhang	2017	IEEE Xplore
5	Interoperability and Integration Testing Methods for IoT Systems: A Systematic Review	Bures, M., Klima, M., Rechtberger, V., Bellekens, X., Tachtatzis, C., Atkinson, R., Ahmed, B. S.	2020	Springer Link Book
6	Lean automated hardware/software integration test strategy for embedded systems	M. Florian, W. Stefan, S. Thilo	2021	IEEE Xplore
7	Measuring and Testing the Scalability of Cloud-based Software Services	A.Amro, A. Peter	2018	IEEE Xplore
8	Microservices	L. Xabier, S. Izaskun, C.-P. Ricardo, E. Christof	2018	IEEE Xplore
9	Report on the Metrics and Standards for Software Testing (MaSST) Workshop 2012	B. Paul, F. Elizabeth	2012	NIST
10	Scalability analysis comparisons of cloud-based software services	A.Amro & A. Peter	2019	Journal of Cloud Computing
11	Software Quality Metrics – Research, Analysis and Recommendation	M. Tsvetelina	2020	IEEE Xplore

12	A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming	T. Sharvari, K. Sowmya	2019	Cornell University Arxiv
13	System Integration for Large-Scale Software Projects: Models, Approaches, and Challenges	S. Mohamed, H. Iman, M. Sherif	2021	Springer Link Book
14	System Integration Testing in Large Scale Agile: dealing with challenges and pitfalls	B-G. Kristian, C. Daniela	2020	IEEE Xplore
14	A Review on Software Defined Content Delivery Network: A Novel Combination of CDN and SDN	Y. Huixiang, P. Hanlin, M. Lin	2023	IEEE Xplore
15	Investigating Performance Metrics for Evaluation of Content Delivery Networks	J. Seyed, N. HamidReza, J. Massoumeh	2018	Springer Link Book
16	Challenges and Opportunities in Content Distribution Networks: A Case Study	E. Muslim, A. Karl	2019	Paper
17	Scalability analysis comparisons of cloud-based software services	A.Amro, A. Peter	2019	Journal of Cloud Computing
18	Industrial Challenges of Scaling Agile in Mass-Produced Embedded Systems	E. Ulrik, O. Helena, S. Niels	2014	Springer Link Book
19	Qualitative Analysis Techniques for the Review of the Literature	O. Anthony, L. Nancy, C. Kathleen	2012	Paper
20	How to Write a Literature Review Paper?	Wee B. V., Banister D.	2016	Paper
21	A Survey on Unit Testing Practices and Problems	D. Ermira, F. Gordon	2014	IEEE Xplore
22	Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices	Shahin, Mojtaba; Ali Babar, Muhammad; Zhu, Liming	2012	IEEE Xplore
23	Performance and Load Testing: Tools and Challenges	Lenka R, Rani Dey M, Bhanse P, Barik R	2018	IEEE Xplore
24	Software Testing Techniques: A Literature Review	Jamil M, Arif M, Abubakar N, Ahmad A	2017	IEEE Xplore
25	How Socially Sustainable Is Social Media Gamification? A Look into Snapchat, Facebook, Twitter, and Instagram	Hristova D, Lieberoth A	2021	Springer Link Book

26	Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems	Chen T, Syer M, Shang W, Jiang Z, Hassan A, Nasser M, Flora P	2017	IEEE Xplore
27	Diário da República, 2.ª série PARTE E Artigo 2.º	Diário da República	2020	Paper
28	Performance Testing And Profiling Of Web Based Application In Real Time	Kajol Mittal, Rizwan Khan	2018	Paper
29	Efficient Caching Mechanisms Using Redis in High-Performance Web Applications	Li, Y., et al.	2015	Paper
30	Scalable Distributed Caching with Redis for High-Performance Applications	Zhao, J., et al.	2014	Paper
31	Redis Replication and Fault Tolerance in Cloud Environments	Mohan, C., et al.	2013	Paper
32	Comparative analysis of metaheuristic load balancing algorithms for efficient load balancing in cloud computing	Zhou, J., Lilhore, U.K., M, P. *et al.*	2023	Paper
33	Online social networks security and privacy: comprehensive review and analysis	Jain, A.K., Sahoo, S.R. & Kaubiyal	2022	Article
34	Software Engineering Code	Don Gotterbarn, Keith Miller, and Simon Rogerson	1997	Paper

2.3. Chapter Summary

The research methodology used in this thesis prioritizes academic integrity by conducting a comprehensive literature review. This review meticulously examines existing scholarly sources, including relevant theories, methodologies, and research gaps related to scaling techniques for various applications. The inclusion and exclusion criteria are rigorously applied to select only the most relevant articles, based on their relevance to scaling principles, practical cases, testing methodologies, and publication types. This systematic approach is intended to make a meaningful contribution to the field of Software Engineering.

3. State of the Art

In software development, effective testing and scaling methods are crucial for the success and reliability of modern applications. The State-of-the-Art chapter reviews the key aspects of testing and scaling in software architecture, highlighting their importance in supporting robust systems.

This chapter explores various testing methods currently used in software development, providing practical insights and guidance. It also examines scaling techniques, focusing on their role in meeting the demands of a growing user base.

By looking at industry standards and best practices, this chapter aims to evaluate and compare different approaches to testing and scaling. The goal is to identify the most relevant strategies for the evolving needs of Friendzone.

3.1. Tests and Scaling Methods

3.1.1. Overview of Testing Methods

In software development, rigorous testing is crucial to ensure reliability, functionality, and performance. This section covers key testing methodologies—performance testing, load testing, and scalability testing—that are vital for building a robust software architecture.

3.1.1.1. Performance Testing

Performance testing is an important part of software development since it ensures that applications can withstand real-world usage and satisfy performance requirements. It entails assessing an application's efficiency, stability, and scalability under varied workloads and circumstances.

“Software Quality Testing has always been a crucial part of the software development process and lately, there has been a rise in the usage of testing applications. While a well-planned and performed test, regardless of its nature - automated or manual, is a key factor when deciding on the results of the test, it is often not enough to give a more deep and thorough view of the whole process. That can be achieved with properly selected software metrics that can be used for proper risk assessment and evaluation of the development.” (Mladenova, 2020). This highlights the significance of software quality testing, including performance testing, as well as the use of metrics in evaluating and improving the development process. Developers may construct applications that provide optimal performance and user experiences by following a disciplined performance testing methodology.

By following a structured performance testing process, developers can create applications that deliver optimal performance and user experiences. As shown by (Lenka et al., 2018) the process of performance testing can be explained by the following:

1. **Determine your key performance indicators (KPIs):** Define the critical metrics for your application. Response time, throughput, and error rates are examples of such parameters.

2. **Create Performance Test Scenarios**: Develop scenarios that simulate real-world usage of the application. These scenarios should cover a variety of user interactions with the application.

3. **Configure the Test Environment:** Make sure the test environment closely resembles the production environment. Hardware, software, and network setups are all included.

4. **Execute the Test**: Run the performance test scenarios under varying loads. This could range from light load (few users) to stress load (beyond peak expected users).

5. **Analyze findings and Identify Bottlenecks:** After running the tests, examine the findings to identify any performance issues or bottlenecks. This could include delayed response times, system crashes under high load, or other issues that could negatively influence the user experience.

6. **Report Findings**: Document the results of the performance test, including the identified issues and recommended actions for improvement.

Tools: **Apache JMeter** is a popular tool for performance testing. Other options include **Gatling** and **LoadRunner**.

3.1.1.2. Load Testing

Load testing is a critical aspect of scaling software systems. It involves simulating a heavy load on a software system to evaluate its behavior under peak load conditions.

"Many large-scale software systems must service thousands or millions of concurrent requests. These systems must be load tested to ensure that they can function correctly under load (e.g. the rate of the incoming requests)" (Lenka et al., 2018). This emphasizes the importance of load testing in ensuring the reliability and functionality of large-scale software systems.

Similarly, in the paper (Chen et al., 2017), the authors note, "Assessing how large-scale software systems behave under load is essential, because many problems cannot be uncovered without executing tests of large volumes of concurrent requests. Load-related problems can directly affect the customer-perceived quality of systems and often cost companies millions of dollars". This highlights the value of load testing in controlling the complexity and interoperability of modern software systems.

To summarize, load testing is an essential feature of current software development processes since it ensures the correct interaction and functionality of software components in complex systems.

Load Testing, according to (Lenka et al., 2018) can be achieved by following the presented steps:

1. **Establish the performance criteria:** Identifying key performance indicators (KPIs) such as response time, throughput, and error rates is part of this process.

2. **Identify the workload**: Understand the typical user behavior on the application and define the different types of user interactions that need to be simulated.

3. **Create the test scripts**: Develop scripts that simulate these user interactions. These scripts will be run concurrently to simulate the load.

4. **Set up the testing environment:** Set up the test environment to be as close to the production environment as possible. This comprises software, hardware, and network settings.

5. **Run the load test**: Execute the test scripts with varying load levels. Start with a small number of users and gradually increase the load to the maximum level the application is expected to handle.

6. **Monitor and measure**: Collect data about how the application performs under load. This includes monitoring the KPIs defined earlier.

7. **Analyze the results**: Evaluate the data collected during testing to identify any performance bottlenecks.

8. **Report**: Document the findings, including any identified issues and recommendations for improvement.

Tools: **Apache JMeter** is widely used for load testing. **BlazeMeter** and **Locust** are also effective tools for this purpose.

3.1.2. Current Industry Standards and Best Practices in Testing

Software testing stands as a cornerstone in ensuring the reliability and functionality of modern software. In this rapidly advancing technological era, adhering to industry standards, and embracing best practices is fundamental for seamless testing processes. In this chapter, established industry standards are researched, explained, and exemplified to congregate the previously mentioned information about the theory of software testing and its types.

3.1.2.1. Industry Standard in Software Testing

Industry standards provide a framework for consistent and effective software testing. They ensure that testing processes are carried out systematically and efficiently. One of the most recognized standards in software testing is the ISO/IEC/IEEE 29119.

The ISO/IEC/IEEE 29119 standard is an internationally agreed set of standards for software testing that can be used by any organization when performing any form of software testing.

As stated by (Patrício et al., 2021), "ISO/IEC/IEEE 29119 is an internationally agreed set of standards for software testing that must be adopted during all software development processes and incorporated by every software company when conducting every software testing development".

3.1.2.2. Best Practices in Software Testing

Best practices in software testing are strategies and approaches that have been proven to be effective in the software testing process. They help ensure the delivery of high-quality products.

Here are some of the best practices, according to (Jamil et al., 2017):

- **Understand the Software Under Test (SUT)**: Before starting the testing process, it’s crucial to understand the software’s functionality, features, and target users.

- **Plan the Testing Process**: A well-planned testing process can help identify potential issues early and ensure that the testing is conducted systematically and efficiently.

- **Use Automated Testing Tools**: Automated testing tools can help speed up the testing process and increase its accuracy.

- **Perform Different Types of Testing**: Different types of testing like unit testing, integration testing, load testing, performance testing, and scalability testing should be performed to ensure comprehensive coverage.

- **Continuous Testing**: Continuous testing throughout the development process can help identify and fix issues early, reducing the cost and effort of fixing them later.

- **Review and Improve the Testing Process**: The testing process should be regularly reviewed and improved based on feedback and results.

3.1.3. Apache JMeter in Performance and Load Testing

Apache JMeter is an open-source testing tool widely used for performance and load testing of applications. It allows developers to simulate multiple users sending requests to a server, enabling them to measure and analyze the performance of web applications, APIs, and other services.

Along with being recommended during the degree this thesis is being written for, it also has the following advantages that made it the choice for testing Friendzone’s capabilities:

- **Versatility:** JMeter supports various protocols, including HTTP, HTTPS, FTP, and JDBC, making it a versatile tool for testing different types of applications.

- **Ease of Use:** With its user-friendly GUI, JMeter allows users to create test plans easily, making it accessible even to those with limited scripting knowledge.

- **Extensibility:** JMeter supports plugins that extend its functionality, allowing for more advanced testing scenarios.

As noted by (Mittal & Khan, 2018), JMeter is particularly effective for integrating performance testing into continuous integration frameworks, which helps in automating performance tests and maintaining the application's response time under various loads.

3.1.4. Scaling Methods and Techniques

3.1.4.1. Microservices

Microservices are small, independent services that work together to form a complex software application. According to (Larrucea et al., 2018), microservices are described as "small applications with a single responsibility that can be deployed, scaled, and tested independently".

This highlights the core principle of microservices - each service is designed to do one thing well and can operate independently of other services. This allows for greater flexibility and scalability in software development.

Pros:

- Scalability: Microservices can be scaled independently based on demand.
- Flexibility: Different microservices can use different technologies, making it easier to adopt new technologies.
- Fault Isolation: Failure in one microservice doesn't directly affect others.

Cons:

- Complexity: Managing and coordinating multiple services can be complex.
- Data Consistency: Ensuring data consistency across services can be challenging.
- When to Implement: Microservices are beneficial when you need to scale different parts of an application independently, or when you want to use different technologies for different services.
- Cost of Changing: Transitioning from a monolithic architecture to microservices involves significant effort and resources. It requires changes in team structure, development practices, and deployment processes.

3.1.4.2. Load Balancers

Load balancers are essential for distributing workloads across multiple computing resources, such as servers or virtual machines (VMs), in cloud environments. They prevent any single resource from becoming

overloaded, ensuring that applications remain responsive and reliable. As described in research, "Cloud load balancing (CLB) is a process that distributes workloads and computing resources in a cloud environment", which is critical to managing system load and optimizing resource use (Zhou, 2023).

There are several types of load balancing techniques, both static and dynamic. Static methods, like Round-Robin, distribute requests in a pre-determined manner, but do not adapt to real-time conditions. On the other hand, dynamic techniques, such as metaheuristic algorithms, respond to changing conditions in the environment, offering more flexibility and efficiency. As noted, "The most popular static load balancing methods are Round-Robin, Weighted Round-Robin, Min-Min, and Max-Min", while dynamic methods can adjust more flexibly under fluctuating workloads (Zhou, 2023).

Research highlights that load balancing is particularly crucial in cloud environments for maintaining key performance metrics like response time, which is "the time required for the system to finish a job". Efficient load balancing algorithms help reduce response times, improving overall user satisfaction. Additionally, "fault tolerance" and "scalability" are key factors, ensuring that systems can handle failures and scale as demand grows without a decline in performance (Zhou, 2023).

3.1.4.3. Message Brokers

Message brokers, such as RabbitMQ or Kafka, are pivotal in enabling software applications to communicate and process data. They serve as intermediaries that facilitate the exchange of messages between different components of a system, which is essential for scaling and optimizing performance.

"The middleware infrastructure for big data streaming, microservices and cloud-based applications" (Sharvari T & Sowmya Nag K, 2019). They further explain that "Real-time data analytics, website tracking, logging and recent boom in IoT devices has increased the demand for fault tolerant, highly available messaging systems" (Sharvari T & Sowmya Nag K, 2019).

This highlights the role of message brokers in facilitating communication between different parts of a software system, which can be crucial for scaling and optimizing performance. The use of message brokers like RabbitMQ or Kafka can help ensure that data is reliably transmitted between different components of a system, thereby enhancing its scalability and performance.

Pros:

- Reliability: They ensure data is not lost in transit.
- Scalability: They can handle high volumes of messages.
- Decoupling: Producers and consumers of data are decoupled.

Cons:

- Complexity: Setting up and managing a message broker can be complex.
- Latency: There can be a delay in message delivery.

When to Implement: Message brokers are beneficial when an app needs to process high volumes of data or when it needs to decouple data producers and consumers.

3.1.4.4. Command Query Responsibility Segregation (CQRS)

CQRS is a pattern that separates read and update operations for a data store. "The pattern is separated from the business by modifying the behavior of modifying (adding, deleting, modifying the system state) and querying (without modifying the system state), making the logic clearer and easier to target different parts Optimization". (Long, 2017) This separation of concerns allows for more efficient scaling and optimization of different parts of a system.

Pros:

- Scalability: Read and write operations can be scaled independently3.
- Performance Optimization: Allows optimization of read and write sides for their specific needs4.

Cons:

- Complexity: Adds complexity to the system due to the segregation of commands and queries3.
- Data Synchronization: Ensuring consistency between the read and write sides can be challenging4.

When to Implement: CQRS is beneficial when the system has high demands for read and write operations that need to be scaled independently.

3.1.4.5. Content Delivery Network (CDN)

CDNs are a pivotal element in frontend scaling techniques, significantly enhancing the performance of web applications. "Traditionally, content delivery network (CDN) service providers deploy servers as close to clients as possible to reduce latency. CDN is an overlay network mainly responsible for routing requests, distribution, delivery, and audit. As a supplement to computing and storage capacity, CDN service providers have begun migrating some of their services to the cloud to focus on the delivery process. At the same time, the collaboration between CDN service providers promotes scalability to meet the growing number of requests" (Yang et al., 2023).

Furthermore, "Content Delivery Networks are one of the most common services in order to overcome performance problems caused by massive data requests in popular web applications. CDNs improve clients' perceived quality of service by placing replica servers scattered around the globe and consequently redirecting users to closer servers" (Jafari et al., 2018).

These insights underscore the role of CDNs in enhancing the scalability and performance of web applications, making them a vital component in modern software scaling strategies.

Pros:

- Performance: CDNs can significantly improve the load time of your website.
- Scalability: CDNs can handle high traffic loads.
- Reliability: CDNs can provide redundancy and failover.

Cons:

- Cost: Using a CDN can add to the cost of running your website.
- Cache Invalidation: Ensuring the CDN serves the most recent content can be challenging.

When to Implement: Implementing a CDN is beneficial when an app has users in various geographical locations, or when it needs to serve large files like images and videos.

3.1.4.6. Caching with Redis

Redis is a powerful in-memory data structure store that plays a crucial role in backend scaling strategies. It significantly enhances the performance and scalability of applications by caching frequently accessed data, which reduces the load on primary databases and speeds up data retrieval times.

Redis operates as an in-memory key-value store, where data is cached in memory for fast access. When an application requests data, Redis checks if the data is already in the cache (a cache hit). If the data is present, Redis serves it directly from memory, resulting in significantly faster response times compared to fetching it from a disk-based database. If the data is not in the cache (a cache miss), the application retrieves the data from the primary database, stores a copy in Redis for future use, and then serves the data to the user. That being said, Redis can improve in the following manners:

- **Performance:** By caching frequently accessed data in memory, Redis reduces the need to query the primary database repeatedly, which lowers latency and improves the overall speed of the application. (Li, 2015) notes that "Using Redis for caching can reduce the load on a database by up to 80%, resulting in faster response times and improved user experiences."

- **Scalability:** Redis allows for horizontal scaling through data sharding, where data is partitioned across multiple Redis instances. This capability enables applications to handle increasing amounts of traffic without significant performance degradation. (Zhao, 2014) highlights that "Redis' ability to scale horizontally makes it an ideal solution for large-scale distributed applications."

- **Reliability:** Redis supports data replication, where copies of data are maintained across multiple Redis instances. This feature ensures high availability, as data can be retrieved from a replica in case of a failure in the primary instance. (Mohan, 2013) emphasizes that "Redis replication mechanisms provide high availability and fault tolerance, which are crucial for maintaining service continuity in large-scale applications."

3.1.4.7. Horizontal vs Vertical Scaling (Hardware)

3.1.4.7.1. Horizontal Scaling

Horizontal scaling, also known as scaling out, involves adding more machines to the existing pool of resources. This approach increases the capacity of the application by distributing the load across multiple servers.

Pros:

- Improved Fault Tolerance: If one server fails, the system can continue to operate by shifting the load to other servers.
- Flexibility: It’s easier to scale horizontally as you can add more machines as per the demand.

Cons:

- Complexity: Managing and coordinating multiple servers can be complex.
- Data Consistency: Ensuring data consistency across servers can be challenging.

When to Implement: Horizontal scaling is beneficial when the application needs to handle a large amount of traffic and requires high availability.

3.1.4.7.2. Vertical Scaling

Vertical scaling, also known as scaling up, involves adding more resources such as CPU, RAM, or storage to an existing machine. This approach increases the capacity of the application by enhancing the power of individual servers.

Pros:

- Simplicity: It’s simpler to manage one powerful machine than multiple smaller ones.
- Performance: A single powerful machine can often deliver better performance than multiple smaller ones.

Cons:

- Limited Scalability: There’s a limit to how much a company can scale up a single machine.
- Downtime: Scaling up often requires downtime as it needs to shut down the system to add resources.

When to Implement: Vertical scaling is beneficial when the application requires high computational power, and it does not expect a large amount of traffic.

In conclusion, both horizontal and vertical scaling have their own advantages and disadvantages. The choice between the two depends on the specific requirements of the application. It’s also worth noting that these two approaches are not mutually exclusive and can be used together for optimal results.

3.1.5. Tests and Scaling Methods Summary

Testing Methods:

In software development, testing methods ensure reliability, functionality, and performance.

Performance Testing: Determines an application's efficiency, stability, and scalability under varying workloads. It requires defining metrics, simulating real-world scenarios, and analyzing performance metrics like response time, throughput, and error rates.

Load Testing: Assesses software systems' behavior under peak load conditions, ensuring functionality under stress. It involves defining performance criteria, creating test scripts, executing tests, and identifying performance bottlenecks.

