International Journal of Scientific Engineering and Science
Volume 9, Issue 3, pp. 37-40, 2025. ISSN (Online): 2456-7361

Modern Approaches to Web Application Architecture: Microservices and Micro Frontends

Chmelev Andrei
Wildberries LLC, Moscow, Russia

Abstract—*This review article examines the evolution of microservice architecture and the transition from monolithic frontend applications to micro frontends. Special attention is paid to a comparative analysis of orchestration methods, service interaction, and version control, as well as a detailed assessment of performance, development convenience, and maintenance. The work includes extended conceptual diagrams illustrating the principal differences and common features of the two approaches. In addition, a table is presented that visually compares the key characteristics of monolithic applications, microservices, and micro frontends. This article serves as an overview of existing practices and helps determine which tools and technologies are best suited for building modern distributed web applications.*

Keywords— *Microservices, Micro Frontends, Web Application Architecture, Orchestration, Versioning, Performance Analysis, Monolithic Applications.*

I. Introduction

Modern web development often faces conflicting requirements: high scalability, reliability, rapid feature delivery, and ease of maintenance. Traditional monolithic architectures, while simpler to deploy at early stages, frequently become bottlenecks as projects expand [5]. They limit the ability to scale individual components and slow down the release cycle when any single part of the application needs updates. In response, microservice architecture emerged as a means to separate a system into small, autonomous services that can independently evolve and deploy [1], [2].

However, the monolithic paradigm also persists in frontend development, where large Single Page Applications (SPAs) accumulate a significant amount of code, resulting in extended loading times, complex maintenance, and challenges for distributed teams [3]. The concept of micro frontends addresses these issues by applying microservice principles to the client side, thereby splitting the application into smaller, independently deployable UI modules. Ensuring organizational alignment across these teams often follows guidelines similar to those discussed in [4].

Although focusing mainly on software architecture, some classic works have also touched upon network-intensive or infrastructure-heavy designs, which can tangentially inform microservice and micro-frontend strategies. For instance, the discussion in [6] on infrared navigation hardware systems highlights the importance of scalability in complex environments, and [7] examines electromagnetic structures that necessitate robust distributed computing. Likewise, earlier conferences explored how bandwidth constraints and specialized hardware might shape an application’s architecture [5].

This paper reviews the key aspects of microservice architecture, explores how micro frontends extend these ideas to the frontend layer, and provides an in-depth discussion on orchestration, interaction, version management, and the impact on both performance and team organization. By comparing monolithic approaches, microservices, and micro frontends, it illustrates how modern distributed architectures help achieve greater flexibility, scalability, and maintainability for large-scale web projects.

II. Evolution of Microservice Architecture

A. *Transition from Monolithic Applications to Microservices*

In a classic monolithic architecture, the entire backend application runs as a single process, with modules and layers logically separated in the code but physically remaining part of a single application. As the number of users and functional requirements grows, the monolith becomes cumbersome, making it difficult not only to scale but also to release new versions. A pivotal step in solving this issue was the shift toward microservice architecture, in which the system is divided into a set of small, autonomous services, each responsible for a specific business function [1]. This separation facilitates independent development and release of services, providing a more flexible update cycle.

B. *Technology Stack and Patterns*

There is a wide range of tools and patterns that help implement microservice architecture. Popular solutions include Docker and Kubernetes, which allow packaging services into containers and managing their lifecycle while also offering automatic scaling. Additional capabilities are provided by Service Mesh solutions (e.g., Istio or Linkerd), which handle distributed policies, security, and tracing. The API Gateway concept enables consistent routing of requests, as well as centralized authentication and authorization. In some cases, a message broker (RabbitMQ, Kafka) is used to implement asynchronous communication, facilitating event exchange between services and simplifying coordination.

C. *Benefits and Challenges in Implementing Microservices*

One of the key benefits of this approach is the ability to perform independent releases and scale services individually, as each team can use the technology stack and development methodology most suited to its needs. Moreover, system growth becomes more manageable: changes in one service do not directly affect the functionality of others. However, the distributed nature of microservices also introduces new issues.

