Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 7.056

IJCSMC, Vol. 15, Issue. 2, February 2026, pg.43 – 61

A Comparative Study of Micro-Frontend and Modular Monolith Frontend Architectures

Tarik Maljanovic

Department of Information Technologies International Burch University, Bosnia and Herzegovina

tarik.maljanovic@stu.ibu.edu.ba

DOI: https://doi.org/10.47760/ijcsmc.2026.v15i02.006

Abstract: This thesis compares two frontend architecture styles—modular monolith and micro-frontends—by implementing the same React/Vite application in two variants. The micro-frontend version uses a shell that centralizes cross-cutting concerns (routing, authentication, shared UI, theming, and API services) and loads eight independently built remotes via module federation. The modular monolith delivers the same functionality as a single build artifact with internal module boundaries. The evaluation combines: (1) maintainability indicators from a TypeScript dependency graph (coupling/cohesion and cycle detection), (2) code-quality signals from ESLint, (3) build and artifact measurements (build duration and emitted JavaScript size), and (4) runtime performance using Google Lighthouse on the deployed dashboard page, supported by qualitative discussion of fault isolation and team workflow. Results show stronger structural separation in the micro-frontend variant, with higher average cohesion (0.326 vs 0.215) and no detected dependency cycles, while the monolith contains one cycle. This comes with notable overhead: the micro-frontend suite required 113.8 s to build versus 10.7 s, and produced ~50.75 MiB of JavaScript compared to 1.85 MiB. Lighthouse audits also favored the monolith over micro-frontends. Overall, micro-frontends can improve domain boundaries and containment but demand careful optimization to manage performance and tooling costs.

Keywords: Micro-frontend, Modular Monolith, Frontend Architectures, Module Federation, Vite, Web Performance.

I. INTRODUCTION

In recent times, there has been a considerable shift in the architecture of frontend development due to the escalating growth of complex web applications. The traditional approach in frontend development has been based on a monolithic design, which encompasses a holistic implementation of the user interfaces along with their corresponding logics in a consolidated manner. Though this mode of development has been quite straightforward, it does tend to face scalability and update issues at higher levels of growth. Modular Monolith Architecture and Micro-Frontends are two notable architectures that attempt to solve these issues. The Modular Monolith architecture promotes a deployable unit that has a clear distinction between defined modules. The architecture retains several operational benefits associated with monolithic architecture, namely straightforward deployment and efficiency. At the same time, it also encourages better modularity. The Micro-Frontend architecture, on the other hand, follows microservice architecture principles in frontend application development. The Micro-Frontend architecture breaks down the front-end interface of an application into smaller micro-apps. This makes it possible for application development teams to work independently. The Micro-Frontend architecture also allows application changes to be made independent of other changes. Despite the increasing adoption rate among both architectures, empirical studies directly comparing the two in practical settings have remained relatively scarce, especially when analyzing the perspective of frontend developers who apply current JavaScript libraries with React solutions or Node.js environments. The current body of studies predominantly revolves around the backend microservices or conceptual micro-frontends rather than conducting a concrete comparison with the modular monolith system. The research paper aims to fill this research gap by performing a comparative assessment of Micro-Frontend and Modular Monolith architecture in the context of frontend application development using modern JavaScript stacks. The study will make use of the example application developed for both architecture choices and evaluate them on both quantitative and qualitative factors like performance, maintainability, scalability, development effort, and autonomy. It is a subject of especial interest to development teams today, faced as they are with the issue of what architecture to use in scaling frontend apps. Increasing complexity of frontend applications and geographically distributed development teams make it especially important to choose the right architectural style. Through explaining the trade-offs of using one system over the other, it hopes to offer valuable information to software development teams faced with such dilemmas.

II. LITERATURE REVIEW

A. *Overview of Micro-Frontend Architecture*

Micro-frontends architecture is the application of microservices concepts to the client-side by decomposing the frontend part of the web application into smaller, independently deployable micro-applications [1]. The Micro-Frontends technique was coined by the software development company, ThoughtWorks, in 2016 as an experimental approach and later promoted as an excellent technique as it gained popularity [2]. Instead of keeping the client-side source code as a monolithic piece, applications may now be composed of several micro-frontends, focusing on delivering a single business feature and usually developed by one team [2]. The vertical business domain decomposition, following domain-driven design, contrasts with other approaches, which split software into layers [3]. Micro-frontends run independently, depending on their adopted technology stack, being independently deployed, and they all provide, through integration, a single web UI residing in the client-side browser [3]. The overall goal of micro-frontends is, therefore, associating scalability and independence, as they were achieved by microservices for the backend, now scaling and independently-developing the frontend [1]. Micro-frontends were immediately adopted by the software industry, and companies such as DAZN, IKEA, Starbucks, SAP, and Zalando announced the successful deployment of Micro-frontends in production environments to handle web development for various teams and features [1]. This technique allows splitting monolithic frontends into self-containing and smaller micro-applications, and this is very appealing to those organizations interested in gaining flexibility in handling their frontend software [1]. By decomposing the UI into business sub-domains and team-oriented boundaries, organizations pursue, in this case, greater flexibility and independence in frontend deployment, as they successfully transferred this benefit from working with microservices on the backend [1]. The adoption of micro-frontends, on the other hand, introduces vast implications and considerations on software designs. Caught in this dilemma are issues, for example, on “how applications are split into vertical and horizontal cuts, composing the UI in this case, handling routing, and communication between independently running micro-applications, or whether this communication should happen at all” [4] [5]. Some of the considerations, for example, are vertical splitting, whereby “both teams implement entirely different features on the domain level (developing an isolated experience), with horizontal splitting the teams develop components for the same page, meaning that multiple teams can contribute to the same domain [3]. The process of composing the UI may be carried out on the client-side by dynamically loading micro-apps within the browser or on the server-side by composing fragments on the server [3]. Some of the core choices, such as integrating routing and communication between micro-frontends, significantly influence this architecture and must be planned ahead [4] [5].

B. Modular Monolith Architecture vs. Micro-Frontend Architecture

A Modular Monolith Architecture (MMA) refers to software architecture with a system that is deployed as a unitary entity, much like a traditional monolithic application, with the system being further broken down and compartmentalized into well-defined discrete self-contained modules. It basically aims at harnessing the simple deployment philosophy of monolithic software design with the advantages and benefits arising from well-defined separate modules typical of the microservices architecture. Modular monolith architecture has been regarded as a "middle ground approach between monolithic and microservices architecture," aiming to extract the best from monolithic and microservices software system design philosophies with respect to operational simplicity and benefits accruable from a separate concerns approach typical of the microservices architecture design philosophy [19]. In this form of software design architecture, all the discrete system modules run within the same process and share the same database and do not incur the cost and expense of network calls between services; this provides significant benefits and advantages compared to microservices architecture design philosophy because operations between modules happen within memory method calls instead of network calls between APIs [19]. Modular monolith architecture design philosophy requires well-established strong module boundaries between various system modules that can be implemented and made possible through specific design and programming framework approaches and protocols that promote system decoupling while allowing all modules developed and maintained separately even though they run together as a unitary entity system design philosophy approach [20]. Additionally, this form of software system design philosophy approach may be implemented and utilized today as a transitional approach to transforming to the microservices design philosophy architecture approach [20].

On the other hand, the concept of Micro-Frontend is the extension of the microservices approach to the frontend of web applications. In this case, instead of one monolithic frontend for the application interface, multiple smaller and partially autonomous so-called "micro-applications" for the frontend make up the entire frontend interface. Normally, each micro-frontend application is maintained by a separate team and can be developed and deployed separately from other applications with a potentially different tech stack if desired [20]. In this regard, frontend development for web applications can be closely tied to domain thinking, with one domain team responsible for developing the search bar function as a micro-frontend application while other domain Teams might be responsible for other functions such as the cart system functionality developed and separately deployed as a widget micro-frontend application. In this regard, multiple benefits and principles similar to those undertaken by microservices can be realized by the application approach with regard to frontend development and deployment and include principles such as distributed application development and deployment processes for such frontend applications.

Key Differences: The monolithic modular and micro-frontend approaches support modularity at various levels and come with various differences and trade-offs. A monolithic modular system keeps the whole application as a unit for deployment, thus offering high performance and easy deployment at the cost of a joint stack and release cycle. On the other hand, the micro-frontend approach splits the frontend application into separate modules for deployment and thus offers more flexible and autonomous deployment at the cost of more complex integration aspects such as frontend routing between separate components and frontend unit consistency between various frontend modules. For instance, with the micro-frontend approach, one would require solutions for frontend routing between various frontend application modules, and frontend unit consistency between various frontend application modules. Additionally, there is runtime efficiency for such approaches like loading multiple bundles for various frontend application modules and frontend inter-api calls between various frontend application modules. The comparison analysis implies that applications developed with the micro-frontend approach may require marginally higher loading time compared to traditional frontend application deployment approaches for the application due to fetching and initializing multiple frontend application modules for interaction with other frontend application modules at start-up [21]. However, for such approaches, application development can be more scalable for doing small-scale upgrades on the frontend application without the necessity for complete frontend deployment. Any frontend application failure (for instance any frontend application errors and crashes and the like) may be limited from impairing the frontend application experience for the end user and can therefore be more reliable frontend application approaches than traditional approaches for similar deployment needs with various frontend application modules.