Industry Standards, Best Practices, and Scaling Techniques:

ISO/IEC/IEEE 29119 Standard: Internationally recognized for guiding software testing processes across organizations.

Best Practices: Include understanding software, planning, automation, diverse testing, continuous improvement, ensuring high-quality products.

Scaling Methods: Microservices, Message Brokers, CQRS, CDN, Redis, Horizontal/Vertical Scaling—each offering specific advantages and challenges in scalability, catering to distinct application needs.

These testing methods and scaling techniques form the backbone of modern software development, ensuring robust, reliable, and high-performance software systems.

3.2. Digital Marketing and Metric Analysis on Social Media

“Since its inception, technology has transformed the way businesses operate and the consumption of goods and services. For example, technology has revolutionized the way companies promote their products and services, perform their business activities, communicate/exchange information, and manage resources." (Cham et al., 2022).Analyzing crucial metrics on these platforms provides invaluable insights into their performance and user behavior.

3.2.1. Basics of Digital Marketing on Social Media

- **Content Variety**: Social media thrives on diverse content formats like posts, videos, stories, and live sessions. Each format serves a unique purpose in engaging and retaining audiences.

- **Audience Targeting**: Platforms offer sophisticated tools for targeting specific demographics, interests, and behaviors, ensuring content reaches the most relevant audience.

- **Engagement and Interaction**: The essence of social media lies in fostering two-way communication. Engaging with the audience, responding to comments, and encouraging discussions are key.

- **Paid Advertising**: Platforms provide opportunities for paid ads, allowing businesses to target audiences beyond their organic reach.

3.2.2. Crucial Metrics in Social Media Analysis

- **Engagement Metrics**: Likes, comments, shares, and saves gauge the audience’s response and interest in content.

- **Follower Growth**: The rate at which a profile gains new followers indicates the effectiveness of content and marketing efforts.

- **Reach and Impressions**: Understanding how many users see the content (reach) and how often it’s displayed (impressions) is fundamental in assessing visibility.

- **Click-Through Rate (CTR)**: This measures the percentage of users who click on a link or call-to-action, reflecting the effectiveness of campaigns.

- **Conversion Metrics**: Tracking desired actions like sign-ups or purchases helps assess the platform's ability to convert engagement into tangible outcomes.

3.2.3. Significance of Metric Analysis

Analyzing these metrics is instrumental in understanding audience behavior, content performance, and the effectiveness of marketing strategies. Data-driven insights enable optimization, helping businesses refine their approaches for better results.

In conclusion, social media's dynamic nature and extensive reach make it a powerhouse for digital marketing. By comprehending and analyzing key metrics, businesses can fine-tune their strategies, enhance engagement, and achieve their marketing objectives in an ever-evolving digital landscape.

3.2.4. Digital Marketing and Metric Analysis on Social Media Summary

Technology reshapes businesses and consumer habits. Key social media metrics offer crucial insights:

Digital Marketing Basics on Social Media:

- Content Variety: Engaging formats cater uniquely to audiences.
- Audience Targeting: Precision tools reach specific demographics.
- Engagement: Two-way interaction fosters audience participation.
- Paid Advertising: Extends content reach beyond organic limits.

Crucial Metrics for Analysis:

- Engagement Metrics: Gauge audience response - likes, comments, shares.
- Follower Growth: Indicates content and marketing effectiveness.
- Reach and Impressions: Evaluate content visibility and interaction.
- Click-Through Rate (CTR): Measures campaign effectiveness.
- Conversion Metrics: Track actions indicating conversion ability.

Significance of Metric Analysis:

Analyzing metrics aids in understanding audience behavior and refining marketing strategies for better outcomes in a dynamic digital landscape.

Leveraging social media metrics refines strategies, enhances engagement, and achieves marketing goals.

3.3. Monetized Gamification

"Gamification is the use of Elements derived from games, in non-game contexts, to promote participation and engagement [7]. It Is not about building an entire game around some activity: It makes use of the Elements and strategies found in games to that make them work." (Hristova & Lieberoth, 2021). Friendzone's challenge feature, embedded within a social network, integrates this concept to enhance user interaction while creating monetization avenues.

3.3.1. Understanding Monetized Gamification

1. **Gamification Elements**: Incorporating game-like elements such as points, levels, rewards, and challenges into the user experience incentivizes participation and engagement.

2. **Monetization Integration**: Monetized gamification involves strategies where users can spend or earn real or virtual currency within the platform.

3. **User Incentives**: Users are enticed by rewards or recognition for completing challenges, contributing content, or achieving milestones.

3.3.2. Friendzone's Challenge Feature

1. **Challenge Creation**: Users can create challenges, defining tasks or activities for others within the network to undertake.

2. **Monetization Aspect**: Friendzone integrates monetization within challenges, enabling users to create or participate in monetized challenges, involving microtransactions or in-app purchases.

3. **Enhanced Engagement**: Challenges foster community interaction, encouraging users to connect, compete, and collaborate, thereby amplifying engagement within the social network.

3.3.3. Metrics for Monetized Gamification Analysis

1. **Challenge Creation**: Users can create challenges, defining tasks or activities for others within the network to undertake.

2. **Monetization Aspect**: Friendzone integrates monetization within challenges, enabling users to create or participate in monetized challenges, involving microtransactions or in-app purchases.

3. **Enhanced Engagement**: Challenges foster community interaction, encouraging users to connect, compete, and collaborate, thereby amplifying engagement within the social network.

3.3.4. Importance of Monetized Gamification in Social Networks

Based on (Hristova & Lieberoth, 2021)the concept of monetized gamification within Friendzone can be divided into the following benefits:

Enhances User Experience: The use of game-derived elements in non-game contexts, such as Friendzone, promotes participation and engagement. This enhancement of the user experience is not about building an entire game around some activity, but rather making use of the elements and strategies found in games that make them engage.

Fosters a Sense of Community: The ubiquitous use of gamification in social media, including elements like point counters, badges, and rewards, frames interactions among users around the globe. This fosters a sense of community within Friendzone, encouraging users to interact more with each other.

Creates Revenue Streams: While the paper does not provide a direct quote for this step, it implies that the use of gamification elements can lead to increased user engagement, which can indirectly contribute to revenue generation.

Incentivizes User-Generated Content Creation: The paper suggests that gamification can incentivize users to create and share their own content, contributing to the vibrancy of the platform. This not only increases engagement but also provides valuable insights for the platform, helping to inform future development and improvements.

Drives Platform Growth: Increased user engagement and content creation, driven by gamification, can attract more users to the platform, contributing to its growth.

In summary, monetized gamification within Friendzone serves to enhance the user experience, foster a sense of community, create new revenue streams, incentivize user-generated content creation, and drive platform growth.

3.3.5. Monetized Gamification Summary

Gamification within Friendzone employs game elements to boost user engagement and introduce monetization strategies. It enhances interaction without building full games, stimulating user participation through challenges, rewards, and community collaboration. This approach fosters a sense of community, incentivizes content creation, indirectly contributes to revenue, and drives platform growth by attracting more users.

3.4. Chapter Summary

Friendzone's integration of the challenge feature, based on monetized gamification, represents the evolution of social networks. It boosts user engagement, introduces innovative revenue models, and transforms sustainability and user experience. Leveraging game elements without building full games, Friendzone stimulates participation through challenges, rewards, and community collaboration, fostering a sense of community, incentivizing content creation, indirectly contributing to revenue, and driving platform growth.

Key social media metrics offer valuable insights:

- Digital Marketing Basics: Diverse content formats and precise audience targeting foster engagement.
- Crucial Metrics for Analysis: Engagements, growth, reach, and conversions reflect content effectiveness and audience behavior.
- Significance of Metric Analysis: Refining strategies, enhancing engagement, and achieving marketing goals in a dynamic digital landscape.

Testing methods ensure software reliability and functionality:

- Unit, Integration, Performance, Load, Scalability Testing: From verifying individual components to assessing system behavior under peak load conditions, these methods guarantee robust, high-performance software.

Industry standards, best practices, and scaling techniques guide software development:

- ISO/IEC/IEEE 29119 Standard: Ensures systematic software testing across organizations.
- Best Practices: Emphasize planning, automation, diverse testing, and continuous improvement.
- Scaling Methods: Microservices, Message Brokers, CQRS, CDN, Horizontal/Vertical Scaling offer specific advantages and challenges tailored to diverse application needs.

Together, these insights and methodologies form the foundation of modern software development, ensuring reliability, innovation, and high-performance outcomes.

4. Solution Analysis, Design and Tests

This chapter begins with an overview of the current state of the system, focusing on its architecture, tech stack, domain model, and existing functionalities. By understanding how the system is currently structured, we can identify areas that need improvement, especially in terms of scalability and performance. After that, load and performance tests are made to then be able to present an alternative architecture to try to solve the identified problems.

Additionally, the entire design and process described are meant to be applicable across various technologies and systems. The architecture is language-agnostic, ensuring that the principles and improvements can be implemented in any tech stack or framework, allowing for flexibility and adaptability in different environments.

4.1. Domain Model

This chapter introduces the domain model that explains the Friendzone system. This model outlines the important elements and their interactions to fully understand the system's business rules and behavior. These concepts are illustrated in the following figure:

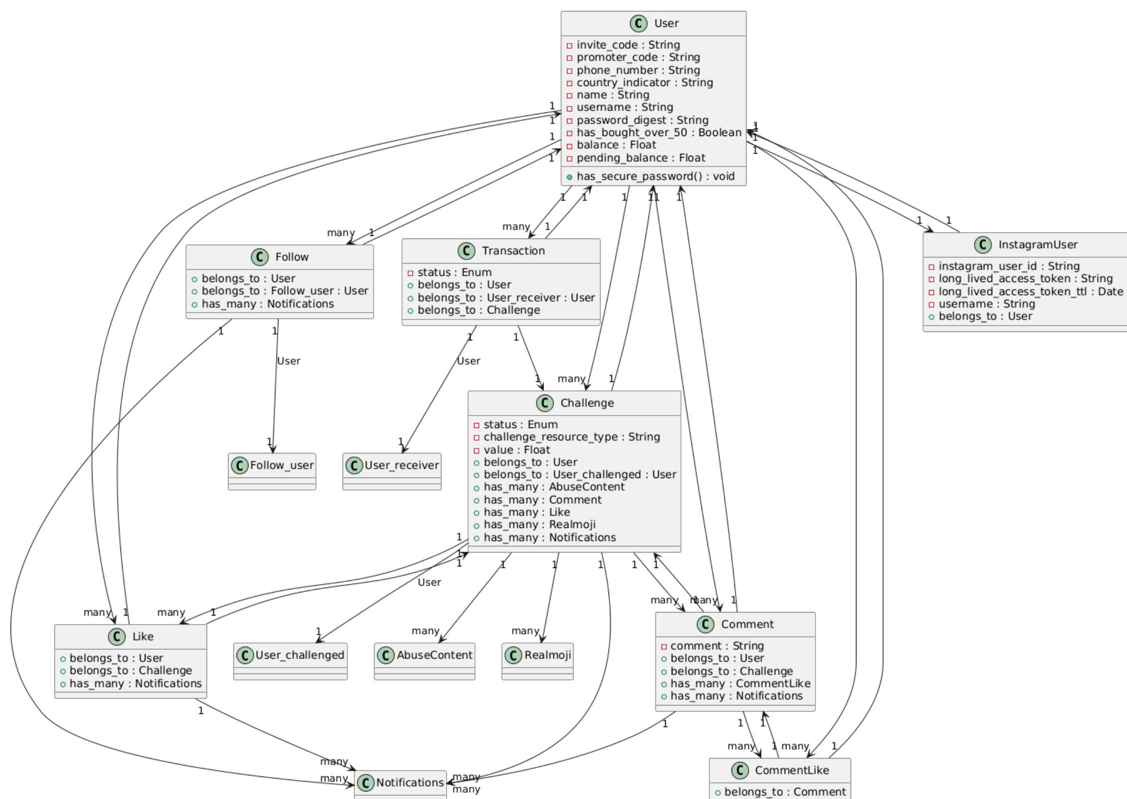


Figure 3 - Friendzone Domain Model

Friendzone is an innovative social networking platform that integrates gamification into user interactions through a system of challenges. This unique approach not only encourages engagement but also fosters a sense of competition and achievement among its users. The platform allows users to issue and accept challenges, which are then shared within the community, creating a vibrant and interactive environment.

The following features can be found within the app:

- **User-Centric Challenges**: The core feature of Friendzone revolves around challenges (Challenge). Users (User) can issue challenges to others, specifying the nature of the task, the resources required (challenge_resource_type), and the reward value in FZ tokens (value). Once a challenge is accepted, the challenged user must complete the task within a stipulated time frame, after which the FZ tokens are transferred, and the challenge becomes a piece of public content on the platform.

- **Social Interactions:** Friendzone incorporates familiar social media interactions, allowing users to like (Like) and comment (Comment) on challenges. This functionality ensures that challenges are not just tasks but also content pieces that generate discussions and feedback within the community, enhancing user engagement.

- **Follow System:** Users can follow (Follow) others, enabling them to stay updated on the activities and challenges undertaken by those they are interested in. This feature helps build a network within the platform, similar to following mechanisms found in other social networks.

- **Hidden Challenges:** An intriguing feature of Friendzone is the ability to create hidden challenges. These challenges are only visible to the participants involved, adding an element of exclusivity and mystery.

- **Content Moderation:** To ensure a safe and respectful community, the platform includes content moderation features. The AbuseContent entity allows users to report inappropriate or offensive material related to challenges, ensuring that the platform remains a welcoming environment for all users.

- **Notification System:** Friendzone keeps users informed of various activities through its notification system (Notifications). Whether it's a new challenge, a comment on a challenge, or a like, users receive notifications that keep them engaged and informed about the latest interactions on the platform.

- **Transactional Model:** The platform operates on a transactional model (Transaction) where FZ tokens are used as currency. This system underpins the challenge mechanism, allowing users to wager tokens on the outcomes of challenges and rewarding them upon successful completion. This not only adds a competitive edge but also provides a tangible reward system within the app.

4.2. Current Tech Stack

The front-end is a React Native application written in TypeScript, providing a robust, cross-platform user experience for both iOS and Android devices. The backend is powered by a Ruby on Rails API, which handles business logic and communication with the database. For data storage, the system uses a PostgreSQL database, known for its reliability and performance with complex queries. To streamline deployment and ensure consistency across environments, the system leverages Docker, allowing for containerized deployment of the system.

4.3. Design

The C4 Model is a structured approach to visualizing software architecture through a series of hierarchical diagrams. It is designed to provide different levels of abstraction that are useful for communicating various aspects of the system to both technical and non-technical stakeholders. The four levels of the C4 Model—Context, Container, Component, and Code—each serve distinct purposes in documenting and understanding the architecture of software systems.

- **Context Diagram**: This diagram offers a high-level view of the software system, showing its interactions with external entities such as users, systems, and services. It is particularly valuable for providing a broad understanding of how the system fits into its environment, making it accessible to everyone involved, from developers to business leaders. The context level provides a high-level overview of the system, depicting it as a black box and demonstrating its interactions with external users, systems, and stakeholders. This identifies the system's boundaries and dependencies (Brown, 2015).

- **Container Diagram**: This diagram dives deeper into the system by breaking it down into containers, which represent applications, services, or databases that execute code or store data. It illustrates the relationships and interactions between these containers, providing a clear understanding of the system's structure and the technologies involved (Brown, 2015).

- **Component Diagram**: The component diagram provides an in-depth view of the internal structure of each container, detailing the major components and their interactions. This level of detail is particularly useful for developers and architects who need to understand the specific responsibilities and dependencies within a container (Brown, 2015).

- **Code Diagram**: At the most granular level, the code diagram focuses on the internal structure of individual components, such as classes and methods. While this level of detail is often documented directly in the codebase, the code diagram can be helpful in situations where a visual representation of the code's structure is necessary (Brown, 2015).

4.3.1. Level 1 – Context

As seen in the figure below, the first level of the C4 model shows 2 main components, the Friendzone and the Instagram Component, as well as a User to represent it's interaction with the Friendzone system.

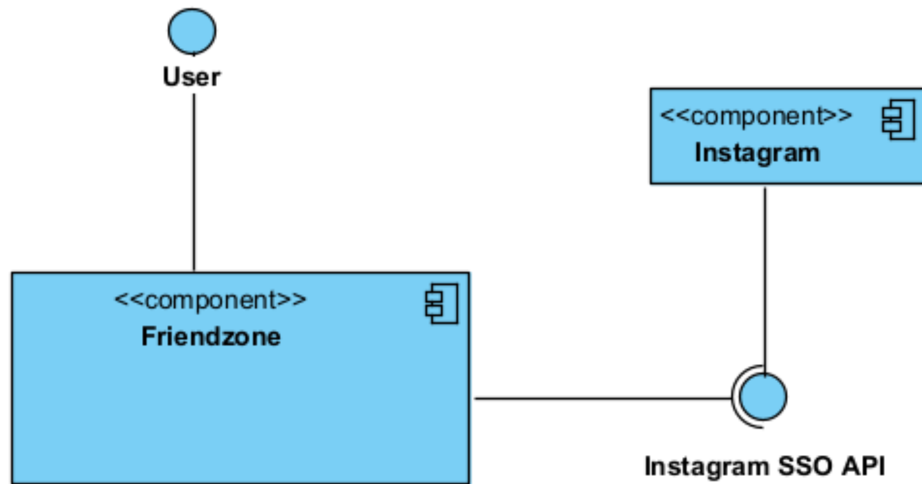


Figure 4 - Level 1 Diagram

4.3.2. Level 2 – Container

As the diagrams increase in detail, the current structure, at container level, that composes the Friendzone app is now visible. There's a mobile app, an API and a database, a very simple and straightforward system whose only external dependency is Instagram's SSO API.

4.3.2.1. Logic View

In the context of the C4+1 model, this level shows the architectural view of the Friendzone application's components and interactions. At this phase, structural aspects determine how different components work together to fulfill the application's objective.

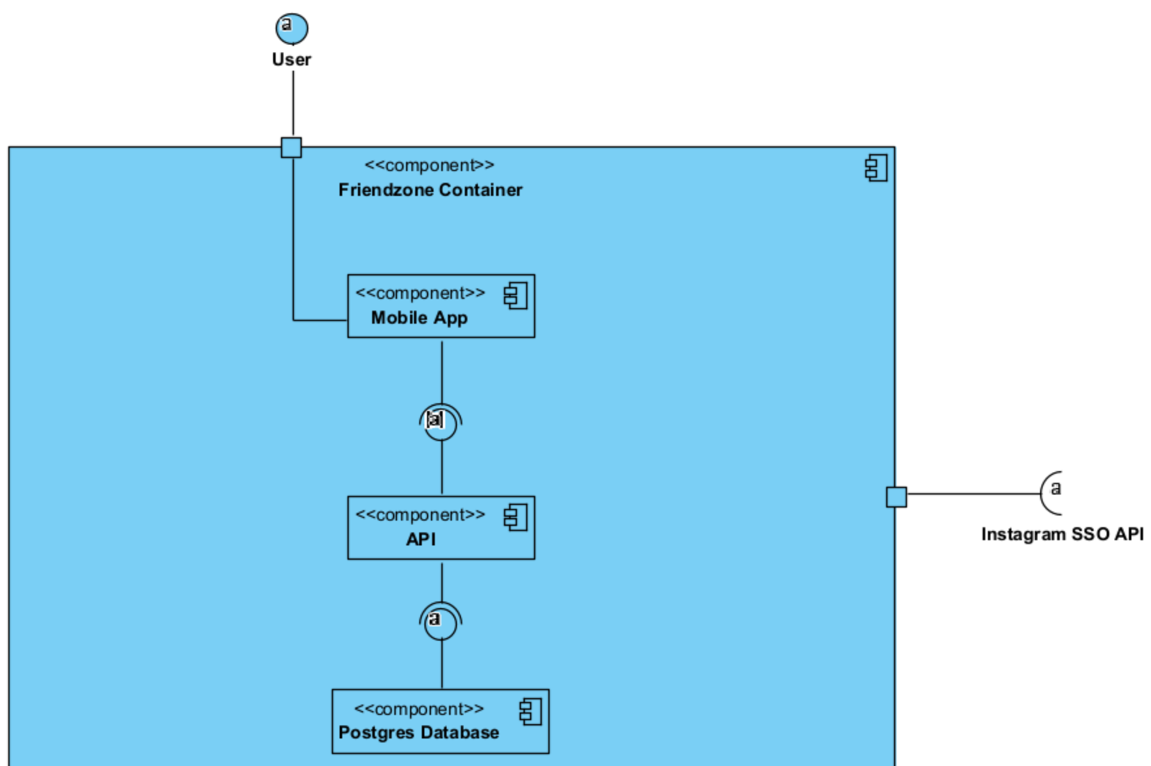


Figure 5 - Level 2 Logic View Diagram

4.3.2.2. Process View

Due to this thesis' focus not being on new functionalities, the following process view, Figure 6, is a simplified example of what a regular interaction would be between a user and Friendzone's system, reiterating the simplicity of the system.

