



Uanderson Ricardo Ferreira da Silva (urfs@cin.ufpe.br)

**Micro Frontend Architecture for Multi-Robot User Interfaces: A  
Systematic Approach with Design Rationale**



Federal University of Pernambuco  
secgrad@cin.ufpe.br  
[www.cin.ufpe.br/~graduacao](http://www.cin.ufpe.br/~graduacao)

Recife  
2024

Uanderson Ricardo Ferreira da Silva (urfs@cin.ufpe.br)

**Micro Frontend Architecture for Multi-Robot User Interfaces: A  
Systematic Approach with Design Rationale**

A B.Sc. Dissertation presented to the Center of Informatics  
of Federal University of Pernambuco in partial fulfillment  
of the requirements for the degree of Bachelor in Computer  
Engineering.

***Concentration Area:*** *Software Engineering*

***Advisor:*** *Augusto Cezar Alves Sampaio (acas@cin.ufpe.br)*

Recife  
2024

Ficha de identificação da obra elaborada pelo autor,  
através do programa de geração automática do SIB/UFPE

Silva, Uanderson Ricardo Ferreira da.

Micro frontend architecture for multi-robot user interfaces: a systematic approach with design rationale / Uanderson Ricardo Ferreira da Silva. - Recife, 2024.

83 p. : il., tab.

Orientador(a): Augusto Cezar Alves Sampaio

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2024.

Inclui referências.

1. Micro frontends. 2. Desenvolvimento web. 3. Arquitetura de software. 4. Interfaces gráficas do usuário. 5. Sistemas multi-robôs. I. Sampaio, Augusto Cezar Alves. (Orientação). II. Título.

000 CDD (22.ed.)

Uanderson Ricardo Ferreira da Silva (urfs@cin.ufpe.br)

# **Micro Frontend Architecture for Multi-Robot User Interfaces: A Systematic Approach with Design Rationale**

A B.Sc. Dissertation presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering. Defended and approved on October 23, 2024, by the following examination board:

---

**Augusto Cezar Alves Sampaio (acas@cin.ufpe.br)**  
Advisor/CIn-UFPE

---

**Kiev Santos da Gama**  
Examiner/CIn-UFPE

# ACKNOWLEDGEMENTS

I want to thank my family for their constant love and support throughout my educational journey. To my mom, Sinara, and my dad, Ubiratan, thank you for everything you have done for me. To my siblings, Ubiratan Junior and Maria Clara, I cherish the bond we share and know we will always be there for each other.

I want to thank my friends Danilo Vaz, Matheus Andrade, Alexandre Burle, Rodrigo Duarte, and Humberto Lopes for the countless laughs the laughs and great memories we created during the graduation. Being with you made everything better.

I want to thank my colleagues in RobôCIn's Software Infra Team: José Victor, Matheus Andrade, Andresa Almeida, Karine Gomes, and Alexandre Burle. Thank you for the many hours spent researching, writing, and developing the SSL Core system. I couldn't have asked for better teammates to work with on this project.

I want to thank Marina Paixão for your love, support, and inspiration. I appreciate your presence in my life and all the ways you have helped me grow. Your positivity and dedication are inspiring, and I am grateful for everything you have done for me. Thank you for inspiring me to be my best.

I want to thank CIn for supporting my studies and providing the infrastructure necessary for my education, as well as for its commitment to excellence in computing. I am also grateful to all the professors who guided me throughout my journey, especially Prof. Augusto Sampaio. Your extensive knowledge and experience have profoundly influenced my academic development and personal growth. I also thank Madiel Conserva and Filipe Arruda for their support during the development of the project.

To all of you, my sincere thanks.

# ABSTRACT

Traditional frontend systems were initially conceived as thin presentation layers within larger monolithic applications. However, as user interaction requirements became more sophisticated, modern frontends started to integrate complex logic and domain-specific business rules. This shift is particularly evident in highly interactive and dynamic applications, such as robotic systems, where frontends must manage more than just user input and output. The resulting large codebases have become increasingly challenging to maintain, driving the need for more robust architectural solutions. This work presents a novel software architectural approach for developing frontends (including graphical user interfaces) in multi-robot systems using micro frontends. The proposed solution was designed through a systematic approach that combines Object-Oriented Modeling and Domain-Driven Design to address key challenges in this domain, leading to a discussion of major decisions such as splitting, composition, communication, routing, performance, and consistency. The architecture was evaluated based on the ISO/IEC 25010 quality model, achieving significant improvements over monolithic systems in performance tests, with higher frame rates and lower latency, as well as enhanced maintainability, reliability, and portability.

**Keywords:** Micro frontends. Web development. Software architecture. Graphical user interfaces. Multi-robot systems.

# RESUMO

Os sistemas de *frontend* tradicionais foram inicialmente concebidos como camadas de apresentação enxutas dentro de aplicações monolíticas maiores. No entanto, à medida que os requisitos de interação do usuário se tornaram mais sofisticados, os *frontends* modernos começaram a integrar lógica complexa e regras de negócio específicas do domínio. Essa mudança é particularmente evidente em aplicações altamente interativas e dinâmicas, como sistemas robóticos, nos quais os *frontends* precisam gerenciar mais do que apenas entrada e saída de dados. O aumento resultante no tamanho das bases de código tornou a manutenção cada vez mais desafiadora, exigindo soluções arquiteturais mais robustas. Este trabalho apresenta uma abordagem arquitetural de software moderna para o desenvolvimento de *frontends* (incluindo interfaces gráficas do usuário) em sistemas multi-robôs usando *micro frontends*. A solução proposta foi projetada através de uma abordagem sistemática que combina *Object-Oriented Modeling* e *Domain-Driven Design* para enfrentar os principais desafios nesse domínio, levando a uma discussão sobre decisões importantes, como divisão, composição, comunicação, roteamento, desempenho e consistência. A arquitetura foi avaliada de acordo com o modelo de qualidade ISO/IEC 25010, apresentando melhorias significativas em relação a sistemas monolíticos em testes de desempenho, com melhores taxas de quadros e menor latência, além de maior manutenibilidade, confiabilidade e portabilidade.

**Palavras-chave:** Micro frontends. Desenvolvimento web. Arquitetura de software. Interfaces gráficas do usuário. Sistemas multi-robôs.

# LIST OF FIGURES

Figure 1	– RoboCup Small Size League robot control . . . . .	17
Figure 2	– Example of an UML class diagram . . . . .	18
Figure 3	– Example of an DDD informal diagram . . . . .	20
Figure 4	– Example of a monolithic architecture . . . . .	21
Figure 5	– Example of a Service-Oriented Architecture . . . . .	22
Figure 6	– Example of a microservices architecture . . . . .	23
Figure 7	– Static-Page Website approach . . . . .	24
Figure 8	– Multi-page applications approach . . . . .	25
Figure 9	– Single-Page Applications approach . . . . .	26
Figure 10	– Micro frontends approach . . . . .	27
Figure 11	– Horizontal split . . . . .	28
Figure 12	– Vertical split . . . . .	29
Figure 13	– Client-side composition . . . . .	30
Figure 14	– Edge-side composition . . . . .	31
Figure 15	– Server-side composition . . . . .	31
Figure 16	– Build-time composition . . . . .	32
Figure 17	– Service registry pattern . . . . .	36
Figure 18	– API gateway pattern . . . . .	37
Figure 19	– Backend for Frontend pattern . . . . .	37
Figure 20	– Proposed software lifecycle model . . . . .	39
Figure 21	– System’s use case diagram . . . . .	41
Figure 22	– System’s exploratory domain model . . . . .	44
Figure 23	– System’s bounded contexts . . . . .	46
Figure 24	– System’s refined domain models inside each bounded context . . . . .	47
Figure 25	– System’s architecture . . . . .	50
Figure 26	– Service interaction model for UC001 . . . . .	51
Figure 27	– System’s component diagram . . . . .	52
Figure 28	– Discarded server-side micro frontends approach . . . . .	54
Figure 29	– Defined design tokens . . . . .	55
Figure 30	– Application shell with micro frontends . . . . .	57
Figure 31	– Player micro frontend . . . . .	58
Figure 32	– Viewer micro frontend . . . . .	59
Figure 33	– Scoreboard micro frontend . . . . .	60
Figure 34	– Parameters micro frontend . . . . .	61



Figure 35 – Comparison between monolithic and micro frontend approach . . . . . 67

## LIST OF TABLES

Table 1 – System’s functional requirements . . . . .	40
Table 2 – System’s quality attributes and non-functional requirements . . . . .	42
Table 3 – End-to-end tests . . . . .	65
Table 4 – Resource utilization for each component . . . . .	66
Table 5 – Micro frontend Performance Results . . . . .	68
Table 6 – Lighthouse performance metrics . . . . .	69
Table 7 – Comparison of Maintainability Index between the components . . . . .	70
Table 8 – Embold’s maintainability metrics . . . . .	71
Table 9 – SonarQube’s maintainability metrics . . . . .	71
Table 10 – Embold’s reliability metrics . . . . .	72
Table 11 – SonarQube’s reliability metrics . . . . .	72
Table 12 – Embold’s portability metrics . . . . .	73
Table 13 – Characteristics and subcharacteristics of micro frontend and monolithic architectures . . . . .	73

# CONTENTS

<b>1</b>	<b>INTRODUCTION .....</b>	<b>13</b>
<b>2</b>	<b>BACKGROUND.....</b>	<b>15</b>
2.1	ROBOTIC SYSTEMS.....	15
2.1.1	Multi-Robot Systems.....	15
2.1.2	RoboCup Small Size League.....	16
2.2	MODERN SOFTWARE ENGINEERING TECHNIQUES .....	18
2.2.1	Object-Oriented Modeling .....	18
2.2.2	Domain-Driven Design .....	19
2.3	SOFTWARE ARCHITECTURE PATTERNS .....	20
2.3.1	Monolithic Architecture.....	20
2.3.2	Service-Oriented Architecture.....	21
2.3.3	Microservices Architecture .....	22
2.4	WEB DEVELOPMENT .....	23
2.4.1	Static-Page Websites.....	24
2.4.2	Multi-Page Applications .....	24
2.4.3	Single-Page Applications.....	25
2.4.4	Micro frontends.....	26
2.5	MICRO FRONTEND STRATEGIES .....	27
2.5.1	Splitting.....	28
2.5.1.1	Horizontal Split.....	28
2.5.1.2	Vertical Split.....	28
2.5.2	Composition.....	29
2.5.2.1	Client-Side Composition .....	29
2.5.2.2	Edge-Side Composition .....	30
2.5.2.3	Server-Side Composition.....	31
2.5.2.4	Build-Time Composition .....	32
2.5.3	Routing.....	33
2.5.3.1	Client-Side Routing.....	33
2.5.3.2	Server-Side Routing .....	33
2.5.3.3	Edge-Side Routing.....	34
2.5.4	Communication .....	34
2.5.4.1	Event-Based Communication .....	34
2.5.4.2	State-Based Communication.....	35
2.5.4.3	Integration-Based Communication.....	35

<b>2.5.5</b>	<b>Backend Integration.....</b>	<b>36</b>
2.5.5.1	<i>Service Registry .....</i>	36
2.5.5.2	<i>API Gateway .....</i>	36
2.5.5.3	<i>Backend for Frontend .....</i>	37
2.6	SOFTWARE QUALITY .....	38
<b>2.6.1</b>	<b>ISO/IEC 25010 .....</b>	<b>38</b>
<b>3</b>	<b>SOLUTION DESIGN.....</b>	<b>39</b>
3.1	REQUIREMENT ANALYSIS .....	39
<b>3.1.1</b>	<b>Functional Requirements.....</b>	<b>40</b>
<b>3.1.2</b>	<b>Non-Functional Requirements.....</b>	<b>41</b>
<b>3.1.3</b>	<b>Design Constraints .....</b>	<b>42</b>
<b>3.1.4</b>	<b>Implementation Constraints.....</b>	<b>43</b>
3.2	SYSTEM ANALYSIS .....	43
<b>3.2.1</b>	<b>Domain Analysis.....</b>	<b>43</b>
<b>3.2.2</b>	<b>Bounded Contexts.....</b>	<b>45</b>
3.3	SYSTEM DESIGN .....	47
<b>3.3.1</b>	<b>Micro Frontends.....</b>	<b>47</b>
<b>3.3.2</b>	<b>Architectural Design Patterns.....</b>	<b>48</b>
3.3.2.1	<i>Domain Events.....</i>	48
3.3.2.2	<i>Backends for Frontends .....</i>	49
3.3.2.3	<i>Application Shell.....</i>	50
<b>3.3.3</b>	<b>Service Interaction Model .....</b>	<b>50</b>
<b>3.3.4</b>	<b>Service Component Model .....</b>	<b>51</b>
<b>4</b>	<b>SOLUTION IMPLEMENTATION .....</b>	<b>53</b>
4.1	OVERVIEW .....	53
4.2	USER INTERFACE CONSISTENCY .....	55
4.3	TECHNOLOGIES.....	56
4.4	COMPONENTS .....	56
<b>4.4.1</b>	<b>Application Shell.....</b>	<b>56</b>
<b>4.4.2</b>	<b>Player Micro Frontend .....</b>	<b>58</b>
<b>4.4.3</b>	<b>Viewer Micro Frontend .....</b>	<b>59</b>
<b>4.4.4</b>	<b>Scoreboard Micro Frontend .....</b>	<b>60</b>
<b>4.4.5</b>	<b>Parameters Micro Frontend .....</b>	<b>60</b>
4.5	TESTING STRATEGIES.....	61
4.6	PERFORMANCE OPTIMIZATION .....	62
4.7	CONTINUOUS INTEGRATION AND DELIVERY .....	62
4.8	SYSTEM MONITORING.....	63

<b>5</b>	<b>EVALUATION .....</b>	<b>64</b>
5.1	ASSESSMENT METHODOLOGY .....	64
5.2	FUNCTIONAL EVALUATION .....	65
5.3	NON-FUNCTIONAL EVALUATION .....	66
<b>5.3.1</b>	<b>Performance Efficiency.....</b>	<b>66</b>
5.3.1.1	<i>Resource Utilization.....</i>	66
5.3.1.2	<i>Rendering Performance Comparison.....</i>	66
5.3.1.3	<i>Rendering Performance in Competition Environment.....</i>	68
5.3.1.4	<i>Lighthouse Audit .....</i>	68
<b>5.3.2</b>	<b>Maintainability .....</b>	<b>69</b>
5.3.2.1	<i>Maintainability Index .....</i>	69
5.3.2.2	<i>Embold Analysis.....</i>	70
5.3.2.3	<i>SonarQube Analysis.....</i>	71
<b>5.3.3</b>	<b>Reliability .....</b>	<b>71</b>
5.3.3.1	<i>Embold Analysis.....</i>	72
5.3.3.2	<i>SonarQube Analysis.....</i>	72
<b>5.3.4</b>	<b>Portability.....</b>	<b>72</b>
5.3.4.1	<i>Embold Analysis.....</i>	73
5.3.4.2	<i>Subcharacteristics Comparison .....</i>	73
<b>6</b>	<b>RELATED WORK .....</b>	<b>75</b>
6.1	MICROSERVICES IDENTIFICATION AND BOUNDARIES .....	75
6.2	EVOLUTION OF ROBOTIC INTERFACES TO BROWSER GUIS .....	76
6.3	MICRO FRONTENDS IN ROBOTIC SYSTEMS .....	77
<b>7</b>	<b>CONCLUSION .....</b>	<b>78</b>
	<b>REFERENCES.....</b>	<b>80</b>

# 1

## INTRODUCTION

In recent years, the complexity of modern applications has grown at an extraordinary pace, especially in domains that demand sophisticated frontends (including graphical user interfaces, GUIs) and integration with complex backend systems. Multi-Robot Systems (MRS) represent one such domain where GUIs are expected to do far more than present static data; they must support dynamic tasks such as 3D real-time visualization, physics simulations, and interactive replays of robotic actions. As systems like MRS evolve, traditional monolithic approaches to GUI development are increasingly showing their limitations [13], resulting in large codebases that are difficult to maintain and slower development cycles.

Historically, frontend applications served as simple layers for data presentation, with minimal business logic, and much of the system complexity was handled by the backend [24]. However, the rise of Single-Page Applications (SPAs), which provide rich user experiences and real-time interactions, has placed a significant burden on frontend architectures. No longer merely consumers of data, modern frontend systems now manage significant portions of domain logic themselves. This change has had profound implications, particularly for systems as demanding as MRS, which require flexibility, responsiveness, and modularity.

Micro frontends (MFEs) offer an alternative solution to these challenges by extending the principles of microservices – widely used in backend development – to the frontend [22], allowing a large application to be divided into smaller, independently developed, and deployable modules. This approach promotes modularity by design and enables teams to work autonomously, adopting different technologies and workflows for different parts of the frontend while ensuring that the overall application remains cohesive.

This work proposes a systematic approach for developing a micro frontend architecture focused on MRS GUIs, grounded in Domain-Driven Design (DDD) and Object-Oriented Modeling (OOM) principles. At the core of this approach is the concept of bounded contexts [9], which isolate different parts of the system to reduce dependencies and simplify development. This concept is particularly applicable to MRS, where different GUIs are required for distinct tasks such as telemetry, robot control, simulation, and data replay. By decoupling these areas, each GUI component can evolve independently.

While this framework is primarily designed for robotics, its versatility extends to any

complex system with evolving requirements and sophisticated user interaction needs. Nevertheless, the focus of this research is on MRS, with an emphasis on GUIs used in robot soccer competitions, where interactive simulations and real-time data processing are critical.

The evaluation of the framework uses the ISO/IEC 25010 product quality model [11], focusing on key characteristics such as performance, maintainability, reliability, and portability. These quality attributes provide a structured way to assess the effectiveness of the proposed architecture in addressing the unique challenges of MRS GUIs. The evaluation also includes an examination of communication between micro frontends, routing, UI consistency, performance optimization, and integration with continuous integration/continuous deployment (CI/CD) pipelines.

The main contribution of this research is the introduction of a micro frontend architecture designed to meet the particular demands of MRS GUIs, combining the strengths of micro frontends, DDD, and OOM to create a scalable, maintainable, and modular solution. This approach contrasts to traditional views of frontend development, positioning the frontend as a core component in the architectural design rather than a simple presentation layer. By treating the frontend as an independent entity capable of managing its own domain logic, this work proposes a more robust and adaptable solution for complex GUI applications.

This document is structured as follows: Chapter 2 provides a detailed background on the trends in web development, focusing on the rise of micro frontends and their application in complex systems. It also explores Domain-Driven Design, Object-Oriented Modeling, and software quality models, setting the stage for the proposed solution. Chapter 3 presents the solution design, discussing the use case modeling, bounded contexts, and architectural decisions that shape the proposed framework. Chapter 4 describes the implementation of the multi-robot system GUI using the proposed micro frontend architecture, while Chapter 5 evaluates the solution through experiments, results, and a comparison of architectural approaches. Chapter 6 reviews related work, offering an analysis of existing solutions and frameworks that address similar challenges in frontend architecture and robotics systems. Finally, Chapter 7 concludes the work with insights into future research directions and potential applications of the framework beyond the domain of robotics.

# 2

## BACKGROUND

This chapter provides an overview of the key concepts and technologies relevant to this work. It begins by introducing robotic systems, focusing on multi-robot systems and their application in the RoboCup Small Size League. The chapter then explores important software engineering practices and architectural patterns that support scalable and maintainable systems. Modern web development techniques, particularly micro frontends, are discussed as essential components for building modular and efficient applications. Lastly, the chapter addresses software quality, with an emphasis on the ISO/IEC 25010 standard.

### 2.1 ROBOTIC SYSTEMS

A robot can be defined as a reprogrammable, multifunctional manipulator capable of performing a variety of tasks by moving materials, parts, tools, or specialized devices through programmable motions [6]. The concept of robotics, however, extends far beyond this basic definition, encapsulating a field of study that has evolved significantly over the decades.

With the ongoing advancements in technology, robotic systems have become integral to everyday life. Beyond their extensive use in manufacturing and industrial operations, robots are now being implemented across critical sectors, including healthcare, education, and transportation [1].

Robotic systems can be categorized into two main types: single-robot systems (SRS) and multi-robot systems [8]. Early research efforts primarily concentrated on SRS, which are typically less complex, as they do not require coordination or task-sharing among multiple units. However, with the rise of the Internet of Things (IoT) and evolution of robotics, the increasing complexity of applications has driven the need for MRS. This work focuses explicitly on the study and application of multi-robot systems.

#### 2.1.1 Multi-Robot Systems

A multi-robot system consists of a group of robots organized into a multi-agent architecture to collaboratively perform a shared task [41]. Over the past decade, MRS has attracted



growing interest within the robotics community due to its exceptional capabilities, such as cooperative behavior, robustness, parallel operation, and scalability. The ability of multiple robots to work together in a coordinated manner offers substantial benefits over traditional single-robot systems, particularly in complex environments where tasks may exceed the capabilities of a single robot.

The literature highlights several advantages of MRS compared to SRS. One commonly discussed benefit is cost efficiency. In many scenarios, the overall system cost can be reduced by employing multiple simpler and less expensive robots, rather than relying on a single, more complex and costly machine [14]. Additionally, in environments with high complexity or dynamic conditions, the capabilities required may be too extensive for a single robot to handle effectively. MRS also inherently increases system efficiency and robustness by enabling parallel task execution and incorporating redundancy. The use of multiple robots allows for a more flexible and adaptable system, capable of continuing operations even if one robot encounters a failure, thus improving the overall reliability of the system.

An essential aspect of managing MRS is the design of a user interface that supports effective operator control and decision-making. Given the increased cognitive load of overseeing multiple robots simultaneously, a well-crafted UI can reduce operator fatigue and enhance performance. Previous research [28] has demonstrated the impact of UI design on operator workload and task performance in MRS. While controlling multiple platforms can increase productivity by providing additional resources, it can also lead to increased cognitive load, potentially affecting accuracy and response time. Thus, optimizing UI design requires a careful balance between the advantages of multi-platform control and the risks associated with increased operator workload.

