BENEFITS, CHALLENGES, AND PERFORMANCE ANALYSIS OF A SCALABLE WEB ARCHITECTURE BASED ON MICRO-FRONTENDS

Adrian Petcu[1], Madalin Frunzete[2], Dan Alexandru Stoichescu[3]

In recent years, there has been significant growth in software development in terms of how code is managed, load distribution, and the ease of adding new features to existing codebases. While there are established patterns for achieving these goals in backend development, front-end applications or Graphical User Interfaces (GUIs) do not have a simple, scalable implementation pattern. Micro-frontends can help with this issue by making front-end development more scalable and straightforward. This is especially useful since teams often struggle with monolithic applications that are difficult to maintain or enhance. This paper will explore the benefits and challenges of a scalable architecture based on micro-frontends.

Keywords: micro-frontends, scalability, architecture, code management, GUIs

1. Introduction

Web applications have evolved to support working on multiple layers without developers stepping on each other's toes while parallelizing work on the same feature between different teams and technologies.

In recent years, software development has been approached differently. The focus has shifted from having developers work on features from start to finish to having them concurrently work on various aspects of the application. Currently, methods have been implemented to enable multiple developers to collaborate on the same feature, with each developer primarily focusing on a single application layer instead of multiple layers.

For example, for web applications that have specialized front-end, back-end, and database developers features can be developed in parallel on all three layers, increasing efficiency and expertise. An advantage of working on layers

[1]PHD Student, Faculty of Electronics Telecommunications and Information Technology, University "Politehnica" of Bucharest, Romania, e-mail: adrian.petcu@stud.etti.upb.ro

[2]Lecturer, Faculty of Electronics Telecommunications and Information Technology, University "Politehnica" of Bucharest, Romania

[3]Professor, Faculty of Electronics Telecommunications and Information Technology, University "Politehnica" of Bucharest, Romania

is that communication between the front-end and back-end can be abstracted, and any layers can be changed without the other layer's announcement.

Another trend when writing this paper is splitting large applications into smaller apps, specialized and coordinated by a central proxy application. This is commonly referred to as "splitting monolith applications into microservices." This reduces the deployment times and allows developers to work on specialized services rather than everybody working on the same application.

This paper analyzes the existing solutions for creating micro-frontend applications and highlights obstacles and advantages.

In the first section, we analyze current architecture patterns and their benefits and drawbacks. Then we deep-dive into micro-frontend architecture, investigating patterns and implementation techniques. Finally, we research the available methods for implementing micro-frontends, each with advantages and disadvantages based on predefined criteria, and a practical comparison between three implementations of choice.

2. **Architectural overview**

In recent years, new technologies and methodologies have enabled the development and deployment of software applications to be performed more quickly and efficiently. One of the most significant trends in software development is the shift from monolithic to microservice-based applications. Typically, monolithic applications are large, complex applications that are created and deployed as a singular entity. Conversely, microservices are small, independent services collaborating to form a larger application.

For many years, monolithic applications have been the predominant architecture for software development, but they have several drawbacks. They can be challenging to maintain and scale, and revisions to one part of the application can significantly affect other areas. On the other hand, microservices offer a more modular approach to software development, with each service accountable for a particular function or feature. This enables greater flexibility and scalability, as each service can be developed and deployed independently, making adding new features simpler or extending the application to meet fluctuating demand.

2.1. **Monolithic Architecture**

The word 'Monolith' in software development refers to a single-tiered application in which multiple components and services are combined and served under the same application infrastructure, which is served by only one platform. Such applications usually serve multiple areas of concern[1] from one big application. Such components and services might include the following in a regular application:

(1) User authorization and authentication are used to manage the user's identity and application browsing session in the application.

(2) Business logic responsible for flows and application actions
(3) The presentation layer is provided by the User Interface, which interacts with the application layer[2] via HTTP requests
(4) Integration with third-party applications via network protocols and REST communication Notification services and monitoring, responsible for raising alerts when applications are down

The monolithic Architecture has been present and is implemented in most of the applications [3] we interact with in our current days. Such architecture is easy to follow as it is straightforward in all processes, from development and testing to deployment. Applications with higher complexity usually become problematic as they are hard to maintain, and more complexity is introduced by multiple teams working on the same codebase. There are several drawbacks to monolithic architectures, and this can be seen in the following list:

(1) The entire application must be shut down and redeployed for each minor change, and users might be affected.
(2) With a higher complexity, the codebase increases along with the build time, disk space, and startup time.
(3) Maintenance becomes stressful as the application becomes too large and complex to understand fully. It is difficult to make changes quickly and accurately.
(4) Reliability: a failure in a module can potentially stop the entire process or the application instance.
(5) Although it may seem simple in the early stages, monolithic applications have difficulty introducing new technologies[4] because framework changes can affect the entire application.
(6) Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements.