C. Motivations and Benefits of Micro-Frontends

Organisations are increasingly turning to micro-frontends as they attempt to overcome the drawbacks of large, monolithic frontends, which are difficult to maintain and scale as the team and the code base grow [1]. Most single-page applications (SPAs) typically grow into frontends, becoming monolithic and hindering parallel development, making it difficult for software engineers to develop independently and introducing complexity into the adoption of emerging features and technology [2]. Micro-frontends help address the same concerns by allowing development and deployment independently of the UI components, hence improving team autonomy and speeding up software development [1] [2]. Micro-frontends, according to the multivocals literature review by Peltonen et al., are adopted by companies to allow greater autonomy for teams and simplicity in managing

the frontend, especially as the applications and software development teams grow [1]. The technique enables all cross-functional teams to handle part of the overall user interface and often integrates this part with the microservice, giving teams autonomy to develop and deploy software independently [1] [2]. The pros of micro-frontends are similar to those of microservices, where they are domain-driven and hide implementation details, hence scaling software development within an organisation as required by the business [1]. The suggested pros may involve smaller, more maintainable source bases, improved opportunities to incrementally improve and reimplement frontends, and improved scalability for software development team configurations and eliminating team dependencies [2]. For instance, Jackson argues that micro-frontends make it easier for teams to improve and replace dated frontend components with latest technology incrementally, hence reducing the risks normally associated with full-frontend replacements [2]. Another touted benefit is increased opportunities to implement newly adopted framework and language features in designated areas of applications without any need to modify other parts of the applications, hence allowing software development to evolve in a modular manner [2]. The other crucial benefit is the chance to deploy features independently. The technique enables deployment of micro-frontends independently, hence giving software development teams the autonomy to implement updates in the designated UI parts without necessarily requiring a global frontend deployment [2]. The scalability of development processes is also assured by organizations that implement micro-frontends, as confirmed by Peltonen et al., since fully functional cross-functional teams are capable of managing separate product parts independently [1]. The above-cited theories are supported by various case studies, which confirm that organizations benefit greatly by introducing micro-frontends, as suggested by various studies, stating that, in the complex SPA project, the implementation of micro-frontends worked well in large applications and large development teams, meaning simultaneous development of various parts of the whole application, and all this was achieved without negatively affecting the entire UX, considering all micro-applications are merged into one product for the final consumer [6]. In another case study, an educational platform was built and the success was very evident through its dramatically reduced build time of 2 minutes, in comparison to its monolith counterpart, which has a build time of 30-50 minutes [7].

Additionally, various teams are capable of choosing different technologies as they see fit, and an example is one sub-application being developed with React and another with Angular, meaning full scalability and allowing easy migration from one technology to another, as well as team-specific expertise, which is not feasible in monolithic architecture [1].

D. Challenges and Drawbacks

Ensuring a uniform UX within the micro-frontends is an issue of critical magnitude. When working independently, the possibility of inconsistent UX may arise if no governance is in place [1]. Taibi and Mezzalira observe, "A broken user experience may happen if components look and act in different ways and there is no consistency in the software's overall look and feel" [1]. The aforementioned issue is usually resolved by either taking an overall UI guideline or communicating through a centralized design, thereby requiring synchronization in an already-decentralized process. Communication and coupling between micro-frontends must also be managed well. Ideally, independence can be achieved with self-contained micro-applications, but this is often required to communicate in some manner, as with global navigation events or shared contexts related to user authentication. If mishandled, this can easily increase the coupling or require event orchestration, which defeats independence. The communication between micro-frontends must be well-handled through techniques like events in the browser, shared stores, and potentially through intermediaries like the API Gateways or BFF [4] [8].

The debug and testing process in the micro-frontend setup is more complicated. In cases where there are issues in the combined interface, debugging through various independently deployed components, which may be developed through diverse technology sets, can be difficult. In relation to DevOps, this increases the complexity and need for advanced management practices because the more components, the more sophisticated management practices are required. However, monitoring complexity stands out as an issue, as understanding and comprehending the components of the overall system and monitoring the various micro-frontends' status is necessary [1]. The process of tracing or logging various micro-frontends might be complicated, and the deployment of various smaller services calls for advanced automations. There is also greater complexity in managing and governing micro-frontends [2]. Independent operation by various teams creates various choices and approaches on the use of tools, software, and processes, including various approaches on handling state in several micro-frontends. Autonomy is one goal and aim, yet without management, it may advance towards what is interpretable as "micro-frontend anarchy," wherein various competing and different platforms, as well as various processes, coexist [9]. Any case involving the utilization of various platforms (e.g., React, Angular, Vue, and so on) within one product is technically practicable, yet "far from advisable" unless within an advanced migration plan [9]. The management, in this case, must strike balance and dictate which ones will remain constant and identical on all levels as teams and which ones will be distinct and varying [9]. In discussions by Wanjala, (2022), simply introducing and extending various micro-frontends through various platforms without strict and advanced management and control creates complexity without positive

improvements, as various multi-frontend applications are technically and pragmatically viable, yet whether they ease or reduce overall development stress and expenses is doubtful and inconclusive [14]. In conclusion, the cons of micro-frontends are the cost of performance, as they may feature large payloads, duplicate work, complexities in handling routing, and complexities in managing micro-frontends [1]. The cons mentioned above show that the adoption of micro-frontends is not enough on its own, as they require appropriate engineering practices to overcome monolithic problems and the cost of distributed UI [1]. Most companies cope with the cons by focusing on infrastructure, like creating a joint shell for composing applications, enforcing global standards, like the use of one framework per app and an overall build process, and management tools to ease debugging for micro-frontends [1]. It is paramount to strike the right balance between consistency and modularity, as Savani (2023) asserts, and adopt the right micro-frontends strategy tailored to the requirements of the task at hand [10].

E. Architecture and Composition Strategies

When working on micro-frontends, it is important to choose how to carve up and assemble the applications. Two main architectural strategies are often cited: vertically decomposing applications by features or domains, and horizontally decomposing by layers or page areas. In a vertical cut, a micro-frontend handles an entire feature from top to bottom, from UI elements to backend calls, usually related to a domain subarea (e.g., shopping cart micro-frontend, products catalog micro-frontend, and so on) [3]. This approach aligns with microservice domains' philosophy for bounded contexts, and in this approach, teams are really cross-functional, working on entire slices or domains of functionality [3]. For horizontally decomposing, UIs are carved up into page areas or into very general functions (like header page teams, footer page teams, or 'search box' function teams, for example) [3]. In this approach, more than one micro-frontend needs to work together to provide data for one page, meaning many teams need to work on one page to design its layout [3]. One page can contain many team inter-dependencies, making pages or workflows vertically-splitting teams isolated [3]. Horizontal decomposition can also introduce more interdependence, where many teams are working on one page, and vertically decomposing aligns with microservices domains for contextual boundaries, where teams are also cross-functional.

The other important decision is on the composition method – where and how to stitch together the micro-frontend pieces to create a cohesive UI. In general, composition can take place at build-time, server-side, or client-side (run-time) compositions. Server-side compositions involve rendering the full HTML page on the server from the output of various microservices or templates. This method, also often referred to as "Edge Side Includes" or template compositions, can result in fast page loads for visitors as the full page is delivered to the client directly by the browser from the start [3]. This method has famously been implemented by e-commerce giants such as Amazon and Ikea for their e-commerce websites, where every aspect of the web page – for example, recommendations, reviews, and product details – is provided separately by various services but are combined on the same page at the server level [3]. Server-side compositions also provide seamless SEO capabilities and can utilize CDNs to cache composed pages but need to handle complex dynamic content in client-specific pages.

Client-side composition, on the other hand, means that the client or browser side loads and integrates various micro-frontends usually after the basic shell application has started. This can also happen on the client-side using JavaScript, for example, via single-SPA or Web Components to encapsulate each micro-frontend work separately in its own encapsulation technology for micro-frontends [3]. In this client-side integration method, for example, micro-frontends can consist of independently deployable bundles or bundles that can also essentially be sourced from other domains, where rendering is controlled by client-side routers for rendering dependent on their respective routes for rendering. Client-side integration also makes extensive usage of frameworks or technologies for dynamic code integration for other bundles at runtime using technologies such as Module Federation or Webpack 5 or even using iframes for very isolated ecosystems. One big advantage of client-side integration or compositions is its loose integration, where no actual integration has to happen on the server or on its side, meaning that no change on one side affects integration on the other side directly on the server. Another disadvantage sometimes for client-side integration or compositions is its decelerated start or load phase resulting from making perhaps many trips to obtain micro-application versions. Furthermore, its processes for link integration or link manipulation can sometimes also seem complicated or more burdensome on client-side applications or integration or compositions than others.