Robot soccer competitions, such as the RoboCup Small Size League (SSL), exemplify the practical application of multi-robot systems. In these environments, teams of robots collaborate in real time to achieve common objectives, demonstrating the capacity of MRS to perform tasks that require high levels of coordination, communication, and adaptability. This competition provides a concrete example of the power and potential of multi-robot systems, which will be discussed in further detail later in this work.

### **2.1.2 RoboCup Small Size League**

The Robot World Cup Initiative (RoboCup) was created in 1997 to promote advancements in the field of collaborative mobile robots for addressing dynamic problem-solving scenarios [18]. The competition involves teams of autonomous robots playing soccer against one another, with the long-term objective of defeating the FIFA World Cup champions by the year 2050 [4].

RoboCup features various leagues, each focusing on developing key technologies in the field. These leagues focus on areas such as real-time sensor fusion, reactive behaviors, strategy acquisition, machine learning, real-time planning, multi-agent cooperation, coordination

strategies, context recognition, vision processing, strategic decision-making, motor control, and intelligent robot control systems [18].

One of the most traditional and competitive leagues within RoboCup is the Small Size League. In this league, robots are constrained to fit within a cylindrical shape, with a maximum height of 15 cm and a diameter of 18 cm. These size constraints allow for precise and dynamic movements within a fast-paced game environment, where rapid decision-making is essential. Division A matches, consisting of advanced teams, involve two teams of eleven robots each, playing on a field measuring 12 meters by 9 meters, while Division B matches, typically for new or less competitive teams, use a smaller field measuring 9 meters by 6 meters and involve six robots per team. In both cases, an orange golf ball is used as the game ball to facilitate tracking and control. Figure 1 illustrates the dynamics involved in controlling robots during a typical RoboCup SSL match.

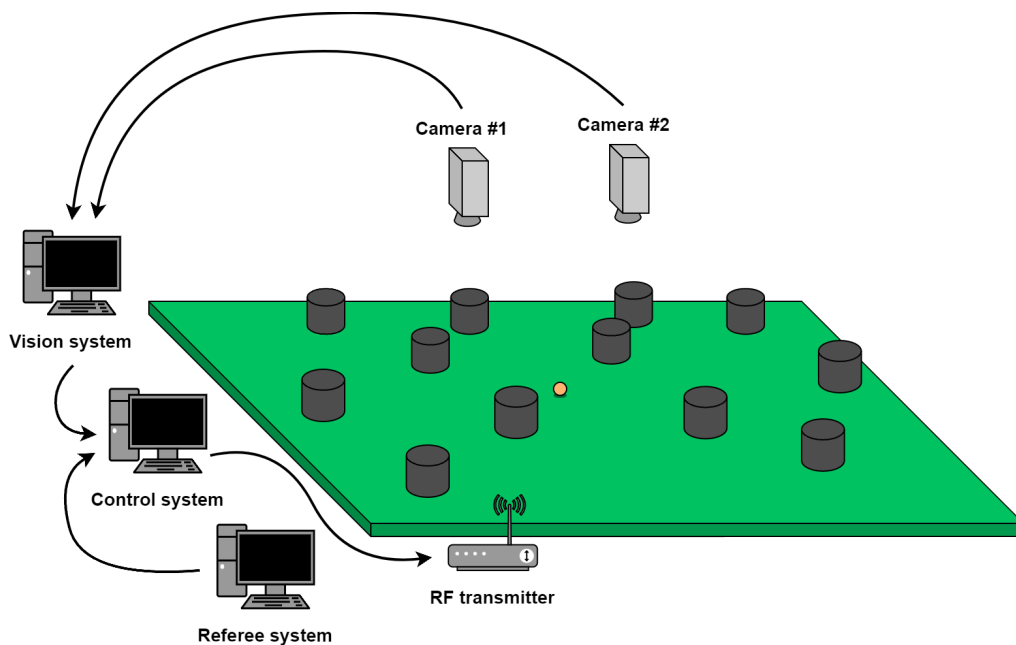


Figure 1: RoboCup Small Size League robot control

Over the past decade, RoboCup evolved into a well-recognized platform for testing and benchmarking strategies in multi-robot coordination. Although the performance of RoboCup teams has significantly improved over the years, several aspects of the competition are intentionally simplified to facilitate multi-robot coordination. For example, in the SSL, global overhead cameras provide a centralized view of the field, simplifying visual processing. Additionally, a central computer coordinates all robots on the field, issuing commands to each robot individually. The relatively small size of the playing field also mitigates communication challenges that would otherwise be more pronounced in larger and more complex environments.

## 2.2 MODERN SOFTWARE ENGINEERING TECHNIQUES

As software systems permeate nearly every aspect of modern life, the necessity for robust, scalable, and adaptable development processes has become more pronounced. Key approaches in software engineering, such as Object-Oriented Modeling and Domain-Driven Design, provide frameworks that enable the creation of systems that are not only functional but also maintainable, reusable, and aligned with business objectives.

### 2.2.1 Object-Oriented Modeling

Real-world systems are often highly complex, making it difficult to fully comprehend them. To understand such systems, models are created that abstract non-essential details, focusing on key aspects relevant for comprehension.

Object-Oriented Modeling provides a structured way of thinking about these systems, organizing them around real-world concepts. It represents software systems as collections of discrete objects that encapsulate both data structures and behaviors [3].

The core concepts of OOM include classes, inheritance, polymorphism, and encapsulation, which form the foundation of modern object-oriented design. To support these concepts, the Unified Modeling Language (UML) was developed as a standardized notation for visualizing, specifying, constructing, and documenting software systems [43]. Figure 2 illustrates an example of a UML class diagram. This models a class **Person** with an association to another class **Address**, along with two subclasses (**Student** and **Professor**) that inherit from **Person**. Additionally, it includes a supervision association between **Professor** and **Student**.

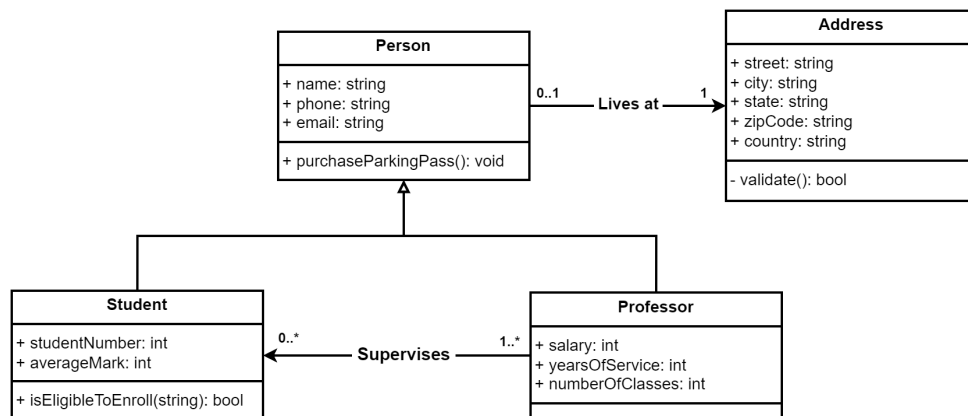


Figure 2: Example of an UML class diagram

OOM has become widely adopted due to its ability to break down complex problems into more manageable parts, promoting better understanding of requirements, cleaner designs, and more maintainable systems [3]. Since objects encapsulate both data and behavior, changes can be made locally without disrupting the entire system, enhancing scalability and flexibility.

OOM is foundational to modern software development and has influenced many design patterns and methodologies.

In the following chapter, the proposed software architecture will be systematically designed using various UML diagrams. Specifically, use case diagrams will capture functional requirements from the user's perspective, class diagrams will model entities and relationships, sequence diagrams will represent interactions among services, and component diagrams will provide a high-level view of the system's modular structure. Together, these diagrams provide a multi-faceted view of the architecture, ensuring clarity and alignment across design, implementation, and release phases.

### 2.2.2 Domain-Driven Design

Domain-Driven Design is a software development approach that prioritizes the core domain of a problem and its inherent complexities [9], fostering a close alignment between software architecture and business requirements. Unlike Object-Oriented Modeling, which primarily focuses on structuring software around objects and their interactions, DDD prioritizes a deep understanding of the domain itself. This methodology promotes collaboration between developers and domain experts [40], enabling a shared comprehension of the problem space and ensuring that software solutions are grounded in domain knowledge rather than technical concerns.

At the heart of DDD lies the domain model, which captures key concepts and relationships within the domain. This model forms the basis of the Ubiquitous Language – a shared vocabulary consistently used across the project by both technical and non-technical stakeholders [9]. The use of a Ubiquitous Language is essential for bridging the gap between business and technical teams, ensuring alignment throughout the development lifecycle.

Communication in DDD is not confined to formal tools like UML diagrams. The model and Ubiquitous Language should permeate all forms of communication, whether in written documents, casual conversations, or informal diagrams [9]. Figure 3 provides an illustrative example of such a diagram.

During the implementation phase, DDD employs specific design patterns such as entities, value objects, aggregates, repositories, and services. These patterns help encapsulate domain logic within appropriate structures, ensuring that the software remains aligned with the domain model while supporting scalability and flexibility.

DDD is particularly useful in complex systems with rich and interconnected domains. By identifying bounded contexts, DDD encourages the separation of different subdomains, each with its own model. This modularity reduces complexity and allows each bounded context to evolve independently, making it easier to adapt the system to changing business needs while maintaining a clear separation of concerns.

This work will extensively apply key concepts from DDD, including domain models,

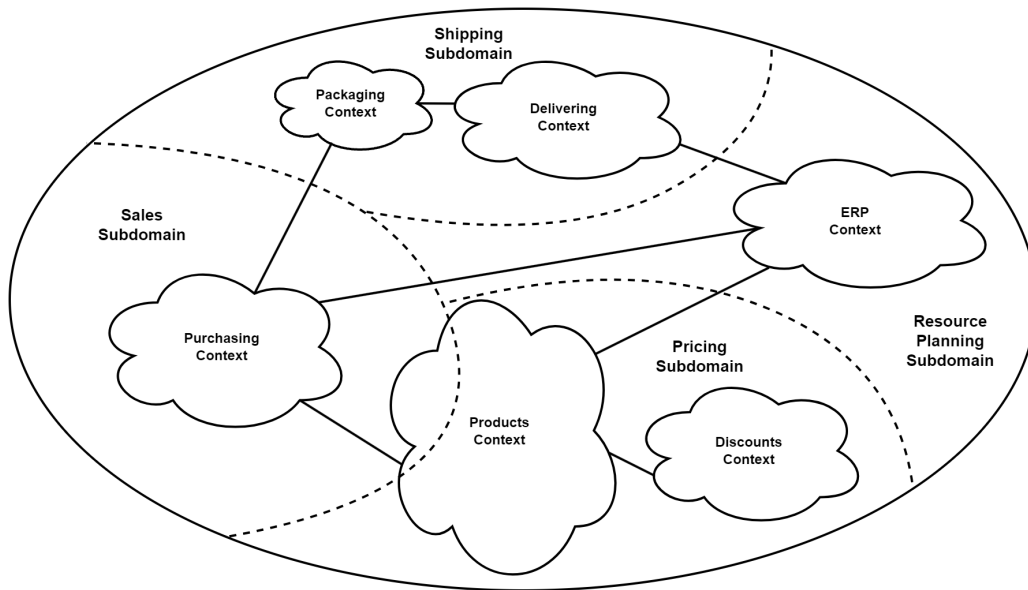


Figure 3: Example of an DDD informal diagram

bounded contexts, and Ubiquitous Language, throughout the analysis and design phases, along with tactical design patterns such as domain events. Rather than replacing OOM and UML, DDD complements these approaches by infusing domain expert knowledge into the models, bridging the gap between software structure and business logic.

## 2.3 SOFTWARE ARCHITECTURE PATTERNS

Software architecture patterns define the high-level structure of software systems, defining the software components, the externally visible properties of those components, and the relationships among them [32]. Choosing the appropriate architectural style is critical for developing robust, maintainable, and scalable applications. Among the most widely adopted architectural styles are Monolithic Architecture, Service-Oriented Architecture, and Microservices Architecture. Each of these patterns has its distinct advantages and trade-offs, making their suitability dependent on the application's size, complexity, and long-term objectives.

### 2.3.1 Monolithic Architecture

Monolithic architecture is a traditional and widely adopted architectural style in the software industry [2]. In this approach, softwares are developed as self-contained units where the different functionalities of the application –such as the user interface, business logic, and database access – are tightly interconnected and interdependent, all running as a single executable or deployable entity. The simplicity of this structure facilitates easier development and deployment during the early stages of a project, as all components reside within a unified codebase that can be managed cohesively. Figure 4 illustrates a monolithic architecture.

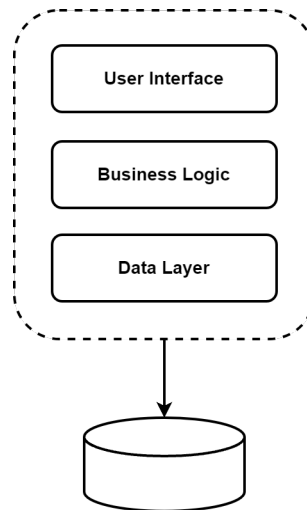


Figure 4: Example of a monolithic architecture

However, as applications grow, monolithic architectures often face significant challenges related to scalability, maintainability, and flexibility. The tight coupling of components means that even minor changes in one aspect can trigger widespread effects, resulting in longer and more error-prone development cycles. Scalability becomes problematic as the entire application must be adjusted, even if only a single component requires additional resources. Furthermore, increased team sizes complicate coordination, with multiple developers attempting to modify the same code segments and teams struggling to align deployment schedules. This environment can lead to confusion over ownership and decision-making responsibilities within the project [25].

Despite these challenges, monolithic architecture remains a viable option for smaller applications or early-stage products, where the overhead of managing distributed systems is not yet justified. Its single deployment unit and straightforward communication between components make monolithic systems suitable for less complex applications with predictable scaling needs. Additionally, the simplified deployment topology facilitates smoother developer workflows and enhances monitoring, troubleshooting, and end-to-end testing. Code reuse is also more straightforward, as developers can easily access existing code without the complications of distributed systems – such as deciding between code duplication, library creation, or service integration [25].

Furthermore, the maintenance of monolithic architectures can be improved by using patterns like Model-View-Controller (MVC) and by further structuring the model with layers that separate concerns, such as business logic and persistence. Nevertheless, the overall system remains a (sequential) monolith, and scalability continues to be a concern.

### 2.3.2 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is an architectural pattern that breaks down applications into multiple services that collaborate to provide a set of capabilities [24]. Each service is designed to perform a specific business function [2] and operates as a completely separate

operating system process. Communication between these services occurs over a network through an Enterprise Service Bus (ESB), typically using standardized protocols such as SOAP or REST, rather than through method calls within the process.

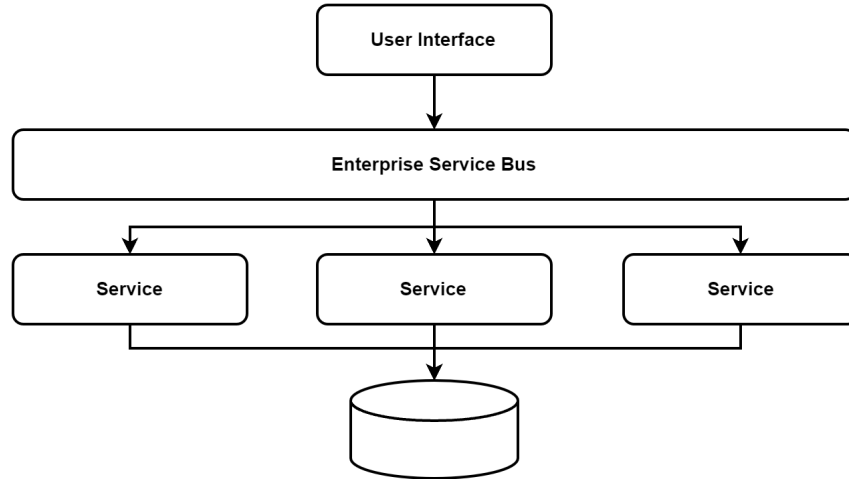


Figure 5: Example of a Service-Oriented Architecture

SOA emerged as a solution to the challenges of large monolithic applications. It promotes software reusability, allowing multiple applications to share common services. SOA also aims to simplify maintenance and rewriting by enabling the replacement of individual services without affecting others, as long as the service interface remains unchanged [24].

However, SOA also has its own set of limitations. The reliance on standardized communication protocols (e.g., SOAP, WSL-\*) and middleware can introduce performance overhead and complicate infrastructure management. Furthermore, the centralization of persistence (database, as seen in Figure 5) can create bottlenecks, impacting overall system efficiency. Additionally, there is often a lack of guidance regarding service granularity, which can lead to suboptimal decisions about system decomposition. SOA is generally better suited for large enterprise applications with diverse business requirements, where the benefits of service reusability and flexibility justify the added complexity.

### 2.3.3 Microservices Architecture

Microservices architecture has evolved as a modern approach of SOA, incorporating lessons learned from real-world implementations [24]. This pattern proposes even more granular services, each dedicated to a specific functionality. Unlike SOA, which often relies on centralized communication and persistence systems, microservices typically use lightweight communication mechanisms (e.g., REST, gRPC), allowing services to communicate directly with one another while each service implements its associated persistent data locally. An example of a microservices architecture is shown in Figure 6.

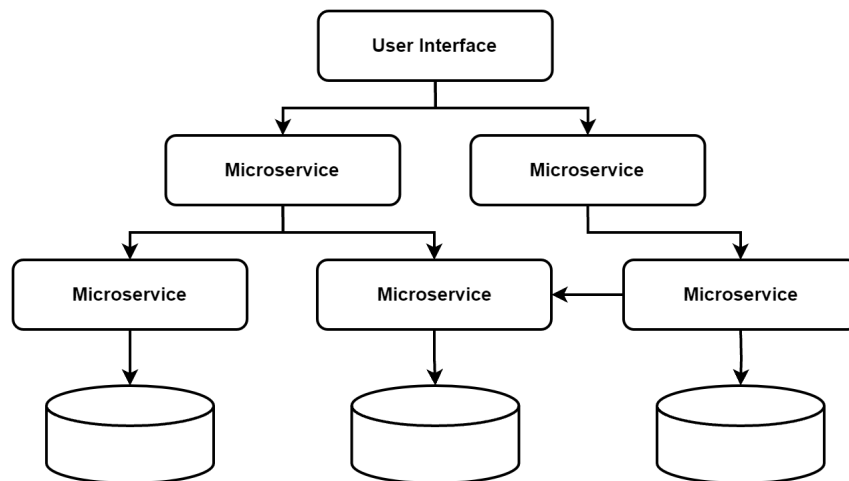


Figure 6: Example of a microservices architecture

The benefits of microservices are numerous. Their independent deployment model improves scalability and system robustness, enabling teams to scale individual services as needed without impacting the entire application. Microservices also allow for greater flexibility in technology choices, as teams can use different programming languages, frameworks, and databases across services. The parallel development of these services offers greater team productivity by reducing bottlenecks, as developers can focus on isolated components without interfering with each other's work. This autonomy also simplifies understanding and maintaining specific parts of the system, as each microservice represents a self-contained unit of functionality.

On the other hand, the distributed nature of microservices introduces new challenges. Managing communication between services, ensuring data consistency across distributed systems, and handling potential failures become more complex. Furthermore, microservices require a more sophisticated infrastructure for deployment, monitoring, and logging, often involving containerization tools and orchestration platforms.

## 2.4 WEB DEVELOPMENT

Originating from the development of the internet, web development utilizes standardized programming languages like HTML, CSS, and JavaScript to create applications that run within web browsers. The ubiquity and cross-platform nature of the web have made it a preferred platform for deploying modern software, ensuring accessibility across a wide range of devices.

Over the past few decades, web development has advanced from basic static pages to sophisticated, dynamic, and scalable applications. This evolution has given rise to a variety of architectural models and technologies, each suited to specific needs and challenges. Four principal approaches define the landscape: Static-Page Websites, Multi-Page Applications (MPAs), Single-Page Applications, and the emerging micro frontends architecture. Each model, detailed in the remaining of this section, presents distinct advantages and limitations, shaping the methods used to build flexible and maintainable web systems at scale.



### 2.4.1 Static-Page Websites

Early websites of the 1990s were published directly as static files of standard web technologies like HTML, CSS, and JavaScript. Each page existed as an individual file on a server, and every time a user clicked a link, a new static page was loaded [22]. This static-page model remains in use today, although it is often considered outdated for more complex applications.

Static-Page Websites represent the simplest form of web development, where each page is a separate HTML file, and content is fixed unless manually updated by developers. These sites are typically used for small-scale projects such as blogs, portfolio websites, or informational sites, where content remains relatively stable and user interaction is minimal. Figure 7 details this approach.

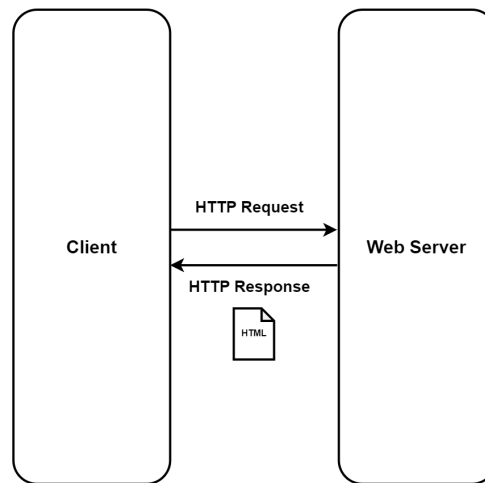


Figure 7: Static-Page Website approach

The primary advantage of static websites lies in their simplicity and performance. Since pages are pre-rendered and do not require server-side processing, they can be delivered quickly to users, resulting in fast load times. Additionally, static sites are inherently more secure, as there is no backend system vulnerable to attacks. However, this simplicity has its limitations. Static websites are incapable of delivering dynamic content or providing personalized user experiences, making them unsuitable for applications that require frequent updates or user interactions.

In recent years, the static-page approach has seen a resurgence through techniques such as Static Site Generation (SSG) or pre-rendering, which modernize the concept by integrating it with more contemporary web development workflows.