Network communication adds overhead, complicating tracing and debugging, while a decentralized architecture demands more advanced

DevOps practices for monitoring, logging, and orchestration. Incorrectly defined service boundaries can lead to interdependencies, resulting in a "distributed monolith" that fails to resolve the original scalability problems. Figure 1 shows one possible arrangement of microservices via an API Gateway. Each request is routed to the appropriate service, while logging is centralized for easier monitoring and debugging.

For more advanced scenarios, a sequence of calls can be traced with the help of service mesh tooling or specialized logging. An example is shown in Figure 2, illustrating how user requests might flow between various microservices during an "order placement" process.

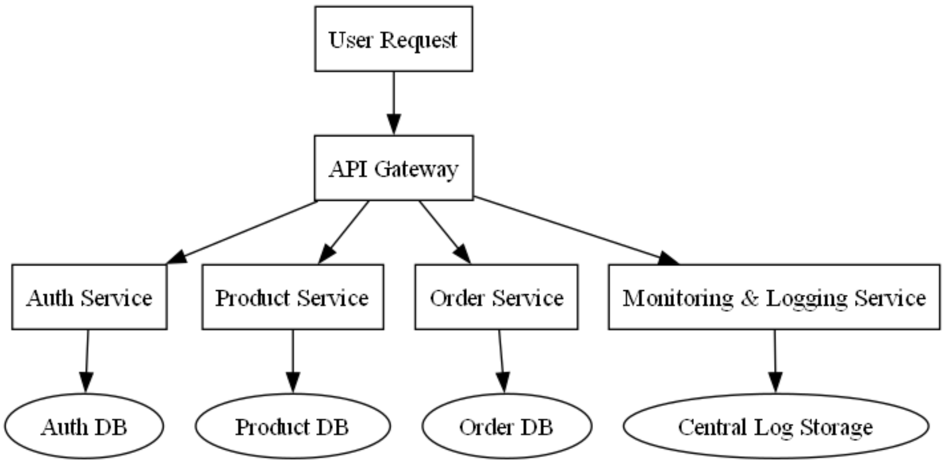

Fig. 1. Example of microservice orchestration through an API Gateway with centralized logging.

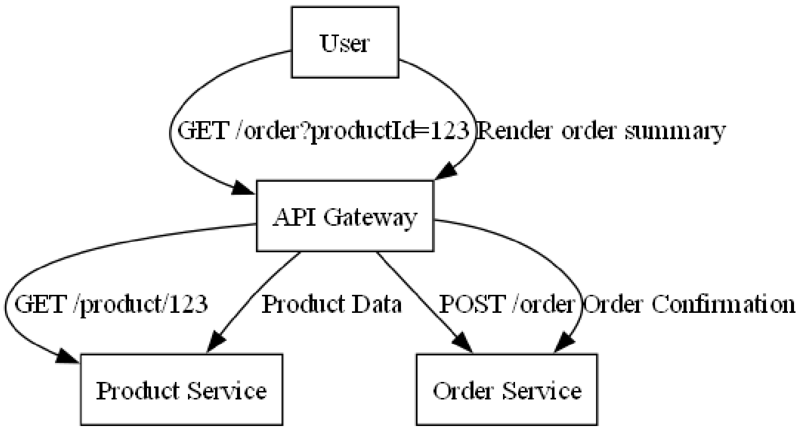

Fig. 2. Example of sequential service calls when placing an order.

III. Transition to Micro Frontends

A. *Challenges with Monolithic SPA Applications*

In the frontend, developing large Single Page Applications (SPAs) remained relevant for a long time. SPAs provide a convenient user experience and high interactivity but, with rapid growth of code and features, lead to difficulties in maintenance and testing. Sometimes even a minor update requires redeploying the entire frontend, and the increasing bundle size degrades performance, especially on clients with slower connections. When large teams work on a single application, synchronization and release speed can suffer significantly.

B. *The Concept of Micro Frontends*

The idea of micro frontends extends microservice principles to the client side. It assumes that a large frontend is split into independent modules, each with its own technology stack, lifecycle, and development team. As a result, the project becomes more flexible since each functional area can evolve independently and release its updates without blocking others. In practice, ensuring a consistent user experience is particularly important so that transitions among micro frontends remain seamless.

For this, the "root" part of the application (often called the Shell) is equipped with routing and a shared UI framework to keep site-wide styling and the header consistent throughout the system.