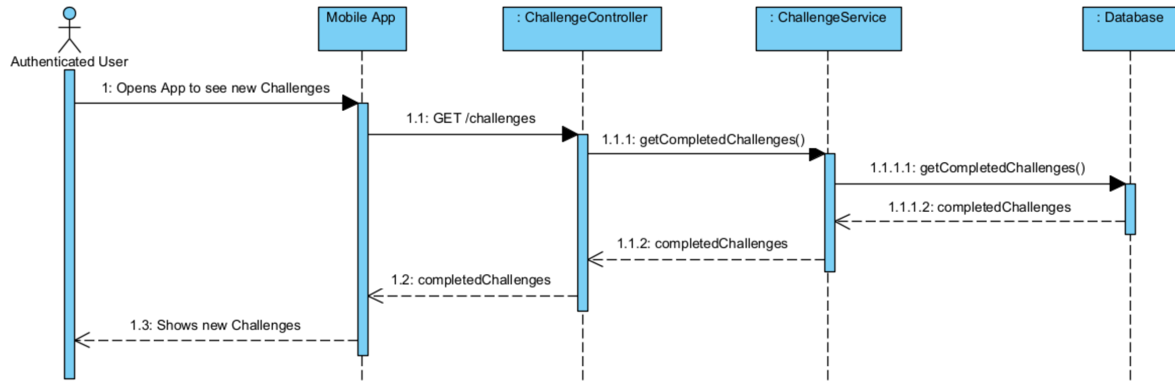


Figure 6 - Level 2 Process Diagram

The process begins with the authenticated user interacting with the app, either by GET or POST request. The mobile app then makes a request to the API which in turn communicates with the Database, fetching or saving the necessary content.

4.3.2.3. Physical View

Moving on to the Physical View, Figure 7 shows what the current system's allocations look like, with both the API and the Database hosted on Digital Ocean. Friendzone uses Digital Ocean as a service provider to host all of its infrastructure. Due to a partnership between the two entities, this is currently the most cost effective solution but may need to change in the future if the current terms expire.

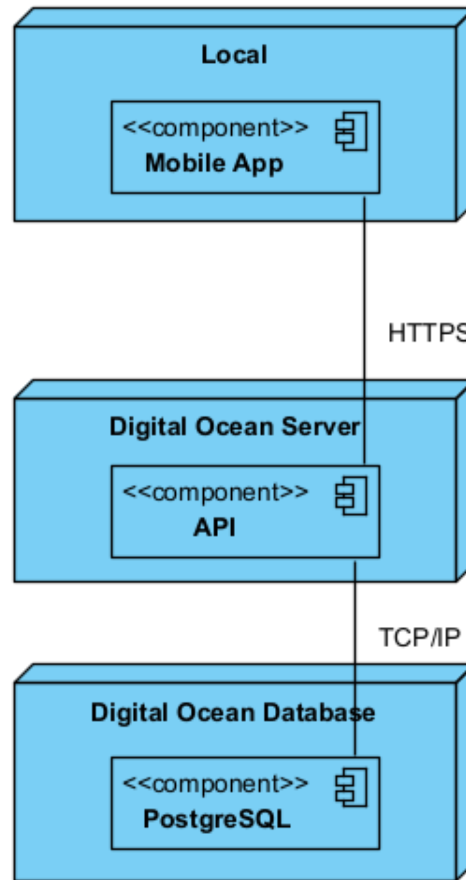


Figure 7 - Level 2 Physical Diagram

4.3.3. Level 3 – Component

Level 3 brings the detail to the component level, where it is now possible to do an in depth analysis of Friendzone's inner workings and systems.

4.3.3.1. Logic View

Within Level 3's logic view diagram, it is now visible the individual components that the API is made of. This shows the chosen software patterns to bring Friendzone to its current state.

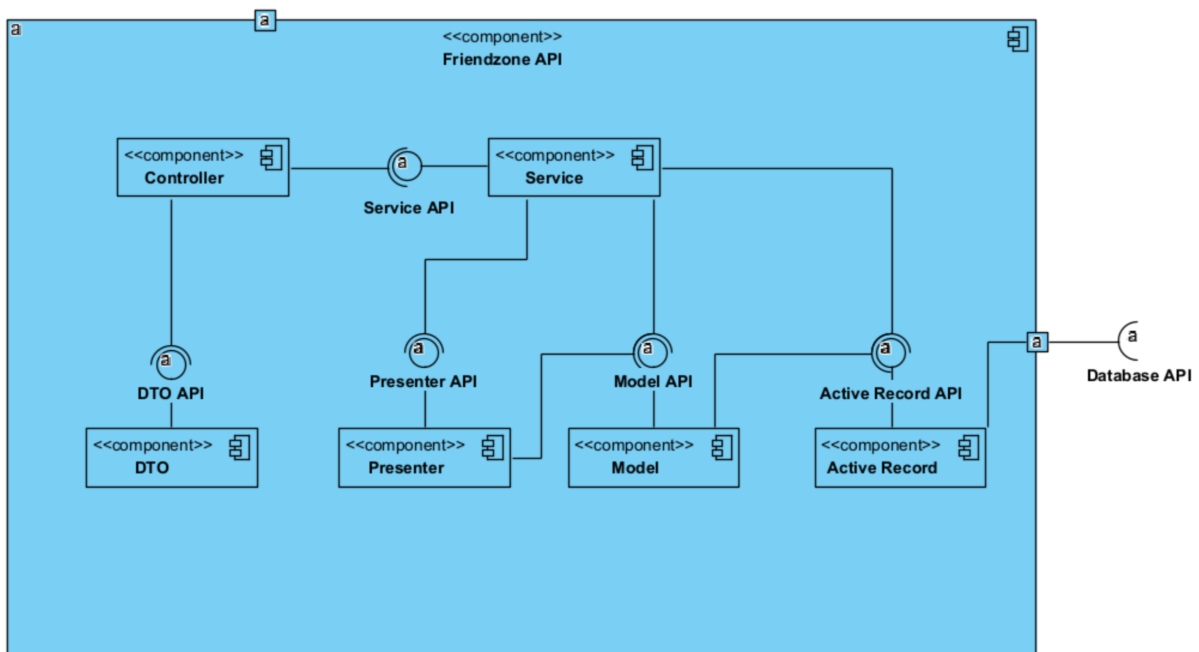


Figure 8 - Level 3 Logic Diagram

From this point on, due to the thesis' focus, it would prove unproductive to continue to go into more detail about the current state of the system, as all the points that need improvement have already been exposed.

4.4. Testing the current system

To establish a benchmark for the system's performance, a series of load tests were conducted on the current API to assess its capabilities. These tests provide key metrics that can be compared against future improvements after the proposed optimizations are implemented. The API was hosted on a local Docker container to simulate real-world conditions, and the testing was performed using Apache JMeter. This section will cover both read and write operations.

In social networks, read operations typically outnumber write operations by a large margin due to the way users engage with content. Most interactions involve activities such as viewing posts, images, or comments (read operations), rather than generating new content (write operations). As one study highlights, "on average 510,000 comments are posted every 60 seconds on Facebook, 298,000 statuses are updated, and 136,000 photos are uploaded, but there are significantly more passive interactions (reads) like viewing or sharing content" (Jain, 2022).

Considering this, it is expected that the API should handle significantly more concurrent reads than writes. Based on typical usage patterns of social media platforms, our load tests will target 10,000 concurrent read operations while simulating 4,000 to 5,000 concurrent write operations or fewer. This aligns with the observed need for more efficient handling of read operations, as systems typically serve far more requests for content retrieval than content creation.

4.4.1. Testing Environment

The testing procedures will be conducted locally, isolating the API in a Docker container. This approach ensures that the performance metrics and load testing results are focused solely on the architecture of the API, without interference from external factors such as server hardware, network latency, or cloud infrastructure optimizations. By isolating the API in a controlled environment, the test results will provide a clear benchmark of the API's capabilities and limitations. This setup removes the complexities introduced by server capacity or external network conditions, allowing for a more accurate assessment of how well the API's design and implementation handle increasing load and user traffic.

The computer that will be running the tests has the following specifications:

- 16 GB of RAM, of which the docker container will have access to, at most, 12,4GB, as seen in the figure below.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %
58a56e5db92d	friendzone-api_adminer_1	0.00%	14.49MiB / 12.4GiB	0.11%
be85c3c30153	friendzone-api_api_1	0.04%	208.8MiB / 12.4GiB	1.64%

Figure 9 - Docker Container Available Resources

- 4 Cores – Intel Core i5 – 7300HQ CPU @ 2.5 GHz

Once the architecture has been optimized and tested locally, further tests may be conducted on cloud servers to evaluate performance under real-world conditions, if a budget for test servers is available.

4.4.2. Apache JMeter

The load tests were executed with varying user loads to evaluate how well the system handles increasing traffic. Starting with 50 concurrent users, the load gradually increased to 1,000 users, then to 5,000 users, and finally to 10,000 users. Each test had a 5-second ramp-up period, allowing users to connect over a short interval rather than all at once. By running these tests, the goal was to identify the system's limits in terms of throughput, response times, and error rates under different levels of load.

4.4.2.1. 50 Concurrent Users

The first test was made at a level where it was expected the API to have a good performance, since it is the load it's currently taking, daily. That being said, the following parameters were used for testing the API:

Threads (Users): 50

Ramp up time: 5 seconds

Loop: 1

These parameters produced the following results, shown in Figures 10, 11, 12 and 13.

Read Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	50	4648	864	7399	2016.96	0.00%	4.1/sec	10.88	1.44	2740.9
TOTAL	50	4648	864	7399	2016.96	0.00%	4.1/sec	10.88	1.44	2740.9

Figure 10 - 50 Concurrent Users Summary Report – Read Operation

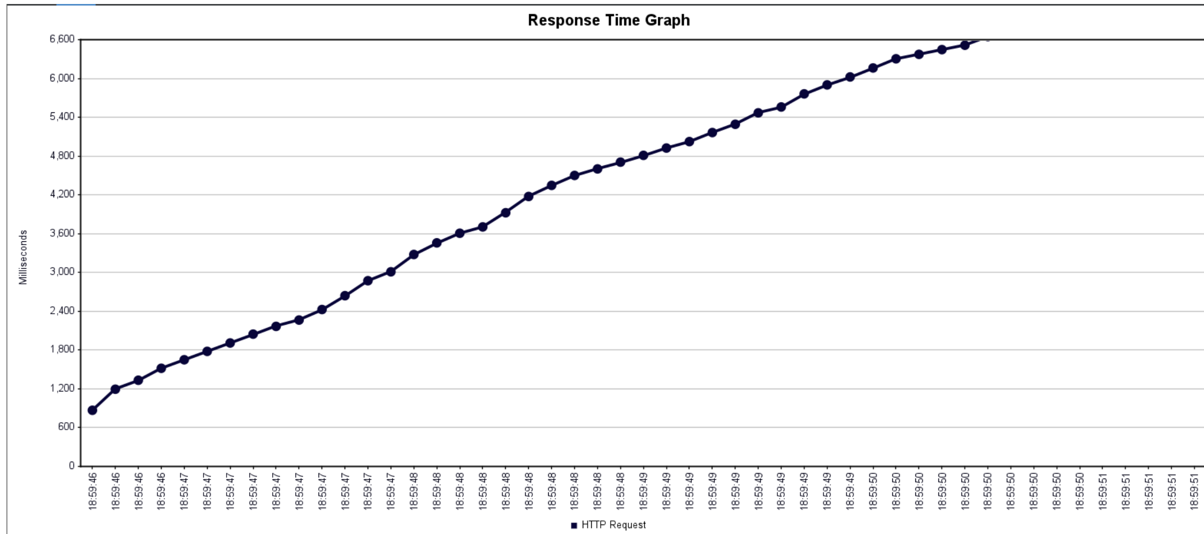


Figure 11 - 50 Concurrent Users Response Time Graph | 100ms interval

Takeaways:

- **0% error rate** – All 50 requests were successfully answered
- **Inconsistent Response times** - The high standard deviation (2016.96 ms) indicates that there is a large variation in the response times. Some requests were processed in under 1 second (864 ms), while others took more than 7 seconds. This inconsistency might be a sign of uneven resource allocation or certain requests hitting bottlenecks like slow database operations or long-running processes.
- **Throughput** - 4.1 requests per second is a relatively low throughput for an API serving 50 users.
- **Crescent Response Times** – The Response Time Graph clearly shows the up trend in response times as requests come in. Confirming what the other data reports indicated.

Write Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST HTTP Request	50	1698	446	2776	710.18	0.00%	6.5/sec	5.03	6.22	792.4
TOTAL	50	1698	446	2776	710.18	0.00%	6.5/sec	5.03	6.22	792.4

Figure 12 - 50 Concurrent Users Summary Report – Write Operation

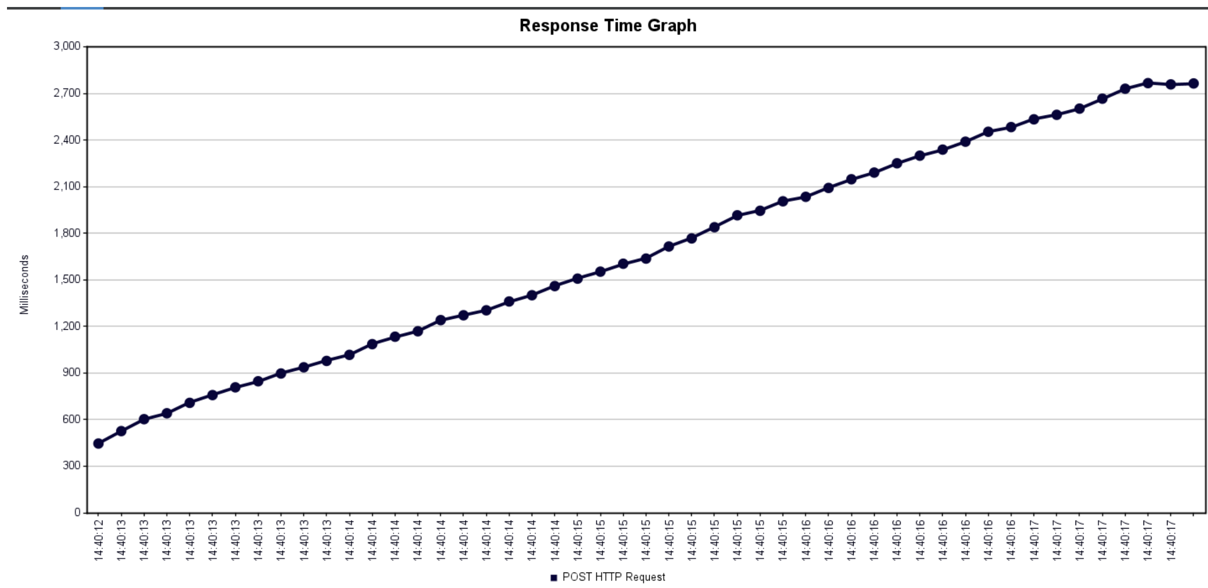


Figure 13 - 50 Concurrent Users Response Time Graph – Write Operation

Takeaways:

- **0% error rate**: Similar to the read operations, all 50 write requests were processed without errors. This confirms that the system can handle concurrent write operations reliably at this load.

- **Inconsistent Response Times**: The write operations show an even higher standard deviation (2016.96 ms), indicating a greater variability in response times. The fastest write operation was completed in 864 ms, while the slowest took more than 7 seconds. Such variation suggests that write operations, which typically involve more database interactions and transaction handling, are more susceptible to delays. This could be due to factors like slow disk writes or locks on database tables.

- **Throughput**: The system's throughput for write operations was 4.1 requests per second, which is notably lower than the read operations' throughput of 6.5 requests per second. Write operations tend to require more resources, such as locking mechanisms and database updates, which can explain the lower throughput.

- **Crescent Response Times**: As with read operations, the response time graph for write requests also shows an upward trend. This indicates that as the number of write operations increases, the system's ability to handle them in a timely manner decreases. The consistent increase could point to database bottlenecks or inefficient transaction handling under load.

4.4.2.2. 1000 Concurrent Users

Upping the level to 1000 concurrent users should prove to be a difficult task for the current API as it has never needed this level of availability.

The following parameters were used for testing the API:

Threads (Users): 1000

Ramp up: 5 seconds

Loop: 1

These parameters produced the following results, shown in Figure 14, 15, 16 and 17.

Read Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	122984	1096	246030	71374.58	0.00%	4.0/sec	10.66	1.42	2741.9
TOTAL	1000	122984	1096	246030	71374.58	0.00%	4.0/sec	10.66	1.42	2741.9

Figure 14 - 1000 Concurrent Users Summary Report - Read Operation

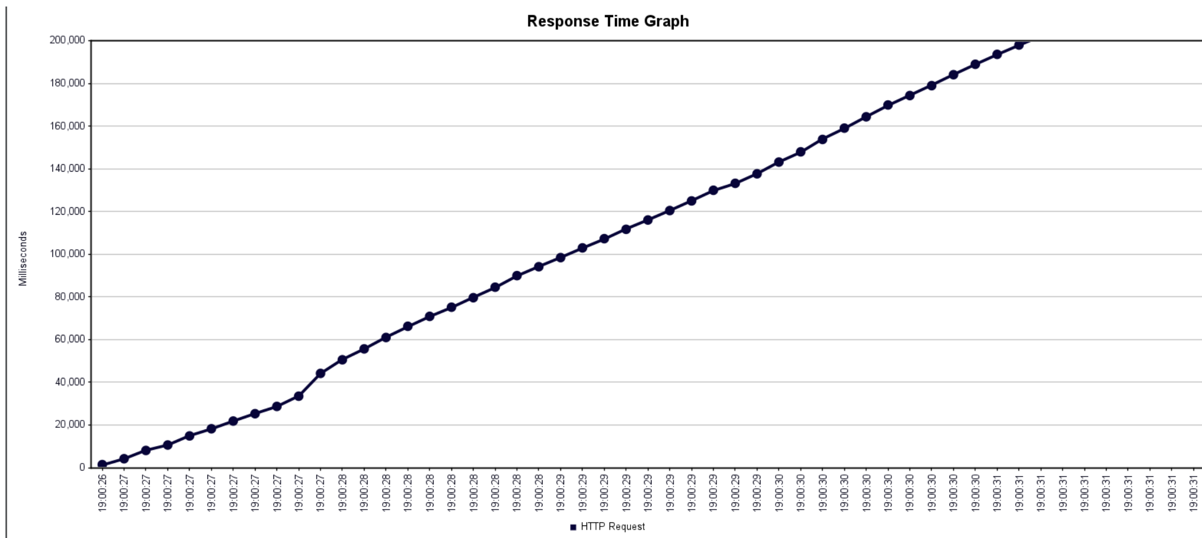


Figure 15 - 1000 Concurrent Users Response Time Graph - Read Operation

Takeaways:

- **0% error rate** – All 1000 requests were successfully answered
- **High Response times** - The average response time for all requests is extremely high. On average, it took about **123 seconds** for the API to respond to a request, which is far from an acceptable performance level for an API.
- **Throughput** - The API processed **4 requests per second** on average, which is similar to the previous test result. This throughput rate indicates that the API is struggling to scale with the increased load (1000 requests), as it has not improved in throughput even though the number of requests increased.
- **Crescent Response Times** – The Response Time Graph clearly shows the uptrend in response times as requests come in. Confirming what the other data reports indicated.

Write Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST HTTP Request	1000	92426	880	172973	49603.98	0.00%	5.6/sec	4.34	5.36	793.0
TOTAL	1000	92426	880	172973	49603.98	0.00%	5.6/sec	4.34	5.36	793.0

Figure 16 - 1000 Concurrent Users Summary Report - Write Operations

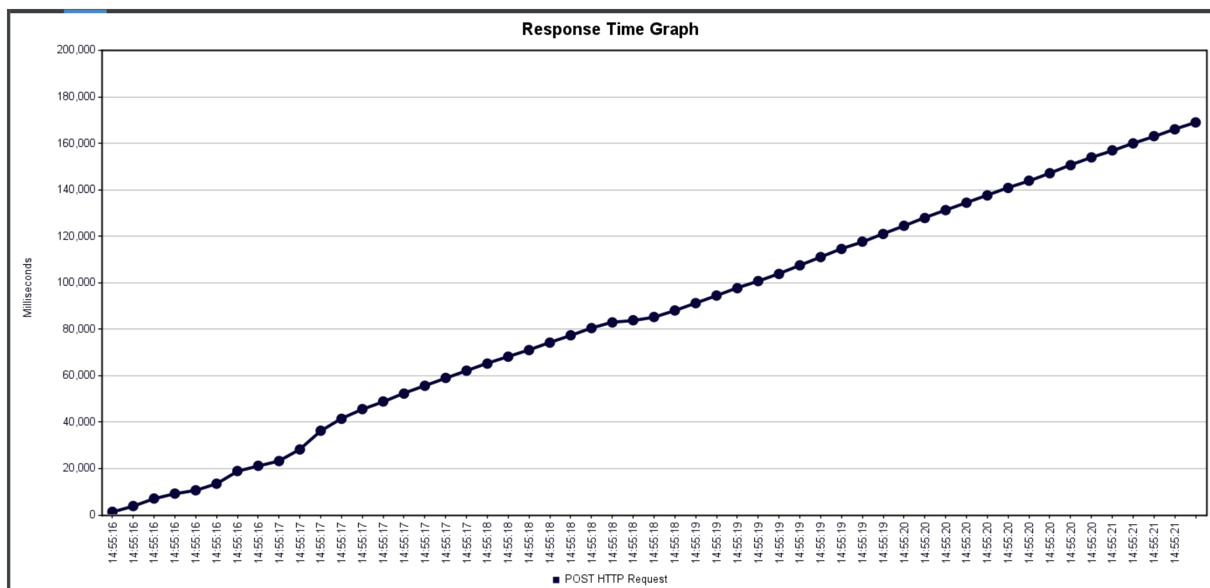


Figure 17 - 1000 Concurrent Users Response Time Graph - Write Operations

- **0% error rate**: All 1000 write requests were successfully processed without any errors, demonstrating stability under this increased load. This is a good indication that the system can handle a higher number of concurrent requests without failure.