### 2.4.2 Multi-Page Applications

Multi-Page Application is a traditional architecture for building large-scale websites. In this model, the application state resides on the server, and each user interaction – such as navigating to a new page – triggers a full page reload through a process called Server-Side Rendering (SSR). When the browser sends a request, the server processes it, generates the

appropriate HTML, and returns it to the client. MPAs are commonly used in enterprise systems, e-commerce platforms, and content-heavy websites requiring multiple distinct pages to organize content and functionality. This approach is illustrated in Figure 8.

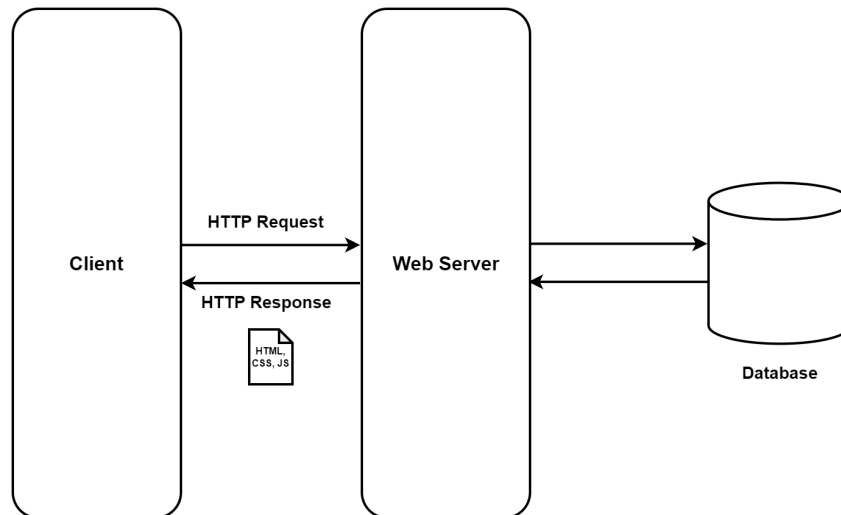


Figure 8: Multi-page applications approach

One of the advantages of MPAs is their straightforward development model. Each page can be developed and managed independently, with unique URLs for each section of the application. This approach also integrates well with search engines, as the structure of individual pages is naturally indexed by crawlers, improving Search Engine Optimization (SEO) performance.

MPAs also have some drawbacks, particularly in terms of performance. Each navigation requires a full page reload, leading to slower response times and increased server load due to frequent requests. Complex pages that involve extensive data and numerous JavaScript elements can also strain both the browser and the client's computer. To address these limitations, AJAX was introduced in the early 2000s, allowing parts of a page to be refreshed without reloading the entire content. While this improved user experience by reducing loading times, it also increased the complexity of the source code, particularly as web applications became more feature-rich and dynamic [15].

### 2.4.3 Single-Page Applications

Single-Page Application is a modern web architecture where the entire frontend is typically bundled into a single JavaScript file that is downloaded by the client [22]. Unlike MPAs, SPAs load a single HTML page initially and then dynamically update the content (see Figure 9) without requiring full page reloads as users navigate or interact with the application. This approach allows for a more fluid, app-like user experience, closely resembling that of native desktop or mobile applications.

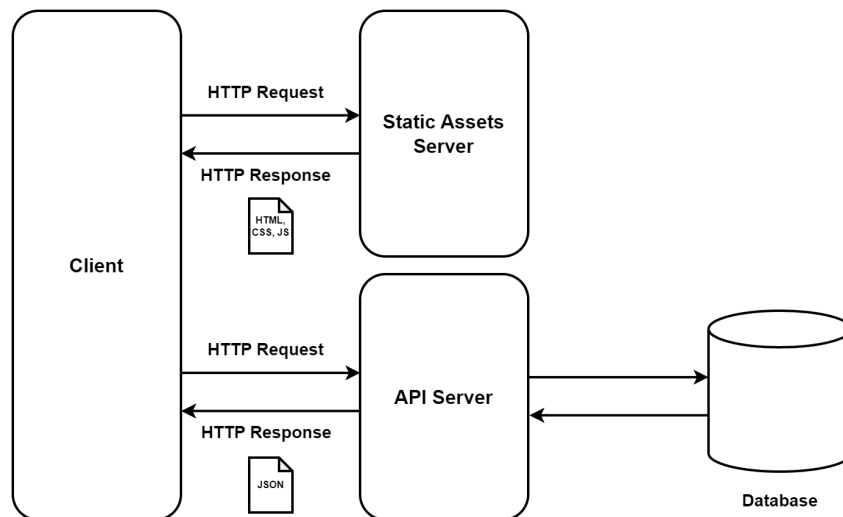


Figure 9: Single-Page Applications approach

SPAs heavily rely on client-side JavaScript to manage aspects such as routing, state management, and rendering. For the routing mechanism, the application rewrites the URL as the user navigates between views, allowing the users to share the page link or bookmark, even though the content is being dynamically generated [22]. This functionality allows SPAs to maintain state on the client side, enabling the application to respond more quickly to user interactions by updating only the relevant parts of the page. Since only the data needed for specific interactions is transferred between the server and client, SPAs can be more bandwidth-efficient than MPAs, particularly in long-running or complex web applications [39].

However, SPAs have some disadvantages for certain types of applications. The first load time can be significantly longer than in other web architectures since the entire application is loaded upfront, rather than loading only the content required for the current view. This can be particularly problematic for users with unstable or low-bandwidth connections, such as those using mobile devices [39]. SPAs also face challenges related to SEO, as the dynamic nature of the content can prevent search engines from effectively crawling and indexing the pages.

To address these challenges, modern SPAs have evolved to adopt hybrid approaches that combine the benefits of SPAs with performance optimizations typically seen in other architectures. Techniques such as partial hydration, island architecture, and resumability have emerged to reduce the initial load time and improve performance on mobile devices [39].

#### 2.4.4 Micro frontends

Micro frontends are a novel architectural approach inspired by microservices [22]. By extending microservices principles to the frontend, micro frontends allow web applications to be divided into independently developed and deployed components. Each micro frontend acts as a self-contained segment of the user interface, with dedicated teams responsible for developing and releasing their components independently of the rest of the application. This

modularity can provide numerous benefits for the development process, especially in large-scale applications where multiple teams are working on different features or services. Figure 10 details this architecture.

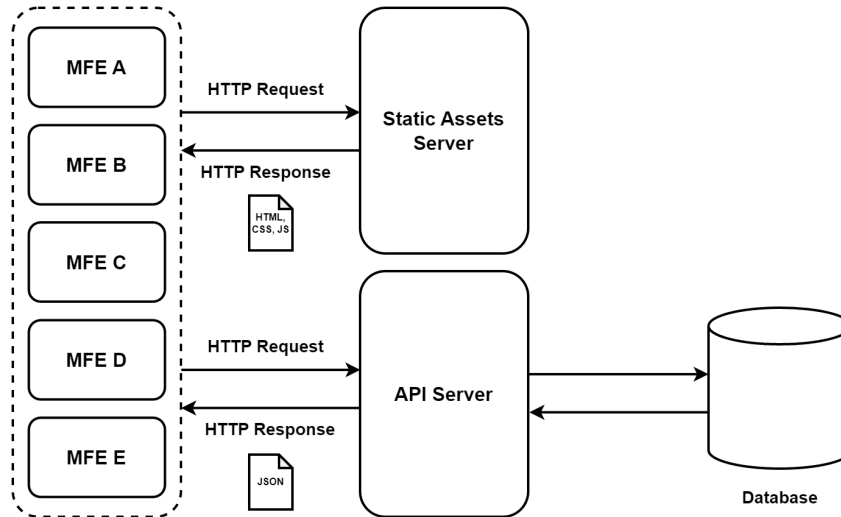


Figure 10: Micro frontends approach

In addition to the gains in maintainability and performance through smaller codebases and optimizations for use cases, the micro frontend architecture encourages more gradual updates and deployments, reducing the risk of introducing failures that could impact the entire application and improving system reliability. Furthermore, the architecture’s technology-agnostic and future-proof nature ensures flexibility, enabling teams to adopt new technologies and methodologies without disrupting the integrity of the overall system.

While working with micro frontends simplifies business logic, they introduce inherent complexities related to networking, persistence, communication protocols, security, and more [22]. The distributed nature of micro frontends brings additional challenges, including managing shared dependencies, ensuring a consistent user experience across components, and handling communication across micro frontends.

## 2.5 MICRO FRONTEND STRATEGIES

Architecting a successful micro frontend application requires careful consideration of several strategies to ensure an efficient operation [22]. The choice of each strategy depends heavily on the project’s specific context, and certain architectural decisions must be made early to provide direction for all subsequent development. These decisions range from how micro frontends will be composed and splitted, to how routing is managed, how inter-component communication is handled, and how the frontend integrates with backend services. This section explores the main strategies that support the development of a robust micro frontend architecture.

## 2.5.1 Splitting

A key consideration when developing micro frontends is how to split and organize them within the application. This can be done by either incorporating multiple micro frontends into the same page or by assigning one micro frontend to each page.

### 2.5.1.1 Horizontal Split

The horizontal split strategy involves placing multiple micro frontends within a single page [22]. In this approach, different teams are responsible for individual sections of the same interface, leading to a collaborative effort to build and maintain the page. This strategy offers significant flexibility, as it allows individual micro frontends to be reused across different views. For instance, a navigation bar or a search component developed as a micro frontend can appear in various parts of the application, improving modularity and reducing redundancy. Figure 11 illustrates this strategy.

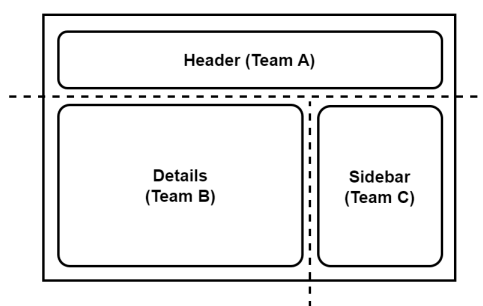


Figure 11: Horizontal split

However, the horizontal split also introduces several challenges. Since multiple teams are working on different parts of the same view, coordination becomes critical. This requires greater discipline and governance to avoid an excessive number of micro frontends within a single project, which could lead to increased complexity in both development and maintenance.

Additionally, this strategy demands more integrated testing efforts, as multiple components must work together seamlessly in the same view. Dependency management can also become more complicated, as different micro frontends may rely on shared libraries or services, leading to potential conflicts or versioning issues.

### 2.5.1.2 Vertical Split

In a vertical split, each micro frontend occupies one or more entire pages and is responsible for a specific business domain or feature [22], such as authentication, product catalog, or user profile. This approach has a stronger alignment with DDD, where each micro frontend fully encapsulates the frontend logic for its domain, providing a clearer separation of concerns and

greater autonomy for development teams. With this strategy, each team can operate independently, focusing on its specific domain without needing to coordinate with other teams working on the same page. This strategy is illustrated in Figure 12.

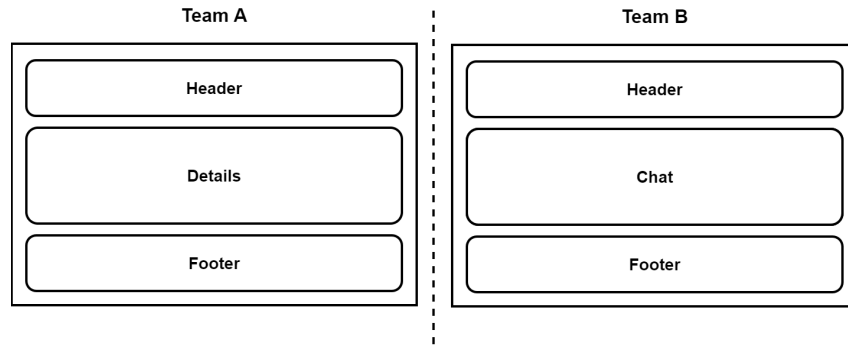


Figure 12: Vertical split

Vertical splitting typically results in a simpler structure, as each micro frontend owns an entire page of the application. However, this approach can reduce flexibility and reusability, as micro frontends are confined to specific views and are not easily shared or capable of real-time communication with other micro frontends. Additionally, maintaining visual consistency across the application requires extra effort, particularly if teams are using different tools or design patterns.

## 2.5.2 Composition

Composing a micro frontend application can be accomplished through various approaches, each with its own advantages and trade-offs. The main strategies for composition include client-side composition, edge-side composition, server-side composition, and build-time composition.

### 2.5.2.1 Client-Side Composition

Client-side composition involves dynamically fetching micro frontends and mounting them into the page on the client side, enabling greater flexibility and responsiveness. This method is often implemented using JavaScript frameworks that facilitate the injection and management of micro frontends directly within the framework environment. Figure 13 illustrates this strategy.

In this approach, a client-side orchestrator, known as the Application Shell, dynamically assembles the page within the browser, loading the necessary micro frontends as required. This method is similar to code-splitting techniques used in SPAs, allowing for parallel downloads of bundles and runtime updates that enhance the user experience. In this context, each micro frontend should expose a JavaScript or HTML file as an entry point, enabling the Application Shell to dynamically append DOM nodes from the HTML file or initialize the JavaScript application [22]. Client-side composition can be achieved through several techniques, including:

- **Iframes:** The Iframe element can encapsulate and isolate micro frontends, embedding another document within the current HTML document while ensuring that they function independently within the main application.
- **Web Components:** This technique utilizes a set of native browser APIs to create reusable custom HTML elements that encapsulate micro frontend logic and styles.
- **JavaScript and DOM Manipulation:** By using JavaScript to manipulate the Document Object Model (DOM) directly, micro frontends can be exported in a custom format that the Application Shell parses to append or modify elements dynamically.

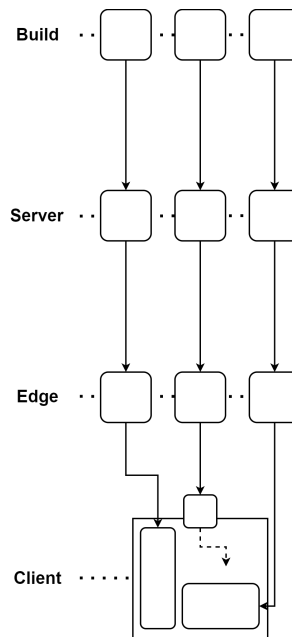


Figure 13: Client-side composition

### 2.5.2.2 Edge-Side Composition

Edge-side composition is a more recent approach where the composition process occurs at the CDN, a distributed network of servers that deliver web content and resources to users based on their geographic location, improving load times and reducing latency. By integrating the strengths of both client-side and server-side, this method optimizes overall performance while preserving some level of flexibility. Figure 14 shows this strategy.

With edge-side composition, the page is assembled at the Content Delivery Networks (CDN) level using techniques such as Edge Side Includes (ESI) [22]. ESI allows for scalable web infrastructure by leveraging the extensive network of CDN points of presence globally. While this approach is advantageous for static content, such as blogs or e-commerce pages, it may not perform well in multi-CDN scenarios due to inconsistent implementations across providers. Adopting a multi-CDN strategy can result in substantial refactoring efforts and the need for new logic when transitioning from one provider to another.

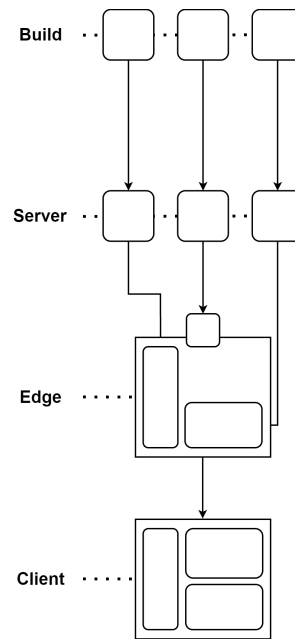


Figure 14: Edge-side composition

### 2.5.2.3 *Server-Side Composition*

Server-side composition composes micro frontends at the server level before sending the assembled HTML to the client. This method is efficient, as it reduces the client's load by offloading rendering tasks to the server. However, it limits runtime flexibility since the composition occurs before the page is delivered. Figure 15 illustrates this strategy.

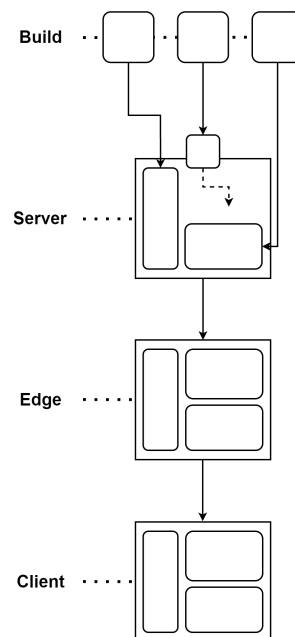


Figure 15: Server-side composition

In this approach, a backend service builds the page using a rendering engine that retrieves



the required micro frontends and composes the final output. If the page content is highly cacheable, leveraging CDNs can significantly enhance performance by serving pre-assembled pages [22]. However, when dealing with dynamic, personalized content, scalability becomes a concern, especially under high traffic conditions.

#### 2.5.2.4 Build-Time Composition

Build-time composition is a micro frontend approach in which individual applications are published as packages and included as dependencies of the container application during the build process. While this method can offer performance gains and simplify dependency management, it introduces tighter coupling between applications, as any minor change requires the recompilation of all micro frontends. This strategy is illustrated in Figure 16.

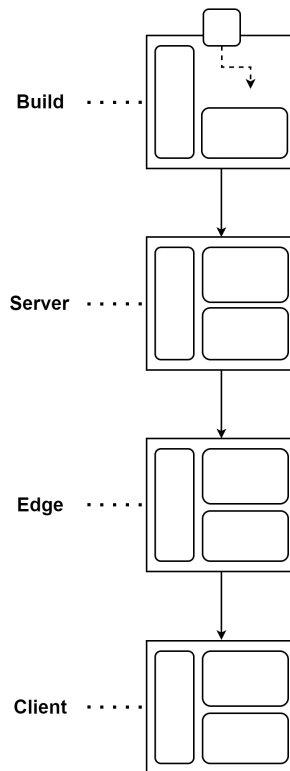


Figure 16: Build-time composition

Some experts argue that build-time composition should not be classified as a true micro frontend approach due to its lack of deployment independence [13]. In this model, the flexibility associated with micro frontends is diminished, as the deployment and integration processes resemble those of traditional monolithic applications, potentially hindering the overall agility of development.

### 2.5.3 Routing

In a micro frontend architecture, two levels of routing must be addressed: team-level routing, which handles navigation within each individual micro frontend, and top-level routing, which manages navigation across different micro frontends. The primary challenge is ensuring that both internal (team-level) and external (top-level) routing work together seamlessly to deliver a consistent user experience across independently deployed applications.

Internal routing decisions are typically left to the discretion of each team, allowing them the flexibility to implement routing patterns suited to their specific micro frontend, much like in any standalone application. However, external routing decisions are closely tied to the composition mechanism selected for the project. Depending on the composition strategy, top-level routing can be managed on the client-side, server-side, or edge-side [22].

#### 2.5.3.1 *Client-Side Routing*

Client-side routing dynamically manages routes within the browser, enabling an app-like user experience by loading micro frontends without requiring full page reloads from the server. As users navigate between different pages or application states, the browser updates the URL, and the application shell determines which micro frontend to load based on the route and business rules.

This approach is particularly advantageous for complex routing scenarios, such as handling user authentication, geolocation-based content, or conditional logic [22]. For instance, the application shell might load an authenticated user area if the user is logged in, or a landing page if the user is visiting for the first time. In this setup, the application shell owns the routing logic and decides which micro frontend to load based on predefined configurations.

#### 2.5.3.2 *Server-Side Routing*

In server-side routing, page requests are handled on the server, which assembles the required micro frontends before sending a fully composed page to the client. This method places the routing logic on the server, allowing the server to determine which micro frontends are needed based on the requested URL.

While this approach simplifies the client-side experience by delivering a pre-assembled page, it can introduce scalability challenges. Handling high volumes of traffic or burst traffic requires a robust server infrastructure capable of managing rapid horizontal scaling. Each server must retrieve the necessary micro frontends and compose the page for the client, which can become resource-intensive in high-demand environments [22].

Server-side routing is well-suited for applications that require centralized control over routing and need to offload much of the complexity to the server. However, it may limit the flexibility and responsiveness that client-side routing provides.

### 2.5.3.3 *Edge-Side Routing*

Edge-side routing handles the routing logic at the CDN level, distributing routing responsibilities closer to the user to enhance performance. This approach offers performance benefits by reducing the distance between the user and the server, leading to faster load times. However, it typically allows for less complex routing logic [22]. Since the CDN operates based on simple URL matching, there is limited flexibility for handling sophisticated business rules or conditional routing.

Edge-side routing is a good fit for simple applications that prioritize performance and scalability over routing complexity, especially when the focus is on delivering static or pre-rendered micro frontend compositions.

## 2.5.4 Communication

Communication between MFEs within a micro frontend architecture is often necessary for complex systems. This becomes especially important when multiple micro frontends are displayed on the same page, as user interactions in one application may need to trigger updates or actions in others. Communication methods in micro frontends can be broadly categorized into event-based, state-based, and integration-based patterns.

It is important to ensure that each micro frontend remains unaware of the internal workings of others to preserve the principle of independent deployment [22]. Overly tight coupling between micro frontends would undermine their autonomy, making coordination between teams and deployment more difficult.

### 2.5.4.1 *Event-Based Communication*

Event-based communication relies on broadcasting messages or events between micro frontends to notify changes, share data, and trigger actions across different components. This approach decouples the micro frontends, allowing for more flexibility and modularity.

- **Event Bus:** The event bus is a global event-based communication mechanism typically implemented in the Application Shell. It allows independent micro frontends to communicate by broadcasting events, with other micro frontends that are interested in a particular event listening for it and reacting accordingly.
- **Custom Events API:** This native browser API enables the creation and dispatching of custom events within a web page. These events are typically dispatched through a globally accessible object like `window`, which makes them available across all micro frontends. However, using custom events in architectures with iframes can be challenging, as each iframe has its own window object, complicating event propagation.