Another form is middle-ground, where integration is done at build or deployment stage to form one artifact out of codes from various micro-frontends, for example using monorepos where everything is built together. This offers various advantages when working separately but acts similarly to just one app once in operation. Build-time integration also means losing the ability to deploy separately, making it far less common in its pure form but applied in development stages for simplicity.

No matter where the composition takes place, to implement micro-frontend architecture, it is usually necessary to provide a routing system and a hosting container. Taibi and Mezzalira identify four main architectural points to focus on in every micro-frontend project: 1) dividing horizontally or vertically, 2)

whether to compose on either the client or server sides, 3) routing approach, or 4) communication between micro-frontends [4] [5]. For example, routing can take place in either a central or distributed fashion, meaning either with one router on the shell redirecting to micro-apps or distributed where every micro-frontend has its own routes and only routes top-level routes on the shell to micro-apps. "Single-spa" routers are usually how most micro-frontend architectural implementations handle routers by mapping various URL routes to respective micro-frontends, meaning mounting every micro-application for its route. Another approach can utilize network-level routes via edge proxies or API gateway serving various fragments depending on URL routes.

Communication between frontends is also another concern in terms of architecture. It is preferred that micro-frontends rarely communicate directly with each other – they would ideally do so via APIs or perhaps via events within the browser, depending on need. This is preferably accomplished via publish-subscribe models within the browser for loose coupling (for example, via a search box UI element sending out an event that can be received by a web results element implemented in another group), or via HTML5 Custom Events for others. Within more directly integrated UI elements, development teams can access or share global state via one or more shared state management libraries, though this introduces its own coupling and so is often eschewed for micro-frontends except where absolutely needed. Some other frameworks exist, including within single-SPA, to provide APIs for apps to publish announcements to other apps. The ideal route is to limit communication between micro-frontends as much as possible, perhaps via using state within the backend or URL for communication points between them to act on. It is also important to mention that industry-agnostic solutions for making micro-frontends work together exist. "Single-SPA" is an open-source library solution that enables living together on the same app with various frontends with seamless orchestration of their lifecycles. "Webpack Module Federation" enables runtime sharing for web applications built separately but wanting to consume each other's capabilities on runtime and has also emerged as a very popular method for creating micro-frontends so that various teams can develop, deploy, but also consume each other's dynamic capabilities on runtime levels very effectively [11]. Many companies create "shells" or "container applications" for integration to provide services for common tasks including routing, authorization, design services, into which micro-apps can plug seamlessly into once they are created. For example, the approach taken by Spotify and others has also implemented various versions where essentially every page has only one thing: various iframes or shadow DOM elements for various micro-frontends — this provides strong isolation between micro-frontends but increases greatly the complexity for basic look-and-feel coordination - and finally SAP has created Luigi, which is essentially a general-purpose micro-frontends framework for enterprise Apps - but also including seamless navigation for various enterprise applications on its own domain [12]. Typically, many applications are implemented using what can almost be called a hybrid approach: server rendering for initial loads (maybe assembling two to three key micro-frontends on initial page load), followed by client-driven rendering for additional interactive elements or on navigation events. For example, an app can implement server rendering for its header and navigation areas, sourced from one team for optimized TTFB, and client-driven rendering for switching to other micro-apps on navigation. It is important to ensure consistency and shared resources are also considered in terms of design. Teams usually collaborate on various resources, for example, CSS frameworks or UI component libraries to avoid duplication. Some micro-frontend frameworks follow a shared dependency approach where common libraries are imported via the shell to ensure micro-frontends do not include their own versions, making them lightweight but ensuring versioning is consistent. Some frameworks force consistency in versions for various frameworks for easier integration, but this defeats various benefits related to technology freedom. The degree to which independence and speed are balanced is important, as Parab (2025) recommends that technology heterogeneity can exist between micro-frontends but can introduce guidelines on versioning core libraries to ensure consistency [13]. To conclude, when it comes to micro-frontend design, it can be very beneficial to implement an orchestration solution for routing and composition, provide integration contracts in terms of APIs or event contracts, or develop a mechanism for sharing or isolating resources. This enables developers to ensure that pieces they develop work well together to form one big application.

F. Adoption Strategies and Best Practices

Applying micro-frontends in a practical organizational setup is not only about technical transition but also needs attention to process and culture evolution. This can be accomplished by adhering to a guide or framework for how micro-frontend development and execution will happen. For this purpose, Amorim et al. (2025) introduced "Guide for Adoption of Micro-Frontends (GAM)," assigning organized factors to businesses willing to migrate to micro-frontend topology [5]. This context also emphasizes harmoniously aligning micro-frontends to "business needs" and adhering to "conventions to overcome known difficulties" in micro-frontend adoption. Primarily, recommendations include "Architecture and Design Patterns that enable consistency and sharing between micro-frontends" and maintaining "CI/CD patterns for coordinated but independent deployment" for efficient execution in micro-frontend topologies [5]. In their respective research on organizations using micro-frontend topology, their experience at big corporations introduced the need to specifically concentrate on important factors in micro-frontend adoption, including "performance, scalability, and parallel development"

but also worsening factors including “module complexity and developer experience” for pro-active intervention in micro-frontend topology execution [5].

Organizational structure is also an important aspect to adopt when using micro-frontends. Modular structure suits micro-frontends. Conway's law applies here. The idea is to ensure every micro-frontend belongs to one team that has resources to implement that part of functionality – frontend, backend, QA, etc. This enables end-to-end ownership. This is supported by Wanjala, who explains that it is important to ensure that “there is clarity on how things are delivered, including how change happens between teams. This can include outlining boundaries and responsibility areas for teams and how they coordinate their work on change that affects more than one micro-frontend. It is also essential to ensure that senior leadership embraces this approach and provides resources to support it, for example, ensuring each team has its own version of the pipe line or dev environment” [14]. It is also essential to provide training for developers on how to work well under new micro-frontend architectural requirements.

Shared Infrastructure: Another best practice for simplifying adoption is shared infrastructure. Many successful adopters choose to provide a “platform” or toolkit for micro-frontends. This could include things such as a shared routing library, common authentication support, or design toolkit. For example, common navigation bars or session management support can be created once and then be common to every micro-frontend. Some organizations also create a central micro-frontend “shell team” which handles common support for everyone else's needs, where every “feature team” communicates via their own micro-app to this core framework.

Nevertheless, avoiding global coupling is also important. Best practices recommend keeping to a minimum the number of shared dependencies, as too many shared services can introduce global coupling and deployment coordination, defeating the micro-frontend approach's purpose. This is rightly asserted by Amorim et al.: “although many tech stacks can be combined, it does not follow that every team ought to create its own combination," as using many frameworks adds to heterogeneity and integration difficulties [5]. Indeed, Geers in 2020 [27] points out that if every team is allowed to choose to utilize the same set of core technologies, many implications of autonomy are preserved while reducing complexity, hinting that organizations tend to standardize on a few “sweet-spot” decisions to achieve balance between hetero- and homogenization. For example, all micro-frontends are built using React and TypeScript, but can select their own state management solution or build tooling, ensuring that the ecosystem remains coherent enough to forestall incompatibilities. Secondly, ThinkWorks recommends establishing which are team decisions and which are company decisions for a micro-frontend program [9].

