Micro Frontend Architecture for Robotic Systems: A Systematic Approach with Design Rationale

Uanderson Silva, Matheus Andrade, José Cruz, Andresa Silva, Augusto Sampaio, Kiev Gama
Universidade Federal de Pernambuco (UFPE)
Recife, Brazil
{urfs, mvtna, jvsc, aas10, acas, kiev}@cin.ufpe.br

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's 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 in robotic 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 regarding the system design. 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 and reliability.**

Index Terms—**micro frontends, web development, software architecture, graphical user interfaces, robotic systems**

I. Introduction

Traditionally, frontend applications primarily served as simple data presentation layers with minimal business logic, leaving most system complexity to be managed by the backend [1]. However, in recent years, the complexity of modern applications has grown at an extraordinary pace, especially in domains that demand sophisticated frontends integrated with complex backend systems. In this scenario, robotic systems represent one such domain where frontends 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 robotic systems evolve, traditional monolithic approaches to Graphical User Interfaces (GUIs) development reveal their limitations [2], resulting in large codebases that are difficult to maintain and slower development cycles. To mitigate these issues, micro frontends (MFEs) have emerged as an alternative solution to the challenges imposed by monolithic architectures by extending the principles of microservices – widely used in backend development – to the frontend [3], 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.

To support the design of modular and scalable software architectures, such as microservices, Object-Oriented Modeling (OOM) and Domain-Driven Design (DDD) are key tools used during the process [4]. OOM represents software systems as collections of discrete objects that encapsulate both data structures and behaviors [5], often using Unified Modeling Language (UML) as a standardized notation. DDD, on the other hand, focuses on modeling systems through a deep understanding of the core domain, promoting collaboration between developers and domain experts. Together, these approaches complement each other, supporting architectures that are both technically robust and domain-aligned.

Therefore, this work proposes a systematic approach for developing a micro frontend architecture focused on robotic systems, grounded in a hybrid approach that applies DDD and OOM principles. 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 robotic systems, with an emphasis on GUIs used in robotic applications, where interactive simulations and real-time data processing are critical.

The remaining sections are structured as follows: Section II reviews related work on frontend architecture and robotic systems. Section III presents the solution design, exploring the architectural decisions. Section IV describes the implementation of the robotic system GUI using the proposed micro frontend architecture in the RoboCup Small Size League. Section V evaluates the solution through experiments, results, and comparisons of architectural approaches. Finally, Section VI concludes with future research directions and potential applications beyond robotics.

II. Background and Related Work

Robotic systems are rapidly expanding across multiple environments [6], driving the development of increasingly complex systems and creating a growing demand for flexible, scalable software architectures, which leads to significant innovation in this area. For instance, Georgiades *et al.* [7] propose a microservice framework for multi-drone autonomous systems, emphasizing fault tolerance and expandability. Furthermore, Zhou *et al.* [8] demonstrate how a virtual microservices model simplifies task composition and control in robotic swarms by abstracting individual actions into high-level virtual services, as seen in a rescue mission scenario. However, while these studies explore novelties in robotic systems, they largely overlook frontend and user interface aspects.

This gap highlights an opportunity to explore new architectures in robotics GUIs. Early GUIs enabled basic robot control through visual feedback, utilizing maps, sensor displays, and simple inputs like buttons and sliders. Rajapaksha *et al.* [9] advanced these interfaces in Robot Operating System (ROS) environments, making robot control more accessible by bridging complex algorithms and user interaction. The rise of browser-based GUIs further transformed robotic interfaces, enabling control from any device via web browsers and platform-independent technologies like HTML5 and JavaScript. Di Nuovo *et al.* [10] highlighted the accessibility of these GUIs in elder care. In complex tasks like urban search and rescue (USAR), Niroui *et al.* [11] demonstrated multi-robot control through browser-based interfaces, enhancing situational awareness with real-time video and data. However, these studies lack an in-depth architectural analysis to compare monolithic and distributed approaches, leaving the trade-offs in flexibility, scalability, and maintainability underexplored.