- **Broadcast Channel API:** This browser feature allows the creation of a bidirectional communication channel between different browsing contexts – such as windows, tabs, workers, and iframes – within the same origin. It supports real-time message exchange between micro frontends, making it particularly useful when managing communication between isolated contexts like multiple tabs or iframes.
- **window.postMessage:** This global method provides a simple way to send messages between different windows, tabs, or iframes in a web browser. It is especially valuable in iframe-based compositions, where micro frontends can communicate by passing messages through this API, ensuring relevant information is shared securely and efficiently across frames.

#### 2.5.4.2 *State-Based Communication*

State-based communication uses shared state mechanisms to pass data between micro frontends, providing indirect yet persistent means of coordination. This method results in tighter coupling compared to event-based communication, as it requires micro frontends to access and manipulate a common state. However, for vertical split architectures, it is a viable method of communication.

- **Storage:** Browser storage options like session storage, local storage, and cookies can be used to store data on the client side. Micro frontends can access this data to share information indirectly, which is particularly useful for persisting state across different sessions or micro frontends.
- **Query Strings:** Query parameters in the page URL can be used to pass information between micro frontends. By appending parameters to the URL and having each micro frontend extract and interpret these parameters, micro frontends can share context without requiring a direct interaction mechanism.

#### 2.5.4.3 *Integration-Based Communication*

Integration-based communication involves direct interaction between micro frontends, allowing them to exchange data or trigger actions through more tightly coupled mechanisms.

- **Attributes:** In this approach, micro frontends share contextual information by passing data through HTML attributes. This method usually follows a unidirectional data flow model, where parent components pass data down to child components, ensuring a clear hierarchy and data flow between fragments.
- **Direct Function Calls:** One micro frontend can expose global functions (e.g., in the global scope) that other micro frontends can invoke. This approach is highly

discouraged, as it relies on one micro frontend having knowledge of the internal structure of another.

## 2.5.5 Backend Integration

There are several approaches for integrating micro frontends with backend layers, each designed to serve distinct purposes. These include service registries, API gateways, and the Backend for Frontend pattern.

### 2.5.5.1 Service Registry

A service registry is a centralized directory that contains information about all the available services that can be consumed by the client. The use of a service registry eliminates the need for shared libraries, environment variables, or configurations that require injection during the continuous integration process, as well as the necessity of hardcoding all endpoints in the frontend codebase [22]. Figure 17 shows this strategy.

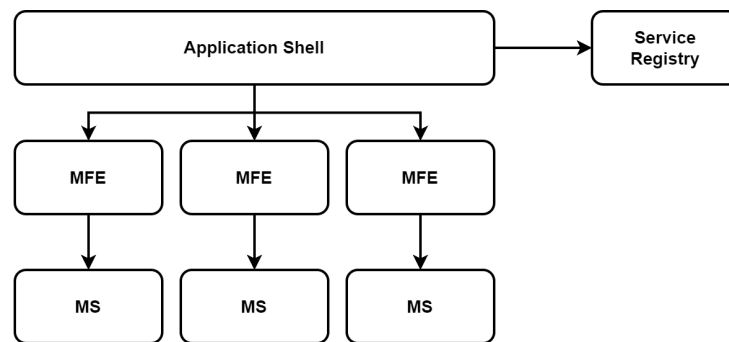


Figure 17: Service registry pattern

When the system loads for the first time, the Application Shell fetches the service registry and provides it to the micro frontends, which then retrieve the necessary service URLs directly. This approach improves flexibility, as changes to service endpoints no longer require modifications to individual micro frontend codebases. Typically, the service registry is implemented using a static JSON file or through a request to an API.

### 2.5.5.2 API Gateway

The API gateway is an intermediary layer that mediates communication between micro frontends and backend microservices, as seen in Figure 18. It centralizes information about service endpoints, improves decoupling between frontend and backend, and provides additional security and traffic management capabilities.

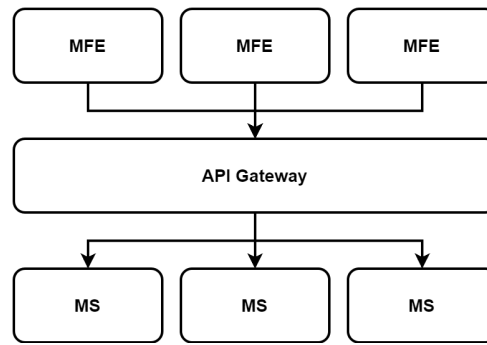


Figure 18: API gateway pattern

As the primary entry point in a microservices architecture [22], the API gateway separates the public external APIs from the private internal APIs, allowing clients to interact with the public APIs. It also centralizes common shared functionalities such as authorization, monitoring, logging, and rate limiting. Furthermore, the API gateway can proxy requests between legacy monolithic systems and newly implemented microservices, ensuring smooth transitions during system migrations.

### 2.5.5.3 Backend for Frontend

The Backend for Frontend (BFF) pattern extends the API gateway pattern [22] by providing a single entry point for each client type. In the context of micro frontend architectures, each micro frontend has its own dedicated BFF that aggregates data from multiple services and optimizes it for the frontend. This design abstraction prevents the frontend from being overloaded with complex API orchestration, thereby improving performance and simplifying data retrieval. The BFF pattern is shown in Figure 19.

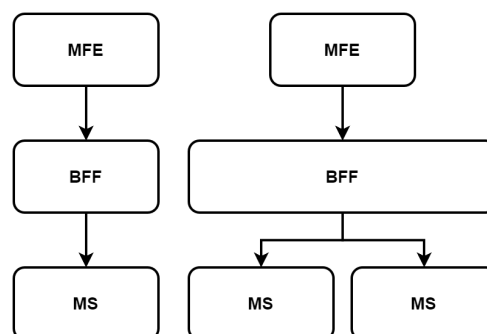


Figure 19: Backend for Frontend pattern

This pattern focuses on creating backend APIs that solves the specific needs of each frontend application. Rather than relying on a single backend to serve all functionalities for various client types of different business domains, the BFF approach promotes the use of specialized backends that address the unique requirements of each frontend.

## 2.6 SOFTWARE QUALITY

Quality refers to how well a product or service aligns with customers' expectations and requirements. Achieving continuous improvement requires that quality be clearly defined and measured. Quality models offer valuable frameworks for predicting reliability, managing quality during development, and assessing software complexity [16].

In modern software development, ensuring high quality is more important than ever, particularly when working with complex architectures like micro frontends. Among the various software quality models available, this work adopts the ISO/IEC 25010 standard, which provides a comprehensive set of quality attributes well-suited for evaluating micro frontends.

### 2.6.1 ISO/IEC 25010

The ISO/IEC 25010 standard defines system quality as the degree to which a system meets the stated and implied needs of its various stakeholders [11]. It includes two models: the quality in use model, which is beyond the scope of this work, and the product quality model, which addresses both static and dynamic properties of a software.

The product quality model supports the specification and evaluation of software from multiple perspectives, including those involved in acquisition, requirements, development, use, evaluation, support, maintenance, quality assurance, control, and auditing. It organizes the product quality into eight key characteristics, each composed of related subcharacteristics:

- **Functionality suitability:** capacity to provide features and capabilities that enable users to complete the specified tasks.
- **Performance efficiency:** capacity to deliver appropriate performance in terms of resource utilization and response time.
- **Compatibility:** capacity to operate effectively while sharing the same common environment and resources with other systems.
- **Usability:** capacity to be used effectively, efficiently, and satisfactorily by users.
- **Reliability:** capacity to prevent failures and maintain adequate performance under specified conditions, even when disruptions occur.
- **Security:** capacity to protect itself and its data from threats, such as unauthorized access, use, disclosure, disruption, modification, or destruction.
- **Maintainability:** capacity to be modified, improved, corrected, or adapted to meet new requirements.
- **Portability:** capacity to be transferred to or deployed in different environments, including hardware platforms and software systems.

# 3

## SOLUTION DESIGN

This chapter presents a systematic framework for designing a micro frontend-based architecture for complex graphical user interfaces. The approach applies use case modeling and domain analysis, grounded in the principles of Object-Oriented Modeling and Domain-Driven Design, to translate abstract system requirements into a maintainable and scalable architecture aligned with core domain logic. A diagram illustrating the proposed software lifecycle model is shown in Figure 20.

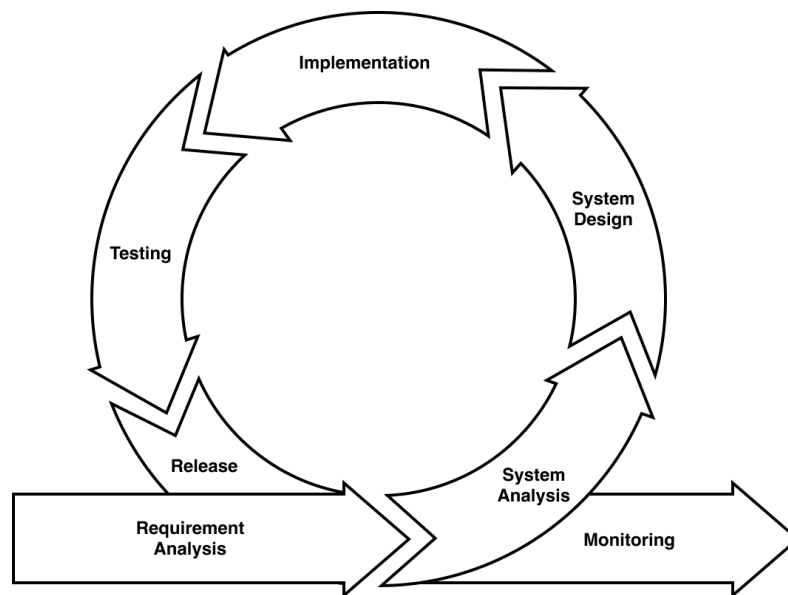


Figure 20: Proposed software lifecycle model

### 3.1 REQUIREMENT ANALYSIS

The scope of this work is particularly significant to developers, engineers, and the broader community involved in the RoboCup SSL environment. A discovery session was conducted with the RobôCIn team – identified as a key stakeholder group representing the community’s interests – to gather requirements, constraints, and pain points. The outcomes of this session were carefully filtered and refined into a requirements document, which serves as the input for all subsequent project work.



Various terminologies exist for classifying requirements. For the purposes of this work, requirements are categorized into functional requirements, non-functional requirements, and design and implementation constraints.

### 3.1.1 Functional Requirements

Functional requirements define the key services a system must provide and how it should behave in response to certain inputs or situations [36]. Although functional requirements are often presented in abstract terms and elaborated further throughout the development process, this work details them directly as use cases. Table 1 summarizes the functional requirements.

Feature	ID	Functional Requirement
Match Playback	UC001	The user shall be able to watch a live match.
	UC002	The user shall be able to play the match.
	UC003	The user shall be able to pause the match, with the system displaying the elapsed time since the pause.
	UC004	The user shall be able to control the current playback time of the match.
	UC005	The user shall be able to adjust the playback speed.
	UC006	The user shall be able to jump to a specific time in the match.
	UC007	The user shall be able to advance or rewind frame by frame.
	UC008	The user shall be able to advance or rewind by specific time increments (e.g., 5s, 10s).
	UC009	The user shall be able to jump to the live broadcast using a "LIVE" button.
3D Visualization	UC010	The user shall be able to lock the view on a specific robot or ball.
	UC011	The user shall be able to view the field coordinates with the mouse cursor.
	UC012	The user shall be able to zoom in on any position on the field.
Match Information	UC013	The user shall be able to view details of a robot or ball by clicking on them.
	UC014	The user shall be able to see overall match information.
	UC015	The user shall be able to view the complete history of match events.
	UC016	The user shall be able to click on a specific event to jump to that point in the match.
	UC017	The user shall be able to filter events in the match history.
Parameter Management	UC018	The user shall be able to search for parameters.
	UC019	The user shall be able to import a parameter file.
	UC020	The user shall be able to export a parameter file.
	UC021	The user shall be able to change parameter types and values.
	UC022	The user shall be able to send parameters to the services.

Table 1: System's functional requirements

The system's functional requirements are aligned with the features identified by stakeholders. For the match playback feature, the use cases focus on viewing and controlling the

playback of a live match. The 3D visualization feature involves rendering match objects within a 3D environment and providing camera controls. In the match information feature, the use cases address real-time tracking of the scoreboard and match events. Lastly, the parameters management feature includes configuring and sending parameters to control backend services.

In UML, use case diagrams visually represent interactions between the system and external entities, such as users or other systems, highlighting the relationships relevant to each use case. While experienced domain experts may sometimes skip this step by identifying system subdomain boundaries directly, use case diagrams remain essential for subsequent phases of this work. Figure 21 presents the system's use case model, capturing these interactions.

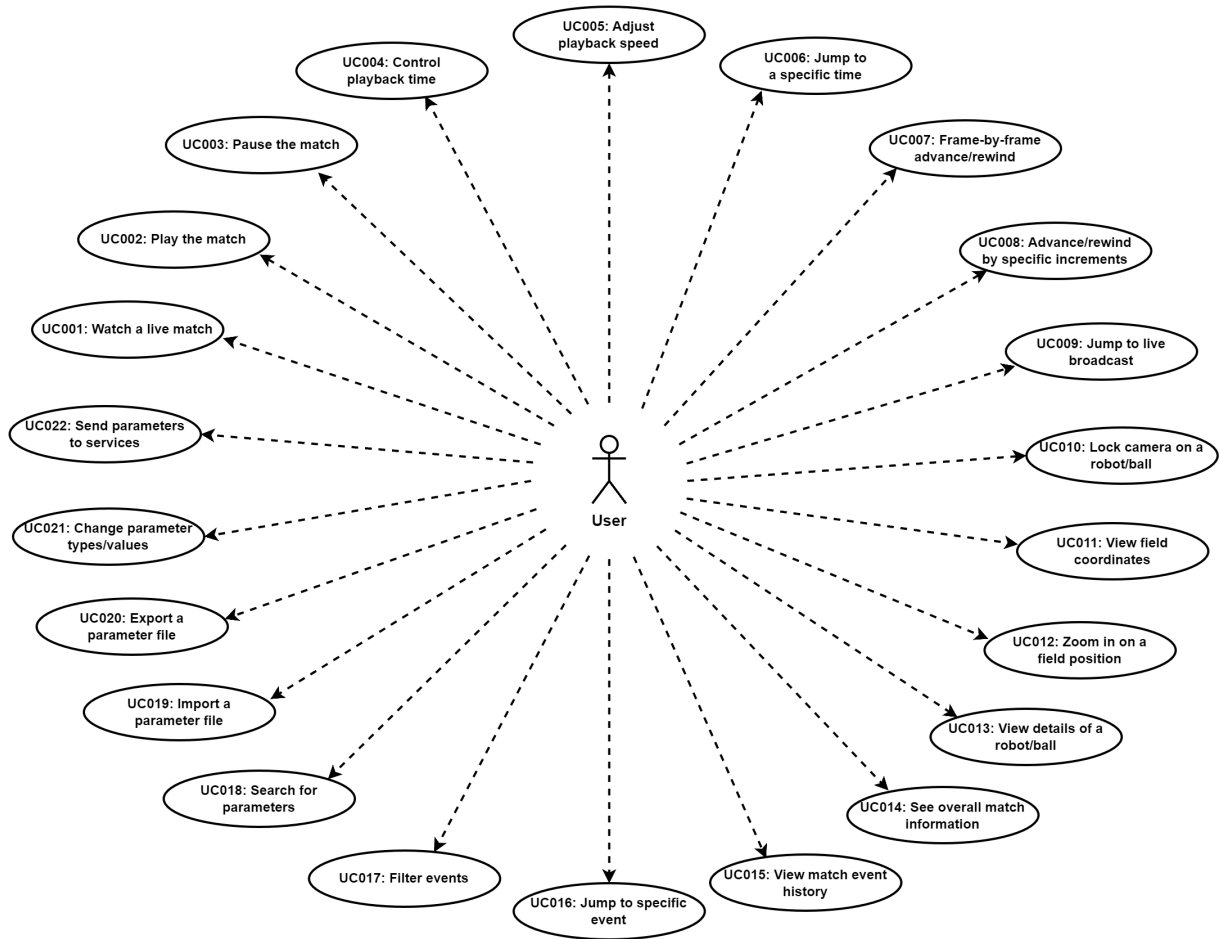


Figure 21: System's use case diagram

### 3.1.2 Non-Functional Requirements

Non-functional requirements (NFRs), or quality attributes [5], place constraints on what the system can do, covering aspects like performance efficiency, reliability, maintainability and portability [36].

This study adopts the ISO/IEC 25010 model [11] to assess non-functional requirements. This model provides a set of quality characteristics and sub-characteristics, enabling a systematic evaluation of relevant quality attributes.

Considering the study's focus on the impact of adopting a distributed micro frontends architecture over a traditional monolithic approach, the analysis prioritizes a specific subset of quality attributes that are particularly influenced by this architectural shift, as detailed in Table 2.

Quality Attribute	Non-Functional Requirement
Performance Efficiency	Respond promptly and within defined time parameters.
	Manage and utilize resources according to specified constraints during functional operations.
	Ensure optimal capacity by managing and meeting the specified maximum limits of parameters.
Reliability	Meet reliability needs under normal operation.
	Be operational and accessible when required for use.
	Operate as intended despite the presence of hardware or software faults.
	Recover data and re-establish system state in case of interruption or failure.
Maintainability	Components should have minimal impact on others when changed.
	Components should be reusable in multiple systems.
	Facilitate effective assessment of intended changes' impact, diagnosis of deficiencies or failures, and identification of parts for modification.
	Support modification without introducing defects or degrading product quality.
	Support effective establishment of test criteria and efficient execution of tests for compliance.
Portability	Be adapted for different or evolving hardware, software, or operational environments.
	Successfully install and uninstall in specified environments.
	Replace another specified software product for the same purpose in the same environment.

Table 2: System's quality attributes and non-functional requirements

### 3.1.3 Design Constraints

Design constraints limit the developer's choices for valid and necessary reasons [42]. Also, when designing of large modular systems, low coupling and high cohesion are foundational design principles that remain critical [9]. By keeping each module focused and reducing dependencies, these principles ensure scalability, maintainability, and architectural integrity, even within imposed constraints.

For this project, the design constraints are as follows:

- **Distributed:** The system must be composed of fully independent applications, enabling them to evolve separately and reducing dependencies across the system.
- **Future-ready:** The architecture of the system must support future changes and new technologies with minimal rework.
- **Backend independent:** The backend of the system must be decoupled from the frontend, allowing independent updates or replacements without affecting the user interface.

### 3.1.4 Implementation Constraints

Implementation constraints address practical considerations in system construction, influenced by the team's expertise and the chosen technology stack. These constraints guide coding and deployment practices to ensure the system remains consistent, maintainable, and adaptable.

For this project, the implementation constraints are:

- **Minimal dependency on external libraries:** The system must favor native technologies to minimize risks and ensure long-term stability.
- **Technology agnosticism:** The system must support various frameworks, languages, and technologies.
- **Web browser compatibility:** The system must be capable of running in a web browser, ensuring broad accessibility and ease of deployment.

## 3.2 SYSTEM ANALYSIS

In software engineering, models are essential tools for simplifying and understanding complex problems. These models are particularly powerful in the context of Object-Oriented Modeling and Domain-Driven Design, where the model closely reflects the real-world domain.

During the system analysis phase, models are created to guide the subsequent design phase. These artifacts capture both functional and non-functional requirements, including domain models and bounded contexts that define how each part of the system will be implemented.

### 3.2.1 Domain Analysis

Domain analysis is the process by which a software engineer acquires the necessary background knowledge to understand a problem and make informed decisions throughout system analysis and subsequent software engineering phases. The term "domain" refers to the specific business or technology area where the software will be applied [19].

The process of domain analysis involves gathering information from various sources, including domain experts, relevant literature, existing software, and documentation. A common technique in this phase is filtering nouns from use case descriptions to identify potential entities, excluding irrelevant elements – which are handled later in the design phase.

Using the identified entities, abstract class diagrams are constructed as part of an exploratory domain model. These diagrams capture the entities and relationships within the domain, but are not intended to model implementation details. Operations, polymorphism, and certain modeling principles are typically not the focus at this stage [19]. The system’s exploratory domain model is depicted in Figure 22.