2.2. Microservices Architecture

The Microservices architecture is a variant of the Service Oriented Architecture. Even though SOA has been available for a long time, microservices were officially adopted in 2012 [5]. The central concept behind microservices is to have multiple small and autonomous services that work together to serve the same purpose of a large application focusing on dividing the large, complex application into smaller chunks [6], organized by sub-domains, thus being easier to maintain. This way, smaller-sized applications are independent and easy to deploy, ensuring that if a particular service is down, the rest of the services and a large proportion of the application will continue functioning properly. There are multiple benefits to microservices. A part of those benefits can be observed in the following list:

(1) Continuous delivery and deployment are greatly simplified because the application is decoupled into smaller services[7] that serve a limited purpose.

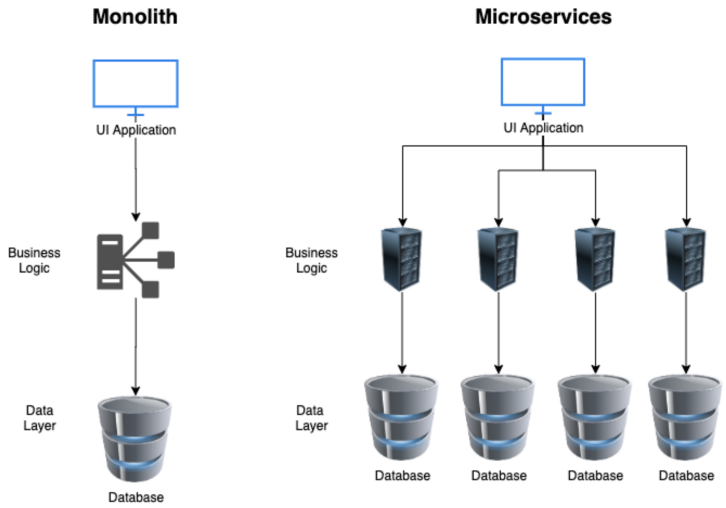

FIGURE 1. Monolith vs Microservice Arhitecture

(2) Development of new features can be done independently, and the number of code conflicts between the development teams is significantly reduced.
(3) Fault isolation is improved. If a service is down due to memory leaks or some other errors, the rest of the services will continue to provide responses to the consumers, and only some parts of the end-user application might be affected.
(4) Automated testing is improved as small-scaled services serve limited scopes.
(5) The microservices architecture allows team independence in deciding the architecture[8] and programming language used for each service, as the communication is done via agreed standards.
(6) Improved testability because of the smaller services and faster deployment.
(7) Development can be scaled and organized between multiple teams on the spot.

2.3. Monoliths vs Microservices

As per the stated advantages and drawbacks, microservices are an obvious choice. If there is a particular demand for a specific area of the application, only the services responsible for that area are scaled to meet consumer needs[9]. The scaling is horizontal and can be instantaneous because the small size of microservices allows them to start very fast. Scalability can also be achieved with monoliths, but the application's resource consumption would significantly increase. Figure 1 presents a comparison between layer separation in both architectures.

3. Micro-frontends

In the cloud web application area, and the business logic being transformed into the microservices architecture, the front-end applications become bigger and bigger. Slowly, the non-essential logic is moved to the front-end applications, and this can cause a strain on the applications, which become harder to maintain. [10] As previously stated, breaking the backend applications from a monolith into microservices brings significant benefits, and the same architecture pattern should also be explored for the User Interface applications. This would break the User Interface into multiple smaller modules that can work independently but under the same application shell [11]. The solution for this problem is using Micro Frontends.

New front-end frameworks are introduced to the public. Along with them, the appetite for working with new and optimized tooling reduces the development time while increasing the application performance. There are multiple ways to organize front-end applications, i.e., Single Page Applications (SPA) and Server-Side Rendered Applications (SSR). However, most of the solutions are monoliths loaded chunk by chunk instead of each area being technology agnostic[12] and independently deployable.

Software developers often adopt technologies based on their experience without considering the whole project's magnitude, thus leading to a complex, monolithic project in which multiple teams are involved. Working on a single codebase developed by multiple teams takes time to maintain.