C. *Common Approaches to Implementing Micro Frontends*

In everyday practice, there are several ways to combine micro frontends into a cohesive whole. Server-Side Composition implies that individual modules are assembled on the server during rendering, using ESI (Edge-Side Includes) or other mechanisms. Client-Side Composition focuses on loading modules directly in the user's browser, sometimes using iframes or Web Components, including Module Federation in Webpack. Build-Time Integration involves jointly building multiple parts into one bundle, provided each team independently develops its segment of the code.

Figure 3 presents one possible schematic of how micro frontends can be composed together. Each module manages a distinct area of the UI but shares common resources, such as styles or libraries, to ensure a unified look and feel.

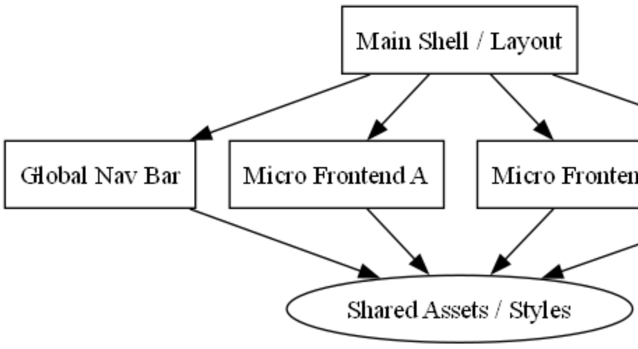

Fig. 3. High-level representation of micro frontends combined by a shell.

IV. Orchestration, Interaction, and Versioning

Orchestration in Microservices

Orchestration is designed to simplify the lifecycle management of multiple services. Kubernetes or Docker Swarm can automate scaling, self-healing, and rolling updates. For microservices, especially in larger projects, a Service Mesh is often used to centrally manage routing, encryption (mTLS), and observability. This approach significantly eases debugging by adding a unified layer for logging, metrics, and request tracing.

Interaction of Micro Frontends

For micro frontends, orchestration comes down to how the user interface is assembled from separate parts into a unified application. The central Shell or Layout not only handles routing but also provides a shared visual environment (header, footer, navigation bar). It is crucial for users that styles, interface elements, and authentication remain consistent. Each micro frontend team can use its own technology stack, provided it does not hinder overall composition. Data exchange among independent modules is handled through events, shared services, or a global store, depending on the project's specifics.

Versioning and Release Management

A common method of coordinating versions in microservices and micro frontends is semantic versioning (semver), which helps clarify compatibility levels. Many companies use Feature Toggles, where new features are initially hidden for most users, making it possible to test them and disable them if needed without halting the entire service. Canary and Blue-Green Deployment allow testing new releases on a subset of users or running two parallel environments, thereby reducing risk during updates. In micro frontends, Module Federation is also frequently used to dynamically load different module versions on the client side.

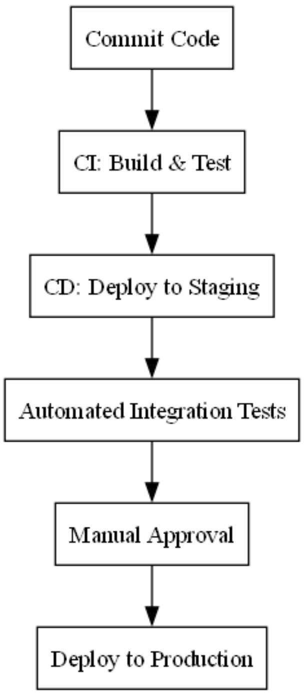

Fig. 4. A typical CI/CD pipeline for microservices and micro frontends.

To ensure code quality and automate deployment, many organizations incorporate a CI/CD pipeline for both microservices and micro frontends, as shown in Figure 4. In this example, commits trigger automated builds and tests, followed by staged deployments and final approval before production release.

V. Performance, Development, and Maintenance Analysis

Performance

A distributed architecture with microservices can increase latency due to numerous network calls. On the other hand, it allows more precise scaling of individual services, potentially saving resources. In micro frontend projects, frontend performance depends on the number of components loaded simultaneously and the degree to which the overall bundle size grows. Proper planning and configuration of caching, compression, and the use of a CDN can minimize the negative impact on web application speed.