- **High Response Times**: The average response time for the write operations was 92426 ms (92.4 seconds), which is significantly higher than acceptable for API performance. The maximum response time peaked at 172973 ms (172.9 seconds), while the minimum was 880 ms, showing significant inconsistency. This indicates potential bottlenecks, likely due to database or resource constraints. These response times are much too high for an efficient system.

- **Throughput**: The throughput of 5.6 requests per second remained consistent, despite the increase in user load from previous tests (e.g., the 50 user tests). This shows that the API is struggling to scale under the increased number of requests. Ideally, as the number of users increases, the throughput should improve or remain stable, but this static throughput suggests resource exhaustion or poor load handling.

- **Crescent Response Times**: The response time graph shows a clear upward trend, confirming that as more requests come in, the system slows down significantly. This behavior is consistent with bottlenecks that worsen as the number of concurrent operations increases, possibly due to database locks, insufficient server resources, or inefficient query handling.

4.4.2.3. 5000 Concurrent Users

Moving on to 5000 concurrent users, this type of demand should start to prove the lack of availability of the current API.

The following parameters were used for testing the API:

Threads (Users): 5000

Ramp up: 5 seconds

Loop: 1

These parameters produced the following results, shown in Figure 18, 19, 20 and 21.

Read Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	5000	78262	0	345811	87859.12	71.06%	14.3/sec	34.27	1.47	2459.9
TOTAL	5000	78262	0	345811	87859.12	71.06%	14.3/sec	34.27	1.47	2459.9

Figure 18 - 5000 Concurrent Users Summary Report

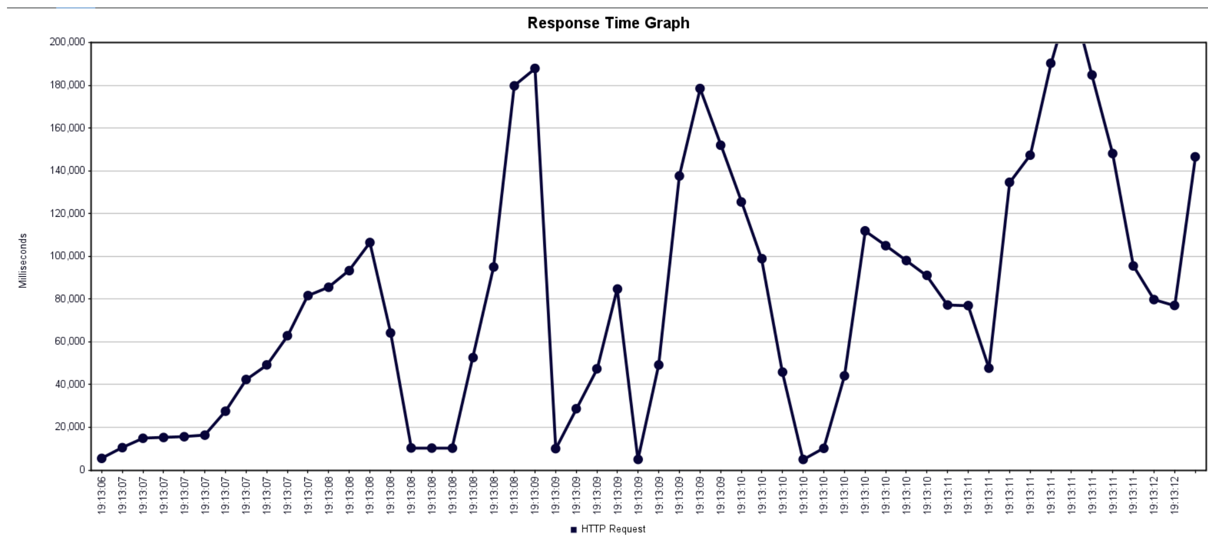


Figure 19 - 5000 Concurrent Users Response Time Graph | 100ms interval

Takeaways:

- **71% error rate** – The majority of the requests failed. This starts to show the lack of availability for the API when met with a spike of requests.

- **High Response times** - The average response time is extremely high. Each request took an average of over a minute to complete, indicating a severe performance issue.

- **Throughput** - The system processed an average of **14.3 successful requests per second**. Although this is a notable improvement compared to previous tests with lower throughput, the fact that a high percentage of requests failed diminishes the relevance of this metric. The throughput is not reliable because most requests were unsuccessful.

- **Erratic Response Times** – The graph shows significant instability in response times for 5000 concurrent users, with frequent spikes reaching up to 180,000 ms (180 seconds) and drops to near zero, indicating severe performance degradation and a high error rate. The system struggles to maintain consistent response times, suggesting it's overwhelmed by the load.

Write Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST HTTP Requ...	5000	78951	0	348384	96313.46	67.82%	14.3/sec	27.25	4.39	1956.5
TOTAL	5000	78951	0	348384	96313.46	67.82%	14.3/sec	27.25	4.39	1956.5

Figure 20 - 5000 Concurrent Users Summary Report - Write Operations

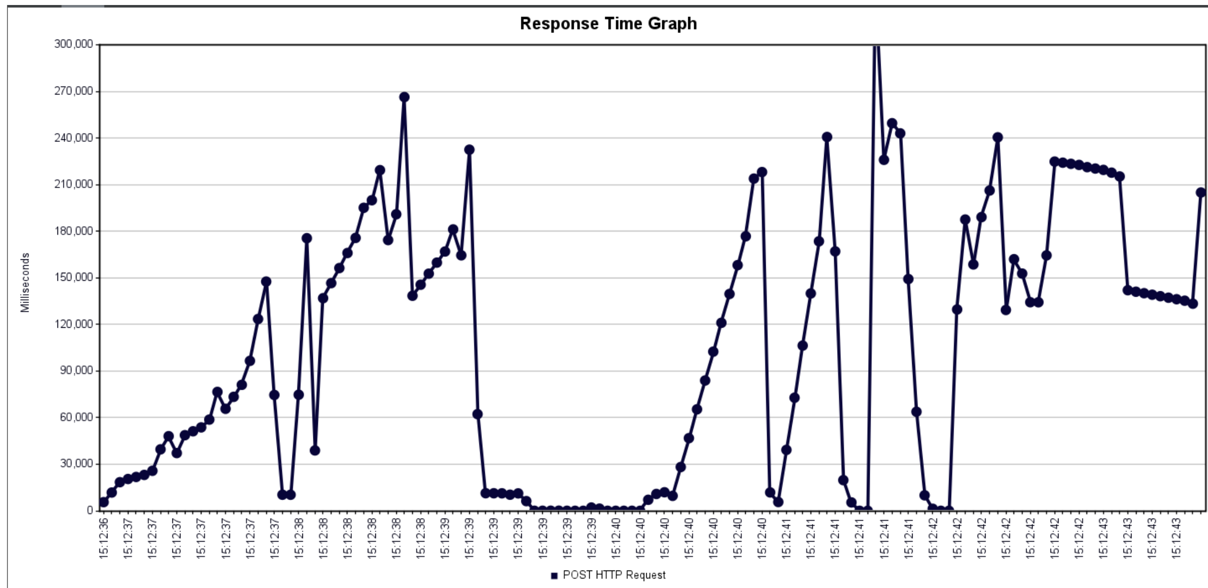


Figure 21 - 5000 Concurrent Users Response Time Graph - Write Operations

4.4.2.4. 10 000 Concurrent Users

Coming to the final test of 10 000 concurrent users, based on the previous test, this should prove to be extremely difficult for the API to handle, as with half the users the error rate was already at 70%. As stated previously, write operations will not be tested at this level, since the 10 000 concurrent user level is only intended for read operations.

The following parameters were used for testing the API:

Threads (Users): 10 000

Ramp up: 5 seconds

Loop: 1

These parameters produced the following results, shown in Figure 22 and 23.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10000	100227	0	479285	97681.06	87.38%	20.0/sec	45.80	0.90	2344.8
TOTAL	10000	100227	0	479285	97681.06	87.38%	20.0/sec	45.80	0.90	2344.8

Figure 22 - 10 000 Concurrent Users Summary Report

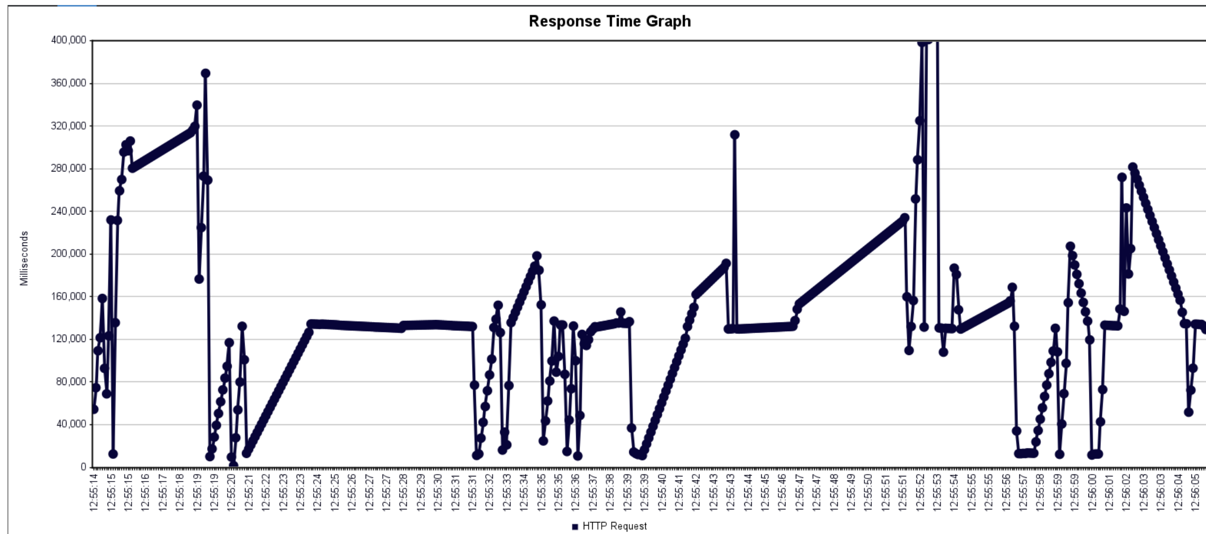


Figure 23 - 10 000 Concurrent Users Response Time Graph | 100 ms Interval

Takeaways:

- **87% error rate** – This indicates that **87.38% of the requests failed**, meaning only about 12.62% of the requests were processed successfully. This extremely high error rate suggests that the system is severely struggling to handle the load of 10,000 concurrent users, with most requests either timing out or failing due to resource exhaustion or other limitations.

- **High Response times** - The average response time is extremely high, with each request taking over **100 seconds** to complete. This suggests that the system is heavily overloaded and cannot process requests efficiently.

- **Throughput** - The system was able to handle an average of 20 requests per second. While this is an improvement over previous tests (likely because many requests are failing quickly), the high error rate diminishes the significance of this number. The system is not able to reliably process even a moderate number of requests due to the high failure rate.

- **Inconsistent Response Times** – The graph shows significant **inconsistency in response times**, with frequent spikes reaching up to **360,000 ms** and occasional drops, indicating the system is struggling under load. The high and fluctuating latency suggests resource bottlenecks and performance issues, with sustained high response times reflecting the system's inability to handle 10,000 concurrent users effectively.

4.5. Improvement Proposal

This section presents alternative design strategies aimed at improving the performance and scalability of the current system. Based on the observed performance issues under heavy load, various architectural patterns and optimizations are evaluated to address bottlenecks and inefficiencies. The improvements focus on scalable solutions such as microservices, Command Query Responsibility Segregation (CQRS), load balancing, and other modern design patterns to better distribute the workload, optimize resource usage, and reduce response times, ensuring the system can handle increasing traffic demands more effectively.

4.5.1. Design

This part introduces alternative architectural diagrams for the system, continuing the use of the C4 Model for consistency with the previously documented architecture. Although not all levels of the C4+1 pattern were presented earlier, as they were not fully relevant to the scope of the thesis, the new diagrams will also focus on the most critical aspects. The key differences between the current and proposed architectures will be explained, with an emphasis on enhancing scalability, performance, and efficiency, before transitioning to the implementation phase. Since the improvement proposal focuses mainly on component level improvements, only level two diagrams will be shown.

4.5.1.1. Level 2 – Container

Logic View

The following image depicts the result from the research made throughout this thesis, showing that a microservice implementation would lead to a better performance of the system.

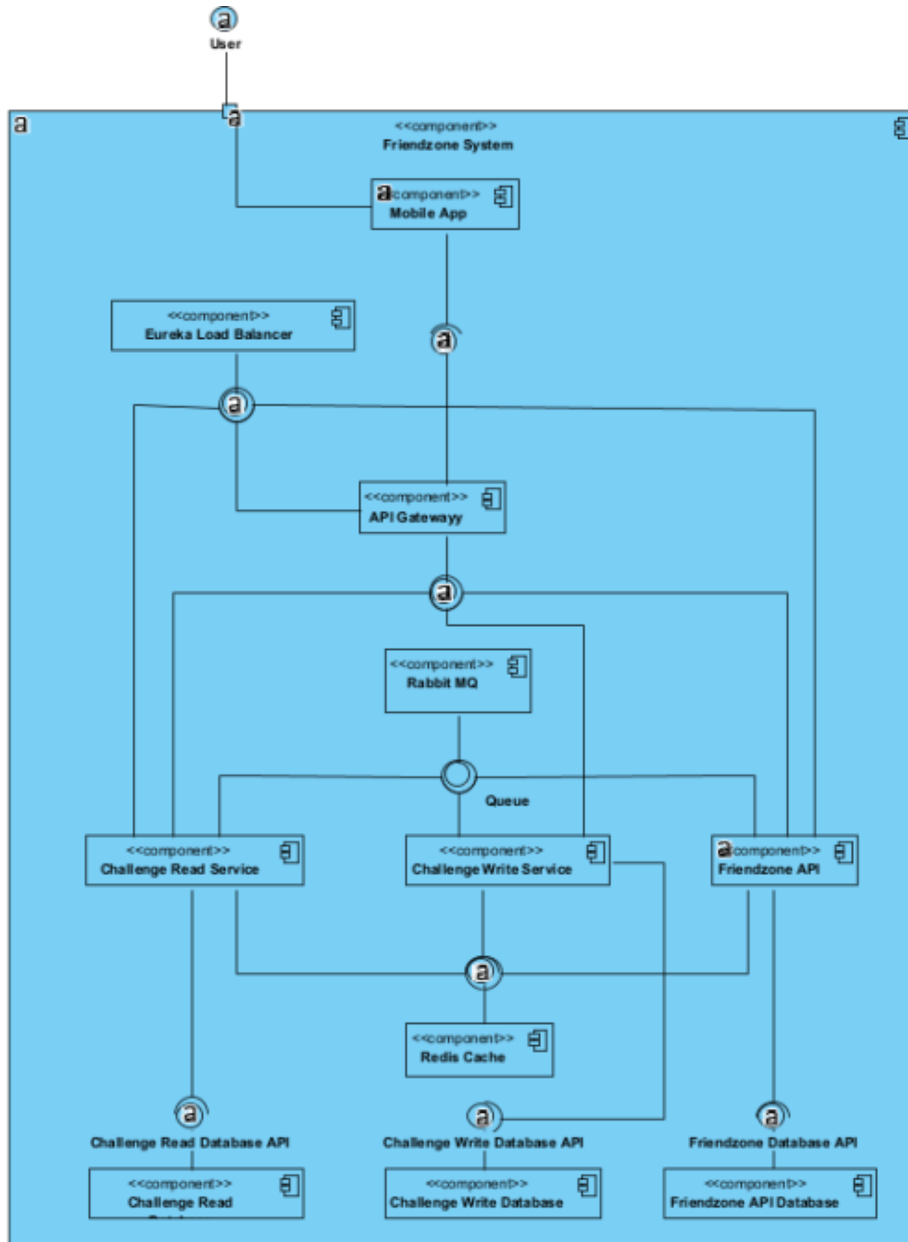


Figure 24 - Level 2 Logic View Diagram - Improved

Figure 24 illustrates the attempt of improving Friendzone's system, employing a microservices architecture with an API Gateway, a load balancer, a message broker (RabbitMQ) for asynchronous communication and a Redis Cache database for caching frequently accessed information. This setup allows the system to handle both read and write operations efficiently while ensuring scalability and resilience.

The API Gateway serves as the primary entry point for incoming requests, routing them to the appropriate microservices based on the load balancer's feedback. The Eureka Load Balancer is responsible for dynamically discovering available service instances, distributing requests across multiple instances, and ensuring that traffic is directed to healthy, available services. This configuration helps the system scale effectively while maintaining fault tolerance.

In the design, RabbitMQ is used to ensure the databases across services remain synchronized and consistent. Rather than being used for direct request routing, RabbitMQ handles asynchronous communication between the services to propagate changes and updates. This mechanism helps reduce the latency of write operations for the user by performing database synchronization in the background. It also enables the system to handle high volumes of concurrent operations more effectively by decoupling the core services and ensuring that each can process tasks without waiting for others to complete their database updates.

The use of separate read and write databases for the Challenge Service aligns with the CQRS (Command Query Responsibility Segregation) pattern, which optimizes database performance by splitting the read and write workloads. The Challenge Read Service is optimized for quick data retrieval, while the Challenge Write Service handles more complex database transactions. This separation ensures efficient scaling and load balancing between services that handle frequent read requests and those responsible for more resource-intensive write operations.

Overall, the architecture depicted in Figure 27 is designed to handle large-scale traffic with modular, scalable microservices, supported by asynchronous messaging and dynamic service discovery. This structure is well-suited for environments where scalability, resilience, and fault tolerance are key requirements, though additional layers of monitoring and security could enhance its robustness in production settings.

Process View

The following image, Figure 25, depicts what a regular read operation looks like under the new design.

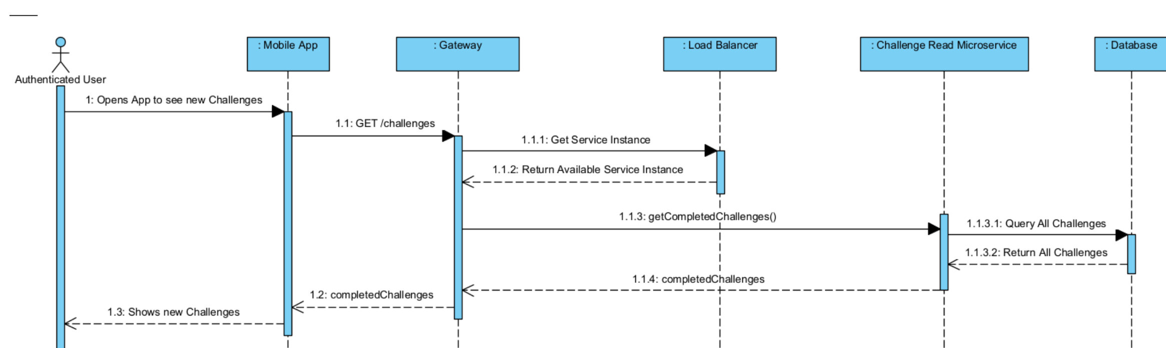


Figure 25 - Level 2 Process View Diagram | Read Operations - Improved

This sequence diagram specifically represents the read process within the system, demonstrating how a user retrieves challenge data, as opposed to any write operations.

The process begins when an authenticated user opens the mobile app, requesting to view new challenges. This action triggers a GET request to the API Gateway, which serves as the entry point for all client requests. The gateway does not handle the request alone but instead communicates with the Eureka Load Balancer to determine which instance of the Challenge Read Microservice is available. The load balancer plays a crucial role in ensuring that the request is routed to the appropriate microservice based on availability and current load.

Once the load balancer provides an available service instance, the gateway forwards the request to the Challenge Read Microservice, which is responsible for retrieving the requested data. The microservice then queries the database to fetch the list of completed challenges. Upon successful completion of the database query, the microservice returns the list of challenges to the API Gateway, which forwards the data back to the mobile app. The mobile app then displays the challenges to the user.

This diagram is focused entirely on the read process, where the system is querying and retrieving information, rather than altering or updating any data. The Challenge Read Microservice and its database interactions are at the heart of this process, emphasizing the system's ability to efficiently fetch and serve data to the user. The flow from the API Gateway to the load balancer and then to the read microservice shows the architecture’s design to handle high levels of concurrent read requests, ensuring data is served quickly and reliably from the system’s databases.

The following image, Figure 26, depicts what a regular write operation looks like under the new design.