Furthermore, while micro frontends have gained attention in software engineering, research focused on their application in distributed frontend systems, particularly within robotic systems, remains limited. The novelty of this architecture has resulted in a scarcity of dedicated literature; however, research from related fields provides relevant examples and potential solutions for modularity and scalability issues. Mena *et al.* [12] present a component-based Progressive Web Application (PWA) for geospatial IoT data acquisition, highlighting the benefits of MFEs for dynamic user interface construction and independent development of visual components. Similarly, Shakil *et al.* [13] propose a modular architecture for industrial HMI using MFEs, demonstrating how engineers can build Human-Machine Interfaces from independent MFEs, with each component encapsulating the entire development lifecycle—from user interface design to data acquisition. Schäffer *et al.* [14] investigate microservices and MFEs in a web-based configurator for robotic automation tools, showcasing how these architectural approaches simplify development and deployment through a divide-and-conquer approach. These works provide real-world applications of micro frontend architectures in robotic contexts, showing that, despite its novelty, this architectural approach can be successfully implemented in complex environments. However, they lack a rigorous architectural modeling approach, and the absence of evaluation methodologies for the proposed architectures limits their practical applicability and assessment.

III. Solution Design

This section introduces a systematic framework for designing a micro frontend architecture for the graphical user interfaces of robotic systems. The proposed approach combines use case modeling and domain analysis to translate abstract system requirements into a maintainable and scalable architecture aligned with domain logic, grounded in the principles of Object-Oriented Modeling and Domain-Driven Design. A diagram illustrating the proposed software lifecycle model is presented in Figure 1, with each phase described in the following subsections.

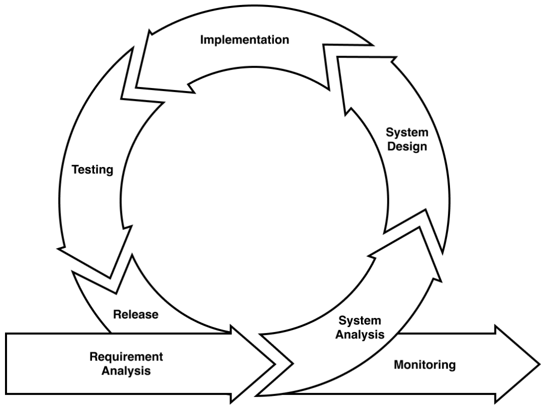

Fig. 1. Proposed software lifecycle model

A. Requirement Analysis

In the requirements analysis phase, stakeholders – including developers, engineers, and end users – are consulted to gather requirements, constraints, and pain points through discovery sessions. The collected information is filtered and refined into a requirements document, which guides all subsequent analysis and design phases.

Requirements are then categorized into functional requirements, non-functional requirements, and design and implementation constraints.

1) Functional Requirements: Functional requirements define the functionalities a system must provide and its expected behavior in response to particular inputs or scenarios [15]. Although these requirements are typically defined at a high level and refined throughout development, this approach captures them directly as use cases.

To represent the interactions between the system and external entities in each use case, a UML use case diagram is utilized. While experienced domain experts may sometimes bypass this step by identifying system subdomain boundaries directly, this artifact is necessary for subsequent project phases.

2) Non-Functional Requirements: Non-functional requirements, or quality attributes [16], place constraints on how the system should perform its functionalities [15]. Quality models provide a structured approach for selecting and evaluating these requirements. Considering this study's focus on the impact of adopting a distributed micro-frontends architecture compared to a traditional monolithic approach, the analysis emphasizes a specific subset of quality attributes from the ISO/IEC 25010:2011 product quality model [17] that are directly influenced by this architectural shift: performance efficiency, maintainability, and reliability.

3) Design Constraints: Design constraints limit developers' choices for valid reasons [18]. When designing large modular systems, low coupling and high cohesion are basic design principles that promote scalability, maintainability, and architectural integrity [19]. Additionally, to support the applicability of the architecture across diverse robotic systems, the proposed 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 without affecting the user interface.

4) Implementation Constraints: Implementation constraints address practical considerations in system construction, influenced by the team's expertise and the selected technology stack. These constraints guide coding and deployment practices to ensure the system remains consistent, maintainable, and adaptable. Based on the micro frontend principles [2], the proposed 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.