Micro-Frontend is a Microservice-like approach to front-end web technologies. The primary purpose behind micro-frontends is to split the application into multiple application units based on pieces of functionality or screens that represent a specific domain instead of creating an actual monolithic front-end application[13]. De-composing the application is the team's responsibility so independent parts coexist and share a common codebase when needed.

As can be seen in figure 2, a Micro-Fronted application is a composition of features or micro-applications that work together to create a more extensive application where independent units are controlled end-to-end by cross-functional teams. Thus, there is a loose coupling between the apps, and they coexist using well-defined contracts. Such micro-applications or features can belong to one or more application suites, thus strengthening the idea of reusability. A perfect example of a micro-frontend unit would be a shopping cart shared between multiple e-commerce solutions developed by the same company.

3.1. Composition types

To build a Micro-Frontends application, there are a few different options. For example, with Micro-Frontends architecture, certain architectural decisions must be made in advance[14] because these decisions will shape the future decisions made on the side of the project.

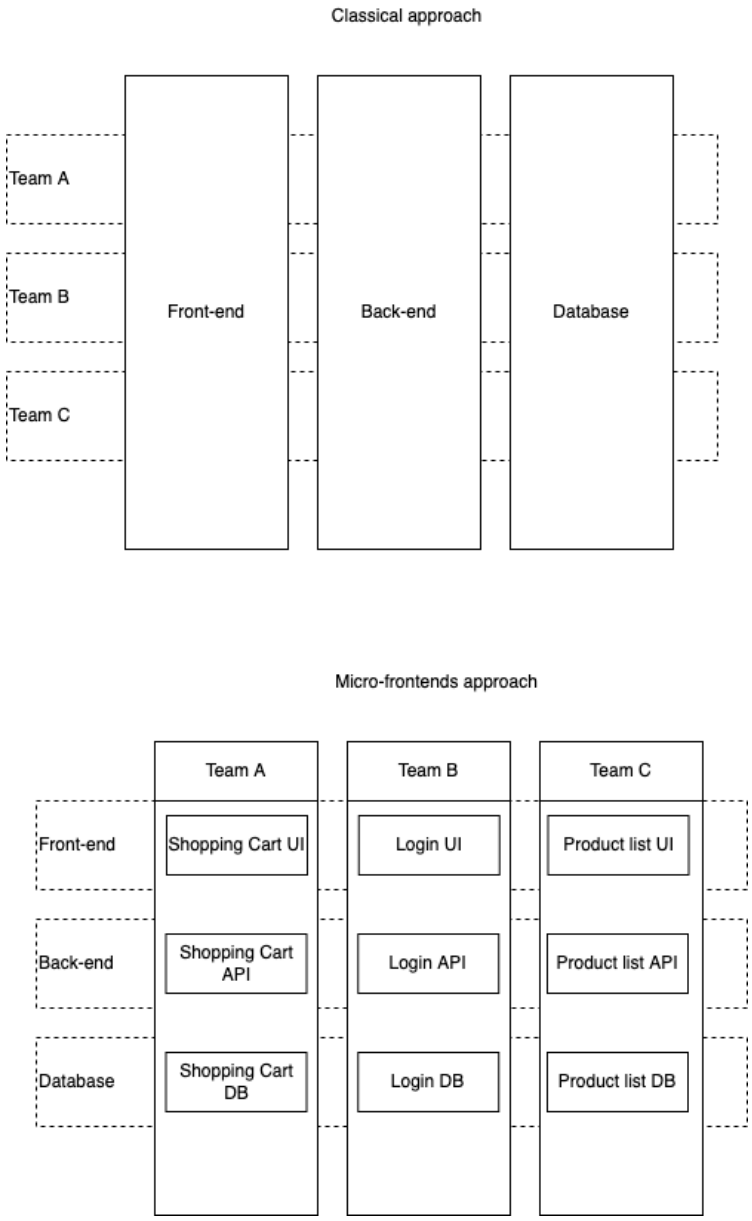

FIGURE 2. Cross-functional teams with micro-frontends

The most critical decision in defining Micro-Frontends is to identify the need to consider a Micro-Frontend from a technical point of view. There are two options for this:

- Horizontal split: several Micro-Frontends on the same page.
- Vertical split: one Micro-Frontend at a time.

In a horizontal split (figure 3), several smaller applications are loaded onto the same page, requiring several teams to coordinate their efforts. Each team is responsible for a part of the screen.