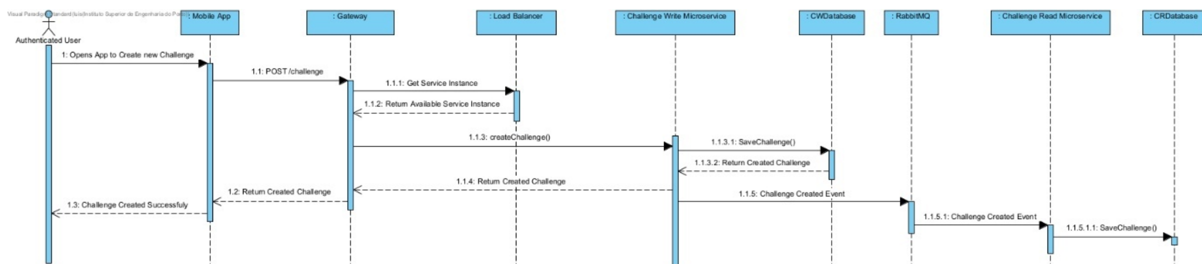


Figure 26 - Level 2 Process View Diagram | Write Operations - Improved

This sequence diagram illustrates the write process for creating a new challenge in the system. The process begins with an authenticated user interacting with the mobile app to create a new challenge. Upon initiating the action, the mobile app sends a POST request to the /challenge endpoint via the API Gateway. This request contains the necessary data for creating the challenge.

The API Gateway serves as the entry point for all incoming requests, including this write operation. Upon receiving the request, the gateway interacts with the Eureka Load Balancer to determine which instance of the Challenge Write Microservice is available to process the request. The load balancer's role is crucial in ensuring that the request is routed to the appropriate service instance, balancing traffic and maintaining system efficiency.

Once the Challenge Write Microservice receives the request, it proceeds to save the new challenge data by performing a database write operation. It interacts with its associated Challenge Write Database (*CWDatabase*) to execute the *SaveChallenge*() operation, ensuring that the challenge data is stored persistently. After successfully saving the challenge, the write microservice returns the created challenge information back through the API Gateway to the mobile app, providing the user with confirmation that the challenge was created successfully.

In addition to completing the immediate write operation, the Challenge Write Microservice triggers a Challenge Created Event. This event is sent asynchronously via the RabbitMQ message broker to notify the Challenge Read Microservice about the newly created challenge. RabbitMQ allows the system to decouple the write and read operations, ensuring that the read microservice can update its database without direct coupling or blocking the write process.

Upon receiving the event from RabbitMQ, the Challenge Read Microservice processes it and performs its own *SaveChallenge*() operation, ensuring that the Challenge Read Database (*CRDatabase*) is updated with the newly created challenge. This mechanism ensures that the data across the system's read and write services remains consistent, while also maintaining the asynchronous nature of the communication between the microservices.

This entire process highlights the system's ability to handle write operations efficiently, while ensuring data consistency across the system through asynchronous updates facilitated by RabbitMQ. By leveraging asynchronous communication and decoupling the read and write processes, the system remains scalable and responsive, even under high load or complex interactions.

Physical View

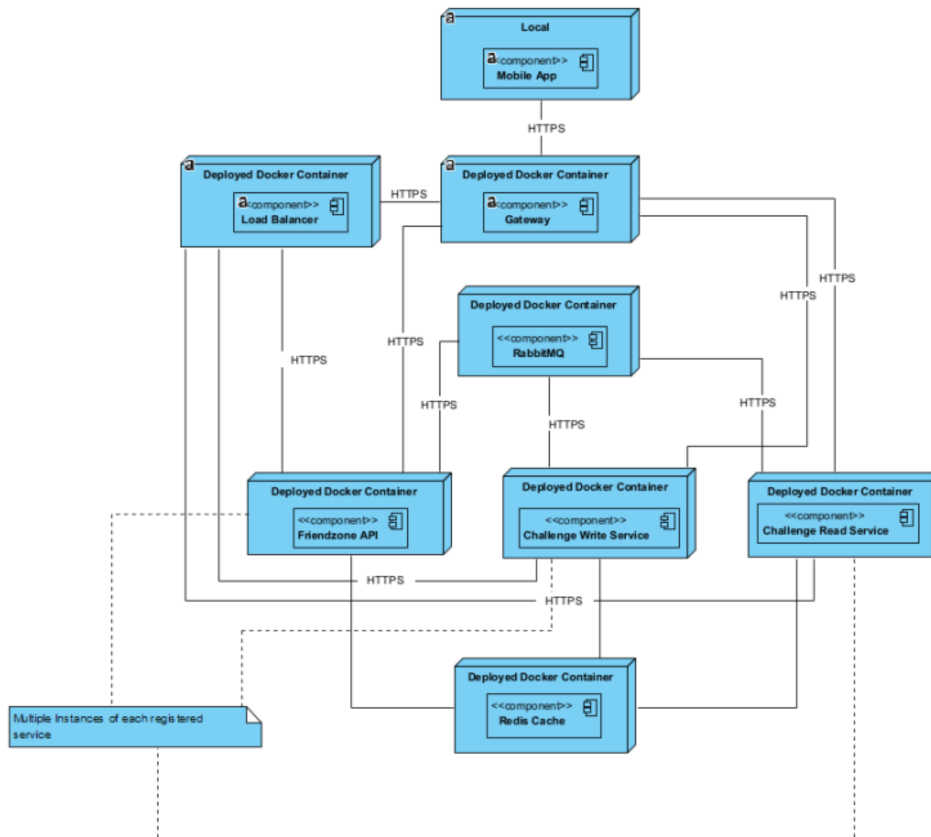


Figure 27 - Level 2 Physical View - Improved

This diagram, Figure 27, showcases the architecture of the system deployed using Docker containers for each of the core services, ensuring that each component is isolated, scalable, and easily deployable across various environments.

Starting from the local environment, we have the mobile app, which interacts with the system via HTTPS. The requests from the mobile app are sent to the API Gateway, which serves as the entry point for routing the requests to the appropriate services.

The Load Balancer, running in its own container, works alongside the gateway to distribute incoming requests across multiple instances of the microservices, ensuring that the system can handle high traffic and maintain high availability. When a request arrives, the load balancer identifies available instances of services and routes the traffic accordingly.

At the core of the architecture, the system makes use of RabbitMQ, also deployed within its own container, for asynchronous message passing between services. This is particularly useful in ensuring that the system remains decoupled and can efficiently process tasks such as keeping the databases in sync or handling events triggered by the write service.

Finally, the Redis Cache component is also visible, where all microservices share the same instance of cached data, in order to improve response speeds and reduce database overload.

The diagram shows multiple core microservices, such as:

- **Challenge Write Service**: Responsible for handling write operations, such as creating or updating challenges. It communicates with the Challenge Write Database and sends messages through RabbitMQ to update other services.

- **Challenge Read Service**: This service handles read operations, fetching data from the Challenge Read Database and delivering it to the user.

- **Friendzone API**: The remainder of the Friendzone system, responsible for every other business logic use case previously available.

The architecture follows a microservices approach with Docker containers allowing for easy scalability—meaning each microservice can have multiple instances (as noted in the bottom right) to ensure the system can handle larger loads without performance degradation.

5. Implementation

This chapter focuses on the implementation phase, detailing the transition from a traditional monolithic API to a more modular and scalable microservices architecture. The chapter will describe the evolution of the system using architectural patterns like the Strangler Fig pattern, which enables a smooth and incremental migration of functionality from the monolith to microservices without disrupting existing operations. This approach ensures that portions of the monolithic API are gradually replaced by microservices, allowing the system to evolve organically while maintaining stability.

The Strangler Fig pattern is a common technique used during the migration from monolithic to microservices architecture. This pattern is especially effective because it allows different parts of the monolith to be replaced over time, reducing risk and allowing the system to evolve incrementally. As described in the literature, "the Strangler Fig pattern has proven useful for refactoring legacy systems by replacing them incrementally with microservices, ensuring minimal disruption and avoiding the complexities of a complete system rewrite" (Henry & Ridene, 2019). This pattern aligns well with the goal of keeping systems operational while evolving and scaling.

Additionally, the chapter will dive into the codebase, showcasing how various microservices were developed and how they interact with each other. It will illustrate how components like the API Gateway, message brokers, and load balancers are integrated to create a cohesive microservices architecture. Examples of the code and deployment strategies will be provided to demonstrate how these microservices are orchestrated and how they handle specific tasks.

Finally, the same performance and load tests performed in the analysis chapter will be rerun on the new microservices system. These tests will provide a direct comparison between the previous monolithic system and the new architecture, highlighting improvements in scalability, response times, and overall system resilience. This will help illustrate the effectiveness of the microservices transformation in addressing the performance bottlenecks identified earlier.

5.1. New Tech Stack

As stated in the previous chapter, the technology stack in use consisted of a simple Ruby on Rails API with a PostgreSQL database and a React Native frontend.

For this restructuring of the system, based on previous experience and research, the chosen technology will migrate the system to a Java-based implementation. Following the Strangler Fig pattern, only one core component of the monolithic system—the Challenge Service—will be migrated initially. The Challenge Service was divided into two microservices: a read service and a write service, both created as part of a Maven Java

project. This separation follows the CQRS (Command Query Responsibility Segregation) approach, allowing read and write operations to be handled independently and efficiently.

In addition to the migration of the Challenge Service, the API Gateway and Load Balancer were also implemented in Java, leveraging frameworks like *Spring Cloud Gateway* for routing and *Spring Cloud Netflix Eureka* for service discovery and load balancing. These components ensure that requests are appropriately routed to the correct microservices, while the load balancer distributes incoming traffic across available instances, maintaining scalability and performance.

To ensure a smooth migration, the existing Ruby on Rails API will continue to run alongside the new Java-based microservices. It connects to both the gateway and load balancer, allowing it to handle other services and endpoints that have not yet been migrated. This hybrid approach ensures that the system can maintain full functionality during the migration process without disruptions to users.

Additionally, Redis Cache has been integrated into the tech stack to enhance performance by caching frequently accessed data. This reduces the load on the database and speeds up read-heavy operations like fetching challenges. By using Redis as a distributed cache, all microservice instances can quickly access cached data, ensuring consistent and fast responses even under heavy traffic. This caching strategy helps decrease latency, improve scalability, and maintain system reliability during peak loads.

To orchestrate the deployment of all services, including the new Java microservices, the API Gateway, the load balancer, and the Rails API, Docker was utilized. Each component is containerized, ensuring consistency across development, testing, and production environments. Docker Compose is used to manage the services, allowing for seamless orchestration and communication between them, as well as easy scalability. This setup provides a flexible deployment process and supports future migrations of other monolithic services to microservices in a controlled and incremental manner.

5.2. New Codebase

As previously mentioned, Java has been selected as the language of choice for migrating key parts of the system. The implementation of the Challenge Service as two distinct microservices—one for read operations and another for write operations—forms the foundation of the new codebase. Both services are developed using Java, structured within a Maven project, which enables efficient dependency management and a modular architecture. The following sections will dive into the code structure and key functionalities of the microservices, illustrating how they integrate into the broader system architecture through the API Gateway and Load Balancer.

5.2.1. Read and Write Service Configuration

Both the Challenge Read and Write Services were implemented as a Java-based microservice using Spring Boot, and its main class is crucial for setting up the essential configurations for this service. These services rely on annotations to enable key functionalities like service discovery and messaging integration.

The following code snippet, Code Snippet 1, shows the main class of the Challenge Read Service, equal to the Challenge Write Service in terms of configuration:

```
@EnableRabbit
@EnableEurekaClient
@SpringBootApplication
public class ChallengeServiceReadApplication {
    public static void main(String[] args) {
        SpringApplication.run(ChallengeServiceReadApplication.class, args);
    }
}
```

Code Snippet 1 - Challenge Service Read Main Class

- **@SpringBootApplication**: This annotation marks the entry point for the Spring Boot application. It combines @Configuration, @EnableAutoConfiguration, and @ComponentScan, enabling automatic configuration and component scanning.
- **@EnableEurekaClient**: This annotation registers the service with **Eureka**, which is the service discovery component in the architecture. By registering as a **Eureka client**, the service can discover other services and be discovered by the **API Gateway** for request routing and load balancing.
- **@EnableRabbit**: This enables the **RabbitMQ** integration, allowing the Challenge Read Service to subscribe to message queues. This is essential for asynchronously receiving updates when challenge data changes elsewhere in the system, ensuring that the read service remains consistent with the data in the write service.

These configurations are fundamental to integrating both the **Challenge Read and Write Services** into the microservices ecosystem, allowing it to function within a distributed system while maintaining scalability and reliability. Since this is a Maven project, all necessary dependencies are registered in the module's *pom.xml*, as seen in Code Snippet 2, the main dependencies for the web application, the Eureka client and the RabbitMQ message broker.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot</artifactId>
    <version>2.7.6</version>
</dependency>
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
</dependency>
```


```
Code Snippet 2 - Challenge Service Read Pom.XML Main
Dependencies<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot</artifactId>
    <version>2.7.6</version>
</dependency>
<dependency>
    <groupId>org.springframework.amqp</groupId>
    <artifactId>spring-rabbit</artifactId>
</dependency>
```


Code Snippet 2 - Challenge Service Read Pom.XML Main Dependencies

Furthermore, another important configuration file is the *application.yml*, as it is responsible for connecting and registering the application in all the necessary services.

The *application.yml*, Code Snippet 3, file contains the configuration settings for the Challenge Read Service. This configuration controls various aspects of the service, such as the server port, RabbitMQ, PostgreSQL database connections, and Eureka service discovery.

```yaml
server:
  port: 8081
spring:
  application:
    name: ChallengeServiceRead
  rabbitmq:
    host: rabbitmq
    username: guest
    password: guest
    port: 5672
  datasource:
    password: password
    url: jdbc:postgresql://postgres-challenge-
r:5433/challenge-db-r
    username: root
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect:
org.hibernate.dialect.PostgreSQLDialect
        format_sql: true
    show-sql: true
eureka:
  client:
    service-url:
      defaultZone: http://eureka-
server:8761/eureka/  # Eureka server URL
  instance:
    prefer-ip-address: true
file:
  uploadDir: DEV/FileUploads
```

Code Snippet 3 - Challenge Service Read application.yml file

This configuration is similar for both the Read and Write services, as both connect to the same gateway and load balancer and have overall similar dependencies.

5.2.2. Gateway

The Gateway module has a straightforward code structure, as its complexity largely resides in its configuration rather than its core code. The following code snippet, Code Snippet 4, shows the main class of the gateway application, which is relatively simple yet plays a crucial role in enabling the gateway to function within a microservices environment.

The GatewayApplication class is a Spring Boot application marked with the @SpringBootApplication annotation, which enables Spring Boot's auto-configuration and component scanning features. Additionally, the class is annotated with @EnableDiscoveryClient, allowing the gateway to register with Eureka for service

discovery. By integrating with Eureka, the gateway can dynamically route incoming requests to available microservice instances.

In terms of dependencies, the API Gateway requires several key libraries, including Spring Cloud Gateway for managing routing and API features, Spring Cloud Netflix Eureka Client for service discovery, and Spring Boot Starter Web to support web-related functionalities. These dependencies allow the gateway to route requests efficiently while ensuring high availability and scalability across the microservices it interacts with.

```
@SpringBootApplication
@EnableDiscoveryClient
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

Code Snippet 6 - Gateway Application.yml file

```
@SpringBootApplication
@EnableDiscoveryClient
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

Code Snippet 4 - Gateway Main Class

In Code Snippet 5, the application.yml file configures the API Gateway, setting the foundation for routing and service discovery in the microservices architecture. The gateway is configured to run on port 8080, with the application name specified as gateway.

This configuration uses Eureka for service discovery, enabling the gateway to dynamically find and route requests to services registered with the Eureka server. The locator is enabled under spring.cloud.gateway.discovery, allowing the gateway to automatically discover services like the Challenge Read Service, Challenge Write Service, and the Rails API.

The routes section defines specific routing rules:

- Requests with the path */read/*** are routed to the Challenge Read Service *(lb://challengeserviceread).*

- Requests with the path */write/*** are routed to the Challenge Write Service *(lb://challengeservicewrite).*

- Requests with the path */api/*** are routed to the legacy Rails API *(lb://railsapiservice).*

Each route is associated with a predicate that specifies how requests are matched and forwarded, with paths clearly defining how different requests are routed to the correct services.

Additionally, Eureka client settings ensure the gateway is properly registered with the Eureka server, and debug logging is enabled for Spring Cloud Gateway to assist in monitoring and troubleshooting. This configuration is essential for ensuring that the gateway efficiently routes traffic within the microservices architecture while maintaining flexibility and scalability.

```yaml
server:
  port: 8080

spring:
  application:
    name: gateway

  cloud:
    gateway:
      discovery:
        locator:
          enabled: true   # Enable service discovery
through Eureka
          lower-case-service-id: true   # Use lower case
service IDs when routing
      routes:
        - id: challenge-service-read
          uri: lb://challengeserviceread   # Load balanced
URI to ChallengeServiceRead
          predicates:
            - Path=/read/**   # Route requests starting
with /read to this service

        - id: challenge-service-write
          uri: lb://challengeservicewrite   # Load balanced
URI to ChallengeServiceWrite
          predicates:
            - Path=/write/**   # Route requests starting
with /write to this service

        - id: rails-api
          uri: lb://railsapiservice   # Load balanced URI
to RailsAPIService
          predicates:
            - Path=/api/**   # Route requests starting with
/api to this service

eureka:
  client:
    serviceUrl:
      defaultZone: http://eureka-server:8761/eureka/   #
URL of the Eureka server
  instance:
    preferIpAddress: true   # Prefer IP address over
hostname for registration

logging:
  level:
    org.springframework.cloud.gateway: DEBUG   # Enable
debug logging for Gateway (optional)
```

Code Snippet 5 - Gateway Application.yml file

5.2.3. Eureka Load Balancer

Similar to the Gateway module, the Eureka load balancer has a very simple main class, its intricacy is in the configuration. In the following snippet, Code Snippet 6, Eureka's main class demonstrates its annotations and configurations.

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class,
args);
    }
}
```

Code Snippet 6 - Eureka Main Class

The EurekaServerApplication class shown in the code snippet sets up the Eureka Server, which is a critical part of the microservices architecture for service discovery. Annotated with @EnableEurekaServer, this class launches a Spring Boot application that serves as the Eureka server.

@EnableEurekaServer: This annotation configures the application as a Eureka Server, enabling other microservices in the architecture (like the gateway, challenge services, etc.) to register themselves and discover one another dynamically.

In this configuration file, Code Snippet 7, the Eureka Server is set up to manage service discovery for the microservices architecture. The server is configured to run on port 8761, which is the default port for a Eureka server.

Key configuration properties include:

- **eureka.client.register-with-eureka: false** : This instructs the Eureka server not to register itself. Since this is the central service registry, it does not need to register with other Eureka servers.

- **eureka.client.fetch-registry: false** : This disables the fetching of the service registry from other Eureka servers. This is because this instance is the authoritative registry for the microservices, meaning it does not need to communicate with other servers to obtain service details.

- **eureka.server.wait-time-in-ms-when-sync-empty: 0** : This property ensures that the server starts immediately, even if no services are registered at startup. This can be useful in development environments or when the service discovery system is being initialized.

- **spring.application.name: eureka-server** : This specifies the name of the application in the Spring ecosystem as eureka-server, allowing other services to recognize and interact with it easily.

This configuration establishes the Eureka server as the central service registry, allowing other microservices to register themselves and discover each other in the system.

```yaml
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false  # Eureka server should not register itself
    fetch-registry: false        # Disable fetching registry from other Eureka servers
  server:
    wait-time-in-ms-when-sync-empty: 0  # Immediately start even if the registry is
empty

spring:
  application:
    name: eureka-server
```

Code Snippet 7 - Eureka Application.yml file

5.2.4. RabbitMQ Broker

In this section, the focus will be on the configuration and integration of the RabbitMQ message broker within the system, specifically as it relates to the Challenge Write Service. By using the Challenge Write Service as an example, the following implementation will demonstrate how RabbitMQ is configured and utilized for handling message queues and inter-service communication.

Starting with the RabbitMQConfig class, it is a configuration class for setting up RabbitMQ in the Challenge Write Service. It defines key RabbitMQ components, such as exchanges, queues, and bindings, allowing the service to communicate asynchronously with other parts of the system.