B. System Analysis

In software engineering, models simplify complex problems and provide a clearer understanding of them. These models are particularly powerful in the context of OOM and DDD, where they closely reflect the real-world domain (the robotic environment).

During the system analysis phase, models are created to guide the following 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.

1) Domain Analysis: 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 requirements and use cases to identify potential entities, excluding irrelevant elements – which are handled later in the design phase.

Exploratory domain models, expressed as abstract UML class diagrams, are constructed based on the identified entities. 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 [20].

As the domain model evolves, subdomains and their interrelationships emerge, bridging the gap between business understanding and technical implementation. A key aspect is the development of a Ubiquitous Language – a shared vocabulary derived from domain experts' jargon – refined for clarity to ensure alignment among all stakeholders, from developers to domain experts, have aligned discussions that translate into code [19].

2) Bounded Contexts: A bounded context in DDD defines the scope where a particular model is applicable. Within this context, the model remains coherent and focused on its domain, without considering relevance outside its boundaries. Different contexts may adopt distinct models, terminology, and rules, each reflecting their own version of the Ubiquitous Language [19].

Identifying bounded contexts involves functional decomposition, by grouping use cases with similar goals and domain concepts. This process reveals natural boundaries within the system, organizing related functionalities into distinct contexts where a consistent domain model can be applied.

Once the bounded contexts are identified, the next step is to distribute the entities from the exploratory domain model to the appropriate contexts. Each context should encapsulate the entities that best represent its core concepts, aligning with its specific requirements and preserving well-defined boundaries to minimize overlap.

As each context is refined, new entities may emerge, and existing ones may be renamed or redefined. In some cases, entities may only represent a partial view of the whole. The result is not a single model but a collection of models, each with 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, some contexts may overlap significantly and should be merged, while others may emerge as the domain becomes clearer. If a context becomes overly complex, it may need to be divided into smaller, more manageable parts. These adjustments are an inherent and necessary part of refining the system's architecture.

C. System Design

In the system design phase, concrete models are developed to define the micro frontend architecture, building on the knowledge from the analysis phase. Unlike analysis models, which capture domain concepts, design models translate these concepts into architectural solutions. These models include detailed representations, such as component diagrams, interaction diagrams, and design patterns that delineate the implementation of each system component.

1) Micro Frontends Definition: The definition and implementation of micro frontends is influenced by several factors, including the size of the development team and their familiarity with this architectural paradigm. Even small teams can benefit from a distributed frontend architecture, improving scalability, resilience, and maintainability, especially in complex domains like robotics graphical user interfaces.

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. As the system evolves, this initial alignment of MFEs to bounded contexts should be iteratively refined. Some MFEs may need to be split into smaller components, while others could be consolidated to reduce complexity or improve performance.

To support the micro frontend architecture, several architectural design patterns are adopted. These patterns, as described by the Gang of Four [21], offer reusable solutions to common design challenges. While patterns like Observer are essential for reactivity within individual micro frontends, three architecturally significant patterns are focused on: Domain Events, Backends for Frontends (BFFs), and the Application Shell.

- **Domain Events:** A DDD pattern [22] that enables autonomous services to operate independently by using asynchronous messaging instead of direct calls. Originating from domain entities, these events align with the Ubiquitous Language and are mapped to new structures when crossing boundaries to maintain their meaning. This pattern is well-suited for web frontend architectures, which are inherently event-driven and reactive, and is used to model significant communications within and across micro frontends.
- **Backends for Frontends:** A consumer-focused API design pattern where each BFF acts as an intermediary, providing data in the format required by its specific client. This allows core backend services to remain generic and reusable while improving frontend-backend interaction. BFFs can also map backend domains to frontend domains, maintaining architectural boundaries and promoting flexibility [3]. Essentially, BFFs serve as specialized backends for user experiences, protecting frontends from backend complexities and exposing only the necessary data and services, acting as the anti-corruption layer in DDD [19].
- **Application Shell:** An orchestrator for micro frontends that is loaded first and remains active throughout the user's session [3]. It dynamically loads and unloads micro frontends based on navigation, optimizing performance by retrieving only relevant components. As a mediator [21], the Application Shell coordinates communication between micro frontends, ensuring their decoupling and enabling them to focus on their respective tasks.