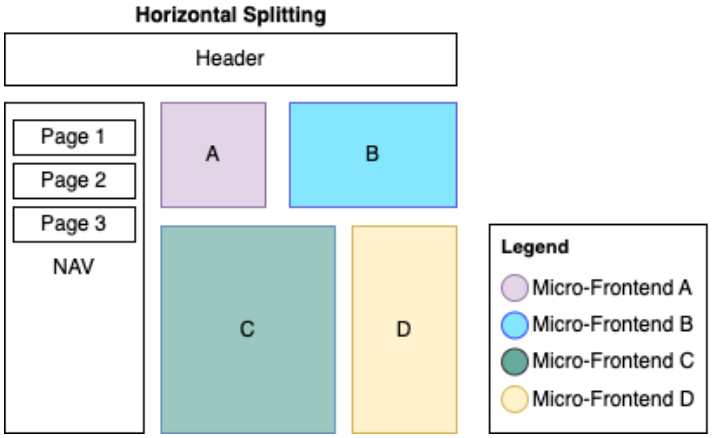

FIGURE 3. Horizontal splitting

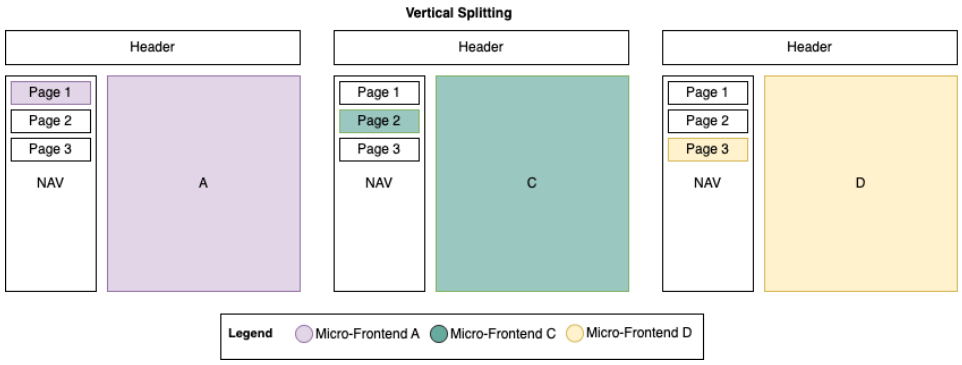

FIGURE 4. Vertical splitting

In a vertical split (figure 4), each application represents an entire page of the overall application. Each team is responsible for one page and the integration with other pages.

3.2. Challenges of using a micro-frontend architecture

To the best of our knowledge, only a limited number of studies were created to find the best approach for adopting a micro-frontend architecture. Thus, there is no standard in place. The following challenges in implementing a micro-frontend architecture have been identified:

- Working on a separate application section, a robust framework for coordinating events and communication between the different units must be maintained.
- Backward compatibility must be respected when modifying the coordinator application.
- A standard for communication between units must be implemented[15] with well-defined, standardized inputs and outputs.

- Communication should be centralized, and the Publisher/Subscriber pattern will facilitate the communication between many components at once
- Overall bundle size must be controlled by instantiating framework core bundles only in need and not for every unit.
- A styling library must be shared between all the independent units to keep aspect consistency and reduce code duplication and overall bundle size.

3.3. Benefits

Emphasizing the emerging attributes and the benefits they bring rather than a specific technical approach, the micro-frontend architecture brings many benefits to the development teams.

3.3.1. *Incremental updates.* Many organizations are slow in releasing new features since the release process is often clogged by procedures or the need to align teams and approvals.

Shifting from a monolithic to a microservice approach is done gradually, tearing the application part by part and extracting it into smaller, easily-maintainable chunks, easy to release independently[16] while releasing new features on the base module in the regular cadence which the organization is accustomed to.

Migration from old to new is easily achievable by intertwining the monolithic application with micro-front-ends. Incrementally updating parts of the application allow small experiments to be run in isolation. i.e., trying a different communication method or a new pattern of writing code

3.3.2. *Simple, decoupled codebases.* Codebases for monolithic applications tend to get bigger due to increasing complexity and feature add-ins. This leads to code smells and code duplications that result in developer frustration and slowness in reaching a business goal. Coupling different areas of the application also bring an extra layer of complexity because any code change can lead to breaking some other working parts of the application

Since the micro-frontends are independent working pieces of the bigger functionality, their codebase is significantly smaller and easy to maintain. Coupling is avoided by separating the codebases and keeping a clean communication channel applicable to all plug-and-play microservices.