Communication and sharing between and within teams is also considered as one of the factors for adoption. Although micro-frontend teams are decoupled, they can still share their experiences and work together on cross-cutting tasks. This can ensure, for example, that instead of each team reimplementing something, they can work together to reuse one team's approach to error handling or analytics. Antunes et al., in 2024, studied that for companies to overcome difficulties in large web applications using micro-frontends, they organized full company-wide developer workshops to discuss perceived benefits and drawbacks [15]. This group evaluation allowed both for collecting feedback about difficulties such as dependency management, debugging, or integration tests, but also for gaining support for new ideas and technologies, as they forged support within the company for using micro-frontends by sharing information about perceived benefits and drawbacks [15]. Developers concluded that they could greatly benefit but also reserved judgment on unidentified issues to ensure they are mitigated correctly via involvement from within their own group to overcome skepticism and point out areas for improvement. Testing and DevOps pipelines must also change to support micro-frontends. This involves ensuring each micro-frontend has tests at the component level, and also integration tests for the entire application. This can also involve using either feature flags or containerization to improve deploying micro-frontends separately. For example, using Docker or even Kubernetes can ensure deployment is decoupled, meaning that failures in one micro-frontend can't automatically impact other micro-frontends. Some companies also use canary releases or even A/B tests for micro-frontends separately to rollout modifications. Some companies also need to ensure monitoring for micro-frontends is distributed, where each micro-frontend provides data on its own status to ensure monitoring on one platform. This approach will ensure that whether an issue lies in, for example, the Payment micro-frontend or in Search micro-frontend, observability will easily identify this. Crucially, companies need to implement micro-frontends incrementally. Often, “the strangler fig method” – imported from microservices – can be applied: essentially, “the existing monolithic app remains in place, but certain parts of its UI are progressively routed to new micro-frontends” [2]. This approach means that while migration happens, “business value can still flow” instead of undergoing “a massive rewrite effort and long downtimes for deployment” or other dramatics. For instance, on e-commerce websites, “the account management part can start as the very first micro-frontend peeled off”, followed later by “the view details page for products” and so on, until “the legacy frontend is finally strangled – i.e., effectively removed”, but with no disruptions to the UX. User-centric concepts must not be forgotten in the adoption phase. Although it is primarily about coding and teams, finally, it also has to remain intuitive and performative on the UI side. As

stated above, therefore, micro-frontend adoption must also not reduce UI consistency or responsiveness to the end-consumer. This means investments in optimizations for performances (such as overarching caching for common resources, support for HTTP/2 or HTTP/3 for concurrent load, and so on) are part of responsible adoption efforts. Taibi and Mezzalira provide one recommendation to improve adoption work: to create within their project a common design system, so that no matter how many technology stacks are introduced, finally, every micro-frontend has to look and perform similarly to others [1]. Finally, it is essential to assess when micro-frontends are applicable to your organization. This approach to applications is not always beneficial. As recent studies suggest, "Hangai Kojo et al. identified that micro-frontends would provide most benefit to organizations where frontend teams are encountering scaling barriers within the monolith or where the need for independent deployment of features is important to their business strategies" [16]. Kojo's research provides valuable information on how they applied micro-frontends effectively to a marketplace web app but stated "not strictly necessary for meeting its goals—a well-organized monolith could have provided comparable outcomes for this company" [16]. Kojo proposes that micro-frontends are beneficial depending on whether "the company is already onboarding new microservices for its backend infrastructural revamp and trying to monolitize its monoliths. Applying micro-frontends became very beneficial for them as they could reuse many infrastructural resources they implemented for their new microservices approach" [16]. This can also provide valuable information on when to implement micro-frontends within organizations.

G. *Applications and Case Studies*

The micro-frontend architectural style has been utilized in different applications ever since its inception, thus proving its versatility. The idea of micro-frontend architecture was first experimented with in the context of content management systems. Yang et al. in 2019 developed a micro-frontend architecture in the context of the intranet CMS of a corporation. This proved that dividing the frontend logic of the system into modules may help in scaling medium to large-sized applications [25]. This proves that the idea of extending the microservices pattern to frontend applications dates back to 2019.

Within the e-commerce industry, in scenarios in which multiple teams develop different parts of the same customer-facing application (for example, the product page or checkout process), micro-frontends appear to have received considerable attention. For an e-marketplace in 2025, it was observed by this case study publication that micro-frontends enabled this particular organization by helping unplug a Kuphryn-connected frontend using both the API Gateway and the Backend-For-Frontend (BFF) pattern together with microservices to support the provision of a more modular frontend Web App [16]. Feature teams enabled each other in utilizing modern frameworks selected based upon appropriateness to each particular feature (Svelte being utilized in one particular piece, for example), and developers now enjoy increased autonomy in this particular system. However, this particular study admits that other strategies like the Modular Monolith may accomplish the same aims.

Providers of enterprise software have also seen success in implementing micro-frontends. For instance, DAZN, a sports streaming platform, implemented micro-frontends in order to allow multiple teams to contribute simultaneously towards the development of their web app in response to the need for enhanced feature deployment in various markets across the world [1]. Zalando, another leading retailer in the industry, developed Mosaic. The software platform developed Mosaic based on micro-frontends in order to allow various teams to deploy new features independently in their storefronts while improving A/B testing in production. The above-mentioned examples highlight that there exist significant opportunities for software platforms of this scale in the effective utilization of micro-frontends.

Micro-frontends have also made inroads in industry and business applications. Within manufacturing, Schäffer et al. in 2019 implemented the architecture of microservices and micro-frontends in an engineering software for the configuration of solutions in robot automation [17]. This allowed multiple users in collaboration to use the shared web interface through multiple frontends that were later combined in one single software tool. This allowed more than one engineer to contribute in parallel towards various facets in the overall robot cell configuration. Within the context of HMIs in industry applications of HMIs (Human-Machine Interface), Shakil and Zoitl in 2020 developed a novel HMI architecture influenced by the idea of micro-frontends [26] with the objective of packing and playing back the modules of the user interface in industry control systems in order to increase malleability in comparison to the standard SCADA interface in HMIs. While both applications are industry-specific in objective, both applications tend to validate the benefit of micro-frontends in manufacturing or IoT dashboard applications in terms of enhanced modularity and parallel software development.

Educational software would be another significant use case. According to Wang et al. in 2020, a micro-frontend approach was utilized in the development of the management information system of the graduate school of a university. The system consisted of various modules like student data or course management done through different micro-frontend modules or components. This allowed the IT staff of the university to build different modules like admissions, transcript analysis, or finances independently based on the organizational structure of the university [18]. The system became more maintainable by allowing the university's IT staff to implement system evolution based on updates like implementing a new student interface in the student portion

of the system independently of the faculty system modules [18]. Even mobile apps have seen advancements in micro-frontends. For mobile apps (iOS/Android), although they are not web applications in nature, the idea of modular frontend architecture has been attempted using web technologies in mobile domains. Capdepon et al. in 2023 proposed the idea of micro-frontends in a Flutter mobile application perfectly embedding micro-frontends in a Flutter framework echoing the successful utilization of micro-frontends in other domains like the Web [8]. This project in progress rightly titled "micro-frontends for mobile applications", smooth working seems to consolidate that the ideology can be adapted in any frontend-like system ready for decomposing and self-governance. The areas of routing, composition, and communication are equally crucial in this mobile world, and the future challenges include multiple UI modules in small screens in mobile devices and measuring the constraints of mobile devices [8]. At last, micro-frontends have inspired the creation of tooling and frameworks revolving entirely around this style of working. After Module Federation and single-SPA, there are now starter kits and libraries available (such as Piral, Luigi, Mosaic) and those works in progress, including various forthcoming books and community-developed guides (such as Micro Frontends in Action by Geers, 2020 [27] in various research). This growing body of works seems indicative of the ripening of this process—that the pioneering works done by the first adopters have now solved many of the same problems (cross-routing in micro-frontends or shared state management that needs consideration), in favor of simply being accessible by those following on behind. Various patterns have been floated in the community, including using a "micro-frontend gateway" that determines what route-associated micro-application needs loading or in other ways integrating authentication across all micro-application frontends in use. To conclude, micro-frontends have been utilized in various industry domains, including e-commerce platforms and streaming services, educational platforms, and industry software solutions, in scenarios typically involving multiple teams working together or requiring the amalgamation of various feature sets in the web UI. Taken together, the various case studies highlighted in this section tend to show that micro-frontends implemented in the proper contexts can create more nimble software development and maintenance procedures in complex web applications in particular. At the same time, however, this growing number of case studies tends to highlight that there do exist contexts in which the micro-frontend approach may be overkill or superfluous in particular domains or scaling levels [16].

H. Tools and Metrics for Comparing Architectures

In assessing a monolith architecture in a modular form in relation to a micro-frontend or other architectures, various metrics are used for evaluation. Some of the key metrics in such a comparative analysis are performance, maintainability, and scalability. Some key metrics used in a comparative analysis of architectures along with the techniques employed in their measurement include:

- Performance Metrics: To estimate performance, certain metrics, like response time, throughput, and use of resources, are measured per architecture. To test backend throughput or API performance, for example, load testing software like Apache JMeter can be used. An example of performance analysis, where stress tests were performed with JMeter, can be seen in Al-Debagy and Martinek, where a monolithic architecture was stress tested in relation to a microservices architecture, which resembled a stress test of a "single-unit system" and a "distributed system," respectively, with both analyses showing that, in fact, a monolithic architecture enabled about 6% higher throughput per request when concurrent connections were high; this proved that no communication between services would optimize performance [22]. To test performance in a micro-frontend, analysis would relate to webpage loading speed as well as size. Using such software as either Google's Lighthouse Tool for webpage analysis or WebPageTest.org, for example, software will provide analysis of First Contentful Paint, total loading time, and other webpage performance metrics in a "modular monolith frontend" setup as opposed to a "micro-frontend" configuration of a webpage. Kaushik, S. et. Al. provide a test of a web application using a "micro-frontend" setup with a "microservices" backend as compared to a traditional "monolithic frontend" architecture, where analysis showed that, with careful implementation, it should be feasible to optimize performance in this way, as it would facilitate parallel development as well as parallel loading of "multiple distinct UI components" among other benefits in this "micro-frontend" application setup analysis [23]. This analysis would disprove that a "micro-frontend" application would provide substandard performance due to certain inherent negatives that all "micro-frontends" would share in common, as it would evidence certain benefits, like fine-grained scaling, which would, in reality, provide benefits that would improve performance in this application setup.
- Maintainability and Code Quality Metrics: The choice of architecture also influences maintainability, which can be measured as "system understandability, changeability, and modifiability". To facilitate a comparison of maintainability, code analysis tools (SonarQube, Structure101, and ArchUnit) can estimate other metrics like coupling, cohesion, complexity, and module autonomy for a given codebase with different architectures. There are models of maintainability using these metrics to compare service-oriented architectures with a monolith. For example, in migrating from a monolith to a microservices architecture, coupling, cohesion, complexity, and size (code volume per module) are structural parameters identified as most important by Hasan et al. In a similar application of maintainability metrics in a comparison of a modular monolith with a micro-frontend solution, modularity of code in both approaches would be analyzed, where a good modulo-monolith would ideally offer

low module coupling and high module cohesion, characteristic of high maintainability. While in a micro-frontend architecture, code for different features would be separate, automatically providing low coupling between modules, as in a micro-frontend solution a module would be a separate artifact, and possibly enhancing maintainability by constraining a codebase to a given feature. However, code analysis would possibly point out potential code redundancy and complexity due to interconnected use of different software frameworks and build pipelines. Metrics for code complexity in modules would provide insight into how much individual codebases in a micro-frontend solution are simpler than an equivalent monolith, which would, in most cases, be true, but also how complex it would be to manage all those codebases. Qualitative analysis would also take into consideration, for example, how complex it would be to test and implement in a micro-frontend solution, which would involve individual testing of all codebases, as opposed to a monolith, which would involve just one pipeline but encompasses a larger body of change.

- Scalability and Flexibility: While it is hard to provide a direct quantitative value for a characteristic like scalability, it can be evaluated in terms of how architectures fare with respect to scaling in user bases or in features. This would be done by scaling in a controlled lab environment and analyzing performance (comparing scaling a monolith by introducing additional instances of it, as opposed to scaling individual microservices for the frontend portion of it, for example). A monolith is vertically scalable, which involves scaling all of it using a powerful server, as opposed to scaling a portion of it using a load balancer, whereas micro-frontends provide a much finer degree of scaling, where individual micro-frontends can be scaled/deployed as needed/requested. These characteristics of a monolith and micro-frontends can be measured in terms of deployment time and deployment rate, where architectures inspired from microservices, including micro-frontends, are able to offer a much higher deployment rate, which can be measured using DevOps metrics like deployment rate and lead time for changes. Platforms like container orchestration – Docker and Kubernetes – make it easy to deploy as well as maintain scaling for micro-frontends, providing a means to compare how many independent units of deployment exist in both models. A greater number of units available for deployment provide a flexible benefit but also necessitate automated support for infrastructure. While doing comparative evaluations, it also becomes important to consider fault isolation properties, which would ideally use a Chaos testing tool that temporarily disables a module and then tests how it affects the larger application. While these are relatively less studies than their performance or maintainability counterparts, it becomes important to consider in a comprehensive evaluation. This would help, for example, in understanding that if a micro-frontend fails, it would be completely independent of other functionalities, which would continue to run. It would offer a benefit of persistence, which a monolith would possibly not. This becomes a qualitative evaluation metric for either reliability and robustness. In reality, a combination of techniques has been employed in order to get a rounded perspective. A comparative study might involve developing prototype implementations of a particular system using both architectures, followed by a series of experiments using load testing, analysis, and simulation of maintenance tasks. Finally, both architectures may be compared on a number of different quality parameters like performance, maintainability, scalability, and so on, using standard methodologies like ISO/IEC 25010 for software quality. The application of a set of parameters becomes necessary since every architecture suits a different domain of application. A micro-frontend architecture, for example, would rate better in terms of team productivity, independent deployability, whereas a modular monolith would do better in terms of efficiency of operation and ease of tests, among others. A science-based study using objective parameters like time, resource usage, and code metrics, in conjunction with proper software tools like profilers, code analysis tools, and test harnesses, would decide which of the architectures needs to be applied in a particular setting. This would involve a set of trade-offs that a series of key parameters indicate, with evidence from experiments and software tools [22] [23] [24].

I. Final Reflections

Micro-frontends sit at the intersection of ambition and caution. They promise a more modular, autonomous way to build big web apps that line up nicely with microservice backends. Teams that have adopted this approach say it helps break down sprawling frontends into bite-sized pieces owned by different squads, accelerating development and giving you freedom to mix and match technologies [1] [2]. However, you pay in performance overhead, the risk of drift between pieces, and additional operational load that has to be tamed by solid architecture and governance [1] [9]. The best practice today is clear: plan and govern. Define conventions for routing, communication, and styling. Provide shared infrastructure that you need such as shell and CI/CD pipelines. And continuously weigh trade-offs to keep the benefits ahead of costs [1] [5].

Micro-frontends pay dividends in large, multi-team projects where independent deployability and domain separability yield actual value. For smaller efforts, the overhead may well override the benefits. As Hangai Kojo and co-authors indicate, one should always question if a micro-frontend architecture is truly needed, or if a well-structured monolith would suffice [16]. The result is a practical message: instead of following every trend, the engineers evaluate what their systems really need.

Looking ahead, the field is pushing to smooth out the rough edges. New tools aim to cut duplicate downloads and boost runtime performance; design systems and style guides are evolving to support distributed UIs; and

empirical work-mapping studies, surveys-is helping quantify how micro-frontends affect development productivity and user experience [1]. As the approach matures, it should become more standardized in parallel with microservices with clearer guidance on when and how to implement it. In brief, micro-frontends indicate a meaningful evolution of frontend architecture, carrying modularity and scalability from microservices to the client side. Drivers of this paradigm find their place in literature alongside hurdles like managing complexity and tuning performance. The organization considering micro-frontends will want to draw on emergent best practices from industry and research: perform the migration gradually, impose sensible consistency where it matters most, invest in tooling and automation, and keep mindful of trade-offs. Applied in the right context and executed with discipline, micro-frontends can help enterprises deliver large web applications with greater flexibility and autonomy, keeping frontend work aligned with modern distributed software systems [1].

III. DATASET

A. *Codebase Overview*

The core source of data of this research paper are the codebases on which the tests are conducted, and the results provided by the measurement tools. Endpoints provided by the backend codebase are used by both frontend versions (Modular Monolith and Micro-Frontend). The objective is to perform tests on a project which could be used in a real-life scenario, thus the frontends will not be static by containing hard-coded data. The two frontend versions are two separate codebases which have the implementation of the same website concept, and use the same measurement tools.

The website concept is a dashboard website where users can create and manage AI-powered voicebots, create and manage knowledgebases used by the voicebots, provide an overview of the conversations, request transcripts from audio or video files, along with a statistical overview of conversations. The AI-powered services are provided by Elevenlabs' REST API. The services include creating the voicebots, knowledgebases, and performing the transcription of audio/video files, and delivering the results in a JSON format.

B. *Backend Architecture*

The backend codebase is used by both frontend versions, its purpose is to expose the necessary endpoints for the frontends to be able to perform identical operations in a Node.js environment, meaning that the code is written solely in JavaScript. It performs basic CRUD operations on data stored within a MongoDB database, using the Mongoose ODM framework. Additionally, the backend makes API calls to the external Elevenlabs REST API using Axios (HTTP calls). The structure of the codebase follows the Model-View-Controller architecture pattern to a certain degree, by separating the logic, routes, and the Mongoose models into separate directories.

C. *Modular Monolith Architecture*

The Modular Monolith frontend is an isolated codebase which will be used for performing the necessary measurements in order to evaluate the architecture. It is a React Application, initialised using Vite.js, and further developed using various libraries for functionality and the User Interface/User Experience. The root component features the "Login Page" component, which is an independent component and a direct child of the root component, and the "Layout" component which contains all the other pages as its child components. This means that when the user navigates to a different page, it renders a different child component inside the Layout component, in order to avoid unnecessary re-rendering of components which do not change, such as the Navigation Menu and the rest of the page.

Inside the source directory of the codebase is the "components" directory, which features all the pages and the common components. Each page has its own directory with components making up the page (Dashboard, Agents, Knowledgebases, etc.). While the "common components" directory features all the components used by multiple pages (Navbar, Button, Input, etc.).

The application uses TailwindCSS and MaterialUI for the majority of the styling and responsiveness, external libraries to support page-translation, making HTTP calls to the backend, and Typescript in order to support static typing and introduce features such as Enums and Interfaces.