2) Service Interaction Model: Interaction diagrams are important tools for modeling the dynamic behavior of software systems, visually representing the steps involved in executing a use case or any specific functionality. Collectively, these steps form what is known as an interaction [20]. 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 help refine the system's architecture by making the flow of interactions explicit and identifying potential inefficiencies. Figure 2 demonstrates a Service Interaction Model.

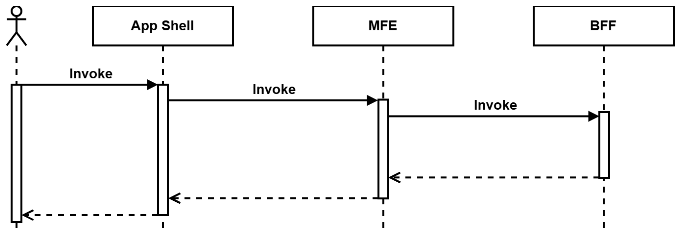

Fig. 2. Service interaction model

3) Service Component Model: As the system architecture is refined, high-level design decisions must be translated 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 representation of the system's major building blocks and their connections, serving as a blueprint for both developers and stakeholders.

Figure 3 presents a 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 monolithic or distributed.

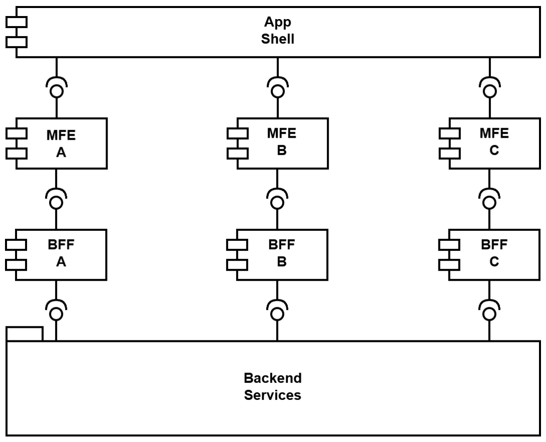

Fig. 3. Service component model

D. Implementation

This phase bridges the gap between high-level architectural concepts and actual implementation, translating the design into actual code. Even with a defined architecture, implementing each micro frontend requires careful decisions about communication, coordination, and composition. These choices establish how components interact, exchange data, and contribute to a cohesive user experience. The goal is to balance the autonomy of individual MFEs with the need to deliver a unified experience across the system's functionalities.

Key considerations include selecting client-side or server-side composition, choosing an orchestrator, and defining inter-component communication methods. These decisions directly influence performance, scalability, and responsiveness, determining where and how views are constructed and managed.

The orchestrator controls routing and component interactions, coordinating the MFEs into a seamless experience. Each choice supports a system that effectively blends flexibility with consistency, resulting in a robust, cohesive implementation.

E. Testing

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.

Continuous integration (CI) and continuous delivery (CD) are essential to streamline testing, supporting rapid and reliable code deployment while ensuring high-quality standards. The CI/CD pipeline integrates comprehensive testing, including unit, integration, and end-to-end (E2E) tests, to validate the software at all levels.

F. Release

Release management in the proposed micro frontend architecture uses continuous delivery techniques to allow seamless deployment of updates across independent components. Once all tests and performance checks are successfully completed within the CI/CD pipeline, a new deployment artifact is generated and prepared for release. This artifact encapsulates the validated code, ensuring that only rigorously tested and performance-compliant components are deployed.

Through automation, CD minimizes manual intervention and reduces the risk of errors during deployment, enabling faster delivery of new features and improvements. By adopting CD practices, development teams can focus on iterative improvements, with the assurance that each deployment meets predefined quality and performance standards.

G. Monitoring

Post-launch performance monitoring is critical for maintaining software quality, ensuring that applications consistently meet technical standards and business goals over time. Monitoring allows for continuous tracking of key metrics, such as load times, error rates, and system resource usage, which reveal how well the application performs in real-world conditions.