Sharing is encouraged between microservices to ensure consistency and reduce duplication. The key components that have to be shared are:

- **Styling** should be shared since the app needs to follow the same design guideline
- **Stateless** libraries that are referred to as "utilities" (e.g. string parsing libraries, date formatting libraries)

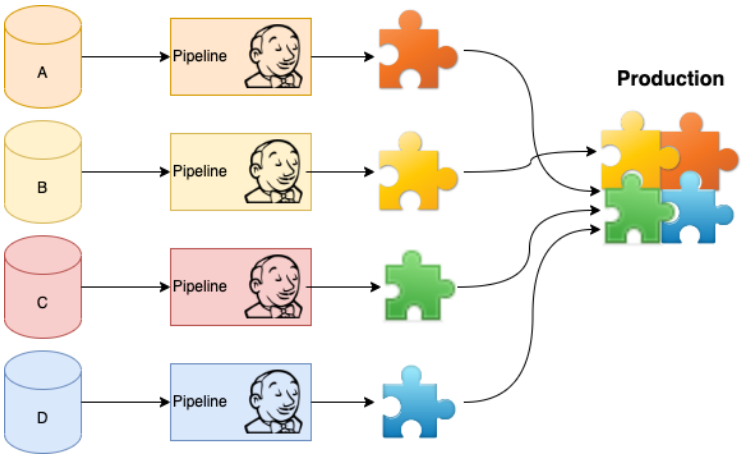

FIGURE 5. Independent deployments

Best coding practices need to be put in place to ensure that even though code bases are separated, the quality of the code and consistency across the application is in place.

3.**3**.3. *Independent deployments.* The main advantage of micro frontends is independent deployability without getting stuck on the broader business process of deploying large applications.

Independent deployments allow more minor, frequent codebase updates. However, if something goes wrong with the targeted deployment, this will only impact one user-interface area[17], and rollback can be easily achieved to restore the functionality.

Release cycles of independent smaller applications can be intertwined. Whenever a microservice is working and ready to be shipped to production, it should be independent of the other microservices' overall context and readiness state.

Figure 5 reflects the independence of deploying multiple parts of the same application into production to form the full functionalities.

3.**3**.4. *Autonomous teams.* Having decoupled the codebases and the release cycles, teams achieve independency to self-organize [18] and deliver their code to the end user whenever it is production ready. Of course, broader business and marketing strategies must be considered.

Teams become total owners of their code quality, business logic, framework, and styling particularities, as a larger framework does not constrain them.

Figure 6 reflects the splitting of layers between cross-functional teams working on the same product.

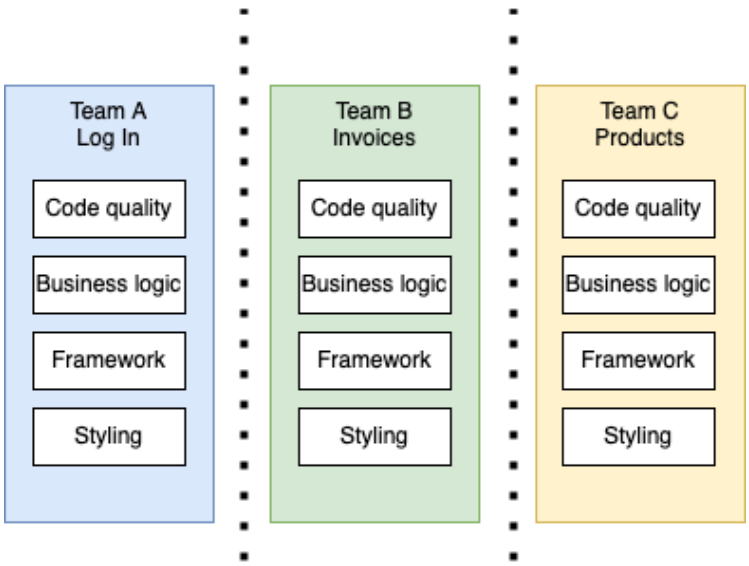

FIGURE 6. Team ownership

3.4. **Micro-frontend solutions**

Several micro-frontend solutions have emerged in recent years, each with benefits and drawbacks. There are multiple out-of-the-box solutions present on the web such as **SingleSPA** or **NX** but also many solutions can be built from scratch. Some of the current solutions are:

- **Routing** Each route represents a different micro frontend
- **Iframe** Each micro-frontend is included in the application via a frame inside a base application
- **Web components** A set of standardized browser technologies that enables the creation of modular components and are framework-agnostic
- **Module federation** Enables the dynamic loading of code and resources

From a performance perspective, web components and module federation perform better than iframe and routing solutions in terms of load time and speed. However, we can expect a higher initial load time for web components since all dependencies need to be loaded up-front. Module federation has an increased performance for resource sharing, and scalability as it allows dynamic loading of code across different parts of the application.