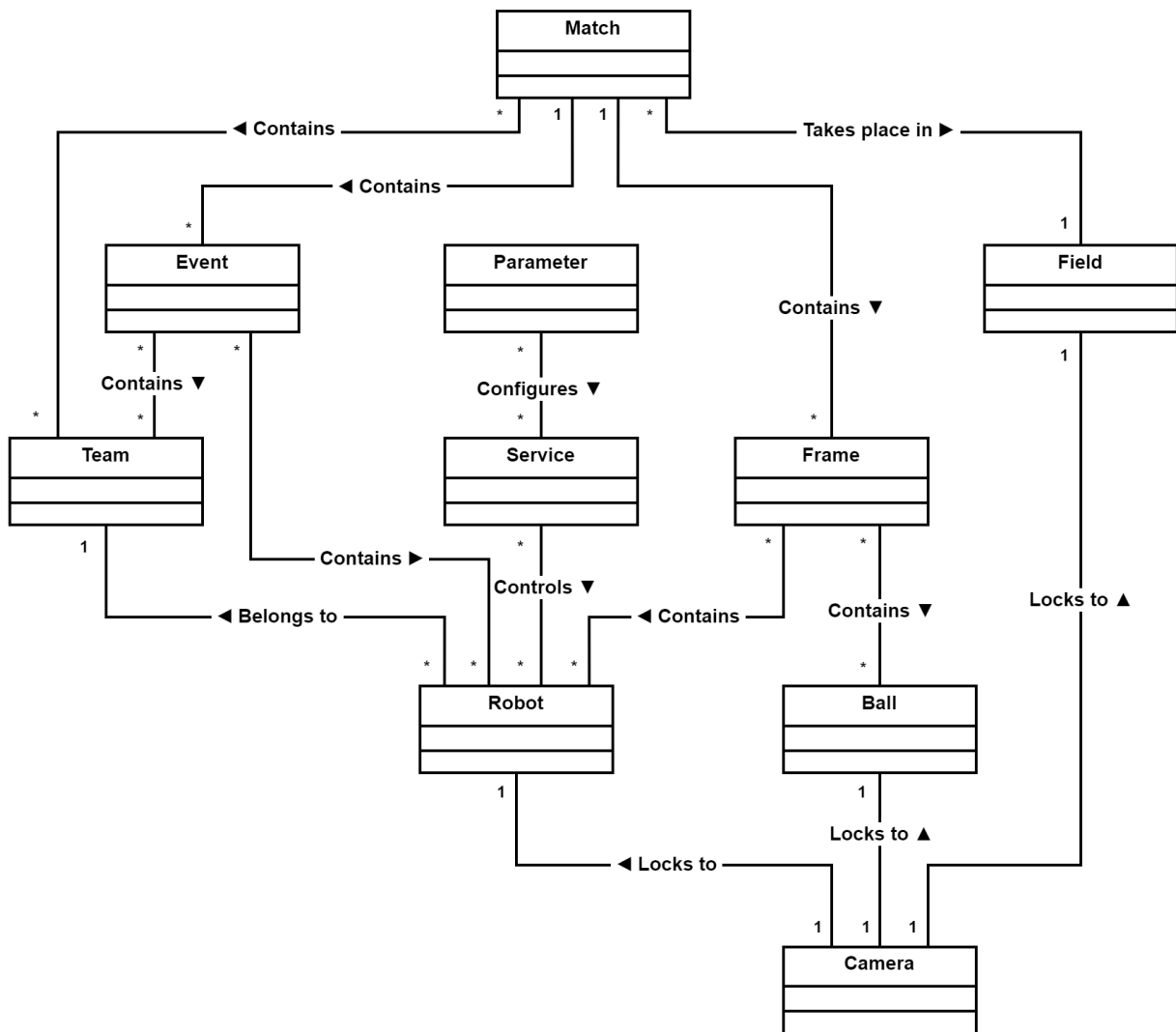


Figure 22: System’s exploratory domain model

At the core of the system is the Match, which encapsulates all aspects of a soccer robot game. A Match consists of multiple Frames, each capturing a snapshot of the game state at a particular moment. These Frames are critical for rendering the match’s progression over time and contain the state of both the Robots and the Ball at each particular instant.

Within the context of a Match, Teams are also an essential entity, with each team

composed of several Robots. These Teams participate in the Match, and their actions and strategies are central to the game's outcome. The Match takes place on a defined Field, which establishes the spatial boundaries and conditions under which the Robots operate. Moreover, the Match is composed of various Events that represent significant occurrences during the game, such as goals, fouls, or other game-changing actions. These Events are associated with both the Robots and the Teams, providing a detailed account of the Match's progression.

In addition to the core gameplay elements, the model also includes Service entities, representing various aspects of robot control mapped to backend services. These Services are configured by Parameters, which define the operational settings for the Robots during the match.

Finally, another important entity is the Camera, which defines the viewpoint in 3D space, allowing users to navigate the match from different perspectives and lock the view onto specific entities, such as Robots, the Field, or the Ball.

As the domain model evolves, it begins to identify subdomains and their interrelationships. This evolving model maps core business functions and dependencies, acting as a bridge between business understanding and technical implementation. A key aspect of this is the development of a Ubiquitous Language – a common vocabulary derived from domain experts' jargon. This language is refined for clarity and precision to ensure all stakeholders, from developers to domain experts, have aligned discussions that translate into code [9].

### 3.2.2 Bounded Contexts

A bounded context in Domain-Driven Design defines where a particular model is applicable, ensuring clarity and consistency within its scope. Within this context, the model remains coherent and focused on its domain, without concern for relevance outside these boundaries. Different contexts may adopt distinct models, terminology, and rules, each reflecting their own version of the Ubiquitous Language [9].

Explicitly defining these boundaries keeps models effective and clear, minimizing confusion when shifting between contexts. Integration across contexts requires careful translation, which helps reduce dependencies between subsystems. This separation allows models to accurately capture domain entities and relationships while reflecting different levels of abstraction.

The process of identifying bounded contexts begins by grouping use cases that share similar goals and domain concepts. This approach reveals natural boundaries within the system, organizing related functionalities into distinct contexts where a consistent domain model can apply, supporting robust and adaptable design. The system's bounded contexts are given in Figure 23.

The identified bounded contexts reflect the same correspondence between features and use cases observed during the requirements analysis phase, resulting in four distinct contexts: Match Playback, 3D Visualization, Match Information, and Parameters Management, each encapsulating a specific aspect of the system's functionality.

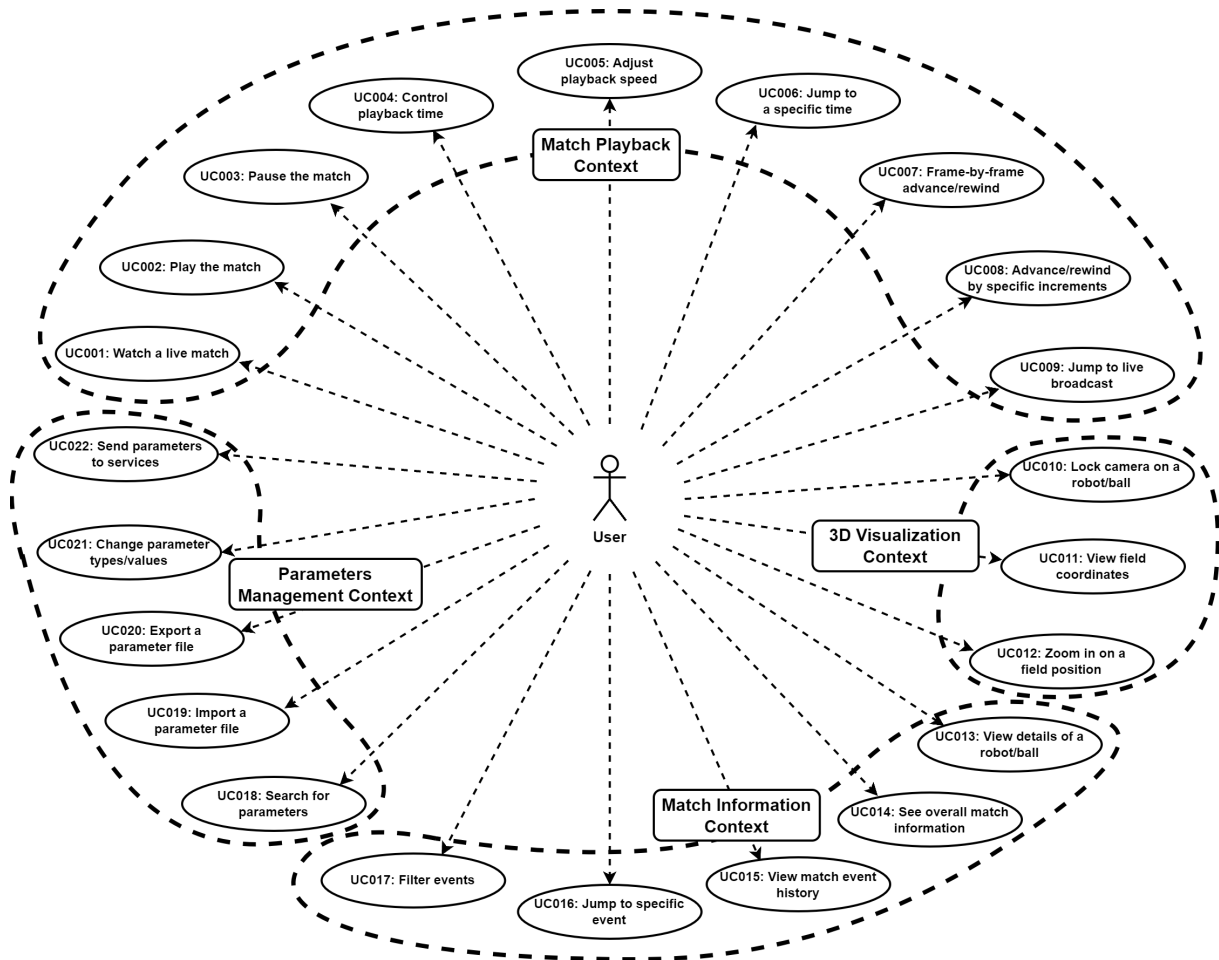


Figure 23: System's bounded contexts

After identifying the bounded contexts, the next step is to distribute the entities from the exploratory domain model among them. Each context should encapsulate the entities that most accurately represent the core concepts of its subdomain. It is important that these entities align with the specific requirements of the context, preserving well-defined boundaries and minimizing overlap between contexts.

As the domain model is refined, new entities may emerge, and existing ones might be renamed. In some cases, entities may only represent a partial view of the whole. The outcome, represented in Figure 24, is not a single domain model, but a collection of models, each characterized by its own Ubiquitous Language, with translation maps allowing communication between them.

It is important to remain flexible during the design and implementation process. As the system evolves, certain contexts may reveal significant overlaps and should be merged, while others may emerge as the domain becomes more defined. Additionally, if a context becomes too complex, it may need to be divided into smaller, more manageable contexts. These adjustments are an inherent and necessary part of refining the system's architecture.

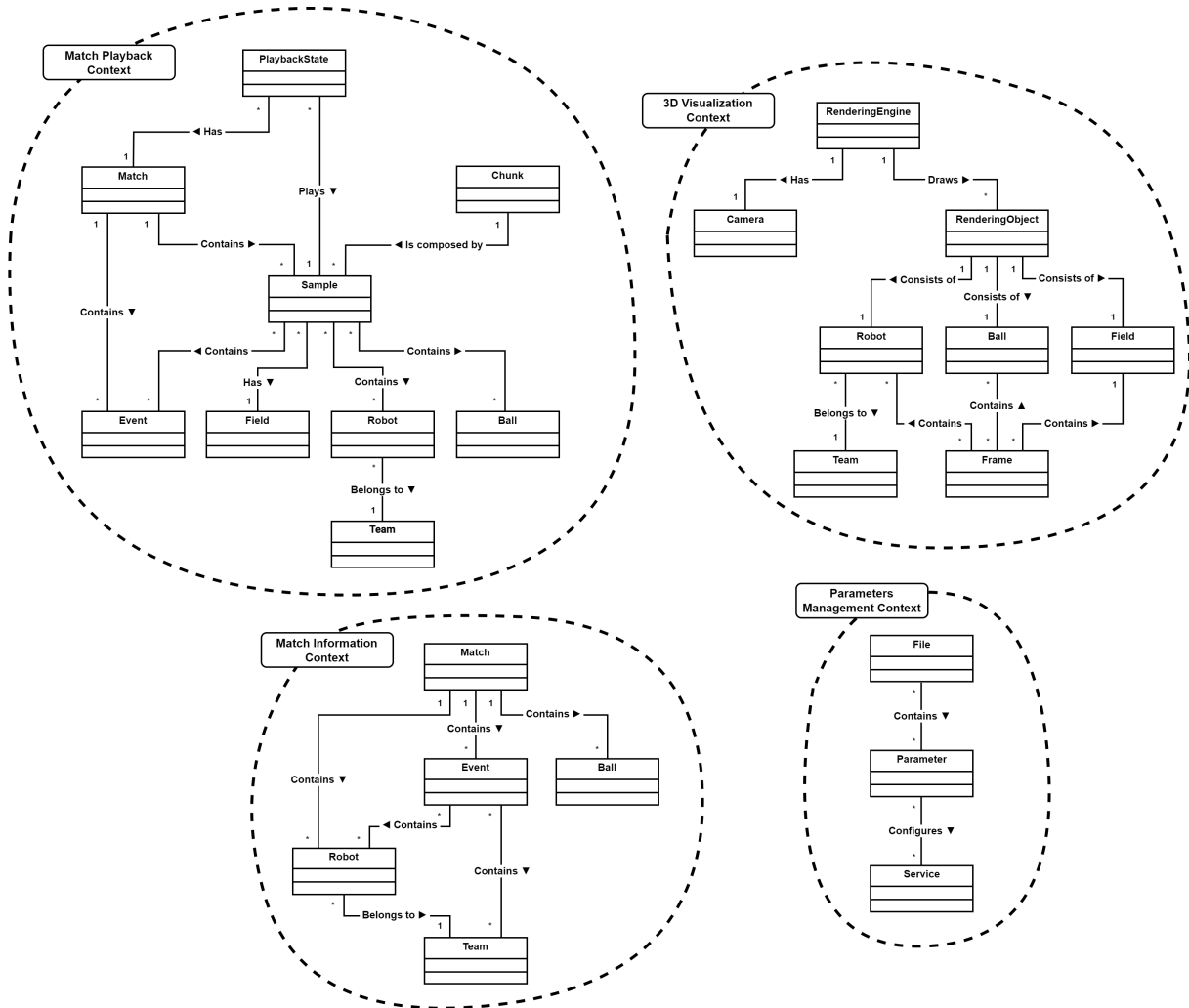


Figure 24: System's refined domain models inside each bounded context

### 3.3 SYSTEM DESIGN

During the system design phase, models are developed to guide the system architecture, building on the insights gained during the analysis phase. In contrast to analysis models, which capture domain concepts, design models translate these concepts into concrete architectural solutions. Design models include detailed representations such as component diagrams, interaction diagrams, and design patterns that define how each component of the system will be implemented. Additionally, bounded contexts identified during analysis are further refined in the design phase, ensuring a clear understanding of how different parts of the system will interact and function together.

#### 3.3.1 Micro Frontends

The definition and implementation of micro frontends are influenced by several factors, including the size of the development team and their familiarity with this architectural paradigm. Achieving the right balance between the benefits of decoupling and the additional complexity it



introduces is critical. Even small teams can obtain significant advantages from adopting a distributed architecture on the frontend, such as improved scalability, resilience, and maintainability. This is particularly true in complex domains like robotics graphical user interfaces, where micro frontends offer a way to build maintainable applications, even with high turnover among team members.

A good starting point for defining MFEs is to align each one with a bounded context. By mapping MFEs to these contexts, the architecture stays cohesive, with each frontend segment reflecting the natural division of the domain. This approach ensures that each MFE encapsulates the functionality and behavior specific to its subdomain, following the principles of domain-driven design.

The system architecture is composed of the following defined micro frontends:

- **Player MFE:** corresponds to the Match Playback context
- **Viewer MFE:** corresponds to the 3D Visualization context
- **Scoreboard MFE:** corresponds to the Match Information context
- **Params MFE:** corresponds to the Parameters Management context

As the system evolves, this initial alignment of MFEs to bounded contexts should be iteratively refined. As the team gains a deeper understanding of the domain and system requirements, some MFEs may need to be split into smaller, more granular components, while others might be consolidated to reduce complexity or improve performance. This iterative refinement process allows the architecture to adapt to changing business needs and technical constraints, ensuring that it remains scalable and maintainable over time.

### 3.3.2 Architectural Design Patterns

The Gang of Four describes design patterns as reusable solutions to common design challenges, structured as descriptions of communicating objects and classes [12]. These patterns facilitate the reuse of proven designs and architectures, enabling designers to arrive at effective solutions more quickly. While many of these patterns are highly valuable within individual micro frontend applications – such as the Observer pattern, which drives much of the reactivity in user interactions – this section focuses on patterns that are architecturally significant: domain events, Backends-for-Frontends (BFFs), and the Application Shell.

#### 3.3.2.1 Domain Events

Domain Events are a DDD pattern described by Vaughn Vernon [40] that enables the design of autonomous services and systems, where each application operates independently by using asynchronous messaging instead of direct calls. These events originate from domain

entities and align with the Ubiquitous Language of their bounded context, being mapped to other structures when crossing bounded contexts to ensure they accurately reflect their meaning within the new context.

This approach is particularly well-suited for web frontend architectures, which are inherently event-driven and reactive through browser APIs and frameworks. Domain Events bridge user interactions with domain logic, guaranteeing that user actions are accurately represented in the domain model. As a result, significant communications within a single micro frontend or across different micro frontends are effectively modeled as Domain Events.

In the local context, inside a bounded contexts, domain events like `PlaybackUpdateEvent` is emitted by the Playback entity in Player MFE when a new sample is processed, to update the state of the UI elements of the micro frontend. A `FrameRenderEvent` is another local domain event, that does not cross the boundary and is emitted by the `RenderingEngine` in the Viewer MFE and received by the `RenderingObjects`, that update their position and physical dimensions. For global domain events, that cross different bounded contexts, `SampleReceiveEvent` is emitted by the Playback entity in the Player MFE, being listened by Viewer and Scoreboard MFE to maintain they synchronized. The `LockToEntityEvent` is a global event emitted by the Scoreboard MFE that is listened by the Viewer MFE, that changes the camera position.

Within a local bounded context, domain events such as the `PlaybackUpdateEvent` are triggered by the Playback entity in the Player MFE whenever a new sample is processed. This event updates the UI elements of the micro frontend. Another example of a local domain event is the `FrameRenderEvent`, which is emitted by the `RenderingEngine` in the Viewer MFE and received by `RenderingObjects` to update their position and physical dimensions. These events remain confined to their respective bounded contexts.

Global domain events, which cross multiple bounded contexts, include the `SampleReceiveEvent`, emitted by the Playback entity in the Player MFE. This event is consumed by both the Viewer and Scoreboard MFEs to ensure synchronization across contexts. Similarly, the `LockToEntityEvent` is emitted by the Scoreboard MFE and is listened to by the Viewer MFE, where it triggers changes to the camera position.

### 3.3.2.2 *Backends for Frontends*

The Backend for Frontend pattern is a consumer-focused approach to API design, where each BFF serves as an intermediary layer that provides data in the format required by its specific client. This allows core backend services to stay generic and reusable, avoiding the need for over-customization for different frontends. By reshaping and combining data, BFFs improve the interaction between frontends and backends, maintaining the scalability and flexibility of the backend while enhancing the user experience on the frontend.

BFFs are usually designed as unique entry points for a given device group, like smartphones or desktops. However, an interesting use case is when BFFs are used to map backend

domains to frontend domains, keeping them separated and independent. This supports cleaner architectural boundaries and promotes a more flexible system [22].

In essence, BFFs work as specialized backends created to each user experience. They act as the anti-corruption layer in Domain-Driven Design [9], protecting frontends from the complexities of backend logic and data, and only exposing the information and services necessary for the user interaction. Figure 25 shows the system’s architecture, including the BFFs.

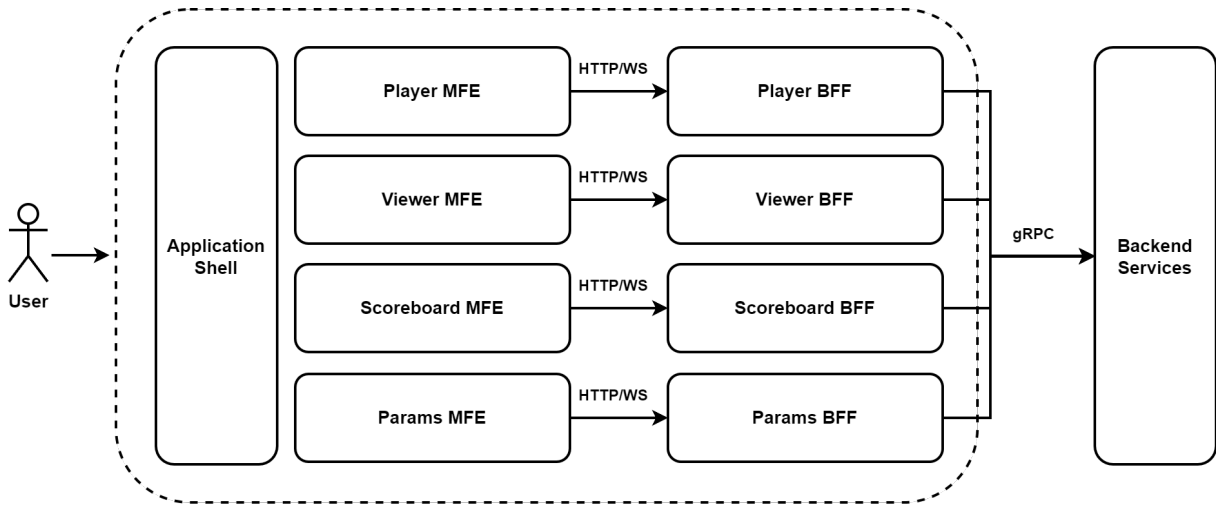


Figure 25: System’s architecture

### 3.3.2.3 Application Shell

The Application Shell is an orchestrator for micro frontends, acting as the first layer loaded when the application is accessed and remaining active throughout the user’s session [22]. It manages the dynamic loading and unloading of micro frontends based on navigation patterns, ensuring that only the relevant components are retrieved. This approach minimizes unnecessary resource consumption, optimizing performance and user experience.

At its core, the Application Shell itself functions as a mediator [12] within the micro frontend architecture. It coordinates communication between individual micro frontends, ensuring they remain decoupled and focused on their respective tasks. By acting as a central authority for these interactions, the Application Shell maintains the principles of reactivity and loose coupling – both fundamental in distributed systems like micro frontends.

As an indispensable component in the architecture, the Application Shell can be observed in Figure 25 and is discussed with more details in the following chapter.

### 3.3.3 Service Interaction Model

Interaction diagrams are essential tools for modeling the dynamic behavior of software systems, providing a clear representation of the steps involved in executing a use case or any specific functionality. Collectively, these steps form what is known as an interaction [19]. One

notable type of interaction diagram, the Service Interaction Model, is particularly useful for illustrating the communication between the micro frontends and the other services. These diagrams guide the refinement of the system's architecture by making the flow of interactions explicit and identifying potential inefficiencies.

As the design process unfolds, new entities often emerge. When such entities are identified, the system's models – whether classes, attributes, or relationships – should be updated to ensure they remain aligned with the evolving domain requirements. This iterative approach is fundamental to maintaining both the scalability and adaptability of the architecture while preserving its structural integrity.

Figure 26 demonstrates the Service Interaction Model for the live match watch use case.