Development Convenience and Team Organization

One of the most notable reasons for adopting a distributed architecture is the convenience of parallel development. Each team can manage its own releases and updates, with the only coordination point being the shared interaction interfaces. The flexibility to choose languages, frameworks, and libraries for specific needs allows adaptation of the tech stack to different tasks. However, maintaining unified styles and UI patterns requires additional agreements and oversight, especially when multiple teams—potentially dozens of specialists—are working on separate micro frontends or microservices.

Maintenance and Future Evolution

As the system grows, maintenance and version control become more complex. Infrastructure costs rise due to Kubernetes, logging systems, and monitoring solutions. Nevertheless, well-designed service boundaries and modularization often pay off by enabling faster rollout of new features and reducing interdependencies among parts of the application. The evolution of such projects becomes easier if documentation on interfaces is maintained and consistent standards exist for coding, security, and testing. It is important to remember that choosing a "distributed" approach is a strategic decision involving investment in DevOps and infrastructure, so that the flexibility and scalability benefits can be realized over time. Table I compares the key characteristics of monolithic applications, microservices, and micro frontends.

TABLE I. Comparison of Key Approaches

Characteristic	**Monolith**	**Microservices**	**Micro Frontends**
Architecture Complexity	Low (single codebase)	High (multiple services, network interactions)	Medium (several UI modules, each is somewhat self-contained)
Scalability	Limited (scales as a whole)	High (each service can be independently scaled)	Medium (only certain frontend modules can be scaled, but they still depend on backend)
Release Cycle	Single release cycle, often bulky	Independent releases for each service, faster feature delivery	Independent release of individual frontend modules, reducing risk of breaking the entire UI
Technology Autonomy	Typically one technology stack	Freedom to choose different languages and frameworks per service	High flexibility (each team can choose its own framework for the micro frontend)
Maintenance Complexity	Simpler environment (just one project)	Complex infrastructure requiring advanced DevOps practices	Requires coordination among multiple modules and consistent UI styling

VI. Conclusion

This review article has highlighted the key aspects of web application evolution from monolithic architecture to microservices, as well as the specifics of transitioning to micro frontends for the client side. Microservice architecture simplifies scalability and optimizes collaboration among teams, reducing the risk that changes in one module might break the entire application. At the same time, micro frontends address similar problems at the frontend layer, splitting large, monolithic SPA applications into self-contained modules with their own release lifecycles.

Despite the numerous advantages, a distributed design approach requires advanced solutions for orchestration, monitoring, and maintaining version compatibility. In the case of micro frontends, there is a need to coordinate the efforts of multiple teams responsible for different parts of one interface. Well-defined boundaries for services and micro frontends, along with a mature DevOps culture and testing practices, enable realization of the main benefits of these approaches, providing a more flexible, reliable, and easily scalable web application architecture. Earlier explorations of network-based system evolution, such as those in [5], [6], and [7], underscore the importance of balancing infrastructure complexity with application needs—an insight that remains relevant when scaling modern distributed architectures.

Acknowledgment

The author extends gratitude to colleagues who provided expert review and support during the preparation of this material, as well as all members of the development team who participated in discussions and peer reviews. Special acknowledgment goes to the scientific community for the ongoing exchange of experience and best practices in the fields of microservices and micro frontends.

References

[1] M. Fowler, "Microservices: a definition of this new architectural term," [Online]. Available: https://martinfowler.com/articles/microservices.html
[2] S. Newman, *Building Microservices*, 2nd ed., O'Reilly Media, 2021.
[3] L. Richardson and M. Fulton, "Micro Frontends in Action," *Manning Publications*, 2021 (to be published).
[4] M. Heath and L. Porcaro, *Team Topologies: Organizing Business and Technology Teams for Fast Flow*, IT Revolution Press, 2019.
[5] D. B. Payne and J. R. Stern, "Wavelength-switched passively coupled single-mode optical network," in *Proceedings IOOC-ECOC*, pp. 585–590, 1985.
[6] J. U. Duncombe, "Infrared navigation—Part I: An assessment of feasibility," *IEEE Transactions Electron Devices*, vol. ED-11, no. 1, pp. 34–39, 1959.
[7] R. J. Vidmar, "On the use of atmospheric plasmas as electromagnetic reflectors," *IEEE Transactions on Plasma Sciences*, vol. 21, issue 3, pp. 876–880, 1992.