- **Queue and Exchange Setup**: The *queue_main()* method creates an anonymous queue, while the *exchange()* method sets up a DirectExchange, as seen in Code Snippet 8, with a specific exchange name (exchange). Direct exchanges are used to route messages to queues based on the routing keys.

```
@Bean
Queue queue_main() {
    return new AnonymousQueue();
}

@Bean
DirectExchange exchange() {
    return new DirectExchange(EXCHANGE);
}
```

Code Snippet 8 - RabbitMQ Queue and Exchange Setup

- **Bindings**: The class sets up three different bindings (*binding_create*, *binding_delete*, and *binding_update*), as seen in Code Snippet 9, that associate the anonymous queue with the direct exchange using specific routing keys for creating, deleting, and updating challenges.

```
@Bean
Binding binding_create(DirectExchange exchange, Queue queue_main) {
    return BindingBuilder.bind(queue_main).to(exchange).with(CHALLENGE_CREATE_RK);
}

@Bean
Binding binding_delete(DirectExchange exchange, Queue queue_main) {
    return BindingBuilder.bind(queue_main).to(exchange).with(CHALLENGE_DELETE_RK);
}

@Bean
Binding binding_update(DirectExchange exchange, Queue queue_main) {
    return BindingBuilder.bind(queue_main).to(exchange).with(CHALLENGE_UPDATE_RK);
}
```

Code Snippet 9 - RabbitMQ Binding with Routing Keys

- **RabbitMQ Listener**: The *receiveMessageAndDistribute* method is annotated with @RabbitListener, indicating that it listens to messages from the specified RabbitMQ queue (in this case, the dynamically generated *queue_main*). When a message is received, it processes the message based on its routing key, which helps determine what action should be taken, to either create, update or delete a challenge, as seen in Code Snippet 10.

```
@RabbitListener(queues = "#{queue_main.name}")
public void receiveMessageAndDistribute(byte[] messageBytes,
@Header("amqp_receivedRoutingKey") String routingKey) {

    CreateChallengeCommand event = (CreateChallengeCommand)
SerializationUtils.deserialize(messageBytes);

    switch (routingKey) {
        case RabbitMQConfig.CHALLENGE_CREATE_RK -> service.create(event);
        case RabbitMQConfig.CHALLENGE_DELETE_RK ->
service.deleteById(event.getId());
        case RabbitMQConfig.CHALLENGE_UPDATE_RK ->
service.updateById(event.getId());
    }
}
```

Code Snippet 10 - RabbitMQ Listener

- **Notifying other services**: In the *create()* method, Code Snippet 11, after the Challenge entity is saved, an asynchronous event is triggered by calling publishChallengeMessage(), which sends the serialized challenge to RabbitMQ using the CHALLENGE_CREATE_RK routing key. This allows other services, like the Challenge Read Service, to asynchronously receive the event and update their data.

```
@Override
public Challenge create(ChallengeDTO challengeDTO) {
    Challenge c = saveChallenge(
            challengeDTO.getStatus(),
            challengeDTO.getChallengeResourceType(),
            challengeDTO.getValue(),
            challengeDTO.getHidden(),
            challengeDTO.getExtraDescription(),
            challengeDTO.getVideoResourceThumbnailId(),
            challengeDTO.getVideoResponseThumbnailId(),
            challengeDTO.getChallengeResourceId(),
            challengeDTO.getChallengeResponseId(),
            challengeDTO.getChallengeResponseType(),
            challengeDTO.getUser().getId(),
            challengeDTO.getUserChallenged().getId()
    );

    publishChallengeMessage(serializeCreateObject(c),
RabbitMQConfig.CHALLENGE_CREATE_RK);

    return c;
}
```

Code Snippet 11 - Challenge Service Write - Create Challenge Method

5.2.5. Redis Cache

In this section, the focus will be on the configuration and integration of Redis as a caching solution within the system, using the Challenge Read Service as an example. By leveraging Redis, the microservices can improve performance by caching frequently accessed data, thus reducing the load on the database. The following implementation will showcase how Redis is configured and integrated into the Java-based microservices architecture to enhance data retrieval efficiency.

Starting with the RedisConfig class, this configuration class sets up Redis within the Challenge Read Service. It defines key Redis components, including the cache manager and cache configuration, allowing the service to interact with the Redis cache. This integration enables the service to store and retrieve cached data, reducing the need for repeated database queries. By using annotations like @Cacheable in service methods, the application can seamlessly utilize the caching layer to enhance response times and handle a larger volume of requests more efficiently, as seen in the following code snippets.

```
@Configuration
@EnableCaching
public class RedisConfig {

    @Bean
    public RedisCacheConfiguration cacheConfiguration() {
        return RedisCacheConfiguration.defaultCacheConfig()
                .entryTtl(Duration.ofMinutes(10))
                .disableCachingNullValues();
    }

    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory
redisConnectionFactory) {
        return RedisCacheManager.builder(redisConnectionFactory)
                .cacheDefaults(cacheConfiguration())
                .build();
    }
}
```

Code Snippet 12 - Redis Configuration Class

```
@Override
@Cacheable(value = "challengesCache")
public List<Challenge> getAllChallenges() {
    return repository.getAllChallenges();
}
```

Code Snippet 13 - Challenge Method with Caching Enabled

5.2.6. Orchestrating with Docker-Compose

The Docker-Compose configuration used in this system orchestrates the entire microservices architecture by managing multiple containers for services like Eureka, the API Gateway, and the Challenge Services. Each service runs in its own container, and Docker-Compose ensures they interact seamlessly within a shared network. Below is a breakdown of the key containers in this setup:

Eureka Server

In the **Eureka Server** container, the ports directive exposes port **8761** for service discovery. The networks section places the Eureka Server on the shared **my-network** bridge, allowing other containers to interact with it. This setup ensures that services can dynamically register and discover each other through Eureka, crucial for handling microservices in dynamic environments, as seen in Code Snippet 14.

```
eureka-server:
  image: eureka-server:latest
  ports:
    - "8761:8761"
  networks:
    - my-network
```

Code Snippet 14 - Docker-Compose Eureka Configuration

API Gateway

The API Gateway is exposed on port 8080 and depends on both the Eureka Server and RabbitMQ services being up before starting (*depends_on*), Code Snippet 15. The gateway is configured to interact with the Eureka Server for service discovery and to communicate with other services like the Challenge Read and Write Services. The networks key ensures that the gateway can interact with the rest of the architecture on the *my-network*.

```
gateway:
  image: gateway:latest
  ports:
    - "8080:8080"
  depends_on:
    - eureka-server
    - rabbitmq
  networks:
    - my-network
```

Code Snippet 15 - Docker-Compose Gateway Configuration

Challenge Read Service (8 Instances)

Each instance of the Challenge Read Service is defined with a unique port mapping, allowing multiple instances to run on different ports (e.g., 8081, 8083, etc.), Code Snippet 16. All instances depend on Eureka, the RabbitMQ message broker, and the PostgreSQL read database. The environment section configures the service with necessary database connection details, while the networks directive connects all instances to the shared network, facilitating interaction. 8 instances of the Challenge Read Service were created in order to meet the anticipated demand of 10 000 concurrent requests and, if needed, more instances can be easily added.

```
challenge-service-read-1:
  image: challenge-service-read:latest
  ports:
    - "8081:8081"
  depends_on:
    - eureka-server
    - postgres-challenge-r
    - rabbitmq
  environment:
    SPRING_DATASOURCE_URL:
jdbc:postgresql://postgres-challenge-
r:5432/challenge-db-r
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: password
  networks:
    - my-network
```

Code Snippet 16 - Docker-Compose Challenge Service Read Configuration

This pattern is replicated across 8 instances, with each instance having a unique external port (e.g., 8083, 8084, 8087).

Challenge Write Service (4 Instances)

The Challenge Write Service is defined similarly to the read service, but there are 4 instances, each mapped to a unique external port (e.g., 8082, 8085, etc.), Code Snippet 17. These services depend on Eureka, RabbitMQ, and the PostgreSQL write database. The configuration for connecting to the write database is provided in the environment variables. 4 instances of the Challenge Write Service were created to meet the demand of around 5000 concurrent requests, if more instances are needed, they can easily be added.

```
challenge-service-write-1:
  image: challenge-service-write:latest
  ports:
    - "8082:8082"
  depends_on:
    - eureka-server
    - postgres-challenge-w
    - rabbitmq
  environment:
    SPRING_DATASOURCE_URL:
jdbc:postgresql://postgres-challenge-
w:5432/challenge-db-w
    SPRING_DATASOURCE_USERNAME: root
    SPRING_DATASOURCE_PASSWORD: password
  networks:
    - my-network
```

Code Snippet 17 - Docker-Compose Challenge Write Service Configuration

PostgreSQL Databases (Read and Write)

The PostgreSQL read and write databases are each defined with their own respective containers. The volumes directive maps data directories from the host to the container to persist data between container restarts. The read database listens on port 5433, while the write database listens on port 5432, allowing the Challenge Read and Write services to interact with them separately, as seen in Code Snippet 18 and 19.

```
postgres-challenge-w:
  container_name: challenge-db-w
  image: postgres:13
  environment:
    POSTGRES_USER: root
    POSTGRES_PASSWORD: password
    POSTGRES_DB: challenge-db-w
    PGDATA: /var/lib/postgresql/data
  volumes:
    - postgres-challenge-w-
data:/var/lib/postgresql/data
  ports:
    - "5432:5432"
  networks:
    - my-network
  restart: unless-stopped
```

Code Snippet 18 - Docker-Compose Challenge Write Database Configuration

```yaml
postgres-challenge-r:
  container_name: challenge-db-r
  image: postgres:13
  environment:
    POSTGRES_USER: root
    POSTGRES_PASSWORD: password
    POSTGRES_DB: challenge-db-r
    PGDATA: /var/lib/postgresql/data
  volumes:
    - postgres-challenge-r-
data:/var/lib/postgresql/data
  ports:
    - "5433:5432"
  networks:
    - my-network
  restart: unless-stopped
```

Code Snippet 19 - Docker-Compose Challenge Read Database Configuration

RabbitMQ

The RabbitMQ container is configured to handle asynchronous messaging between microservices, ensuring efficient communication across different services such as the Challenge Read and Write Services. It uses the management image to provide a management UI, allowing for easier monitoring of message queues and exchanges.

The **ports** directive exposes two key ports:

- **5672**: The default port used by RabbitMQ for communication with microservices.
- **15672**: This port is used for the **RabbitMQ management dashboard**, where users can view queues, exchanges, and routing configurations.

RabbitMQ is defined as a dependency (*depends_on*) for both the **Challenge Read** and **Challenge Write** services, ensuring it is started before these services attempt to use it for messaging. Additionally, the environment section configures the default user credentials for accessing the RabbitMQ instance, as seen in Code Snippet 20.

```yaml
rabbitmq:
  image: rabbitmq:management
  environment:
    - RABBITMQ_DEFAULT_USER=guest
    - RABBITMQ_DEFAULT_PASS=guest
  ports:
    - "5672:5672"
    - "15672:15672"
  expose:
    - 5672
    - 15672
  networks:
    - my-network
  restart: unless-stopped
```

Code Snippet 20 - Docker-Compose RabbitMQ Configuration

Redis

This configuration sets up two Docker containers: one for Redis and another for RedisInsight. The Redis container runs the Redis server, using version 6.0.7, and maps the default Redis port 6379 from the container to the host machine, allowing other services to interact with the Redis instance. The data is persisted using the redis_volume_data volume, ensuring that data stored in Redis remains intact even if the container is restarted.

```
redis:
    image: redis:6.0.7
    container_name: redis
    restart: always
    volumes:
      - redis_volume_data:/data
    ports:
      - 6379:6379

  redis_insight:
    image: redislabs/redisinsight:1.14.0
    container_name: redis_insight
    restart: always
    ports:
      - 8001:8001
    volumes:
      - redis_insight_volume_data:/db
```

Code Snippet 21 – Docker-Compose Redis Cache Configuration

5.2.7. Deployed Result

After running the docker-compose file, if all services turned on correctly, the Eureka management UI should look like the following, Figure 28, where 8 Read, 4 Write, 1 Gateway and 1 Rails API are registered and ready to be used.

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CHALLENGESERVICEREAD	n/a (8)	(8)	UP (8) - f1733d12b610:ChallengeServiceRead:8081 , 0b275d5dbbf7:ChallengeServiceRead:8081 , f3ab730f4e66:ChallengeServiceRead:8081 , 8b13c9e03874:ChallengeServiceRead:8081 , 85953852cea7:ChallengeServiceRead:8081 , 8be3a641e6b6:ChallengeServiceRead:8081 , 333f7844a130:ChallengeServiceRead:8081 , 2948f26c358a:ChallengeServiceRead:8081
CHALLENGESERVICEWRITE	n/a (4)	(4)	UP (4) - 88ed323876be:ChallengeServiceWrite:8082 , 6a4f4099ed96:ChallengeServiceWrite:8082 , e19ec8f70e5f:ChallengeServiceWrite:8082 , a327edc9596c:ChallengeServiceWrite:8082
GATEWAY	n/a (1)	(1)	UP (1) - 407cca98ef75:gateway:8080
RAILSAPISERVICE	n/a (1)	(1)	UP (1) - localhost:RailsAPIService:3000

Figure 28 - Eureka Registered Services

Similarly, after logging in the RabbitMQ management UI, it should now be visible all the registered services in the “exchange” exchange, as seen in Figure 29.

Overview			Details			Network	
Name	**User name**	**State**	**SSL / TLS**	**Protocol**	**Channels**	**From client**	**To client**
172.18.0.10:50292 rabbitConnectionFactory#46a488c2:0	guest	running	○	AMQP 0-9-1	1	2 B/s	0 B/s
172.18.0.11:60056 rabbitConnectionFactory#6242ae3b:5	guest	running	○	AMQP 0-9-1	1	2 B/s	2 B/s
172.18.0.12:53876 rabbitConnectionFactory#49d831c2:5	guest	running	○	AMQP 0-9-1	1	0 B/s	2 B/s
172.18.0.13:56466 rabbitConnectionFactory#3ae126d1:2	guest	running	○	AMQP 0-9-1	1	0 B/s	2 B/s
172.18.0.14:36268 rabbitConnectionFactory#ef60710:2	guest	running	○	AMQP 0-9-1	1	0 B/s	2 B/s
172.18.0.15:48842 rabbitConnectionFactory#7e7f3cfd:4	guest	running	○	AMQP 0-9-1	1	0 B/s	0 B/s
172.18.0.16:39250 rabbitConnectionFactory#56399b9e:4	guest	running	○	AMQP 0-9-1	2	0 B/s	0 B/s
172.18.0.17:33862 rabbitConnectionFactory#3fba233d:5	guest	running	○	AMQP 0-9-1	1	0 B/s	0 B/s
172.18.0.19:39858 rabbitConnectionFactory#69d23296:3	guest	running	○	AMQP 0-9-1	2	0 B/s	0 B/s
172.18.0.7:47576 rabbitConnectionFactory#1a6dc589:1	guest	running	○	AMQP 0-9-1	2	0 B/s	0 B/s
172.18.0.8:57916 rabbitConnectionFactory#65ddee5a:5	guest	running	○	AMQP 0-9-1	1	0 B/s	0 B/s
172.18.0.9:35122 rabbitConnectionFactory#257e0827:3	guest	running	○	AMQP 0-9-1	1	0 B/s	0 B/s

Figure 29 - RabbitMQ Management UI | Exchange Connections

5.3. Load Testing

This chapter focuses on load testing the newly implemented microservices-based API to assess its performance under various levels of traffic. The purpose of these tests is to gather detailed metrics on scalability, response times, error rates, and throughput. These tests will help evaluate how well the restructured system handles concurrent requests and intensive workloads, allowing for a thorough analysis of the new architecture's performance.

Throughout this chapter, different scenarios will be tested, including high volumes of read and write operations through the Challenge Services. These tests aim to measure how the API Gateway, Eureka Server, and RabbitMQ handle distributed requests while ensuring that the system remains reliable and efficient.

To simulate these loads and track performance, Apache JMeter will be used again as the primary tool for conducting the tests, allowing for controlled stress testing and comprehensive monitoring of the API's behavior under varying levels of demand.

5.3.1. 50 Concurrent Users

Starting with the lowest level of demand, 50 concurrent users, both read and write operations will be tested.

The following parameters were used for testing the API:

Threads (Users): 50

Ramp up: 5 seconds

Loop: 1

These parameters produced the following results, shown in Figure 30, 31, 32 and 33.

Read Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET HTTP Request	50	187	13	1097	318.00	0.00%	9.8/sec	1.23	3.57	128.0
TOTAL	50	187	13	1097	318.00	0.00%	9.8/sec	1.23	3.57	128.0

Figure 30 - 50 Concurrent Users Summary Report | Read Operations | Upgraded

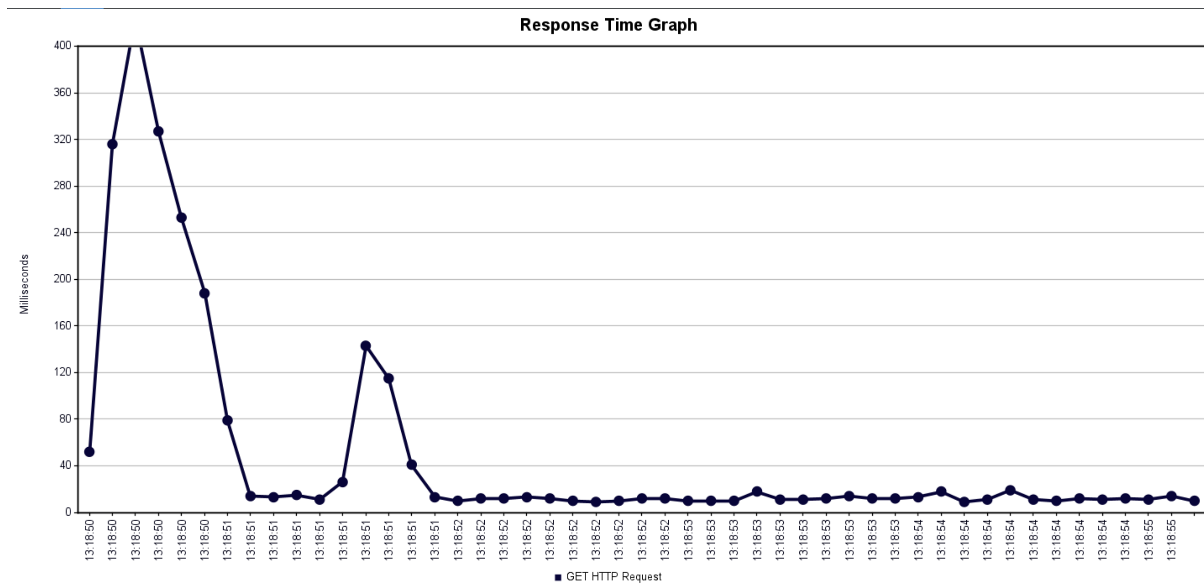


Figure 31 - 50 Concurrent Users Response Time Graph | Read Operations | Upgraded

Takeaways:

- **0% error rate**: All 50 requests were successfully processed, indicating that the API is stable and able to handle the current load without failure. This suggests the API can handle small bursts of traffic efficiently without losing any requests.

- **Low Response Time**: The average response time is 187 ms, with the minimum response time being 13 ms and the maximum reaching 1097 ms. The relatively low average suggests that the API responds quickly under low load. However, the relatively large standard deviation of 318 ms indicates some inconsistency in response times, with certain requests experiencing delays.

- **Throughput**: The system processed 9.8 requests per second, which is a decent throughput for the tested load. This metric shows that the API can handle almost 10 requests per second under the current conditions, maintaining a consistent processing speed.

- **Stable Performance**: With no errors and a solid throughput rate, the system appears to perform reliably for this test case. The variance in response times (as indicated by the standard deviation) should be monitored in higher load scenarios, as it could suggest potential bottlenecks when scaling up.

- **Initial Response Time Spike**: The response time graph shows an initial spike of nearly **400 ms**, followed by a rapid decrease to more stable and consistent times around **50 ms**. This suggests that there may be some warm-up time or initial latency when the first few requests are made, but the system stabilizes quickly and performs efficiently after that. This is likely due to initialization processes or caching mechanisms.

Write Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST HTTP Requ...	50	232	22	1788	329.02	0.00%	10.3/sec	8.81	8.55	877.6
TOTAL	50	232	22	1788	329.02	0.00%	10.3/sec	8.81	8.55	877.6

Figure 32 - 50 Concurrent Users Summary Report | Write Operations | Upgraded

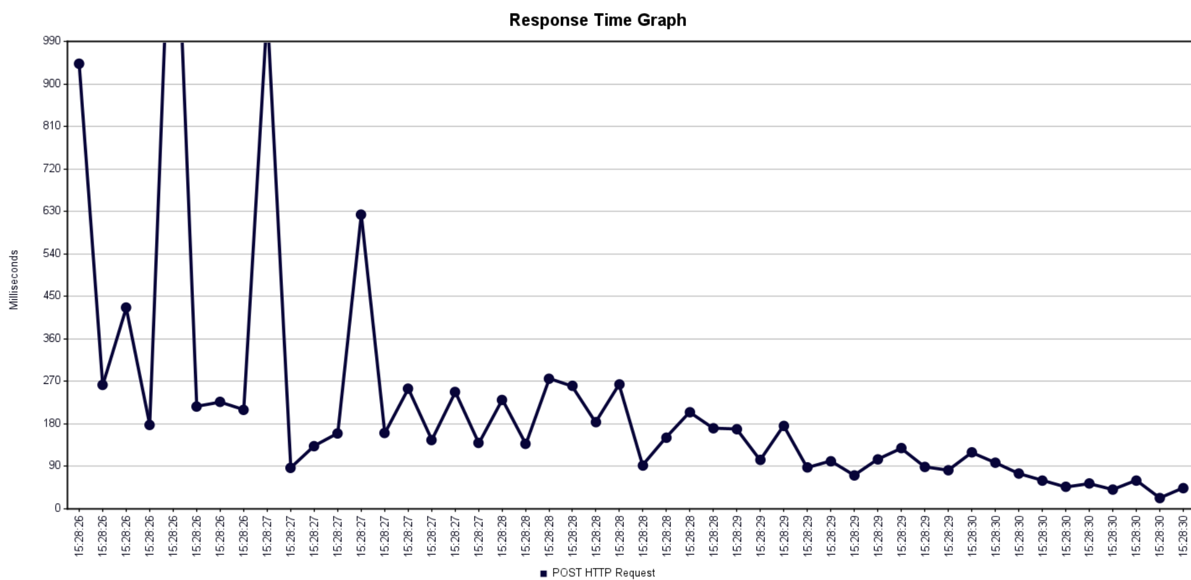


Figure 33 - 50 Concurrent Users Response Time Graph | Write Operations | Upgraded

Takeaways:

- **0% error rate** – All 50 requests were processed successfully, which indicates a stable performance under the load of 50 concurrent users. This demonstrates that the upgraded write operations can handle moderate traffic without causing any failures or errors.