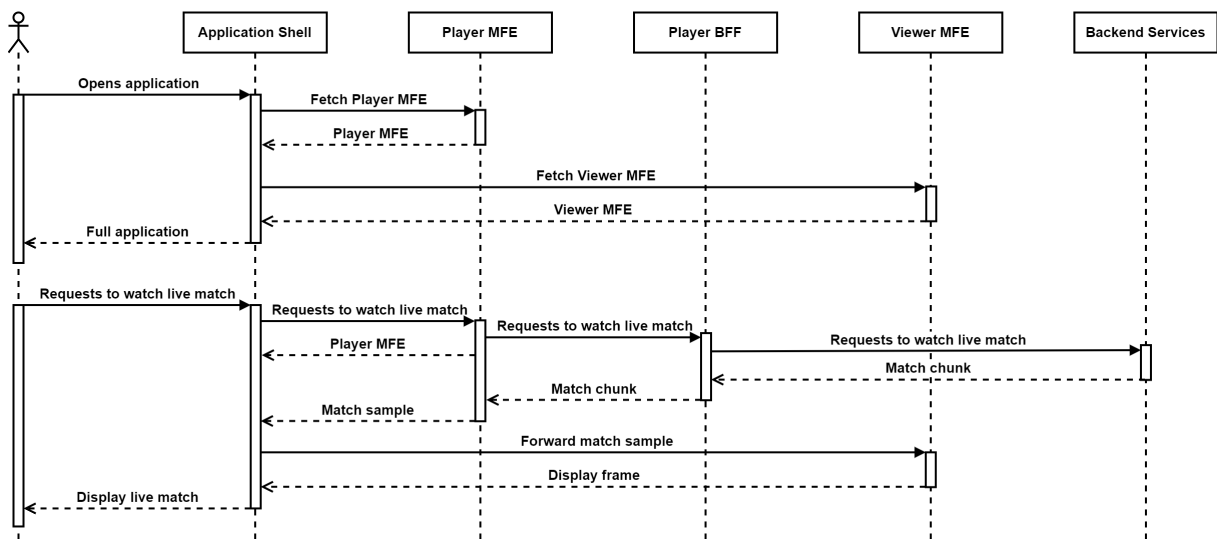


Figure 26: Service interaction model for UC001

### 3.3.4 Service Component Model

As the system architecture is refined, it becomes essential to translate high-level design decisions into a detailed component diagram. This process involves making architectural choices that will shape the system's structure and behavior. The component diagram provides a visual representation of the system's major building blocks and their interactions, serving as a blueprint for both developers and stakeholders. By clearly delineating components and their dependencies, this diagram helps ensure that the system's architecture is robust, scalable, and aligned with the project's requirements.

Figure 27 presents the system's component diagram, which includes the Application Shell, micro frontends, and their respective BFFs. The backend services are represented as a generic layer, illustrating that they can be architected in a variety of forms, whether as a monolithic structure or a distributed system.

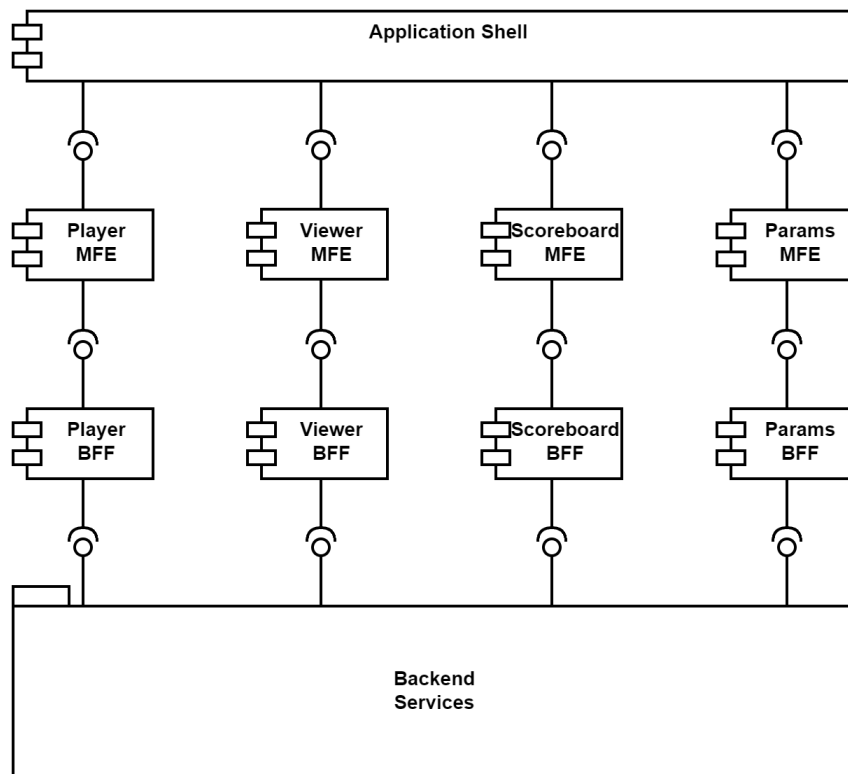


Figure 27: System's component diagram

# 4

## SOLUTION IMPLEMENTATION

This chapter describes the system's implementation, providing detailed justifications for the chosen technologies and architectural decisions. It examines the micro frontend strategies and their impact on system structure. The chapter also covers the user interface design, testing methodologies, and deployment strategies, with attention to how they support continuous integration and delivery.

### 4.1 OVERVIEW

The system was developed using a micro frontend architecture, composed of fully independent applications, each with its own dependencies, teams, and pipelines for testing and deployment. Key challenges in micro frontend design include determining how the components will communicate, how the different views will be orchestrated and how to compose the final user interface into cohesive system [22]. Each of these aspects are explained in detail, as they represent distinct strategies for designing micro frontend architectures.

Given the highly dynamic and customizable nature of the application, a horizontal split architecture was adopted, allowing multiple micro frontends to coexist within the same view. This approach introduces several complexities, such as maintaining design consistency and cohesion across micro frontends on a single page, ensuring real-time communication between them, and managing potential dependency conflicts. Additionally, integration testing becomes more challenging due to the intricate interactions between components.

The horizontal split supports various composition strategies, including client-side, edge-side, server-side, and hybrid approaches. Since the application is designed to run primarily on local machines and is heavily JavaScript-dependent, client-side composition was identified as the most effective solution. This allows the system to leverage the advantages of a single-page application model, where an initially empty shell dynamically loads and composes micro frontends at runtime, enabling personalization, rule enforcement, and flexible adjustments.

Despite the challenges posed by this strategy, the combination of horizontal split and client-side composition was chosen for its flexibility. This approach allows for the easy addition, removal, modification, and reuse of micro frontends while relying on native browser features and

APIs to provide a robust foundation. The system is designed to be extensible, accommodating future requirements and features as needed. Each micro frontend functions as a self-contained unit that integrates seamlessly into the overall system, much like a plugin.

With client-side composition in place, a client-side orchestrator, or Application Shell, is required to manage routing. This app shell controls external routing between different micro frontend-composed pages and communicates internal routing changes to each micro frontend. It ensures that micro frontends respond appropriately to changes in query or deep URL parameters. To facilitate this, the Application Shell enforces a standardized communication pattern across all micro frontends. Event-driven communication, aligned with the principles of decoupling and reactivity, is adopted to maintain consistency.

For backend communication, a layer of Backend for Frontend services intermediates between the frontends and backend services. The BFF layer translates, coordinates, and aggregates requests, ensuring smooth interaction between the frontend (using protocols such as HTTP and WebSockets) and the backend (using gRPC for more efficient communication). This architecture ensures a scalable, adaptable system capable of handling both current and future demands.

The decision to adopt a client-side composition architecture with Backend for Frontends was made based on the requirements and the advantages it offers. While a server-side composition without BFFs was considered (see Figure 28), it was discarded due to its limitations in dynamic interactions and flexibility. In server-side composition, full pages are rendered on the server and sent to the client. This method simplifies client-side logic but restricts dynamic interactions and real-time updates, as pages are static upon delivery. The lack of a BFF layer means direct communication between frontend and backend, leading to tighter coupling and more difficulty managing changes. Performance can be slower due to the need for full-page rendering on the server, and scalability is limited by server resources.

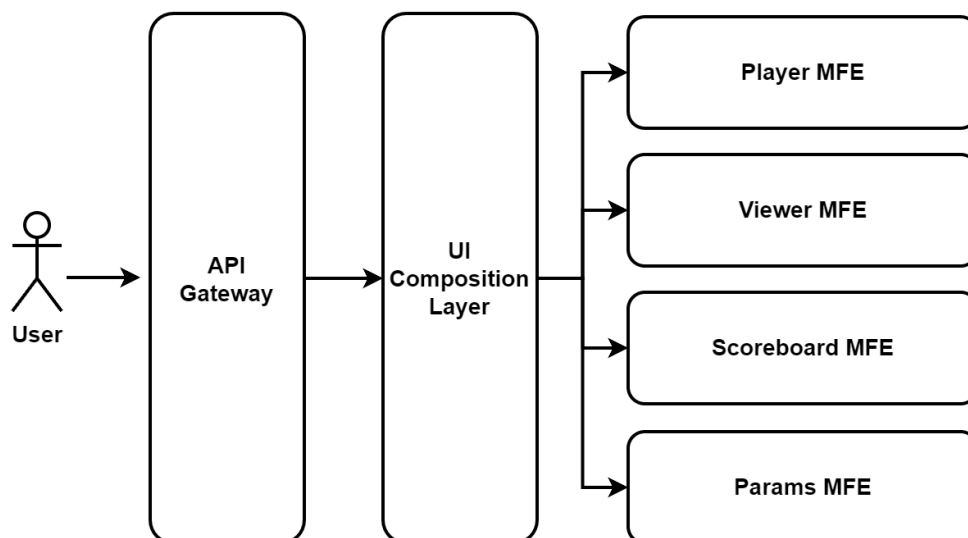


Figure 28: Discarded server-side micro frontends approach

## 4.2 USER INTERFACE CONSISTENCY

Consistency in user interface design is an important aspect for improving user perception, learning, and memory. It applies to spatial properties like menu organization and widget placement, as well as visual elements such as fonts, colors, and layouts. Consistency also ensures uniformity in actions and interaction patterns across an application, promoting skill transfer and making systems easier to learn and operate effectively [20]. However, maintaining design consistency across fully decoupled applications presents challenges.

Building base components, such as buttons, inputs, and dropdowns, can lead to coupling with specific technologies or frameworks. An alternative is to create native components that are framework-agnostic, though this approach still limits flexibility. Thus, there is a trade-off between coupling for consistency and reusability versus reimplementation for flexibility and freedom.

A central design system offers a solution by providing guidelines—defining colors, shapes, spacing, and typography – that ensure consistency across teams without stifling creativity. These design systems can be implemented as reusable components, CSS classes, or separately from the codebase using tools like Figma or Sketch, where design tokens are exported. Although modifying a design system can be costly, incremental updates across teams make it a continuous, iterative process [13]. For this work, only design tokens were defined, as seen in Figure 29, allowing each micro frontend to reimplement its own elements using their specific technologies while adhering to a consistent visual identity.



Figure 29: Defined design tokens

Additionally, each micro frontend in the system is designed within a modular grid layout. A modular grid is a tool in design that brings order and structure to layouts. It consists of uniform geometric shapes — modules — arranged in a defined sequence. In practice, a modular grid breaks a two-dimensional plane into smaller fields, separated by intermediate spaces. These fields, when arranged logically and systematically, help organize text and visual elements in line with objective, functional criteria. This approach, rooted in rational design principles, allows designers to merge modules to create a variety of sizes and shapes, offering flexibility while



maintaining a cohesive structure. In the system context, these modules, referred to as fragments, are directly related to specific micro frontends on a page, enabling modular design at both the visual and technical levels [23].

## 4.3 TECHNOLOGIES

The micro frontend architecture is divided into two key components: client-side MFEs and server-side BFFs. Client-side components are designed to run directly in the web browser, adhering to World Wide Web Consortium (W3C) standard web technologies such as HTML, CSS, and JavaScript. On the server side, a broader range of technology options was available.

To bootstrap the client-side applications, Vite was selected as the build tool, handling tasks such as bundling, development server setup, minification, and asset management. TypeScript, a strongly typed superset of JavaScript that compiles to JavaScript at build time, was used as primary programming language to provide a layer of type safety. In alignment with the principles of micro frontends [13], the applications rely on native browser features rather than on a specific framework, allowing flexibility and the ability to integrate any frontend framework as needed.

On the server side, the BFFs are written in Go, a highly performant and statically typed language designed by Google, known for its strong concurrency model. Go was chosen to ensure efficient handling of multiple requests, making it well-suited for the intermediary role that BFFs play in managing communication between frontends and backends.

All components are containerized using Docker, with Docker Compose orchestrating the containers, ensuring consistent environments for development, testing, and deployment across different stages of the system's lifecycle. This approach simplifies scaling and managing the distributed architecture inherent to micro frontends.

## 4.4 COMPONENTS

This section explores the implementation details of each component within the micro frontend architecture.

### 4.4.1 Application Shell

The Application Shell serves as the entry point to the system, orchestrating dynamic loading, rendering, and communication between micro frontends. It is the first component downloaded when a user accesses the application and remains active throughout the user's journey, coordinating the assembly of the user interface based on the requested routes [22]. Its primary responsibilities include managing client-side routing, handling global configurations, maintaining the user's initial state, and handling errors when a micro frontend fails to load. Figure 30 shows a screenshot of the implemented Application Shell.

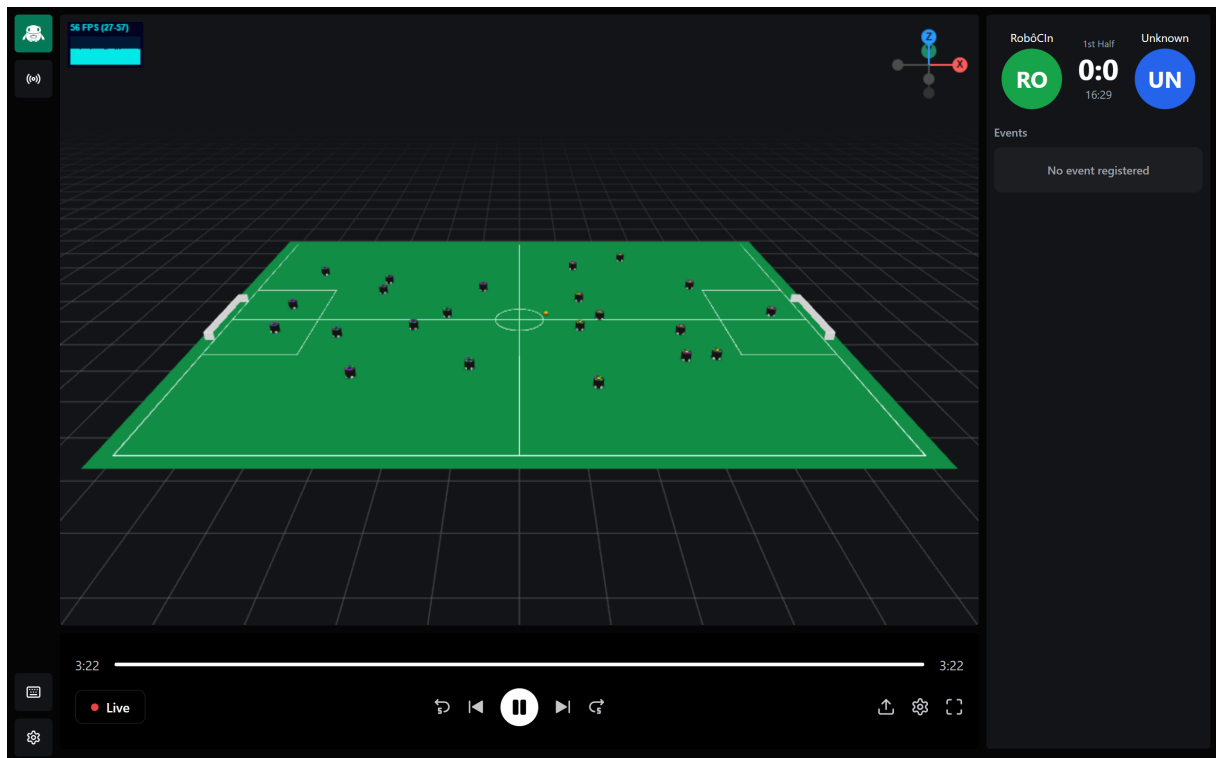


Figure 30: Application shell with micro frontends

At its core, the App Shell functions similarly to a Single-Page Application, allowing navigation between views without full page reloads. This is accomplished through a configuration file – which could potentially be replaced by a dynamic approach – that defines the available routes and specifies the corresponding micro frontends for each path. Each route corresponds to a combination of micro frontends (referred to as fragments) and defines their position, size, and properties within the modular grid layout system.

When a user navigates to a route, the Application Shell fetches the relevant fragments and initially displays a skeleton loading screen in the designated positions. Once the fragments are loaded, the shell manages communication and interactions between them, ensuring consistent behavior and user experience throughout the application. The ability to orchestrate multiple micro frontends on a single page, each operating independently, is a key feature of this architecture.

To provide communication between the micro frontends, the App Shell implements an event-driven model using the native Broadcast Channel API. This creates a shared communication layer, or Event Bus, that allows micro frontends to publish and subscribe to events. This bidirectional communication can span across different contexts, such as browser windows, tabs, frames, or web workers, ensuring real-time synchronization of data and actions. For example, a route change or a keyboard shortcut triggered in one micro frontend can be broadcasted to other micro frontends that subscribe to these events, facilitating coordinated behaviors across the application.

The App Shell supports two methods for integrating micro frontends: WebComponents and IFrames. WebComponents are the default approach due to their native integration with the

browser's DOM and their ability to encapsulate HTML, CSS, and JavaScript within a reusable component. This method aligns with the principles of micro frontends by allowing each fragment to operate independently while maintaining coherence within the broader system. However, IFrames are available as a fallback option for legacy components.

The overall internal architecture of the Application Shell is designed with extensibility in mind. While the current implementation uses WebComponents and IFrames, the system can be extended to support other frontend frameworks or technologies as they emerge. This flexibility, combined with the independence of each micro frontend, ensures that the architecture can evolve over time without significant refactoring.

#### 4.4.2 Player Micro Frontend

The Player micro frontend is a critical component within the system's architecture, responsible for managing the real-time playback of live match data. Its primary function is to synchronize all other micro frontends with the current frame of the match, making it a task-intensive application. To meet the high demands of live playback without overloading the user interface's main thread, the Player MFE utilizes Web Workers. These workers run in a separate thread, ensuring that the real-time match data processing does not cause the UI to freeze or lag. They handle intensive tasks such as receiving, processing, and sending match data to the event bus, all while maintaining smooth playback. Figure 31 shows a screenshot of the implemented Player MFE.

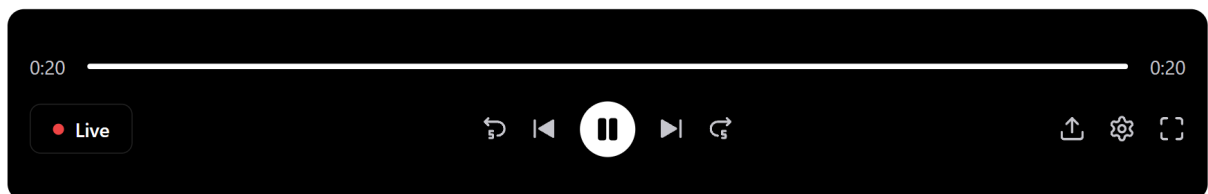


Figure 31: Player micro frontend

A persistent connection is maintained between the Player MFE and its dedicated BFF through WebSockets. This connection allows the Player MFE to receive chunks of samples containing match data, including player positions, ball trajectories, and frame-specific information. The data transmitted via WebSockets is processed by the Player MFE and then broadcast to other micro frontends through the Application Shell's event bus, keeping all the components of the application synchronized with the current match frame.

From a technical perspective, the Player MFE operates with two primary threads: the Main Thread and the Socket Worker. The Main Thread is responsible for instantiating the Web Component, handling user inputs (such as play, pause, and seek), and coordinating match playback on the client side. Meanwhile, the Socket Worker manages the WebSocket connection to the Player BFF, offloading the Main Thread from dealing with the constant stream of data sent

by the backend.

### 4.4.3 Viewer Micro Frontend

The Viewer micro frontend is designed as the 3D rendering engine for the match. Its core responsibility is to render a dynamic 3D match environment, including the robots, ball, and field, by employing the Three.js library – a popular wrapper for WebGL that leverages hardware acceleration through the browser’s GPU. The Viewer MFE listens for updates via the event bus on the App Shell and renders the scene at a fixed rate of 60 frames per second. Due to its computationally intensive nature, the Viewer MFE also utilizes Web Workers to handle heavy tasks in parallel, preventing any freezing or lagging of the user interface. Figure 32 shows a screenshot of the implemented Viewer MFE.