Another core part of the application is the "services" directory, where the API-calls are written as exported functions using the Axios library in order to simplify the process of making HTTP requests and delivering the response. Each service is written in Typescript and represents its backend counterpart.

D. *Micro-Frontend Architecture*

In addition to the modular monolith implementation, the same application was also implemented using a micro-frontend approach. The micro-frontend version is organized as a multi-application workspace based on Vite and React. Instead of one frontend codebase that contains all pages, the system is split into a host (shell) application and a set of domain-specific remote applications. Each remote represents one business area of the product (e.g., agents, dashboard, knowledgebases, conversations, speech-to-text, profile, users, tenants) and exposes a single page-level entry point that can be loaded into the shell at runtime.

The shell acts as the main entry point of the system and centralizes cross-cutting concerns that should remain consistent across the entire user experience. In particular, the shell owns the global layout and navigation

(sidebar/header "chrome"), routing configuration, authentication and authorization, theming, localization, and shared UI building blocks. This makes the shell responsible for maintaining consistency in look-and-feel and access control, while the remotes focus on implementing domain functionality. In practice, routing is defined in the shell and the remote pages are loaded lazily when users navigate to their corresponding route. This allows the overall application to behave like a single SPA from the user's perspective, while still being assembled from multiple independently built artifacts.

To avoid duplication of common logic, the shell also provides shared modules that the remotes reuse rather than re-implement. These shared modules include authentication utilities (such as an authentication context and a protected route mechanism), a shared API layer built around an Axios client that can attach authentication tokens, and a common component library / design system used across all pages. In addition, common domain types and utilities are placed in shared code so that remotes can remain thin and primarily compose existing building blocks. Each remote can still be executed independently for development and preview purposes, but the intended production execution mode is through the shell.

At build and runtime, the micro-frontend system uses module federation to connect the shell with the remote applications. Remote entry URLs and local development ports are centralized so that the shell can resolve each remote's exposed page entry point. To avoid loading duplicate framework code, core dependencies such as React, the router, Redux, UI libraries, internationalization, and HTTP tooling are configured as shared singletons. Overall, this architecture aims to support independent evolution and deployment of domain pages, while the shell enforces system-wide standards for authentication, navigation structure, and user experience.

E. *Measurement Tools*

To compare the two frontend implementations, the evaluation relies primarily on repository-native tooling and small auxiliary scripts. The goal of this choice is to keep the measurement process reproducible and close to the codebase, so that the same procedure can be applied consistently to both architectures.

For architectural modularity metrics (module coupling, cohesion, and cycle detection), a custom Node.js script is used that analyzes the TypeScript/JavaScript dependency graph (imports between files and modules). The script produces quantitative indicators such as fan-in/fan-out patterns, internal vs. external dependency edges, cohesion ratios, and detected cycles. These outputs are exported in a machine-readable form to support later reporting and comparison.

Build-time is measured using wall-clock execution time of the respective build commands. Since the micro-frontend version consists of multiple applications (shell and remotes), its build includes multiple sequential build steps, while the modular monolith is built as a single application.

Bundle-size is measured from production builds by summing the emitted JavaScript artifacts in the generated dist output. Because the two architectures produce different sets of build outputs (one dist folder vs. multiple dist folders), the measurement is defined as the total emitted JavaScript size per architecture, which makes the results comparable from a distribution and CI/CD perspective. Optional bundle visualization (treemap) can be produced using Vite/Rollup visualization tooling where applicable.

Maintainability and complexity are approximated through static analysis by running the repository ESLint configuration and recording lint violations (including complexity-related rules and code-quality issues). Finally, team autonomy and fault isolation are evaluated through a combination of repository mining (commit history and contributor patterns) and architectural reasoning about deployment/runtime boundaries (e.g., the blast radius of a failing module or remote entry).

IV. METHODOLOGY

A. *Research Design*

This thesis follows a comparative evaluation design in which two frontend architectures—(1) a modular monolith and (2) a micro-frontend architecture—are assessed using the same application domain and comparable implementation technologies. Both frontends implement the same functional scope and communicate with the same backend API. The independent variable of the study is the architectural decomposition strategy (single modular application vs. shell + domain remotes). The dependent variables are a set of technical and organizational indicators that reflect performance and maintainability-related qualities.

The evaluation combines quantitative and qualitative methods. The quantitative part focuses on measurable properties such as page-load performance (via Lighthouse), build-time, bundle-size, and static-analysis indicators. These metrics are selected because they can be collected reproducibly and provide concrete evidence for trade-offs that teams face in real projects (e.g., runtime performance vs. build/deployment overhead). The qualitative part complements these results by analyzing properties that are harder to capture purely numerically in a small-scale study, such as fault isolation boundaries and team autonomy. These aspects are evaluated through architectural reasoning and repository-mining signals from version control.

The study is best interpreted as an applied software engineering artifact evaluation. The results are intended to provide evidence of trade-offs in a controlled, comparable setup, rather than universal conclusions for all

projects. The measured differences are therefore discussed in relation to the characteristics of the implementations (e.g., runtime composition, shared dependencies, and build structure) and linked back to the research questions regarding when one architecture is more suitable than the other.

B. *Experimental Environments*

Two experimental environments were used, depending on the type of metric under evaluation.

Local environment (development-time measurements). Metrics that depend on the codebase structure or build pipeline were executed on the author's local development machine. This includes build-time measurement, bundle artifact sizing from the build outputs, static analysis (ESLint), and dependency-graph analysis for coupling/cohesion. The local environment represents the "developer workstation" perspective and reflects the cost of building and maintaining the system during development and CI.

Deployed environment (runtime/performance measurements). Metrics that depend on real HTTP delivery and runtime execution in the browser were evaluated on deployed instances. Both frontend implementations and the backend were deployed on DigitalOcean as separate applications. The modular monolith frontend is deployed as a single static build, while the micro-frontend setup is deployed as a set of independent static builds (shell plus remote applications) that are assembled at runtime. Performance measurements (page-load metrics) were executed against these deployed URLs using Lighthouse, which captures network transfer, browser execution, and runtime behavior under a defined Lighthouse configuration. This split between local and deployed environments is used to avoid conflating developer-time and runtime characteristics, and to ensure that each metric is collected in the environment where it is most meaningful.

C. *Evaluation Metrics and Tools*

Throughout the Literature Review of the research process, several metrics have been defined which can be used to compare the architectures. The table below defines these metrics, the tool which will be used for the measurement process, with the environment in which the metric will be measured. The environment will either be "Development" or "Production", meaning that it will either be tested locally on the machine, or as a deployed app. In order to increase the accuracy of the results, the environment-sensitive metrics will be tested on the application deployed using the DigitalOcean cloud platform.

TABLE I
MEASUREMENT METRICS

Metric	**Description**	**Tool(s)**	**Environment**
Load Time	Total time from page request to full load	Google Lighthouse	Deployed
First Contentful Paint (FCP)	Time until first visual element is rendered	Google Lighthouse	Deployed
Time to Interactive (TTI)	Time until app becomes fully interactive	Google Lighthouse	Deployed
Bundle Size	JavaScript Bundle Size	Webpack Bundle Analyzer, Build Logs	Local
Build Time	Time required to compile and bundle the frontend	Build Logs (npm, Vite)	Local
Code Complexity	Measures maintainability and logic complexity	ESLint, SonarQube	Local
Module Coupling & Cohesion	Evaluate architectural modularity and independence	Custom Script	Local
Fault Isolation	Whether a failure in one module affects others	Manual failure tests	Local
Time Autonomy	Degree to which teams can work independently	Commit history analysis	Mixed

D. *Data Collection Procedure*

The data collection process was executed separately for the modular monolith frontend and the micro-frontend suite, using the same high-level steps and recording outputs in a consistent format.

1) Preparation and consistency checks
- Both repositories were installed with the same package manager workflow (dependency installation and lockfile resolution).
- The same backend API was used for both frontends, ensuring that differences in measurements are primarily attributable to frontend architecture rather than backend behavior.

2) Local (developer-time) measurements
- Build-time: For each architecture, the production build command was executed and the wall-clock time was recorded. The micro-frontend build includes building the shell and each remote application, while the monolith build is a single build step
- Bundle-size: Production builds were generated, and the emitted JavaScript artifacts in the dist directories were collected. The total JavaScript size per architecture was calculated by summing the sizes of the emitted .js files. This provides a comparable "total shipped JS artifact" indicator across the two build structures.
- Static analysis (complexity/maintainability proxy): ESLint was executed using the repository configuration. The number and type of lint violations were recorded for later comparison.
- Coupling and cohesion: A dependency-graph analysis script was executed to extract fan-in/fan-out patterns, internal vs. external dependency edges, cohesion ratios, and detected cycles. Outputs were saved so they could be referenced directly.