Web components and module federation tend to be the most tech-agnostic since they can be used with a variety of frameworks. However, since routing and iframe don't necessarily rely on the parent to display content, they can also be technologically agnostic.

Routing and Iframe require the least amount of specialized developer knowledge as they can be developed independently regardless of the technology chosen. Web components and module federation require a literature study for in-depth knowledge.

Resource sharing performs less with routing and iframe since in some scenarios files and resources need to be duplicated since the micro-frontends can work independently.

User experience is impacted mostly on routing and iframe solutions as they either require a full page reload or a specific part of the page to become unavailable until the resources are loaded. In module federation and web components, the user experience is not impacted as long as development discipline is in place.

4. **Research methodology**

A thorough research methodology was employed to compare the advantages of various micro-frontend solutions. The methodology included a comprehensive analysis of prior research and publications on the subject and analyzing third-party providers' documentation for existing micro-frontend solutions.

Identifying the different types of micro-frontend solutions to be evaluated was the initial phase in the research methodology. Among these were routing, iframe, web components, and module federation.

The second phase of the research methodology involved establishing the evaluation criteria for each micro-frontend solution. Time required for the first paint, number of requests, total resource consumption, and load time were identified as the criteria. These criteria were chosen based on their significance to the micro-frontend solution's overall success.

By comparing the advantages and disadvantages of each solution concerning the identified criteria, we determined which solution would be most appropriate for migrating an application to a micro-frontend architecture.

The implementation of a simple web application has taken place as part of this study in order to compare two micro-frontend solutions with the monolith approach. Similar variations of the application have been implemented using iframes and module federation for comparison. Figure 7 portrays the application implemented. The turquoise and mauve blocks represent micro-frontends for the iframe and module federation implementations. The navigation bar and the top-right block belong to the shell application, which subsequently loads the modules. Using the practical implementation we acquired information and insights regarding the performance of each micro-frontend solution in meeting these criteria.

Relevance of the observed criteria:

- **First paint** Time spent until the full page is displayed to the user and is relevant for the general user experience when opening the website. It is an indicator of how quickly the website becomes visually interactive and provides feedback to the user.
- **Requests** Number of network requests made by the application. Each request made by the browser to fetch a specific resource (such as HTML, CSS, JavaScript files, images, etc.) introduces additional overhead and

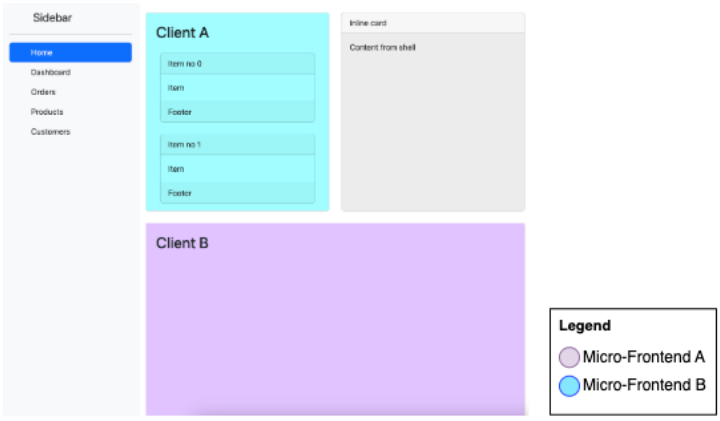

FIGURE 7. Simple application layout

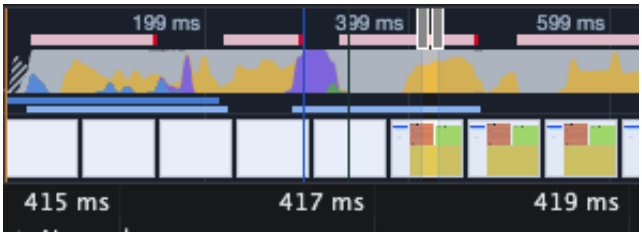

FIGURE 8. Analysis of the first content paint time using Chrome inspector

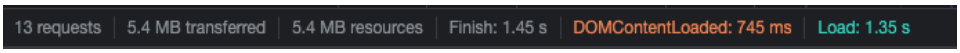

FIGURE 9. Analysis of resources consumption with Chrome inspector

can impact the overall load time and user experience. Handling a larger number of requests may imply additional server costs as well.