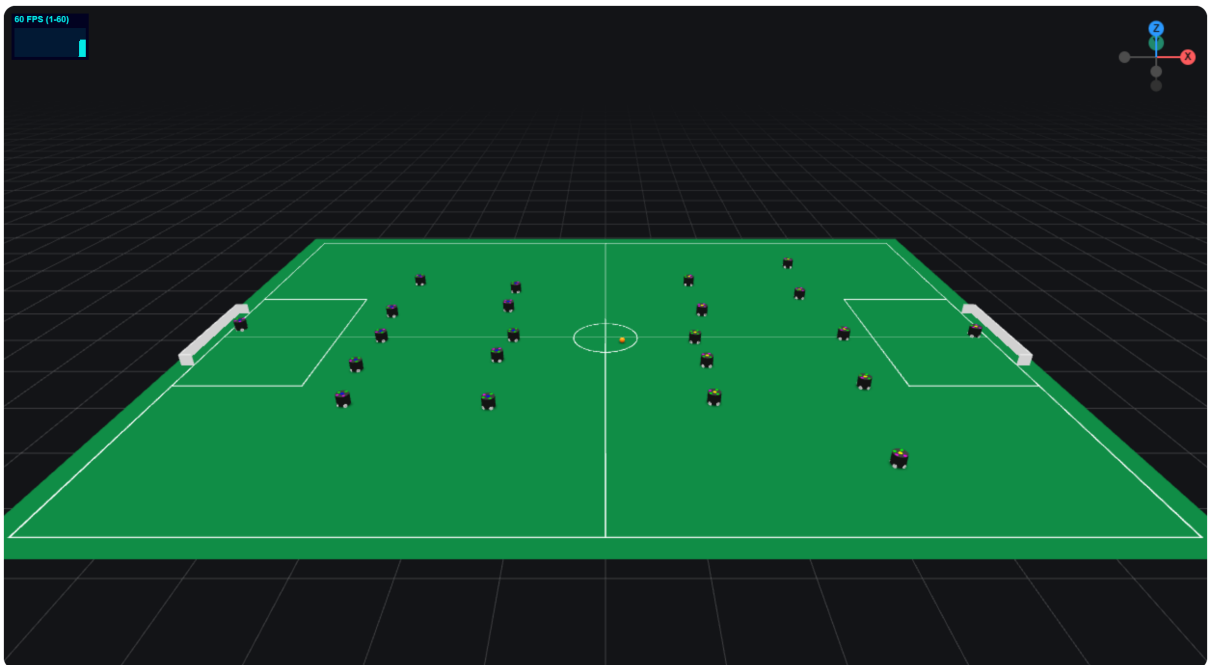


Figure 32: Viewer micro frontend

Integrated into the system as a Web Component, the Viewer MFE operates across three threads: the Main Thread, a Communication Worker, and a Rendering Worker. The Main Thread manages user interactions, such as rotating, zooming, or navigating within the 3D environment, and coordinates the overall rendering process. The Communication Worker listens to the event bus, processes incoming messages, and passes relevant data to the Rendering Worker, which handles the actual 3D rendering using Three.js. This multi-threaded approach ensures efficient real-time rendering while maintaining the system’s performance.

#### 4.4.4 Scoreboard Micro Frontend

The Scoreboard micro frontend plays a important role in the system by providing real-time updates on match information, such as the score and match events. It is designed to display data in a clear and concise format, ensuring that users can easily track the progress of the match. Like other micro frontends in the system, the Scoreboard MFE is developed, tested, and deployed independently, but it is composed dynamically in the browser through the App Shell. Figure 33 shows a screenshot of the implemented Scoreboard MFE.



Figure 33: Scoreboard micro frontend

Receiving its data through the event bus, the Scoreboard MFE also offers interactivity by allowing users to click on specific events, which triggers commands to adjust the match timeline to the selected moment. This functionality enhances the user's ability to review and navigate through key moments in the match.

#### 4.4.5 Parameters Micro Frontend

The Parameters micro frontend serves a specialized function in the system, designed primarily to handle configuration inputs related to the backend connection and system parameters.

Unlike other micro frontends, the Params MFE is implemented as a modal window that does not occupy any space within the grid layout. Its purpose is activated at the start of the system, prompting the user to input the required backend IP address and port, which it uses to establish the necessary connection. Figure 34 shows a screenshot of the implemented Params MFE.

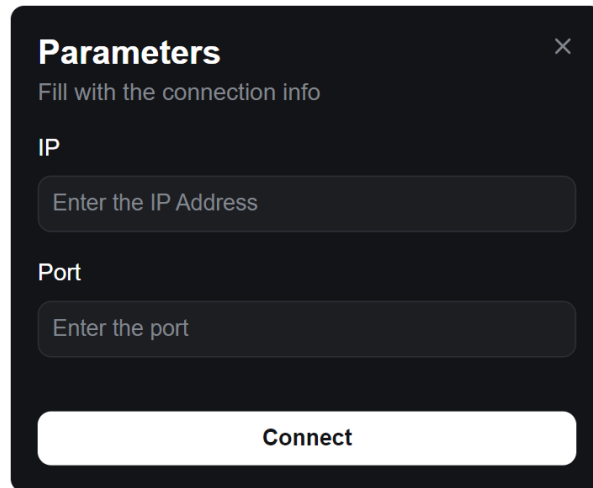
A dark-themed modal window titled "Parameters" with a close button (X) in the top right corner. Below the title is a subtitle "Fill with the connection info". The form contains two input fields: "IP" with a placeholder "Enter the IP Address" and "Port" with a placeholder "Enter the port". At the bottom is a large white button labeled "Connect".

Figure 34: Parameters micro frontend

Once the connection details are provided, the Params MFE communicates with its dedicated BFF, which manages and stores these parameters. The BFF acts as an intermediary between the Params MFE and the various services within the system, ensuring that the input configurations are correctly distributed. Additionally, as the backend services evolve to support new external parameters, the Params MFE will be adapted to send calibration data and other necessary parameters to control specific aspects of the robotic systems.

## 4.5 TESTING STRATEGIES

Testing is essential for ensuring software quality, as it evaluates both fundamental units – such as classes, functions, and components – and their integrations. The micro frontend architecture enhances this testing process by enabling a more granular and manageable approach due to its decoupled nature and adherence to object-oriented principles.

Vitest is utilized as the testing framework, facilitating the evaluation of both unit and integration aspects. Vitest allows for efficient test execution and integrates seamlessly with Vite, making it a suitable choice for the development environment.

In addition to unit and integration testing, end-to-end testing is emphasized to validate the entire user journey across micro frontends. This holistic approach ensures that the application functions cohesively from the user's perspective, thereby enhancing software quality. This topic will be addressed in the next chapter as a means of evaluating functional requirements.

## 4.6 PERFORMANCE OPTIMIZATION

Performance is a critical aspect of user experience and has a significant impact on business metrics. Even when functional requirements are met and good development principles are followed, this does not guarantee optimal performance. To achieve performance goals, it is essential to define them clearly from the outset, starting the journey by establishing a performance budget.

A performance budget acts as a guiding framework that sets clear limits on various performance-related metrics, ensuring that each micro frontend contributes positively to the overall user experience. Specific budgets were established to prevent individual components from adversely affecting page performance. Key metrics included maximum bundle sizes, restrictions on external resources, and compliance with Lighthouse performance metrics.

For this project, the system must satisfy the following performance criteria, formulated with input from the RobôCIn team:

- Each JavaScript bundle must be less than 300 KB when minified and gzipped.
- Micro frontends must load and become interactive in under 5 seconds, even on slow 3G connections.
- Micro frontends must achieve a performance score greater than 80 on Lighthouse audits.
- Each container image for a system component must not exceed 100 MB in size.

## 4.7 CONTINUOUS INTEGRATION AND DELIVERY

Continuous Integration (CI) and Continuous Delivery (CD) are essential practices in modern software development that facilitate rapid and reliable code deployment while maintaining high system quality. Integrating performance budgeting into the CI/CD pipeline is crucial for managing application performance proactively. Open-source tools like `bundlesize` and Lighthouse CI are used to monitor bundle sizes and evaluate key performance metrics, enforcing performance budgets automatically. If predefined thresholds are not met, code integration into the main branches is blocked, preventing potential performance degradation.

In addition to performance monitoring, a suite of tests is integrated into the CI/CD pipeline. This includes unit tests, integration tests, and end-to-end (E2E) tests, all of which validate code changes at different levels. These automated tests are executed during the CI process, ensuring that faulty code is not merged into the main branch if any test fails.

The combination of these practices leads to Continuous Delivery. Once code is pushed to the main branch and successfully passes all tests and performance checks, the CI/CD pipeline triggers the creation of a Docker image, which is then pushed to Docker Hub. This streamlined

process facilitates frequent and reliable deployments, reducing the time between code development and production availability. Automating both the build and deployment processes allows development teams to focus on feature development rather than infrastructure management, enhancing overall productivity and responsiveness to user and community needs.

## 4.8 SYSTEM MONITORING

Maintaining high-quality software requires post-launch performance monitoring to ensure that the application continues to meet technical and business objectives. This process involves continuous tracking of key metrics and collecting real user data to assess how performance fluctuations impact user experience.

In the system, performance data is collected through OpenTelemetry Collectors, which aggregate metrics from multiple sources within the application. These metrics are then visualized using Grafana, a robust monitoring and visualization tool. Grafana's real-time dashboards allow teams to continuously monitor application performance and quickly detect performance regressions or bottlenecks that may arise post-deployment.



# 5

## EVALUATION

This chapter presents the evaluation of functional and non-functional requirements outlined in Section 3.1, along with a comparison against the monolithic baseline to highlight key differences. Limitations encountered during the assessment are also discussed, providing a clear view of the trade-offs and challenges identified throughout the evaluation process.

### 5.1 ASSESSMENT METHODOLOGY

The assessment methodology is divided into two categories: functional and non-functional evaluation. The functional evaluation focuses on ensuring the system behaves as expected from the end-user's perspective. This is achieved through automated end-to-end tests [38], which simulate real-world interactions to verify that the system meets its functional requirements.

Automated end-to-end tests provide a reliable and repeatable means of ensuring functional integrity, reducing the likelihood of human error and increasing the consistency of results. Additionally, they offer scalability by allowing the functional aspects of the system to be tested across a variety of scenarios with minimal manual intervention.

The non-functional evaluation, on the other hand, focuses on assessing the system's quality attributes such as performance, reliability, and maintainability. This is done using a multidimensional approach that combines quantitative and mathematical methods with subjective assessments, providing a broad understanding of the system's operational characteristics.

Some methods utilize static code analysis, defined by the ISO/IEC/IEEE 24765 standard as the process of evaluating a system or component based on its form, structure, content, or documentation [10]. This type of analysis focuses on the code itself rather than its execution, resulting in the derivation of code metrics that can be employed for quantitative assessment of software quality attributes.

To extract these code metrics for the evaluation, various open-source and commercial software tools were used, processing only source code files. Configuration and style files were excluded from the analysis.

## 5.2 FUNCTIONAL EVALUATION

The functional requirements are evaluated through a set of end-to-end tests, each representing a specific use case within the system. Each specification is defined to verify that the system processes inputs correctly, produces expected outputs, and follows the prescribed workflows in accordance with the defined functional specifications.

The tests are implemented using Playwright, an open-source, cross-platform automation tool designed to support all modern web browsers. Playwright enables testing across multiple browsers, including Chromium, Firefox, and WebKit, ensuring that the system is evaluated in diverse environments. Its cross-platform compatibility also allows for tests to be executed on different operating systems.

Table 3 shows the relation between the test name, corresponding use case identifier, and the status of each test.

Test Name	Use Case Identifier	Status
User should watch a live match	UC001	Passed
User should play a match	UC002	Passed
User should pause a match	UC003	Passed
User should control the playback time	UC004	Passed
User should adjust playback speed	UC005	Passed
User should jump to a specific time in the match	UC006	Passed
User should advance/rewind the match frame by frame	UC007	Passed
User should advance/rewind by 5 seconds	UC008	Passed
User should jump to the live broadcast	UC009	Passed
User should lock the view on a robot/ball	UC010	Passed
User should see field coordinates with the cursor	UC011	Passed
User should zoom in on a position of the field	UC012	Passed
User should see details of a robot/ball by clicking	UC013	Passed
User should see overall match information	UC014	Passed
User should see the history of match events	UC015	Passed
User should jump to a specific event in the match	UC016	Passed
User should filter events in the match history	UC017	Passed
User should search for parameters	UC018	Skipped
User should import a parameter file	UC019	Skipped
User should export a parameter file	UC020	Skipped
User should change parameter types and values	UC021	Skipped
User should send parameters to the services	UC022	Passed

Table 3: End-to-end tests

The tests for UC018 to UC021 were skipped due to the current limitations of the backend services, which only support receiving fixed connection parameters and do not yet allow for dynamic parameter changes. Consequently, the implementation of dynamic parameter handling is recommended for future development to enable these tests.

Due to the focus of the requirements gathering process with RobôCIn on identifying

missing features and enhancing existing ones, a direct comparison of use cases with the existing software was not possible. Therefore, the primary goal of this evaluation is to verify that the system functionally aligns with the requirements defined during the analysis, rather than provide a direct comparison.

## 5.3 NON-FUNCTIONAL EVALUATION

### 5.3.1 Performance Efficiency

Performance efficiency quantifies the ability of a system to effectively execute its functions within predetermined time and throughput constraints relative to use of resources, such as CPU, memory, storage, energy, and materials [11].

The evaluation of this quality attribute is conducted through various means, including multiple static analysis tools, web performance assessments, latency measurements, and comparisons against predefined metrics.

#### 5.3.1.1 Resource Utilization

Table 4 summarizes the resource utilization for each component within the system, including lines of code (LOC), bundle sizes, and Docker image sizes. These metrics align with the performance budgets defined in Section 4.5.2.

Component	LOC	Bundle Size (gzip)	Docker Image Size
app-shell	618	7.9 kB	2.81 MB
params-mfe	313	2.6 kB	2.62 MB
player-mfe	1,925	12.7 kB	2.65 MB
scoreboard-mfe	750	6.3 kB	2.66 MB
viewer-mfe	2,419	276 kB	3.4 MB
player-bff	688	-	15.71 MB
<b>TOTAL</b>	6,713	306 kB	29.85 MB

Table 4: Resource utilization for each component

#### 5.3.1.2 Rendering Performance Comparison

To evaluate the rendering performance of the proposed micro frontend approach, a comparative analysis was conducted with a monolithic software developed by RobôCIn, built in C++ using the Qt framework. The monolithic architecture tightly couples robotic control processing with the user interface, leading to high resource consumption, particularly in terms of memory and CPU usage. Although the feature sets between the two systems differ – most notably the use of a 2D environment in the monolithic system compared to a 3D environment

in the micro frontend – this comparison remains valuable for highlighting the performance improvements offered by the new approach.

The tests were performed on an older machine running Ubuntu 22.04, equipped with an Intel® Core™ i5-5200U processor, 8GB DDR3 RAM, a 1TB HDD, and a NVIDIA® GeForce™ 920M GPU with 2GB of VRAM. In terms of evaluation methodology, a 30-second sample was collected for the live match use case, with camera frame rate transmissions ranging from 1 to 120 FPS, generated by Python-based software. The monolithic system received the packets via UDP multicast, while the micro frontend’s BFFs received them through gRPC, both within a local network. Frame rate measurements were taken as the time difference between rendering consecutive frames, using Three.js for the micro frontend and Qt for the monolith. Outliers beyond three standard deviations were discarded, and the average frame rate for each camera configuration was then calculated. The results are presented in Figure 35.

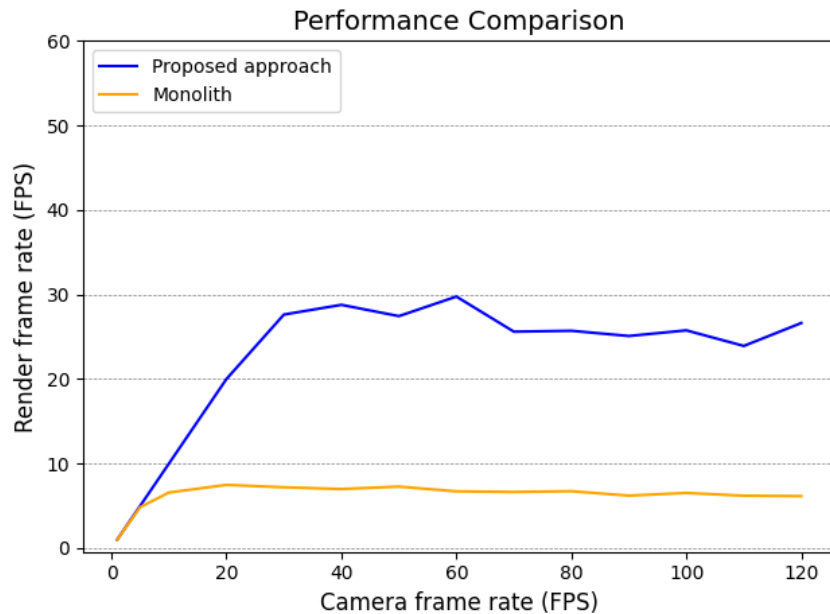


Figure 35: Comparison between monolithic and micro frontend approach

The proposed micro frontend approach consistently delivers rendering performance between 24 FPS and 30 FPS, which aligns with industry standards for web videos, television, and films. This is achieved even on an outdated machine with relatively modest specifications. In contrast, the monolithic application struggles to maintain performance, rendering at only 7 FPS and requiring a significantly more powerful hardware to perform adequately.

Despite being written in a low-level language without communication overhead, the monolith exhibits high RAM and CPU usage, particularly due to the integration of robotic control modules within the Qt-based interface. In contrast, the micro frontends benefit from modern web technologies and WebGL optimizations, which are designed to perform well even on slower devices. Additionally, the communication overhead in the micro frontend architecture differs from traditional microservices, as the components are composed within a single view by the

browser, resulting in minimal communication latency.

An important observation is that the frame rate for the micro frontend approach initially aligns with the camera’s frame rate, reflecting the unique rendered frames. However, the system ensures smooth performance by extrapolating frames when necessary, repeating the last received frame to maintain a stable frame rate and prevent lag or freezing.

### 5.3.1.3 *Rendering Performance in Competition Environment*

Another similar experiment was conducted using a machine from the RoboCup SSL competition environment. The machine was equipped with an Intel® Core™ i7-8565U CPU, 16 GB of RAM, and a 256 GB SSD, running Ubuntu 20.04. Docker containers were used to orchestrate the environment. The evaluation of the micro frontend architecture focused on validating whether the system could consistently achieve a target frame rate of at least 16 frames per second (FPS) – the perceptual lower bound for smooth motion for the human eye [29] – during 3D real-time rendering for live match. Notably, the performance values for the monolithic system were not captured during this assessment.

The micro frontend architecture evaluation yielded an average latency of 17 milliseconds between frame renderings during live streaming, which significantly exceeds the required minimum for smooth motion. The detailed performance results are shown in Table 5.

Metric	Result
Average Latency (ms)	$17.39 \pm 3.47$
Approximate Frame Rate (FPS)	57
Minimum Required Frame Rate (FPS)	16

Table 5: Micro frontend Performance Results

These results satisfy the minimum requirements for real-time rendering, demonstrating that the proposed micro frontend architecture is capable of delivering an innovative approach to robotic GUI applications without compromising the critical robot strategy pipeline.

### 5.3.1.4 *Lighthouse Audit*

Web performance tests were conducted using Google Lighthouse, an open-source tool widely used for auditing web applications. Lighthouse runs a set of performance tests on web pages and generates reports based on various key metrics. These metrics include First Contentful Paint (FCP), Speed Index, Largest Contentful Paint (LCP), Time to Interactive (TTI), Total Blocking Time (TBT), and Cumulative Layout Shift (CLS), all of which are critical indicators of web application responsiveness and user experience. However, it is important to note that these metrics cannot be calculated for applications that do not operate within web browsers, and therefore could not be measured for the monolithic system.

The audits were conducted individually on each micro frontend as well as on the fully composed application. The results of these evaluations are presented in Table 6.

Component	Performance Score	FCP (s)	LCP (s)	TBT (s)	CLS	Speed Index (s)
App Shell	100	0.3	0.3	0	0	0.3
Player MFE	100	0.2	0.2	0	0	0.2
Viewer MFE	100	0.5	N/A	N/A	0	0.5
Scoreboard MFE	100	0.2	0.2	0	0	0.2
Params MFE	100	0.2	0.3	0	0	0.2
Full Application	100	0.4	0.8	0	0	0.4

Table 6: Lighthouse performance metrics

These results align with the performance budget defined in Section 4.5.2, where the target was set to a minimum performance score of 80. All components achieved a perfect performance score of 100. This exceptional performance is attributed to the adoption of native web technologies, such as Web Components and the Broadcast Channel API, which minimize reliance on external libraries. By reducing the bundle size and using efficient browser APIs, the application ensures fast rendering times and smooth user interaction across all micro frontends.

### 5.3.2 Maintainability

Maintainability refers to how easily a system can be modified to adapt to changes or improve its functionality in response to evolving requirements and environmental conditions [11].

Unlike performance efficiency, maintainability is hard to quantify precisely and automatically. It often depends on the team's judgment about the code's structure, readability, and modularity. To provide a more objective framework for evaluating maintainability, standardized metrics, such as the Maintainability Index (MI), have been developed, alongside various tools that implement their own maintainability scores.

The micro frontend architecture inherently promotes maintainability by having smaller, distributed codebases. Each micro frontend operates independently, allowing developers to refactor and improve code without affecting other components. This separation enables teams to reduce the time and effort required for developing compared to monolithic codebases. Consequently, micro frontends offer long-term advantages in maintainability [22]. The following subsections provide further evidence to support this assertion.

#### 5.3.2.1 Maintainability Index

The Maintainability Index (MI) is a widely used metric designed to evaluate the maintainability of software systems. This index combines three traditional code measures – Halstead's

Volume (HV), McCabe’s cyclomatic complexity (CC), and lines of code (LOC) – into a single-value indicator using a polynomial formula [35]. The original formulation of the MI is expressed as follows:

$$MI = 171 - 5.2 \ln(HV) - 0.23 \times CC - 16.2 \ln(LOC) \quad (5.1)$$

The MI was assessed using different software tools tailored to the specific programming languages of each application. For TypeScript files, the Code Health Meter was utilized; for Go files, Go Cyclo; and for C++ files, CppDepend. The results of these assessments are presented in Table 7.

Component	Maintainability Index (MI)
app-shell	131
params-mfe	143
player-mfe	132
scoreboard-mfe	140
viewer-mfe	128
player-bff	130
Full Application	134
Monolith (Baseline)	126

Table 7: Comparison of Maintainability Index between the components

Overall, the results indicate that the micro frontend architecture enhances maintainability compared to traditional monolithic systems. Each micro frontend component achieved a higher Maintainability Index than the monolith, benefiting from optimizations and smaller codebases with reduced technical debt. This improvement contributes to greater adaptability and long-term sustainability of the software. However, the Maintainability Index alone does not capture the full range of benefits and complexities inherent in a distributed system. Thus, the evaluation focuses on demonstrating how the average MI has improved while acknowledging the broader challenges of distributed architectures.