Gathering data directly from real users offers deeper insights into how performance changes impact user experience, identifying patterns or specific areas needing improvement. These metrics provide actionable data, enabling teams to detect potential issues early and make timely adjustments to enhance system stability and responsiveness. Effective monitoring not only helps to quickly resolve emerging issues but also supports ongoing optimization, ensuring the application remains aligned with evolving user needs and expectations.

IV. Case Study: RoboCup SSL

RoboCup is a prestigious international robotics competition designed to advance the fields of autonomous robotics and artificial intelligence. Within the competition, the Small Size League (SSL) [23] is one of the oldest and most challenging categories. The league's emphasis on speed, agility, and decision-making presents unique challenges, requiring teams to design robots capable of navigating dynamically changing environments while executing complex strategies.

In the RoboCup SSL robot soccer competition, two teams of mobile robots compete, guided by the standardized vision system. This system processes data from overhead cameras to track all field objects. A human referee operates a community-maintained game controller to manage the match. Each team's off-field computer receives positional data and referee commands to set strategies, which govern robot actions. Each team's computer performs most of the computation and exchanges information with the robots using wireless communication. Figure 4 illustrates the dynamics of controlling robots during a typical SSL match.

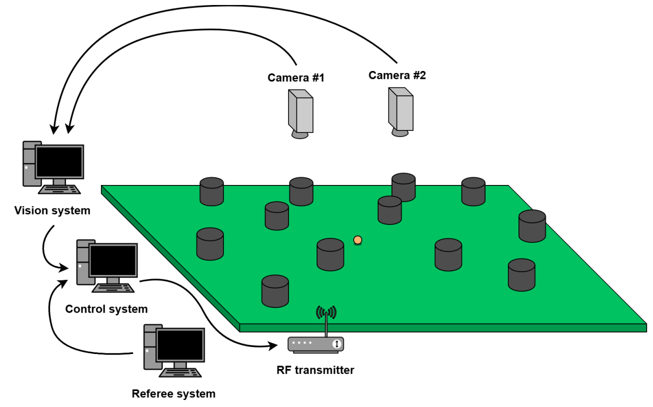

Fig. 4. General dataflow for Small Size League environment.

For the case study, a requirements gathering process was conducted with the system stakeholders, represented by the RobôCIn team – a longstanding competitor in the SSL category, who aims to use the new software as a migration from the previous monolithic architecture. Based on the collected requirements, the system was expected to support several functionalities, including match playback, 3D visualization, real-time match information, and parameters management for controlling backend services.

The system[1] was developed using a micro frontend architecture composed of fully independent applications, as presented in Figure 5. The architecture is composed by 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,

[1]Available at https://github.com/ssl-core/ssl-core

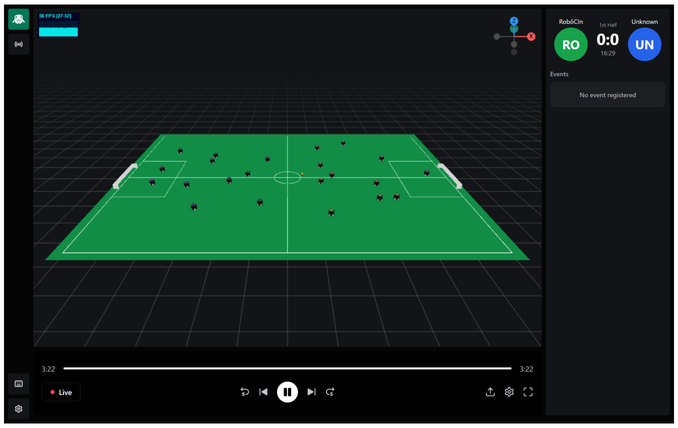

Fig. 5. Screenshot of the developed application

a strongly typed superset of JavaScript that compiles to JavaScript at build time, was used as a primary programming language to provide a layer of type safety. In alignment with the principles of micro frontends [2], 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, known for its robust concurrency model. Go was chosen to ensure efficient handling of multiple requests, making it well-suited for BFFs' intermediary role 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. Each component is designed with a specific responsibility, working independently to handle different aspects of the system while collaborating through the Application Shell. The following is a summary of the main components and their responsibilities:

- **Application Shell:** Serves as the entry point and orchestrates dynamic loading, rendering, and communication between micro frontends. Communication between micro frontends is facilitated through an event-driven model using the Broadcast Channel API. The shell supports integration via WebComponents or IFrames, with WebComponents being the preferred approach for their native integration and reusability.
- **Player MFE:** Responsible for real-time match playback, synchronizing other micro frontends with match data. To meet performance requirements, it uses Web Workers to process match data off the main thread, ensuring smooth playback. A WebSocket connection is maintained with the Player BFF to receive match data, which is broadcast to other components via the event bus.
- **Viewer MFE:** Renders a dynamic 3D match environment using Three.js and the browser's GPU. It also utilizes Web Workers for parallel processing to handle the computational demands of real-time rendering. The Viewer MFE operates across three threads: the main thread for user interaction, the communication worker for event bus data handling, and the rendering worker for 3D rendering.
- **Scoreboard MFE:** Provides real-time match updates, displaying score and event data. It is dynamically integrated through the Application Shell and communicates via the event bus. It also offers interactivity, allowing users to click on match events and adjust the match timeline accordingly.
- **Parameters MFE:** Manages configuration inputs for backend connections and system parameters. It is implemented as a modal window that prompts the user for backend IP and port details. Once the connection is established, the MFE communicates with its BFF to store and distribute the configuration settings to the appropriate services, enabling system calibration and control.

Figure 6 provides a complete visualization of the architecture, including how the components are integrated and communicate through various protocols.

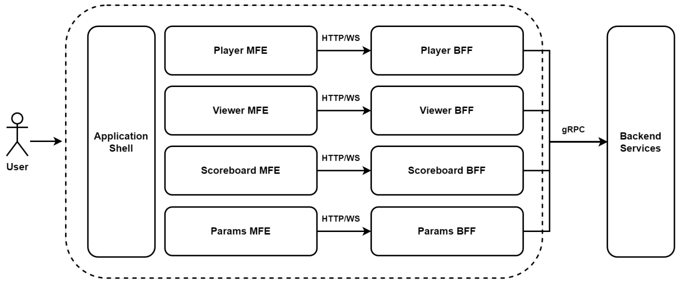

Fig. 6. Micro frontend architecture overview

V. Evaluation

This section evaluates the proposed micro frontend approach by assessing the system's quality attributes outlined in the requirement analysis phase. The assessment uses a multidimensional comparative analysis against a monolithic application developed by the RobôCIn team [24], built in C++ with the Qt framework for Linux platforms.

Static code analysis, as defined by ISO/IEC/IEEE 24765 [25], was used to extract code metrics for this evaluation. This method analyzes code structure, form, and content without executing it, providing quantitative data on software quality attributes. Various open-source and commercial tools processed the system's source code files, excluding configuration and style files.

A. Performance Efficiency

To evaluate the performance of the micro frontend architecture, a rendering comparison was conducted against RobôCIn's monolithic software for real-time rendering of a live match – the system's most critical use case.

Tests were performed on a machine running Ubuntu 22.04, equipped with an Intel® Core^{TM} 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 a workload generator. 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 7.

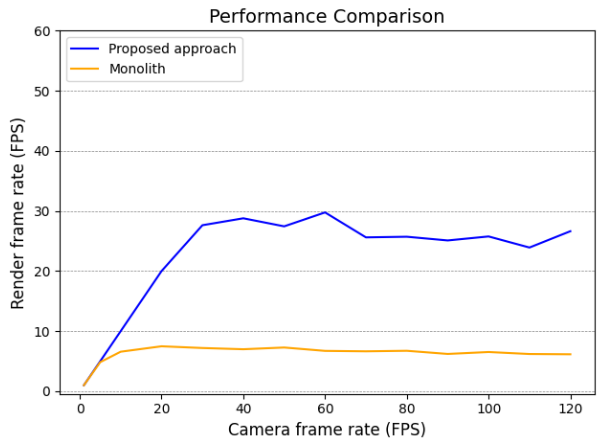

Fig. 7. Comparison between monolithic and micro frontend approach

The proposed micro frontend approach consistently delivers rendering performance between 24 FPS and 30 FPS, aligning 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 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 tight coupling of robotic control modules within the Qt-based interface. In contrast, the micro frontends benefits 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.