- **Resources** Total size of network requests. This metric directly impacts the time it takes for a website to load.
- **Load time** Time spent until all resources for the page have been loaded and it represents the time it takes for a website to fully load and become usable for the user.

Analyzing the implemented solution using the browser inspection tools available. Figure 8 is an example of the first content paint analysis, while figure 9 represents an analysis of the resources loaded for each implemented solution.

5. Results

Several prospective research findings regarding the advantages and disadvantages of various micro-frontend solutions can be deduced from the evaluation criteria enumerated in the research methodology.

We can calculate the relative improvement using the Equation 1 formula, where N represents the new value and O represents the original value.

$$RI = \frac{N - O}{O} \tag{1}$$

Analyzing the information resulting from implementing a simple web application as seen in table 1 and figure 10, we can observe that the module federation solution outperforms the iframe implementation. Figure 10 presents a radar chart in which values obtained from the analysis and outline in table 1 are compared between the three implementations.

All values have been normalized to have a similar graphical representation between the four metrics while maintaining relativity for the three solutions studied.

Regarding the time required for painting the application, loading it, the resources it consumes, and the requests made, the best result is the closest to zero, or the center of the radar chart. We can assume that for larger projects or codebases, the first paint time will increase for the monolith solution as the total bundle size increases.

Module federation shows a 55% decrease in time spent for the **first paint** as compared with iframe and a 29% increase in time spent compared to a monolith application. Still, taking into consideration that the first paint is dependent on the bundle size, for larger applications, the first paint time will increase compared to module federation which leverages asynchronous module loading in which modules are loaded on the fly depending on the need of the user.

The results show that the **bundle size** is reduced in module federation compared to the iframe solution. Resource duplication in an iframe and routing solution is often inevitable, thus, increasing the bundle size. Analyzing the results obtained and listed in table 1, we can observe a 60% decrease in bundle size for the module federation compared with the iframe implementation. Nevertheless, the module federation implementation shows a 22% increase in bundle size compared to the monolith approach.

In terms of the **number of requests** and the total size of resources loaded, module federation outperforms the iframe implementation but is less performant than the monolith implementation. Iframe implementation shows a significant increase compared to the monolith approach and monolith approach, mostly since resources are not shared between the different parts of the application. Module federation shows a 23% decrease in the number of requests as compared to the iframe implementation and a 100% increase in the number of requests compared with the monolith solution.

Load time is calculated based on the initial load of the resources and since module federation asynchronously loads secondary modules, there is a significant improvement in speed as compared to monolith and iframe. The

TABLE 1. Comparison of resources used by each type of application

Type/Criterion	**Monolith**	**Iframe**	**Module**
First paint	418ms	1222ms	540ms
Requests	13	34	26
Resources	5.4MB	16.6MB	6.6MB
Load Time	1.35s	1.12s	0.774s

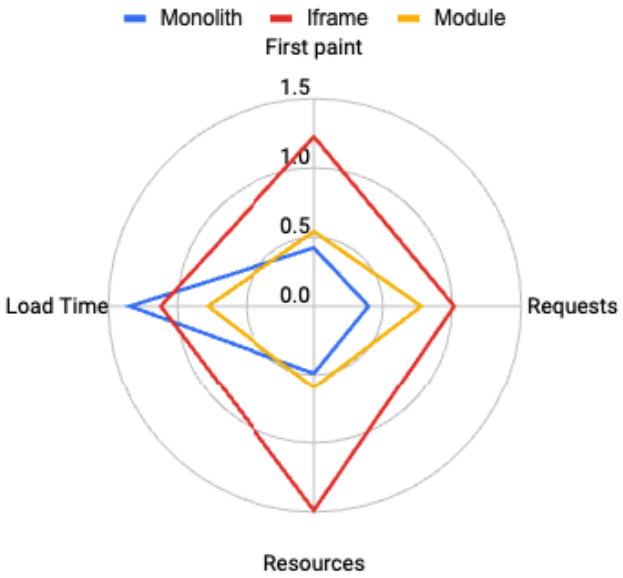

FIGURE 10. Relative Comparison between implemented solutions

results show a 30% decrease when comparing module federation load time to the iframe and a 42% decrease when compared to the monolith approach.

Even though the results of the module federation solution show a lack in performance on some specific chapters for the studied implementation, we can assume that for larger applications, it will outperform the monolith approach. Achieving decoupled codebases that work together to form an entire application offers teams independence and scalability without decreasing performance or degrading the user experience.