### 5.3.2.2 Embold Analysis

The code was further analyzed using Embold, a commercial software analytics platform designed to assist teams in assessing and enhancing software quality. The results of this analysis are presented in Table 8.

Component	Maintainability	Issues
Full Application	91	3
Monolith (Baseline)	1	146

Table 8: Embold’s maintainability metrics

The significant difference in maintainability scores between the full application and the monolith can be attributed to Embold’s scoring methodology, which evaluates both the number and severity of issues in the code. The monolith’s 146 issues, some of which are critical, result in a poor score of 1. In contrast, the full application, with only three less severe issues, achieves a score of 91. This contrast underscores the micro frontend architecture’s effectiveness in enhancing code quality and maintainability.

### 5.3.2.3 SonarQube Analysis

The maintainability was also evaluated using SonarQube, a widely used open-source static analysis tool in the industry. The results of this analysis are presented in Table 9.

System	Rating	Issues	Debt Ratio
Full Application	A	31	0.1%
Monolith (Baseline)	A	6405	3.8%

Table 9: SonarQube’s maintainability metrics

Despite both systems receiving the same rating of A – calculated based on the estimated time required to fix issues relative to the time already invested in the application – the total number of issues and estimated debt ratio for the new approach is significantly improved. The full application exhibits only 31 issues and a debt ratio of 0.1%, contrasting sharply with the monolith’s 6405 issues and a debt ratio of 3.8%.

### 5.3.3 Reliability

System reliability is a key metric for evaluating software solutions, typically defined by how prone the system is to errors and how it performs its specified functions under defined conditions over a specified period of time [11].

In a broader context, metrics such as Mean Time to Failure (MTTF), Mean Time to Repair (MTTR), and Mean Time Between Failures (MTBF) are often used to assess reliability. However, these require prolonged user monitoring post-release to gather accurate data. In this work, reliability was evaluated through metrics provided by tools like Embold and SonarQube.

Micro frontend architecture naturally reduces the risk of system-wide failures by isolating each micro frontend as an independent application, ensuring that an error in one component does



not impact the rest of the system. In contrast, monolithic architectures are more susceptible to reliability issues, where a critical error can affect the entire system. The Application Shell, responsible for loading and managing the micro frontends, remains a potential single point of failure, but the risk is minimized as it handles only routing and system messages.

#### 5.3.3.1 *Embold Analysis*

The results of the Embold analysis for reliability are presented in Table 10.

<b>Application</b>	<b>Reliability</b>	<b>Issues</b>
Full Application	100	0
Monolith (Baseline)	91	1

Table 10: Embold's reliability metrics

The Embold analysis shows the full micro frontend application achieved a perfect reliability score of 100 with zero issues, highlighting its robustness. The monolith, although still rated relatively high with a score of 91, had one issue that could potentially affect system stability.

#### 5.3.3.2 *SonarQube Analysis*

The results of the SonarQube analysis for reliability are presented in Table 11.

<b>System</b>	<b>Rating</b>	<b>Issues</b>
Full Application	A	0
Monolith (Baseline)	C	1

Table 11: SonarQube's reliability metrics

The SonarQube analysis shows the micro frontend application received an "A" rating with zero issues, demonstrating strong reliability. In comparison, the monolith earned a "C" rating with one several issue, reflecting a higher likelihood of system failures.

### 5.3.4 **Portability**

Portability assesses how well a product can adapt to changes in its requirements, usage contexts, or system environments – such as hardware, software, or operational settings.

In this work, portability will be evaluated using Embold's portability metric, complemented by a subjective comparison of subcharacteristics. This multidimensional approach aims to provide a deeper understanding of this quality attribute by incorporating different perspectives.

#### 5.3.4.1 Embold Analysis

The results of the Embold analysis for portability are presented in Table 12.

Application	Portability	Issues
Full Application	100	0
Monolith (Baseline)	83	2

Table 12: Embold's portability metrics

The Embold analysis indicates that the micro frontend application achieved a perfect portability score of 100 with no issues, whereas the monolithic baseline scored 83 with two reported issues. This further emphasizes the enhanced portability of the micro frontend approach compared to the monolithic system.

#### 5.3.4.2 Subcharacteristics Comparison

To evaluate the suitability of each measure, a Likert scale is employed, with the following ratings: Very Good, Good, Neutral, Poor, and Very Poor.

Table 13 summarizes the quality attribute subcharacteristics along with their corresponding ratings for both the micro frontend architecture and the monolithic baseline, following the approach of Karnouskos [17].

Characteristic	Subcharacteristic	Micro frontend Rating	Monolithic Rating
Adaptability	Hardware Environmental Adaptability	Good	Neutral
	System Software Environmental Adaptability	Very Good	Poor
	Operational Environment Adaptability	Very Good	Neutral
Installability	Installation Time Efficiency	Very Good	Poor
	Ease of Installation	Very Good	Very Poor
Replaceability	Usage Similarity	Very Good	Good
	Product Quality Equivalence	Very Good	Good
	Functional Inclusiveness	Good	Good
	Data Reusability/Import Capability	Very Good	Neutral

Table 13: Characteristics and subcharacteristics of micro frontend and monolithic architectures

The ratings reflect the adaptability and ease of use associated with each architecture. The micro frontend architecture scores "Very Good" and "Good" across all categories due to its browser-based nature, which enhances usability, modularity, and installation efficiency. In contrast, the monolithic architecture, which is implemented in C++ and operates only in Linux

environments, receives mostly neutral to poor ratings. Although it is somewhat modular, its adaptability to different hardware and operational environments is limited, resulting in lower ratings across the portability subcharacteristics.

# 6

## RELATED WORK

This chapter presents an in-depth review of the existing literature on robotic graphical user interfaces and micro frontends, with a particular emphasis on the design of micro frontend architectures in robotic systems. The main goal of this review is to identify the current state of the art in micro frontend architecture design and to compare it with the contributions of this work. The examination focuses on the methodologies employed to delineate boundaries for identifying microservices, the evolution of robotic interfaces towards browser-based GUIs, and the practical applications of micro frontends within robotic systems.

### 6.1 MICROSERVICES IDENTIFICATION AND BOUNDARIES

The architectural evolution of multi-robot systems has mirrored broader trends in software design, moving from traditional monolithic approaches to more modular and scalable solutions like microservices. Defining clear service boundaries is a critical aspect of microservices architecture, and micro frontends – as a specialized case of microservices for the frontend – face similar challenges. In this context, existing work on identifying and delineating service boundaries in microservices provides a valuable foundation for designing micro frontends.

Zimmermann [44] offers a detailed review of microservices principles, positioning them as an evolution of Service-Oriented Architecture. He emphasizes fine-grained interfaces and business-driven development in his approach to decomposing monolithic systems into microservices.

Steinegger et al. [37] explore boundary identification by using bounded contexts to guide the decomposition of microservices. Through a case study on a thesis management application, they demonstrate the practical application of DDD concepts and emphasize the importance of defining domain boundaries. They also explore the reuse of functionalities, especially in identity and access management, and systematic methods for deriving web APIs that prioritize quality aspects like evolvability.

Similarly, Rademacher et al. [30] contribute to this field by proposing a UML profile designed to improve the modeling of microservice systems through Domain-Driven Design, emphasizing the role of bounded contexts in identifying microservice candidates. Initial strategies

for mapping profile-based domain models to microservices are also discussed, guided by findings from a literature survey of 92 domain models.

The discussed approaches incorporate DDD strategies to define microservice boundaries, utilizing tools like UML, informal diagrams, and DDD design patterns. However, they diverge from this work by viewing the frontend only as a layer within the architecture, without addressing the concept of micro frontends. They also neglect a variety of object-oriented modeling techniques, such as use case diagrams, sequence diagrams, and component diagrams, and they do not engage in conducting profound requirements analysis to support their architectural decisions. Additionally, the absence of evaluation methodologies for the proposed architectures limits their practical applicability and assessment.

## 6.2 EVOLUTION OF ROBOTIC INTERFACES TO BROWSER GUIS

Robotic systems have evolved significantly over the past few decades, transitioning from hardware-dependent, specialized interfaces to more accessible, web-based graphical user interfaces. This shift has been primarily driven by the need for improved user accessibility, ease of control, and the integration of multi-modal interaction methods, particularly for non-expert users. Early robotic interfaces were highly technical, requiring users to interact with command-line systems or specialized hardware. Systems like the Lego Mindstorms and Nomad 200, as discussed by Paolini and Lee [27], represent initial attempts to create educational robotic platforms that provided basic control capabilities through low-level interfaces like direct sensor inputs and microcontroller commands.

As the field matured, the demand for more intuitive interfaces grew, leading to the development of graphical user interfaces. These early GUIs allowed users to control robots through visual feedback, utilizing maps, sensor data displays, and simple input methods like buttons and sliders. Rajapaksha et al. [31] work on GUIs for Robot Operating System (ROS) environments marked a significant advancement in making robotic control systems more user-friendly. These interfaces bridged the gap between complex robotic control algorithms and user interaction by offering visual, context-aware controls, reducing the cognitive load on users and making robotic programming more accessible.

The advent of browser-based GUIs further revolutionized robotic interfaces, enabling users to interact with robots from any device via standard web browsers, regardless of the underlying operating system or hardware. Technologies like HTML5 and JavaScript facilitated the creation of responsive, platform-independent interfaces. Di Nuovo et al. [7] work on multi-modal, web-based interfaces for the Robot-Era project demonstrated the flexibility and accessibility of such systems, particularly in applications like elder care, where user experience is critical. Similarly, Rajapaksha et al. [31] ROS-based web interfaces allowed for remote robot control and teleoperation through a browser, significantly lowering the barriers to entry for robotics interaction.

In more complex scenarios, such as urban search and rescue (USAR), browser-based GUIs have proven effective in controlling multiple robots simultaneously. Niroui et al. [26] introduced a multi-robot control interface designed for such applications, offering real-time video, sensor data, and mapping tools to enhance situational awareness and reduce operator workload. This scalability further underscores the role of browser-based GUIs in improving human-robot interaction in both single and multi-robot contexts.

Despite the demonstrated benefits of browser-based GUIs in robotic interfaces, these studies lack a rigorous architectural modeling approach for analyzing the coupling associated with monolithic architectures compared to distributed systems. An in-depth evaluation of the architectural implications and trade-offs of these design choices is necessary to fully understand their impact on system flexibility, scalability, and maintainability.

### 6.3 MICRO FRONTENDS IN ROBOTIC SYSTEMS

While micro frontends have gained attention in software engineering, research focused on their application in distributed frontend systems, particularly within multi-robot systems, remains limited. The novelty of this architecture means there is little dedicated literature on the subject. However, research from related fields, such as the Internet of Things (IoT), Human-Machine Interfaces (HMI), and general robotics applications, provides relevant examples and potential solutions for modularity and scalability issues.

Mena et al. [21] present a component-based Progressive Web Application (PWA) for geospatial IoT data acquisition. Their work highlights the benefits of MFEs for dynamic user interface construction and independent development of visual components.

Shakil et al. [34] propose a modular architecture for industrial HMI using MFEs. They demonstrate how engineers can build HMIs from independent MFEs, where each component encapsulates the entire development lifecycle, from user interface design to data acquisition.

Schäffer et al. [33] investigate the use of microservices and MFEs in a web-based configurator for robotic automation tools using a combination of microservices and MFEs architectures. Their prototype showcases how these architectural approaches simplify development and deployment through a divide-and-conquer approach.

The reviewed works showcase real-world applications of micro frontend architectures in robotic contexts, demonstrating that, despite its novelty, this architectural approach can be successfully implemented in complex environments. However, these studies lack a deep exploration of software modeling techniques and do not address the challenges of coordinating multiple robots with low-latency communication.

# 7

## CONCLUSION

This work presents a novel approach for developing graphical user interfaces in multi-robot systems through the modeling, implementation and evaluation of a micro frontend architecture based on Domain-Driven Design and Object-Oriented Modeling principles. The proposed methodology covers the entire software development lifecycle, including requirements gathering, architectural modeling, implementation, deployment, and the formulation of CI/CD strategies. An extensive evaluation was conducted following the ISO/IEC 25010 standard to assess multiple software quality attributes.

All proposed requirements were successfully validated, demonstrating the architecture's robustness through a multidimensional evaluation utilizing both quantitative metrics and qualitative analysis. The system demonstrated exceptional performance, achieving perfect scores in Lighthouse audits and maintaining bundle sizes consistently below the defined limits. In terms of rendering performance, the micro frontend-based GUI maintained frame rates between 24 FPS and 30 FPS on lower-spec hardware, significantly outperforming the monolithic architecture, which struggled to achieve 7 FPS. Moreover, in live streaming tests conducted in the context of the RoboCup Small Size League, the micro frontend architecture achieved an average latency of 17 milliseconds, corresponding to 57 FPS, highlighting its real-time responsiveness.

The system also demonstrated high maintainability, as shown through code analysis tools such as Embold and SonarQube, outperforming the monolithic system. Improvements were also observed in reliability and portability, underscoring the architecture's focus on scalability and ease of deployment across multiple platforms, thereby establishing it as an adaptable solution for multi-robot environments.

Despite these successes, several challenges and limitations emerged. A notable challenge identified during this research was the lack of detailed documentation addressing software quality attributes specific to micro frontends. Existing literature predominantly focuses on implementation aspects and high-level discussions of benefits and trade-offs, with little attention given to quantitative analysis or performance metrics. The relative novelty of micro frontends within the broader field of software architecture has limited the availability of scientific studies that thoroughly investigate this topic, resulting in a scarcity of in-depth analyses.

The comparative analysis of the micro frontend and monolithic architectures also pre-

sented challenges due to differing use cases, as the monolithic system tightly couples control logic with the user interface. Although the static analysis tools provided useful metrics, they only offer approximate representations of quality attributes and do not fully capture the real-world benefits of modularity, reduced code complexity, and team ownership associated with micro frontend approaches.

Moreover, the use of different programming languages for each architecture introduced additional complexity in comparing metrics like code efficiency and verbosity. More meaningful metrics could be obtained after deployment in production environments, considering interactions with real users and developer contributions to the system.

Future work will address these limitations. Comparative studies on developer productivity between monolithic and micro frontend architectures could provide deeper understanding of development efficiency. Additionally, post-release evaluations of reliability and maintainability using metrics such as Mean Time to Failure (MTTF) and Mean Time Between Failures (MTBF) will offer a clearer understanding of long-term system performance. Further enhancement of the system's functional capabilities may be achieved through the introduction of dynamic parameters within the Parameters MFE, integrated with the backend systems. Moreover, proposing the system for official use in the RoboCup SSL competition and encouraging community-driven development could expand the platform's extensibility, enabling the integration of additional micro frontends and plugins to support a broader range of applications.

Addressing these areas will contribute to the development of a more robust, scalable, and adaptable solution for multi-robot systems, potentially driving further advancements in human-robot interaction and modular software design.



## REFERENCES

- [1] Afzal, A., Goues, C. L., Hilton, M., & Timperley, C. S. (2020). A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 96–107.
- [2] Aroraa, G. K., Kale, L., & Manish, K. (2017). *Building Microservices with .NET Core*. Packt Publishing.
- [3] Blaha, M. & Rumbaugh, J. (2004). *Object-oriented modeling and design with UML*. Pearson, Upper Saddle River, NJ, 2 edition.
- [4] Burkhard, H., Duhaut, D., Fujita, M., Lima, P., Murphy, R., & Rojas, R. (2002). The road to robocup 2050. *IEEE Robotics Automation Magazine*, 9(2):31–38.
- [5] Chung, L., Nixon, B., Yu, E., & Mylopoulos, J. (2012). *Non-Functional Requirements in Software Engineering*. International Series in Software Engineering. Springer, New York, NY.
- [6] Considine, D. M. & Considine, G. D. (1986). *Robot Technology Fundamentals*, 262–320. Springer US, Boston, MA.
- [7] Di Nuovo, A., Broz, F., Belpaeme, T., Cangelosi, A., Cavallo, F., Esposito, R., & Dario, P. (2014). A web based multi-modal interface for elderly users of the robot-era multi-robot services. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2186–2191.
- [8] Duca, R. N. (2024). *Coordination and control of multi-robot systems*. PhD thesis, University of Malta.
- [9] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Educational, Boston, MA.
- [10] for Standardization, I. O. (2010). ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, 1–418.
- [11] for Standardization, I. O. (2016). ISO/IEC 25010, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Standard ISO/IEC 25010:2011, International Organization for Standardization.
- [12] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston, MA.
- [13] Geers, M. (2020). *Micro frontends in action*. Manning Publications, New York, NY.
- [14] Jones, C., Shell, D., Matari, M., & Gerkey, B. (2004). Principled approaches to the design of multi-robot systems. *Proc. of the Workshop on Networked Robotics, IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*.
- [15] Kaluža, M. & Vukelić, B. (2018). Comparison of front-end frameworks for web applications development. *Zbornik Veleučilišta u Rijeci*, 6(1):261–282.

- 
- [16] Kan, S. H., Basili, V. R., & Shapiro, L. N. (1994). Software quality: An overview from the perspective of total quality management. *IBM Systems Journal*, 33(1):4–19.
- [17] Karnouskos, S., Sinha, R., Leitão, P., Ribeiro, L., & Strasser, T. I. (2018). The applicability of ISO/IEC 25023 measures to the integration of agents and automation systems. In *IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, 2927–2934.
- [18] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., & Osawa, E. (1998). Robocup: the robot world cup initiative. *Proceedings of the International Conference on Autonomous Agents*.
- [19] Lethbridge (2002). *Object-oriented software engineering: Practical software development*. McGraw Hill Higher Education, Maidenhead, England.
- [20] Mahajan, R. & Shneiderman, B. (1997). Visual and textual consistency checking tools for graphical user interfaces. *IEEE Transactions on Software Engineering*, 23(11):722–735.
- [21] Mena, M., Corral, A., Iribarne, L., & Criado, J. (2019). A progressive web application based on microservices combining geospatial data and the internet of things. *IEEE Access*, 7:104577–104590.
- [22] Mezzalana, L. (2021). *Building micro-frontends: Scaling Teams and Projects, Empowering Developers*. O'Reilly Media, Sebastopol, CA.
- [23] Muller-Brockmann, J. (1999). *Rastersysteme für die visuelle Gestaltung - Grid systems in Graphic Design*. Niggli Verlag, Sulgen, Switzerland.
- [24] Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Sebastopol, CA.
- [25] Newman, S. (2019). *Monolith to microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Sebastopol, CA.
- [26] Niroui, F., Liu, Y., Bichay, R., Barker, E., Elchami, C., Gillett, O., Ficocelli, M., & Nejat, G. (2016). A graphical user interface for multi-robot control in urban search and rescue applications. In *2016 IEEE International Symposium on Robotics and Intelligent Sensors (IRIS)*, 217–222.
- [27] Paolini, C. & Lee, G. K. (2012). A web-based user interface for a mobile robotic system. In *2012 IEEE 13th International Conference on Information Reuse Integration (IRI)*, 45–50.
- [28] Prewett, M. S., Johnson, R. C., Saboe, K. N., Elliott, L. R., & Coover, M. D. (2010). Managing workload in human–robot interaction: A review of empirical studies. *Computers in Human Behavior*, 26(5):840–856. Advancing Educational Research on Computer-supported Collaborative Learning (CSCL) through the use of gStudy CSCL Tools.
- [29] Pueo, B. (2016). High speed cameras for motion analysis in sports science. *J. Hum. Sport Exerc.*, 11(1).
- [30] Rademacher, F., Sachweh, S., & Zündorf, A. (2018). Towards a UML profile for domain-driven design of microservice architectures. In Cerone, A. & Roveri, M., editors, *Software Engineering and Formal Methods*, 230–245.

- 
- [31] Rajapaksha, D. D., Mohamed Nuhuman, M. N., Gunawardhana, S. D., Sivalingam, A., Mohamed Hassan, M. N., Rajapaksha, S., & Jayawardena, C. (2021). Web based user-friendly graphical interface to control robots with ROS environment. In *2021 6th International Conference on Information Technology Research (ICITR)*, 1–6.
  - [32] Rozanski, N. (2011). *Software systems architecture: Working with stakeholders using viewpoints and perspectives*.
  - [33] Schäffer, E., Mayr, A., Fuchs, J., Sjarov, M., Vorndran, J., & Franke, J. (2019). Microservice-based architecture for engineering tools enabling a collaborative multi-user configuration of robot-based automation solutions. *Procedia CIRP*, 86:86–91. 7th CIRP Global Web Conference – Towards shifted production value stream patterns through inference of data, models, and technology (CIRPe 2019).
  - [34] Shakil, M. & Zoitl, A. (2020). Towards a modular architecture for industrial hmis. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 1:1267–1270.
  - [35] Sjøberg, D. I. K., Anda, B., & Mockus, A. (2012). Questioning software maintenance metrics: A comparative case study. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 107–110.
  - [36] Sommerville, I. (2010). *Software Engineering*. Pearson, Upper Saddle River, NJ, 9 edition.
  - [37] Steinegger, R., Giessler, P., Hippchen, B., & Abeck, S. (2017). Overview of a domain-driven design approach to build microservice-based applications.
  - [38] Tsai, W., Bai, X., Paul, R., Shao, W., & Agarwal, V. (2001). End-to-end integration testing design. In *25th Annual International Computer Software and Applications Conference. COMPSAC 2001*, 166–171.
  - [39] Vepsäläinen, J., Hevery, M., & Vuorimaa, P. (2024). Resumability—a new primitive for developing web applications. *IEEE Access*, 12:9038–9046.
  - [40] Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Educational, Boston, MA.
  - [41] Wang, Y. & de Silva, C. W. (2008). A machine-learning approach to multi-robot coordination. *Engineering Applications of Artificial Intelligence*, 21(3):470–484.
  - [42] Wiegers, K. (2005). *More about software requirements: Thorny issues and practical advice*. Microsoft Press, Redmond, WA.
  - [43] Yacoub, S. & Ammar, H. (2003). *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison Wesley, Boston, MA.
  - [44] Zimmermann, O. (2016). Microservices tenets: Agile approach to service development and deployment. *Computer Science - Research and Development*, 32.