3) Deployed (runtime) measurements
- Deployment: The backend and the two frontend implementations were deployed on DigitalOcean. The deployment ensures the applications can be audited under realistic HTTP delivery conditions.
- Lighthouse audits: Lighthouse was executed against the deployed frontend URLs to collect page-load performance metrics. The JSON output format was used so that results could be extracted consistently and included in tables. Since Lighthouse results can vary with throttling configuration and network conditions, the chosen preset and execution time were documented alongside the results.

4) Documentation and reporting
- Console outputs and generated JSON/metric files were archived and summarized into tables.
- Where a metric produced multiple related values (e.g., coupling/cohesion), the most relevant values were selected based on the research questions and the evaluation metrics defined previously.

V. RESULTS

This chapter presents the empirical results of the comparative evaluation between the modular monolith frontend and the micro-frontend implementation. The results are organized by metric category, following the evaluation plan.

A. *Page-load Performance (Lighthouse)*

Page-load performance was measured using Lighthouse against deployed instances. Table 5.1 reports the primary Lighthouse timing metrics. In this thesis, "Load Time" is approximated by the Speed Index metric, which reflects how quickly the visible content of a page is populated.

TABLE II
LIGHTHOUSE PERFORMANCE RESULTS (DEPLOYED)

Metric	**Micro-Frontend**	**Modular Monolith**
Speed Index (Load Time)	4.86 s	0.94 s
First Contentful Paint (FCP)	4.14 s	0.93 s
Time to Interactive (TTI)	8.55 s	1.26 s

In the collected reports, the micro-frontend run produced substantially higher values for Speed Index and Time to Interactive than the modular monolith. A likely explanation is that the micro-frontend implementation loads and evaluates a larger number of JavaScript resources before it reaches an interactive state, which is consistent with a runtime-composed application where multiple remote bundles are fetched and executed. However, an important limitation of the current dataset is that the two Lighthouse reports were generated using different presets ("mobile" vs. "desktop"), which affects throttling and makes the absolute values not strictly

comparable. For a final thesis version and for publication-quality evidence, both architectures should be audited under identical Lighthouse settings and repeated across multiple runs. To provide additional context beyond the three primary timing metrics, the reports also show a difference in request volume and transferred bytes. The micro-frontend report includes substantially more total requests and script requests than the monolith report, which supports the interpretation that more resources are being fetched in the micro-frontend runtime.

B. *Bundle Size (Production Build Artifact)*

Bundle-size was measured from production build outputs by summing the emitted JavaScript artifacts. This metric reflects the amount of JavaScript produced by the build pipeline that would need to be stored and served (e.g., via a CDN) for each architecture.

TABLE III
TOTAL EMITTED JAVASCRIPT SIZE (LOCAL BUILD)

Metric	**Micro-Frontend**	**Modular Monolith**
Total emitted JS	~50.75 MiB	~1.85 MiB

The modular monolith produces a significantly smaller total JavaScript output. The micro-frontend suite produces multiple dist outputs (shell + remotes), and each remote contains its own compiled code. In a real deployment, compression and caching reduce network transfer, but the larger artifact footprint still impacts build storage, CI/CD execution, caching efficiency, and the cost of invalidating caches across multiple deployments.

C. *Build Time*

Build time was measured via wall-clock time for the production build. The modular monolith performs one build step, while the micro-frontend suite builds the shell and the remote applications.

TABLE IV
BUILD-TIME RESULTS (LOCAL BUILD)

Metric	**Micro-Frontend**	**Modular Monolith**
Build time (wall clock)	113.84 s	10.65 s

The micro-frontend build is noticeably slower, which is expected given that multiple applications are built sequentially. This difference is relevant for developer feedback cycles and CI pipelines, where the cost of building the entire system can become a bottleneck unless builds are parallelized or made incremental.

D. *Static Analysis / Code Complexity Proxy (ESLint)*

ESLint was used as a maintainability and code-quality proxy. The total number of reported lint issues differs between the architectures.

TABLE V
ESLINT RESULTS SUMMARY (LOCAL)

Metric	**Micro-Frontend**	**Modular Monolith**
Errors	249	115
Warnings	12	6

The micro-frontend suite reports more lint issues overall, which may be partially explained by the larger overall code volume across multiple projects. Nevertheless, the presence of lint issues in both architectures indicates that maintainability is not determined solely by architectural boundaries; coding standards, refactoring practices, and consistent enforcement also play a significant role.

E. *Module Coupling, Cohesion, and Cycles*

Coupling/cohesion was analyzed by extracting the import dependency graph and computing summary indicators. The micro-frontend analysis identifies slightly fewer internal edges but more external edges. Most notably, the cohesion ratio is higher for the micro-frontend version, and no cycles were detected in the analyzed module graph. The modular monolith shows one detected cycle.

TABLE VI
COUPLING/COHESION SUMMARY (LOCAL)

Metric	Micro-Frontend	Modular Monolith
Modules analyzed	10	9
Internal edges	58	61
External edges	107	96
Average cohesion ration	0.3263	0.2151
Cycles detected	0	1

A higher cohesion ratio in the micro-frontend version is consistent with the intended domain-oriented decomposition where each remote focuses on a specific business capability. The detected cycle in the modular monolith suggests that, even with modularization, a single-repository application can accumulate cross-module dependencies unless boundaries are actively enforced.

F. *Fault Isolation (Qualitative)*

Fault isolation was assessed qualitatively based on the deployment/runtime boundaries of the two architectures. In the micro-frontend setup, the shell loads individual remote entries at runtime. This structure implies that the failure of a single remote (e.g., a missing remoteEntry or an exception inside that remote's code) primarily affects the routes that depend on that remote, while other remotes may remain functional. In contrast, the modular monolith is deployed as one application bundle; a failure during initialization or a critical runtime crash is more likely to impact the entire UI.

In the current thesis version, fault isolation is primarily reasoned from architecture rather than verified through systematic failure injection experiments. As a next step, this can be strengthened by deliberately simulating remote failures (e.g., returning 404 for a remote entry or throwing errors in a remote module) and recording the observed blast radius and user-visible degradation.

G. *Team Autonomy (Repository Mining)*

Team autonomy was evaluated using repository mining signals such as contributor distribution and commit separation. In the current repositories, the commit history indicates a single primary contributor and a small number of commits, which limits the ability to empirically demonstrate multi-team independence. Therefore, the team autonomy comparison remains largely theoretical in this thesis version: micro-frontends are designed to enable independent ownership and deployments per domain, while the modular monolith encourages shared ownership and coordinated releases.

H. *Summary of Results*

Across the measured metrics, the modular monolith shows clear advantages in build-time and produced artifact size. The micro-frontend implementation shows evidence of stronger modular boundaries in the coupling/cohesion analysis (higher cohesion ratio and no detected cycles), and it provides a stronger theoretical foundation for fault isolation and independent deployments. The results therefore reflect a trade-off: micro-frontends can improve domain separation and isolation but introduce additional build and runtime overhead; modular monoliths remain simpler and more efficient for smaller-scale deployments, but require discipline to prevent erosion of module boundaries over time.

VI. CONCLUSION

This thesis compared two frontend architecture styles—modular monolith and micro-frontends—implemented using the same application domain and modern JavaScript tooling. The goal was to provide evidence for architectural trade-offs that teams face when choosing between simplicity and centralized control (modular monolith) versus runtime decomposition and independent deployability (micro-frontends).

A. *Answering the Research Questions*

RQ1 (Performance differences). The collected evidence suggests that the modular monolith has a clear advantage in developer-time efficiency (build time and produced artifact size) and shows lower Lighthouse timing metrics in the provided dataset. The micro-frontend implementation has higher runtime overhead in the performance audits and a much larger total emitted JavaScript footprint, which is consistent with the cost of assembling a UI from multiple independently built parts. At the same time, the current Lighthouse comparison is limited because the reports were generated using different Lighthouse presets; therefore, the exact magnitude of the performance gap should be treated as preliminary until both applications are audited under identical settings and repeated measurements.

RQ2 (Qualitative attributes such as maintainability and complexity). Static analysis results show lint issues in both architectures, indicating that maintainability is influenced by both architecture and coding practices. In the module dependency analysis, the micro-frontend version exhibited a higher cohesion ratio and no detected dependency cycles, while the modular monolith exhibited one cycle. This suggests that the micro-frontend decomposition can support stronger domain boundaries in practice, whereas a modular monolith needs explicit governance to prevent cross-module coupling over time.

RQ3 (Team structure, deployment frequency, workflow). The study's ability to empirically validate team autonomy is constrained by the limited commit history and single-contributor nature of the repositories. Nevertheless, the architectures differ conceptually: micro-frontends are designed to allow teams to build and deploy domain remotes independently, while modular monoliths typically favor coordinated releases. Future work that includes multi-contributor development and separate release pipelines would be required to make strong claims about deployment frequency and workflow.