- **Moderate Response Times** – The average response time is 232 ms, with a minimum of 22 ms and a maximum of 1788 ms. This suggests that while most requests are processed quickly, there are outliers with significantly higher response times. The high standard deviation of 329 ms reflects this inconsistency in response times.

- **Throughput** – The system managed to process 10.3 requests per second, which is a solid throughput given the current load of 50 concurrent users. This indicates that the system can handle multiple write operations concurrently, maintaining a decent rate of successful requests.

- **Response Time Instability** – The response time graph shows considerable fluctuation, especially in the beginning, where peaks reach up to 900 ms. However, after the initial spikes, response times stabilize and decrease significantly over time, suggesting some warm-up or caching effect, after which the system handles requests more efficiently.

- **Improved Performance** – Despite some spikes in response times, the overall performance appears to be an improvement, with response times steadily decreasing throughout the test. This stabilization indicates that the system becomes more efficient as it processes more requests.

5.3.2. 1000 Concurrent Users

Now with 1000 concurrent users, both read and write operations will be tested and some performance improvements should be noticeable.

The following parameters were used for testing the API:

Threads (Users): 1000

Ramp up: 5 seconds

Loop: 1

These parameters produced the following results, shown in Figure 34, 35, 36 and 37.

Read Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET HTTP Reque...	1000	8053	3724	11838	1438.84	0.00%	77.7/sec	9.71	28.31	128.0
TOTAL	1000	8053	3724	11838	1438.84	0.00%	77.7/sec	9.71	28.31	128.0

Figure 34 - 1000 Concurrent Users Summary Report | Read Operations | Upgraded

Figure 35 - 1000 Concurrent Users Response Time Graph | Read Operations | Upgraded

Takeaways:

- **0% error rate** – All 1000 requests were processed successfully, showing that the system can handle a significant load for read operations without any failures. This is a strong indicator of system reliability under high traffic conditions.
- **High Response Times** – The average response time is **8053 ms**, with a minimum of **3724 ms** and a maximum of **11,838 ms**. These high values indicate that the read operations are struggling under this load, with the average response time being over **8 seconds**. The **standard deviation of 1438.84 ms** further highlights the large variation in response times, suggesting that certain requests take significantly longer to process.

- **Throughput** – The system maintained a throughput of **77.7 requests per second**, which is relatively high considering the extended response times. This indicates that the API can still process a considerable number of requests per second despite the long response times, though the user experience may suffer due to the delay.

- **Increasing Response Time Pattern** – The response time graph shows a steady increase in response times, peaking around **9000 ms**, before gradually decreasing toward the end of the test. This pattern suggests that the system faces an initial bottleneck or resource exhaustion as the load increases, but eventually stabilizes as requests are processed.

Write Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST HTTP Requ...	1000	9477	2555	16029	2046.35	0.00%	58.3/sec	49.96	48.48	878.0
TOTAL	1000	9477	2555	16029	2046.35	0.00%	58.3/sec	49.96	48.48	878.0

Figure 36 - 1000 Concurrent Users Summary Report | Write Operations | Upgraded

Figure 37 - 1000 Concurrent Users Response Time Graph | Write Operations | Upgraded

Takeaways:

- **0% error rate** – All 1000 requests were successfully processed without any errors, demonstrating the system's ability to handle a significant number of concurrent **write** operations without failure. This suggests reliability under heavy write loads.
- **High Response Times** – The average response time for the requests was **9477 ms**, with a minimum of **2555 ms** and a maximum of **16,029 ms**. These response times are significantly high, with a **standard deviation of 2046 ms**, indicating notable variability in performance. Although the system manages to handle the load, the high response times could lead to slower user experiences under similar load levels.
- **Throughput** – The system maintained a throughput of **58.3 requests per second**, which is reasonable given the high response times, indicating that the system can still process a good number of requests, though there are delays in handling each one.
- **Gradual Increase in Response Times** – The response time graph shows a clear upward trend, with response times steadily increasing from **400 ms** to **4800 ms**, peaking midway through the test, and then stabilizing at a high range. This suggests that the system encounters bottlenecks or performance degradation as the load persists over time. However, the graph shows some stabilization after peaking, which could be attributed to the system adapting to the load.

5.3.3. 5000 Concurrent Users

Now with 5000 concurrent users, both read and write operations will be tested and some performance improvements should also be noticeable.

The following parameters were used for testing the API:

Threads (Users): 5000

Ramp up: 5 seconds

Loop: 1

These parameters produced the following results, shown in Figure 38, 39, 40 and 41.

Read Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET HTTP Request	5000	23694	35	32475	5837.53	0.00%	147.7/sec	18.46	53.80	128.0
TOTAL	5000	23694	35	32475	5837.53	0.00%	147.7/sec	18.46	53.80	128.0

Figure 38 - 5000 Concurrent Users Summary Report | Read Operations | Upgraded

Figure 39 - 5000 Concurrent Users Response Time Graph | Read Operations | Upgraded

Takeaways:

- **0% error rate** – All 5000 requests were processed successfully, with no errors, showing that the system can handle a high volume of read requests without failures. This indicates strong stability and reliability under load, even with a large number of concurrent users.

- **Moderate to High Response Times** – The average response time was 23,694 ms, with a minimum of 35 ms and a maximum of 32,475 ms. These numbers suggest that, while the system can manage the load, some requests experience substantial delays, with certain requests taking over 32 seconds. The standard deviation of 5837 ms further illustrates the inconsistency in response times, pointing to potential performance bottlenecks under heavy load.

- **Throughput** – The system achieved an impressive throughput of 147.7 requests per second, which is strong for a test involving 5000 concurrent users. Despite the longer response times, the system was able to handle a large volume of requests concurrently, maintaining an efficient flow of operations.

- **Response Time Trend** – The response time graph shows a gradual upward trend, peaking at around 27,000 ms, and then stabilizing at that level. This pattern suggests that the system initially struggles with the increased load but manages to stabilize at a higher response time. While the throughput remains consistent, the response time trends indicate the need for further optimizations to reduce latency.

Write Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
POST HTTP Requ...	5000	9633	606	23078	6504.15	35.50%	195.8/sec	298.29	105.09	1559.9
TOTAL	5000	9633	606	23078	6504.15	35.50%	195.8/sec	298.29	105.09	1559.9

Figure 41 - 5000 Concurrent Users Summary Report| Write Operations | Upgraded

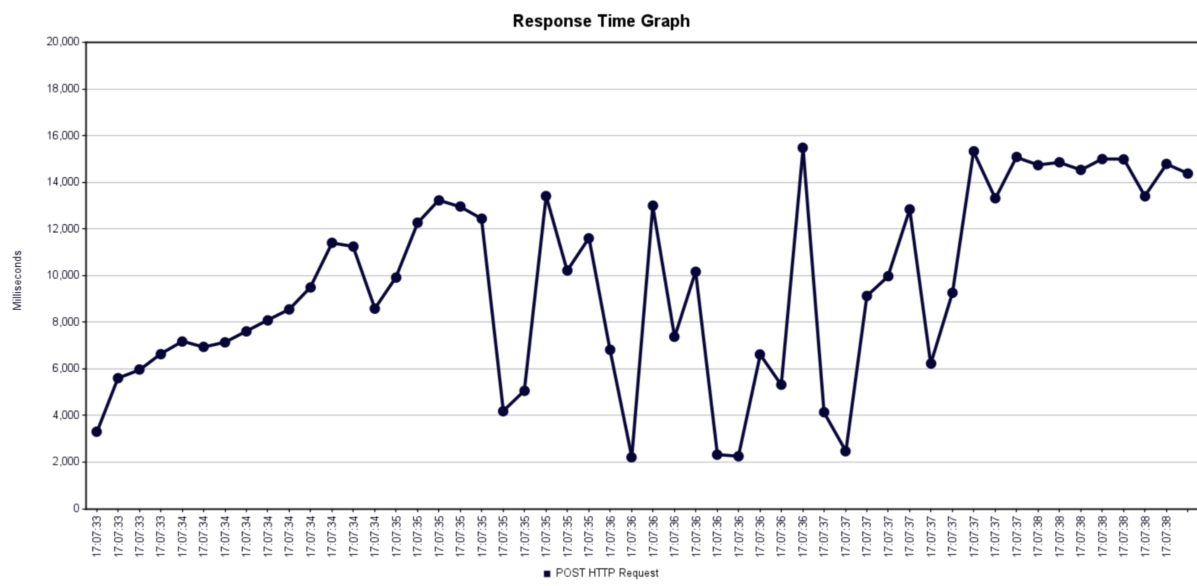


Figure 40 - 5000 Concurrent Users Response Time Graph | Write Operations | Upgraded

Takeaways:

- **35.5% error rate** – A high percentage of the requests, 35.5%, failed, indicating that the system struggled to handle the load of 5000 concurrent write requests. This high error rate highlights a clear issue with the system's ability to scale effectively under heavy write operations, leading to a substantial number of failed requests.

- **High Response Times** – The average response time was 9633 ms, with a minimum of 606 ms and a maximum of 23,078 ms (over 23 seconds). The standard deviation of 6504 ms shows significant variability in response times, with some requests processed relatively quickly while others experienced severe delays. These high and inconsistent response times indicate performance degradation under load.

- **Throughput** – The system achieved a throughput of 195.8 requests per second, which is fairly high given the error rate and response times. While the system processed a large number of requests, the high error rate diminishes the significance of this metric, as many requests were not completed successfully.

- **Erratic Response Time Pattern** – The response time graph shows frequent spikes, with response times fluctuating dramatically between 2000 ms and 16,000 ms. The erratic pattern suggests that the system is facing bottlenecks or resource limitations, causing inconsistent performance throughout the test. Despite some recovery periods, the response times remain unstable, which could lead to an unreliable user experience.

5.3.4. 10 000 Concurrent Users

Now with 10 000 concurrent users, only read operations will be tested as write operations for this level are not required.

The following parameters were used for testing the API:

Threads (Users): 10 000

Ramp up: 5 seconds

Loop: 1

These parameters produced the following results, shown in Figure 42 and 43.

Read Operations

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET HTTP Reque...	10000	26965	121	59458	20429.90	17.27%	104.1/sec	59.89	31.38	588.9
TOTAL	10000	26965	121	59458	20429.90	17.27%	104.1/sec	59.89	31.38	588.9

Figure 42 - 10 000 Concurrent Users Summary Report | Read Operations | Upgraded

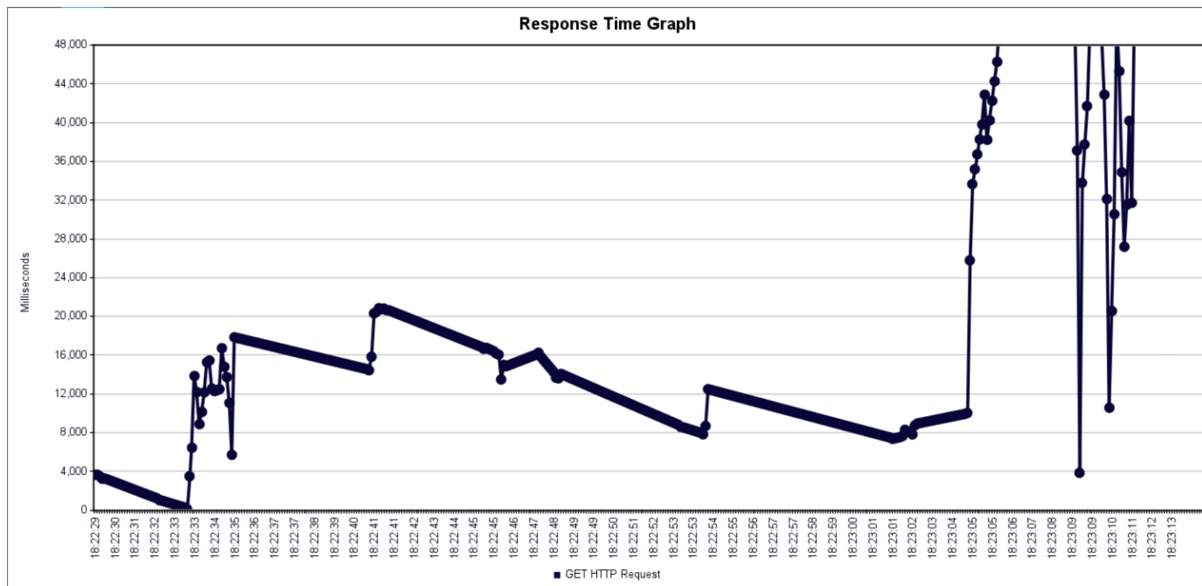


Figure 43 - 10 000 Concurrent Users Response Time Graph | Read Operations | Upgraded

Takeaways:

- **17.27% error rate –** The test resulted in a 17.27% error rate, showing that while the system handled the majority of the 10,000 concurrent read requests, nearly one in five requests failed. This highlights the system's struggle to manage high volumes of traffic efficiently, especially under significant load.
- **High and Erratic Response Times** – The average response time was **26,965 ms**, with a minimum of **121 ms** and a maximum of **59,458 ms** (nearly 60 seconds). The **standard deviation of 20,429 ms** reveals a wide variation in response times, reflecting inconsistencies in performance. Some requests were completed quickly, while others experienced severe delays.

- **Throughput** – The system processed **104.1 requests per second**, which is reasonable given the high response times and error rate. However, despite this throughput, the delays and failed requests would likely result in a poor user experience.

- **Response Time Pattern** – The response time graph displays an erratic pattern, with large spikes in response times reaching over **40,000 ms** before stabilizing slightly toward the end of the test. These fluctuations suggest performance bottlenecks or resource exhaustion as the system attempted to handle the load, leading to inconsistent performance across requests.

6. Review, Impact and Answers

In this section, a detailed analysis of the results from the implementation phase will be presented, focusing on the impact of migrating to a microservices architecture and the improvements made through the restructuring. This chapter will examine how the system's performance metrics evolved following the transition, including scalability, response times, error rates, and throughput under different levels of load. Additionally, this section will explore how the key goals, such as reducing bottlenecks, improving resilience, and handling large concurrent loads, were met. The insights gathered from these findings will provide a comprehensive evaluation of the overall impact of the new architecture and will address any remaining challenges, offering answers to the questions raised in the previous phases of the project.

6.1. Improvement Impact and Review

Now, the focus will shift to comparing the test results obtained before and after the migration to the microservices architecture. This section will provide a detailed comparison of key performance metrics such as response times, error rates, throughput, and scalability across different levels of load. By analyzing these metrics, we can gauge the effectiveness of the architectural changes and the integration of new components like Redis and RabbitMQ. The aim is to highlight the improvements brought about by the restructuring and identify any areas that still require further optimization. This comparison will provide a comprehensive view of how the migration has impacted the overall system performance and whether it has successfully met the initial objectives.

6.1.1. Read Operations

This section compares read operation performance before and after the migration. It includes a comparison of all the gathered data, and an overall summary of improvements to highlight the impact of the changes, as seen in Table 4.

Table 4 - Read Operations Comparison Table

Concurrent Users	Average Response Time Initial VS Improved		Error Percentage Initial VS Improved		Throughput Initial VS Improved		Min Response Time Initial VS Improved		Max Response Time Initial VS Improved	
50	4648 ms	187 ms	0%	0%	4.1/sec	9.8/sec	864 ms	13 ms	7399 ms	1097 ms
1000	122984 ms	8053 ms	0%	0%	4.0/sec	77.7/sec	1096 ms	3724 ms	246030 ms	11838 ms
5000	78262 ms	23694 ms	71.06%	0%	14.3/sec	147.7/sec	0 ms	35 ms	345811 ms	32475 ms
10 000	100227 ms	26965 ms	87.38%	17.27%	20.0/sec	104.1/sec	0 ms	121 ms	479285 ms	59458 ms
Improvement %	83%		70.59%		833.5%		−85.375%		89.5%	

- **Average Response Time:**
    - **50 Users:** Improved API has a significantly lower average response time (187 ms) compared to the Initial API (4648 ms), showing a drastic improvement in performance for lower concurrent users.
    - **1000 Users:** The average response time for the Improved API (8053 ms) is much lower than the Initial API (122984 ms), indicating enhanced scalability and efficiency under moderate load.
    - **5000 Users:** The Improved API shows a reduced average response time of 23694 ms compared to the Initial API's 78262 ms. Despite handling more requests, the improved system performs considerably better.
    - **10 000 Users:** Even under very high load, the Improved API maintains a better response time (26965 ms) than the Initial API (100227 ms), showing its ability to handle extreme loads more effectively.
- **Error Percentage:**
    - **50 Users:** Both APIs maintain a 0% error rate, indicating stability at a low number of concurrent users.
    - **1000 Users:** Both APIs continue to show a 0% error rate, reflecting reliability under increased load.

- **5000 Users:** The Improved API completely eliminates the error rate (0%) seen in the Initial API (71.06%), indicating significant enhancement in the system's fault tolerance.
- **10000 Users:** While the Initial API has a very high error rate (87.38%), the Improved API reduces this to 17.27%, demonstrating marked improvement in stability under high load.

- **Throughput:**
  - **50 Users:** The throughput for the Improved API is more than double that of the Initial API (9.8/sec vs. 4.1/sec), indicating better performance.
  - **1000 Users:** The Improved API exhibits a throughput of 77.7 requests per second, vastly outperforming the Initial API's 4.0/sec, reflecting better scalability.
  - **5000 Users:** The Improved API's throughput (147.7/sec) is significantly higher than the Initial API's (14.3/sec), showing the system can handle much more traffic efficiently.
  - **10000 Users:** Despite the high load, the Improved API achieves a throughput of 104.1 requests per second, compared to just 20.0/sec for the Initial API, demonstrating superior capacity handling.
- **Minimum Response Time:**
  - **50 Users:** The Improved API has a much lower minimum response time (13 ms) than the Initial API (864 ms), indicating faster initial processing.
  - **1000 Users:** The Initial API's minimum response time is 1096 ms, while the Improved API shows a minimum of 3724 ms. This slight increase may be due to initial load distribution.
  - **5000 Users:** The Initial API has a minimum response time of 0 ms, likely due to request failures, while the Improved API maintains a realistic minimum response time of 35 ms.
  - **10000 Users:** The Improved API has a minimum response time of 121 ms compared to 0 ms for the Initial API, again indicating better reliability rather than failures.
- **Maximum Response Time:**
  - **50 Users:** The Improved API significantly reduces the maximum response time to 1097 ms compared to the Initial API's 7399 ms.
  - **1000 Users:** Maximum response time for the Improved API is 11838 ms, a vast improvement over the Initial API's 246030 ms, showing better handling of peak loads.
  - **5000 Users:** Improved API's maximum response time (32475 ms) is lower than the Initial API's (345811 ms), demonstrating better peak performance.

- **10000 Users:** The Improved API has a maximum response time of 59458 ms, significantly less than the Initial API's 479285 ms, indicating enhanced performance under stress.

This comparison shows that the Improved API significantly outperforms the Initial API across all metrics, especially as the number of concurrent users increases. It highlights improved response times, reduced error rates, and increased throughput, indicating a more robust and scalable system overall.

However, it's important to note that the minimum response time metric appears worse in the Improved API. This is likely due to how the initial API handled requests with errors. In the initial API, the minimum response time recorded some instances as 0 ms, which is practically impossible in real-world scenarios and likely reflects failed requests rather than successful, processed responses. The improved API shows a more realistic minimum response time because it correctly handles requests and doesn't artificially inflate its performance by counting errors as instant responses. This honest representation indicates that the improved system processes all requests accurately, even if some take longer than others, thereby providing a more truthful measure of the system's performance.

6.1.2. Write Operations

Similarly to the previous section, this topic compares write operation performance before and after the migration. It includes a comparison of all the gathered data, and an overall summary of improvements to highlight the impact of the changes.

Table 5 - Write Operations Comparison Table

Concurrent Users	Average Response Time Initial VS Improved		Error Percentage Initial VS Improved		Throughput Initial VS Improved		Min Response Time Initial VS Improved		Max Response Time Initial VS Improved	
50	1698 ms	232 ms	0%	0%	6.5/sec	10.3/sec	446 ms	22 ms	2776 ms	1788 ms
1000	92426 ms	9477 ms	0%	0%	5.6/sec	58.3/sec	880 ms	2555 ms	172973 ms	16029 ms
5000	78951 ms	9633 ms	67.82%	35.5%	14.3/sec	195.8/sec	0 ms	606 ms	348384 ms	23078 ms
Improvement %	87.83%		47.7%		756.27%		−52.9%		73.23%	

- **Average Response Time:**
  - **50 Users**: The Improved API exhibits an **86% decrease** in average response time, dropping from 1698 ms to 232 ms.
  - **1000 Users**: There's a substantial **89.7% reduction** in response time, with the Improved API averaging 9477 ms compared to the Initial API's 92426 ms.
  - **5000 Users**: The Improved API shows an **87.8% improvement**, with the average response time reduced from 78951 ms to 9633 ms.