A similar experiment was conducted using a machine from the RoboCup SSL competition environment, equipped with an Intel® Core™ i7-8565U CPU, 16 GB of RAM, and a 256 GB SSD, running Ubuntu 20.04. The evaluation 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 [26] – during 3D real-time rendering for live match. The micro frontend achieved an average latency of 17.39 ± 3.47 ms between frame renderings during live streaming. This translates to an approximate frame rate of 57 FPS, satisfying the minimum requirements.

B. Maintainability

Unlike performance efficiency, maintainability is hard to quantify precisely and automatically, as it often relies on the team's judgment of the code structure, readability, and modularity. To provide a more objective framework for evaluating maintainability, standardized metrics like the Maintainability Index have been developed, alongside various tools with their own scoring systems.

The Maintainability Index (MI) is a widely adopted metric that 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 [27]. The original formulation of the MI is expressed as follows:

$$MI = 171 - 5.2\ln(HV) - 0.23 \times CC - 16.2\ln(LOC) \quad (1)$$

The MI was calculated using specific software tools for each programming language: Code Health Meter for TypeScript, Go Cyclo for Go, and CppDepend for C++. Results are shown in Table I.

TABLE I
COMPARISON OF MAINTAINABILITY INDEX BETWEEN THE COMPONENTS

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

Overall, the results indicate that the micro frontend architecture enhances maintainability compared to traditional monolithic systems. Each micro frontend component achieved a higher MI 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.

C. Reliability

In a broader context, metrics such as Mean Time to Failure (MTTF), Mean Time to Repair (MTTR), and Mean Time Between Failures (MTBF) are commonly used to assess reliability. However, these metrics require prolonged user monitoring post-release to collect accurate data. In this work, reliability was assessed using metrics from Embold, a commercial static analysis tool, and SonarQube, an open-source platform. Table II summarizes the findings.

TABLE II
RELIABILITY METRICS FROM EMBOLD AND SONARQUBE

Application	**Embold Score**	**SonarQube Rating**
Micro frontend	100	A
Monolith (Baseline)	91	C

Both analyses indicate the micro frontend's high reliability, achieving a perfect Embold score of 100 and a SonarQube "A" rating. In contrast, the Monolith scored 91 in Embold with a "C" rating in SonarQube, reflecting a higher likelihood of system failures.

VI. CONCLUSION AND FUTURE WORK

This work presents a novel approach for developing GUIs in robotic systems through the modeling, implementation, and evaluation of a micro frontend architecture based on OOM and DDD principles. The proposed methodology covers the entire software development lifecycle, including requirements gathering, architectural modeling, implementation, testing, deployment, and the formulation of CI/CD strategies.

An evaluation based on ISO/IEC 25010 confirmed the system's robustness across multiple quality metrics. Performance tests revealed strong results, with rendering on lower-spec hardware far exceeding the monolithic system's capabilities. In RoboCup SSL live streaming tests, the system maintained 57 FPS, emphasizing its real-time responsiveness. Embold and SonarQube analyses demonstrated high maintainability, with additional improvements in reliability that reinforce the architecture's scalability.

Future work will focus on comparative studies of developer productivity between monolithic and micro frontend architectures, along with post-release evaluations of reliability and maintainability. Additionally, promoting the system for RoboCup SSL and encouraging community-driven development could improve the platform's extensibility with more micro frontends and plugins to support a broader range of applications.

REFERENCES

[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA: O'Reilly Media, Feb. 2015.

[2] M. Geers, *Micro frontends in action*. New York, NY: Manning Publications, Nov. 2020.

[3] L. Mezzalira, *Building micro-frontends: Scaling Teams and Projects, Empowering Developers*. Sebastopol, CA: O'Reilly Media, Nov. 2021.

[4] R. Steinegger, P. Giessler, B. Hippchen, and S. Abeck, "Overview of a domain-driven design approach to build microservice-based applications," 04 2017.

[5] M. Blaha and J. Rumbaugh, *Object-oriented modeling and design with UML*, 2nd ed. Upper Saddle River, NJ: Pearson, Nov. 2004.