6. **Conclusions**

As time passes, applications become more complex to fulfill user and market demands, code complexity increases, and the release process becomes complicated with multiple variables and interdependencies between teams, layers, and business feature release timings.

In order to follow a clean approach and allow for easier scaling and team independency, cross-functional teams working on independent pieces of the

same applications must be applied by encouraging microservices and micro-frontends.

Based on the analysis conducted in this paper, it can be concluded that micro-frontends offer a promising solution for overcoming the challenges posed by monolithic front-end applications. By dividing the front-end applications into smaller, more manageable pieces, development teams can work more efficiently, resulting in shorter development cycles and enhanced agility.

While using micro-frontends is still a relatively new architectural pattern, the results presented in this paper suggest that they hold significant promise for the future of front-end development. Choosing the right architectural pattern or solution should be dictated by a comprehensive analysis of the entire application code along with an analysis of team competencies, shared resources, and other company applications with which the code can be shared.

The results obtained are an intriguing expansion for this study, and the authors want to broaden the outcomes of the current work by conducting a comprehensive investigation on the impact on performance for enterprise-grade applications with larger codebases.

REFERENCES

[1] Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Rahina Oumarou Mahamane, Pascal Zaragoza, and Christophe Dony. From monolithic architecture style to microservice one based on a semi-automatic approach. In *2020 IEEE International Conference on Software Architecture (ICSA)*, pages 157–168. IEEE, 2020.

[2] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.

[3] Miika Kalske et al. Transforming monolithic architecture towards microservice architecture. Master's thesis, University of Helsinki, 2018.

[4] Chen-Yuan Fan and Shang-Pin Ma. Migrating monolithic mobile application to microservice architecture: An experiment report. In *2017 ieee international conference on ai & mobile services (aims)*, pages 109–112. IEEE, 2017.

[5] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. Migrating from monolithic architecture to microservices: A rapid review. In *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–7. IEEE, 2019.

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto L Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Lara Safina, and Gianluigi Zavattaro. Microservices: yesterday, today, and tomorrow. *Communications of the ACM*, 60(6):36–44, 2017.

[7] Hulya Vural, Murat Koyuncu, and Sinem Guney. A systematic literature review on microservices. In *Computational Science and Its Applications–ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part VI 17*, pages 203–217. Springer, 2017.

[8] Daojiang Wang, DongMing Yang, Huan Zhou, Ye Wang, Daocheng Hong, Qiwen Dong, and Shubing Song. A novel application of educational management information system based on micro frontends. *Procedia Computer Science*, 176:1567–1576, 2020. Knowledge-Based and Intelligent Information and Engineering Systems: Proceedings of the 24th International Conference KES2020.

[9] Calin CONSTANTINOV, Lucian IORDACHE, Adrian GEORGESCU, Paul-Stefan POPESCU, and Mihai MOCANU. Performing social data analysis with neo4j: Workforce trends & corporate information leakage. In *2018 22nd International Conference on System Theory, Control and Computing (ICSTCC)*, pages 403–406, 2018.

[10] Anna Montelius. An exploratory study of micro frontends. Master's thesis, Linköping University, Software and Systems, 2021.

[11] Severi Peltonen, Luca Mezzalira, and Davide Taibi. Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review. *Information and Software Technology*, 136:106571, 2021.

[12] Michael Geers. *Micro frontends in action*. Simon and Schuster, 2020.

[13] Andrey Pavlenko, Nursultan Askarbekuly, Swati Megha, and Manuel Mazzara. Micro-frontends: application of microservices to web front-ends. *J. Internet Serv. Inf. Secur.*, 10(2):49–66, 2020.

[14] Luca Mezzalira. *Building Micro-Frontends*. " O'Reilly Media, Inc.", 2021.

[15] Caifang Yang, Chuanchang Liu, and Zhiyuan Su. Research and application of micro frontends. *IOP Conference Series: Materials Science and Engineering*, 490:062082, 04 2019.

[16] Davide Taibi and Luca Mezzalira. Micro-frontends: Principles, implementations, and pitfalls. *SIGSOFT Softw. Eng. Notes*, 47(4):25–29, sep 2022.

[17] Emilija Stefanovska and Vladimir Trajkovik. Evaluating micro frontend approaches for code reusability. In *International Conference on ICT Innovations*, pages 93–106. Springer, 2022.

[18] P Yedhu Tilak, Vaibhav Yadav, Shah Dhruv Dharmendra, and Narasimha Bolloju. A platform for enhancing application developer productivity using microservices and micro-frontends. In *2020 IEEE-HYDCON*, pages 1–4. IEEE, 2020.