- **Error Percentage:**
  - **50 Users**: Both APIs maintain a **0% error rate**, indicating reliable performance at a low number of concurrent users.
  - **1000 Users**: The error rate remains at **0%** for both APIs, suggesting improved stability even under moderate load.
  - **5000 Users**: The Improved API drastically reduces the error rate by **47.7%**, going from 67.82% in the Initial API to 35.5%.

- **Throughput:**
  - **50 Users**: The Improved API achieves a **58.5% increase** in throughput, from 6.5/sec to 10.3/sec.
  - **1000 Users**: A significant improvement of **941.1%** is observed, with throughput rising from 5.6/sec to 58.3/sec.
  - **5000 Users**: The throughput increases by **1269.2%**, surging from 14.3/sec in the Initial API to 195.8/sec in the Improved API.

- **Minimum Response Time:**
  - **50 Users**: The minimum response time improves by **95.1%**, decreasing from 446 ms to 22 ms.
  - **1000 Users**: The Improved API has a higher minimum response time of 2555 ms compared to the Initial API's 880 ms, resulting in a **190.3% increase**. This may be due to the system initially taking time to distribute the load efficiently.
  - **5000 Users**: Unlike the unrealistic 0 ms minimum response time in the Initial API (likely due to request failures), the Improved API maintains a realistic minimum response time of 606 ms.

- **Maximum Response Time:**
    - **50 Users**: The maximum response time is reduced by **35.6%**, from 2776 ms in the Initial API to 1788 ms in the Improved API.
    - **1000 Users**: The maximum response time drops by **90.7%**, from 172973 ms to 16029 ms in the Improved API.
    - **5000 Users**: The Improved API sees a **93.4% reduction** in maximum response time, lowering it from 348384 ms to 23078 ms.

This comparison shows that the improved API significantly outperforms the initial API across all metrics in write operations, particularly as the number of concurrent users increases. The improved architecture exhibits much lower average response times, reduced error rates, and a remarkable increase in throughput, underscoring a more robust and scalable system overall.

However, like the read operations, the minimum response time metric in the improved API appears less favorable when compared to the initial API. The initial API reported minimum response times as low as 0 ms, especially under high loads. This unrealistic figure suggests that the system was likely failing to process some requests, recording them as 'instant' responses when, in fact, they were errors or dropped requests. The improved API, on the other hand, displays more realistic minimum response times. It accurately captures the time taken to process each request without erroneously marking failed requests as completed instantly. This transparency demonstrates that the improved system is genuinely processing all incoming requests, reflecting a more accurate and reliable measure of performance.

Overall, the transition to a microservices architecture for write operations has resulted in substantial improvements. Average response times were drastically reduced, with reductions of up to 89.7%, and throughput increased dramatically, indicating the system's enhanced capacity to handle large volumes of concurrent traffic. The error rates, particularly under high load, were also significantly reduced, showcasing the improved API's greater resilience and reliability. While the minimum response time appears worse in some cases, this actually indicates a more stable and honest handling of requests rather than an issue with system performance. The improved API provides a more truthful representation of its operational efficiency, resulting in a system that is not only faster and more scalable but also more dependable in handling write operations.

6.2. Research Questions and Answers

This section revisits the initial research questions posed at the beginning of the thesis, aiming to provide informed answers based on the extensive research, implementation, and testing carried out. By evaluating the system's performance before and after the migration to a microservices architecture, this analysis sheds light on key aspects such as scalability, load balancing, and caching. The insights gained here not only address the core challenges of building a high-performance social networking platform but also offer practical recommendations for optimizing similar systems in real-world scenarios.

6.2.1. Research and Test Based Answers

Q1: What are the key challenges and considerations when designing a scalable architecture for social networking platforms with high user interaction and content generation?

- **Null Hypothesis (H0):** The current architecture and design of social networking platforms can handle high user interaction and content generation without requiring specific scaling considerations or architectural changes.
- **Alternative Hypothesis (H1):** Designing a scalable architecture for social networking platforms with high user interaction and content generation requires significant changes, including specific scalability mechanisms such as load balancing, caching, microservices, and fault tolerance to manage system load efficiently.

Answer: The thesis research disproves the null hypothesis, highlighting the necessity of adopting a microservices-based architecture to handle high user interaction effectively. The original monolithic API suffered from scalability issues, with increasing response times and error rates as the user load grew. Transitioning to a microservices architecture, along with implementing load balancing, caching, and other scalability mechanisms, drastically improved performance metrics. Specifically:

- **Concurrent User Capacity:** The microservices-based system handled significantly more concurrent users, demonstrating the enhanced scalability needed for high interaction scenarios.
- **Throughput and Response Time:** The improved architecture showed a drastic reduction in average response times (up to 90% in some cases) and a marked increase in throughput, indicating the ability to efficiently handle the surge in user-generated content and interactions.
- **System Availability (Uptime):** The use of load balancing and microservices increased the system's resilience, ensuring high availability even under heavy loads.

Conclusion: To design a scalable architecture for social networking platforms, it is crucial to implement a microservices-based architecture, alongside load balancing and caching strategies, to effectively manage high user interaction and content generation.

Q2: What are the best practices for implementing microservices architectures to support scalability, and how do these practices improve modularity and fault isolation compared to monolithic architectures?

- **Null Hypothesis (H0):** Implementing microservices architectures does not provide significant improvements in scalability, modularity, or fault isolation compared to monolithic architectures.
- **Alternative Hypothesis (H1):** Implementing microservices architectures significantly improves scalability, modularity, and fault isolation compared to monolithic architectures through best practices such as service independence, API-based communication, and fault-tolerant design.

Answer: The research supports the alternative hypothesis, demonstrating that microservices architecture, when implemented with best practices, significantly improves scalability, modularity, and fault isolation:

- **Service Independence:** Each microservice in the new architecture operates independently, improving modularity. This isolation enables independent scaling of services based on demand, unlike the monolithic approach, which required scaling the entire application.
- **Inter-Service Latency:** The microservices design allows for optimized communication between services, reducing inter-service latency. By using API-based communication and asynchronous messaging (e.g., RabbitMQ), the architecture can efficiently distribute workloads.
- **Fault Isolation:** The microservices architecture isolates faults within individual services, preventing system-wide failures. This design proved beneficial during load testing, as faults in one service did not propagate to others, unlike in the monolithic architecture.

Conclusion: Implementing microservices using best practices such as service independence and API-based communication significantly improves scalability, modularity, and fault isolation, allowing systems to manage high loads more effectively.

Q3: How can load balancing techniques be optimized to ensure both high availability and fault tolerance in social networking platforms with unpredictable traffic patterns?

- **Null Hypothesis (H0):** Optimizing load balancing techniques does not significantly improve high availability and fault tolerance in social networking platforms with unpredictable traffic patterns.
- **Alternative Hypothesis (H1):** Optimizing load balancing techniques significantly improves high availability and fault tolerance in social networking platforms, especially in environments with unpredictable traffic patterns.

Answer: The research validates the alternative hypothesis, showing that optimized load balancing is essential for maintaining high availability and fault tolerance:

- **Traffic Distribution Efficiency:** Load balancing effectively distributed incoming requests across multiple service instances, ensuring no single instance was overwhelmed. This was evident in the improved API, which maintained a high throughput under heavy load.
- **System Downtime Due to Overload:** The improved API showed significantly reduced downtime compared to the initial monolithic API, thanks to load balancing strategies that managed traffic spikes more effectively.
- **Latency Variability and Server Utilization Rate:** By dynamically distributing requests, load balancing minimized latency variability and maximized server utilization. The system could adapt to unpredictable traffic patterns without a degradation in user experience.

Conclusion: Optimizing load balancing is crucial for ensuring high availability and fault tolerance in social networking platforms, particularly under unpredictable traffic patterns. This optimization allows the system to handle varying loads efficiently without sacrificing performance or reliability.

Q4: How does caching improve the scalability and responsiveness of user-driven interactions in high-traffic social networking platforms?

- **Null Hypothesis (H0):** Caching does not significantly improve the scalability and responsiveness of user-driven interactions in high-traffic social networking platforms.
- **Alternative Hypothesis (H1):** Caching significantly improves the scalability and responsiveness of user-driven interactions in high-traffic social networking platforms by reducing database load and lowering response times.

Answer: The thesis findings affirm the alternative hypothesis, indicating that caching plays a vital role in improving scalability and responsiveness:

- **Cache Hit Ratio:** Implementing caching mechanisms, such as Redis, led to a high cache hit ratio, reducing the need for repeated database queries. This directly decreased database load and improved system performance.
- **Latency Reduction:** Caching substantially reduced latency, resulting in faster response times for frequently accessed data. This contributed to the lower average response times observed in the improved API during testing.
- **Backend Load Reduction:** By serving a large proportion of read requests directly from the cache, the system reduced the load on the backend services, enabling them to scale more efficiently.

Conclusion: Caching significantly enhances the scalability and responsiveness of social networking platforms by reducing database load, minimizing latency, and improving the system's overall performance during high traffic.

7. Conclusion

In this concluding chapter, the thesis' journey is revisited, summarizing the research, design, implementation, and evaluation of a scalable architecture for social networking platforms.

The project began by identifying key challenges in handling high user interaction and content generation, leading to the exploration of architectural strategies to enhance system performance. Through the implementation of microservices, load balancing, and caching mechanisms, the thesis demonstrated the potential of these techniques to improve scalability, reliability, and responsiveness. This chapter reflects on the outcomes, discussing the effectiveness of the solutions adopted and their implications for future work in building robust, high-traffic social networking systems.

7.1. Results

The thesis set out to enhance the scalability, performance, and reliability of the Friendzone application by transforming its architecture and employing various optimization techniques. Initially, the current monolithic architecture was carefully reviewed, pinpointing key areas that required improvement for large-scale operations. This evaluation paved the way for a strategic migration to a microservices-based architecture, addressing the need for more flexible and scalable system components.

To establish a performance baseline, comprehensive tests were conducted on the existing system, providing critical benchmarks for later comparison. Based on these initial evaluations, tailored scalability strategies were developed, focusing on areas where microservices would provide the most substantial benefits. This strategic approach led to a detailed roadmap for transitioning from the monolith to a more modular and scalable architecture.

Subsequent performance optimization efforts aimed at identifying and mitigating bottlenecks resulted in significant enhancements in response times and system efficiency. These optimizations were thoroughly tested under various load scenarios to assess the improvements. The enhanced system demonstrated remarkable advancements in handling increased user loads and activities, with noticeable reductions in response times and error rates, especially under high concurrency.

Reliability was another key focus area, particularly in the context of a distributed microservices environment. Strategies implemented to ensure the application's robustness resulted in a more resilient platform, capable of maintaining availability even during peak loads. This reliability was further supported by documentation of the transition process and best practices, ensuring that the architectural improvements were well-understood and reproducible.

Finally, the impact of these changes was rigorously assessed. The re-evaluation of the system against the initial benchmarks revealed substantial gains in operational efficiency and user experience. The analysis of various scaling techniques, including the shift to microservices, demonstrated the effectiveness of the chosen methods in addressing the specific needs of the application. In summary, the thesis successfully met its objectives, providing a robust solution that significantly enhanced the application's scalability, performance, and reliability.

7.2. Limitations and Solution Validity Threats

The thesis faced several limitations that could potentially affect the generalization of the results. A primary limitation was that all performance and scalability tests were conducted on a local machine rather than a dedicated, more powerful server environment. This setup may have restricted the system's full potential, as the hardware limitations of the local machine likely imposed constraints on processing power, memory, and network throughput. While using a virtual server could have been a viable alternative, providing a more powerful and scalable environment for testing, it also comes with its own set of limitations. Virtual servers can incur significant costs, especially when extensive resources are required for accurate stress testing and performance evaluation. Furthermore, configuring a virtual server to match the desired production environment can be complex, potentially introducing additional variables that might affect test outcomes. Testing in a real-world scenario with a more powerful and distributed server infrastructure could have resulted in even more pronounced improvements in performance, scalability, and reliability, but these considerations highlight the balance between practical constraints and the ideal testing environment.

However, it's important to note that both the initial monolithic API and the improved microservices architecture were evaluated under identical conditions. Therefore, the performance enhancements observed—such as reduced response times, increased throughput, and lower error rates—are still valid and indicative of the benefits of the new architecture. The improvements reflect the system's architectural advancements rather than hardware differences.

Other potential threats to the validity of the solution include the synthetic nature of the test scenarios. While efforts were made to simulate realistic user loads, these tests might not fully capture the complexity and variability of traffic patterns in a live production environment. Despite these limitations, the results provide a strong indication of the system's improved scalability and performance, although further testing in a production-grade environment would be necessary to fully understand the real-world impact of the implemented changes.

7.3. Future Work

For future work, one of the most promising challenges is the integration of Kubernetes to orchestrate the multiple instances of the microservices. Kubernetes is a powerful container orchestration platform that would

significantly enhance the system's ability to scale dynamically. Unlike the current setup, which uses Docker Compose for container orchestration, Kubernetes offers advanced capabilities such as auto-scaling, self-healing, and automated deployment of microservices. By deploying the system on a Kubernetes cluster, it would be possible to automatically create and remove instances of microservices in response to varying traffic loads, ensuring optimal resource utilization and maintaining performance during peak times. This dynamic scaling would further improve the system's ability to handle unpredictable traffic patterns, making it even more resilient and efficient.

Kubernetes would also introduce robust fault tolerance mechanisms through features like rolling updates, health checks, and automatic failover. These capabilities would enhance the overall reliability of the system by allowing seamless updates and repairs without downtime, further aligning with the thesis's goal of building a scalable and reliable architecture for social networking platforms.

Beyond Kubernetes, future work could explore the implementation of advanced caching strategies, such as distributed caching across a cluster, to further improve response times and system performance. Additionally, incorporating a more sophisticated monitoring and logging solution, like Prometheus and Grafana, could provide deeper insights into system performance and facilitate proactive maintenance and optimization. Security enhancements, such as implementing robust authentication and authorization mechanisms for microservices communication, would also be a crucial next step to ensure data integrity and privacy as the platform scales.

Another avenue for future exploration is the integration of Artificial Intelligence-driven tools to automate performance testing and generate comprehensive reports. AI can be leveraged to simulate complex user behaviors and traffic patterns more accurately, adjusting parameters in real-time to mimic real-world scenarios. Tools enhanced with AI can identify performance bottlenecks and optimization opportunities faster and more effectively than manual methods, providing valuable insights for improving system scalability and reliability. AI-powered testing could also automatically analyze results and generate detailed reports, highlighting key areas for enhancement and ensuring continuous improvement of the platform.

Finally, migrating testing to a cloud-based environment would allow for performance evaluations under more realistic conditions. Testing on cloud platforms such as AWS, Azure, or Google Cloud would enable the simulation of large-scale user interactions in a distributed setting, offering a more accurate assessment of the system's capabilities and limitations. By addressing these areas in future work, the platform can achieve an even higher level of scalability, reliability, and performance, better preparing it to meet the demands of real-world, high-traffic social networking applications.

7.4. Final Statement

This thesis set out to tackle the complex challenge of building a scalable, reliable, and efficient architecture for social networking platforms with high user interaction and content generation. By systematically transitioning from a monolithic architecture to a microservices-based approach, implementing load balancing,

caching, and optimizing the system for performance, this work demonstrated the tangible benefits of these strategies in real-world scenarios. The findings revealed that a thoughtfully designed microservices architecture can drastically improve response times, throughput, and fault tolerance, even under heavy and unpredictable user loads.

Through extensive testing and analysis, the thesis provided a clear pathway for developers and researchers aiming to scale similar platforms. It showed that scalability is not just about handling more users but also about ensuring a seamless, reliable experience regardless of demand. While the limitations of testing on a local machine were acknowledged, the consistent environment for both the initial and improved APIs made the improvements observed undeniably valid.

The exploration of future work, particularly the potential integration of Kubernetes, opens new avenues for further research and application, pushing the boundaries of what's possible in dynamic scalability and orchestration. This thesis has laid a solid foundation for how social networking platforms can be designed and optimized, providing valuable insights into the practical implementation of microservices, load balancing, and caching in a high-demand context. By contributing to the body of knowledge on scalable system design, this work serves as a guide and inspiration for building more resilient and scalable digital infrastructures in the ever-evolving world of social networking.

References

A. Shibl, M., M. A. Helal, I., & A. Mazen, S. (2021). *System Integration for Large-Scale Software Projects: Models, Approaches, and Challenges* (Mostafa Al-Emran, Mohammed A. Al-Sharafi, Mohammed N. Al-Kabi, & Khaled Shaalan, Eds.). Springer, Cham.

Ahmad, A. A. S., & Andras, P. (2018). Measuring and Testing the Scalability of Cloud-based Software Services. *5th International Symposium on Innovation in Information and Communication Technology, ISIICT 2018*. https://doi.org/10.1109/ISIICT.2018.8613297

Al-Said Ahmad, A., & Andras, P. (2019). Scalability analysis comparisons of cloud-based software services. *Journal of Cloud Computing*, *8*(1). https://doi.org/10.1186/s13677-019-0134-y

Bjerke-Gulstuen, K., Daniela, A., & Cruzes, S. (2020). *System Integration Testing in Large Scale Agile: dealing with challenges and pitfalls*.

Cham, T. H., Cheah, J. H., Memon, M. A., Fam, K. S., & László, J. (2022). Digitalization and its impact on contemporary marketing strategies and practices. In *Journal of Marketing Analytics* (Vol. 10, Issue 2, pp. 103–105). Palgrave Macmillan. https://doi.org/10.1057/s41270-022-00167-6

Chen, T. H., Syer, M. D., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2017). Analytics-driven load testing: An industrial experience report on load testing of large-scale systems. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017*. https://doi.org/10.1109/ICSE-SEIP.2017.26

Daka, E., & Fraser, G. (2014). A survey on unit testing practices and problems. *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*. https://doi.org/10.1109/ISSRE.2014.11

Diário da República. (2020). *Diário da República, 2.ª série PARTE E Artigo 2.º*.

Eklund, U., Olsson, H. H., & Strøm, N. J. (2014). Industrial challenges of scaling agile in mass-produced embedded systems. *Lecture Notes in Business Information Processing*, *199*. https://doi.org/10.1007/978-3-319-14358-3_4

Hristova, D., & Lieberoth, A. (2021). *How Socially Sustainable Is Social Media Gamification? A Look into Snapchat, Facebook, Twitter and Instagram* (Agnessa Spanellis, Ed.).

Jafari, S. J., Naji, H. R., & Jannatifar, M. (2018). Investigating Performance Metrics for Evaluation of Content Delivery Networks. *Lecture Notes in Business Information Processing*, *234*. https://doi.org/10.1007/978-3-319-76587-7_9

Jamil, M. A., Arif, M., Abubakar, N. S. A., & Ahmad, A. (2017). Software testing techniques: A literature review. *Proceedings - 6th International Conference on Information and Communication Technology for the Muslim World, ICT4M 2016*. https://doi.org/10.1109/ICT4M.2016.40

Larrucea, X., Santamaria, I., Colomo-Palacios, R., & Ebert, C. (2018). Microservices. *IEEE Software*, *35*(3), 96–100. https://doi.org/10.1109/MS.2018.2141030

Lenka, R. K., Rani Dey, M., Bhanse, P., & Barik, R. K. (2018). Performance and Load Testing: Tools and Challenges. *2018 International Conference on Recent Innovations in Electrical, Electronics and Communication Engineering, ICRIEECE 2018*. https://doi.org/10.1109/ICRIEECE44171.2018.9009338

Long, Z. (2017). Improvement and Implementation of a High Performance CQRS Architecture. *Proceedings - 2017 International Conference on Robots and Intelligent System, ICRIS 2017*. https://doi.org/10.1109/ICRIS.2017.49

Mladenova, T. (2020). Software Quality Metrics - Research, Analysis and Recommendation. *2020 International Conference Automatics and Informatics, ICAI 2020 - Proceedings*. https://doi.org/10.1109/ICAI50593.2020.9311361

Onwuegbuzie, A. J., Leech, N. L., & Collins, K. M. T. (2012). Qualitative Analysis Techniques for the Review of the Literature. In *The Qualitative Report* (Vol. 17). http://www.nova.edu/ssss/QR/QR17/onwuegbuzie.pdf

Patrício, C., Pinto, R., & Marques, G. (2021). A Study on Software Testing Standard Using ISO/IEC/IEEE 29119-2: 2013. In *Studies in Systems, Decision and Control* (Vol. 295). https://doi.org/10.1007/978-3-030-47411-9_3

Schäfer, M., Nadi, S., Eghbali, A., & Tip, F. (2023). *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation*. http://arxiv.org/abs/2302.06527

Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. In *IEEE Access* (Vol. 5). https://doi.org/10.1109/ACCESS.2017.2685629

Sharvari T, & Sowmya Nag K. (2019). *A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming*.

Wee, B. Van, & Banister, D. (2016). How to Write a Literature Review Paper? *Transport Reviews*, *36*(2). https://doi.org/10.1080/01441647.2015.1065456

Yang, H., Pan, H., & Ma, L. (2023). A Review on Software Defined Content Delivery Network: A Novel Combination of CDN and SDN. *IEEE Access*, *11*. https://doi.org/10.1109/ACCESS.2023.3267737

Attachments