[6] P. M. Fresnillo, S. Vasudevan, J. A. Perez Garcia, and J. L. Martinez Lastra, "An open and reconfigurable user interface to manage complex ros-based robotic systems," *IEEE Access*, vol. 12, pp. 114 601–114 617, 2024.

[7] C. Georgiades, N. Souli, P. Kolios, and G. Ellinas, "Creating a robust and expandable framework for cooperative aerial robots," in *2024 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2024, pp. 1192–1199.

[8] G. Zhou, Y. Zhang, F. Bastani, and I.-L. Yen, "Service-oriented robotic swarm systems: Model and structuring algorithms," in *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2012, pp. 95–102.

[9] D. D. Rajapaksha, M. N. Mohamed Nuhuman, S. D. Gunawardhana, A. Sivalingam, M. N. Mohamed Hassan, S. Rajapaksha, and C. Jayawardena, "Web based user-friendly graphical interface to control robots with ROS environment," in *2021 6th International Conference on Information Technology Research (ICITR)*, 2021, pp. 1–6.

[10] A. Di Nuovo, F. Broz, T. Belpaeme, A. Cangelosi, F. Cavallo, R. Esposito, and P. Dario, "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)*, 2014, pp. 2186–2191.

[11] F. Niroui, Y. Liu, R. Bichay, E. Barker, C. Elchami, O. Gillett, M. Ficocelli, and G. Nejat, "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)*, 2016, pp. 217–222.

[12] M. Mena, A. Corral, L. Iribarne, and J. Criado, "A progressive web application based on microservices combining geospatial data and the internet of things," *IEEE Access*, vol. 7, pp. 104 577–104 590, 2019.

[13] M. Shakil and A. Zoitl, "Towards a modular architecture for industrial hmis," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 1267–1270.

[14] E. Schäffer, A. Mayr, J. Fuchs, M. Sjarov, J. Vorndran, and J. Franke, "Microservice-based architecture for engineering tools enabling a collaborative multi-user configuration of robot-based automation solutions," *Procedia CIRP*, vol. 86, pp. 86–91, 2019, 7th CIRP Global Web Conference – Towards shifted production value stream patterns through inference of data, models, and technology (CIRPe 2019).

[15] I. Sommerville, *Software Engineering*, 9th ed. Upper Saddle River, NJ: Pearson, Mar. 2010.

[16] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, ser. International Series in Software Engineering. New York, NY: Springer, Oct. 2012.

[17] I. O. for Standardization, "ISO/IEC 25010, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," International Organization for Standardization, Standard ISO/IEC 25010:2011, 2016.

[18] K. Wiegers, *More about software requirements: Thorny issues and practical advice*. Redmond, WA: Microsoft Press, Nov. 2005.

[19] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley Educational, Aug. 2003.

[20] Lethbridge, *Object-oriented software engineering: Practical software development*. Maidenhead, England: McGraw Hill Higher Education, Apr. 2002.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison Wesley, Oct. 1994.

[22] V. Vernon, *Implementing Domain-Driven Design*. Boston, MA: Addison-Wesley Educational, Feb. 2013.

[23] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, "Robocup: the robot world cup initiative," *Proceedings of the International Conference on Autonomous Agents*, 04 1998.

[24] V. Araújo, R. Rodrigues, J. Cruz, L. Cavalcanti, M. Andrade, M. Santos, J. Melo, P. Oliveira, R. Morais, and E. Barros, "Robôcin ssl-unification: A modular software architecture for dynamic multi-robot systems," in *RoboCup 2023: Robot World Cup XXVI*, C. Buche, A. Rossi, M. Simões, and U. Visser, Eds. Cham: Springer Nature Switzerland, 2024, pp. 313–324.

[25] I. O. for Standardization, "ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, 2010.

[26] B. Pueo, "High speed cameras for motion analysis in sports science," *J. Hum. Sport Exerc.*, vol. 11, no. 1, 2016.

[27] D. I. K. Sjøberg, B. Anda, and A. Mockus, "Questioning software maintenance metrics: A comparative case study," in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2012, pp. 107–110.