RQ4 (When does one architecture outperform the other, and why?). Based on the measured results and architectural analysis, a modular monolith is the stronger choice for smaller teams, early-stage products, or performance-sensitive applications where minimizing build complexity and runtime overhead is critical. Micro-frontends become more attractive in larger organizations where teams are distributed, releases must be decoupled, and fault isolation or independent evolution of domains is more valuable than a minimal runtime footprint. In such contexts, micro-frontends can pay back their additional operational and performance cost by enabling scaling of teams and ownership.

B. *Limitations*

The results should be interpreted in light of several limitations:

- Lighthouse results can vary with throttling settings and network conditions, and the current dataset contains reports generated under different presets. A fair comparison requires identical Lighthouse configuration and multiple runs per architecture.
- Only a subset of pages was audited (e.g., focusing on dashboard-related routes). Other routes might yield different results, especially for applications where some pages have heavier data requirements.
- Team autonomy could not be validated empirically due to the small and single-contributor commit history.
- Fault isolation was reasoned from architecture rather than validated through systematic failure-injection experiments.

C. *Further Work*

To strengthen the evidence base and make the work suitable for publication, the following extensions are recommended:

- Re-run Lighthouse under the same configuration (desktop and mobile separately) for both architectures with multiple repetitions, report mean and standard deviation, and include additional metrics such as LCP, TBT, CLS, and request counts.
- Expand the performance evaluation to multiple representative pages (dashboard, agents, users, tenants) and to cold-cache vs. warm-cache scenarios.
- Add controlled fault injection tests to empirically measure blast radius in the micro-frontend case (remote entry unavailable, remote runtime exception) and in the monolith case (runtime exception at initialization).
- Complement ESLint with additional maintainability tools (e.g., consistent complexity thresholds, duplication analysis) and track how metrics evolve as new features are added.
- Evaluate workflow and autonomy using multi-contributor development, separate CI/CD pipelines per remote, and release cadence data.

D. *Final Remarks*

The study supports the hypothesis that micro-frontends trade performance and build simplicity for modular boundaries and potential organizational scalability, while modular monoliths deliver strong performance and simplicity but require discipline to maintain module boundaries as the codebase grows. For most small-to-medium applications or early-stage teams, the modular monolith provides the best cost–benefit ratio. Micro-frontends are best justified when independent deployment and domain ownership are first-class requirements and when the organization is prepared to invest in governance, tooling, and performance optimization to manage the added complexity.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to my mentor, Assist. Prof. Dr. Fatima Mašić, for her guidance, support, and constructive feedback throughout the development of this thesis. Her encouragement and insight helped me stay focused and improve both the structure and quality of this work..

REFERENCES

[1]. S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review," Information and Software Technology, vol. 136, p. 106571, 2021.

[2]. C. Jackson, "Micro Frontends – extending the microservice idea to frontend development," MartinFowler.com, June 2019. [Online]. Available: https://martinfowler.com/articles/micro-frontends.html (accessed 2025).

[3]. Y. R. Prajwal, J. V. Parekh, and R. Shettar, "A Brief Review of Micro-frontends," United International Journal for Research & Technology (UIJRT), vol. 2, no. 8, pp. 123–126, 2021.

[4]. G. Cunha de Amorim and E. D. Canedo, "Micro-Frontend Architecture in Software Development: A Systematic Mapping Study," in Proc. 27th Int'l Conf. on Enterprise Information Systems (ICEIS 2025), vol. 2, pp. 105–116, 2025.

[5]. G. Amorim, L. Rocha, F. F. Mendes, R. P. dos Santos, and E. D. Canedo, "Guidelines for Adoption Micro-frontend Architecture," in Proc. SBSI 2025 – Brazilian Symposium on Information Systems, Recife, Brazil, May 2025.

[6]. A. Pavlenko, N. Askarbekuly, S. Megha, and M. Mazzara, "Micro-frontends: Application of microservices to web front-ends," Journal of Internet Services and Information Security (JISIS), vol. 10, no. 2, pp. 49–66, 2020.

[7]. D. C. Hidayat, K. J. Atmaja, and I. B. G. Sarasvananda, "Analysis and Comparison of Micro Frontend and Monolithic Architecture for Web Applications," Jurnal Galaksi (GKAIIS), vol. 1, no. 2, pp. 92–100, Aug. 2024.

[8]. Q. Capdepon, N. Hlad, A.-D. Seriai, and M. Derras, "Migration Process from Monolithic to Micro Frontend Architecture in Mobile Applications," Berger-Levrault & LIRMM, University of Montpellier, France.

[9]. ThoughtWorks Technology Radar, "Micro Frontend Anarchy," Oct. 2020. [Online]. Available: https://www.thoughtworks.com/radar/techniques/micro-frontend-anarchy (accessed 2025).

[10]. N. Savani, "The Future of Web Development: An In-depth Analysis of Micro-Frontend Approaches," International Journal of Computer Trends and Technology, vol. 71, no. 11, pp. 65–69, Nov. 2023.

[11]. L. Mezzalira, Building Micro-Frontends: Scaling Teams and Projects, Empowering Developers, O'Reilly Media, 2021.

[12]. D. Taibi and L. Mezzalira, "Micro-Frontends," ACM SIGSOFT Software Engineering Notes, vol. 47, no. 4, pp. 25–29, 2022.

[13]. A. Parab, "Micro Frontends Architecture – Breaking Down Monolithic Frontend Applications," Journal of Computer Science and Technology Studies, vol. 7, no. 6, pp. 1014–1023, Jun. 2025.

[14]. S. T. Wanjala, "A Framework for Implementing Micro Frontend Architecture," International Journal of Web Engineering and Technology, vol. 17, no. 4, pp. 337–352, 2022.

[15]. F. Antunes, M. J. D. de Lima, M. A. P. Araújo, D. Taibi, and M. Kalinowski, "Investigating Benefits and Limitations of Migrating to a Micro-Frontends Architecture," Tecgraf/PUC-Rio, UFJF, and University of Oulu, 2023.

[16]. R. H. H. Kojo, L. F. Corte Real, R. C. Ferreira, T. O. Rosa, and A. Goldman, "Exploring Micro Frontends: A Case Study Application in E-Commerce," in Proc. European Conf. on Software Architecture (ECSA 2025), LNCS 15982, pp. 187–201, 2025.

[17]. E. Schäffer, M. Brossog, J. Franke, P. Gönnheimer, D. Kupzik, S. Coutandin, and J. Fleischer, "Web-Based Platform for Planning and Configuration of Robot-Based Automation Solutions: A Retrospective View on the Research Project ROBOTOP," in Proc. 1st Int. Conf. on Robotics and Automation Engineering (ICRAE), pp. 387–397, Jan. 2022.

[18]. D. Wang et al., "A Novel Application of Educational Management Information System based on Micro Frontends," Procedia Computer Science, vol. 176, pp. 1567–1576, 2020.

[19]. R. Su and X. Li, "Modular Monolith: Is This the Trend in Software Architecture?," Proc. of the Intl. Workshop on Software Architecture Trends (SATrends 2024), 2024, pp. 1–6.

[20]. D. Taibi and L. Mezzalira, "Micro-Frontends: Principles, Implementations, and Pitfalls," ACM SIGSOFT Software Engineering Notes, vol. 47, no. 4, pp. 25–29, 2022.

[21]. D. C. Hidayat, I. K. J. Atmaja, and I. B. G. Sarasvananda, "Analysis and Comparison of Micro Frontend and Monolithic Architecture for Web Applications," Jurnal Galaksi, vol. 1, no. 2, pp. 92–100, 2024.

[22]. O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," in Proc. of the 18th IEEE International Symposium on Computational Intelligence and Informatics (CINTI), Budapest, Hungary, 2018, pp. 149–154.

[23]. N. Kaushik, H. Kumar, and V. Raj, "Micro Frontend Based Performance Improvement and Prediction for Microservices Using Machine Learning," Journal of Grid Computing, vol. 22, art. no. 44, 2024.

[24]. M. H. Hasan, M. H. Osman, N. I. Admodisastro, and M. S. Muhammad, "From Monolith to Microservice: Measuring Architecture Maintainability," International Journal of Advanced Computer Science and Applications, vol. 14, no. 5, pp. 857–865, 2023.

[25]. C. Yang, C. Liu, and Z. Su, "Research and Application of Micro Frontends," IOP Conf. Ser.: Mater. Sci. Eng., vol. 490, no. 6, p. 062082, 2019, doi:10.1088/1757-899X/490/6/062082

[26]. M. Shakil, and A. Zoitl, "Towards a Modular Architecture for Industrial HMIs," in Proc. 25th IEEE Int'l Conf. on Emerging Technologies and Factory Automation (ETFA), 2020, pp. 1069–1072.

[27]. M. Geers. 2020. Micro Frontends in Action. Manning Publications, https://books.google.com.br/books?id=FFD9DwAAQBAJ

