2026 Volume: 02 Issue: 04 www.ijrpa.com ISSN 2456-9995 Review Article

International Journal Research Publication Analysis

Page: 01-08

MICRO-FRONTENDS & SCALABLE ARCHITECTURES: IMPROVING DEVELOPMENT AGILITY IN FULL-STACK APPLICATIONS

***Rajani Thakan, Anusha Raj, Er. Ram Babu, Dr. Vishal Shrivastava, Dr. Akhil Panday**

Computer Science & Engineering, Arya College of Engineering & IT, Jaipur, India.

Article Received: 03 March 2026
Article Revised: 21 March 2026
Published on: 11 April 2026

***Corresponding Author: Rajani Thakan**
Computer Science & Engineering, Arya College of Engineering & IT, Jaipur, India.

DOI: https://doi-doi.org/101555/ijrpa.4302

ABSTRACT

Modern full-stack applications are large and complex. Traditional monolithic frontend applications cause issues like slow builds, long release cycles, and tight team coupling. Micro-frontend architecture solves these challenges by dividing the frontend into smaller independent parts developed and deployed separately. This paper explains micro-frontends, their tools and languages, benefits, challenges, and migration roadmap. Case studies show how companies improve agility and scalability using this approach.

INTRODUCTION

Traditional frontend development often uses a **monolithic architecture** where all features exist inside a single large application. At first, this design looks simple to manage because there is only one codebase, one deployment pipeline, and one technology stack. However, as applications grow, this model begins to create significant challenges. Build times become very long, testing slows down, and deployments become risky because even a small change in one part of the

application requires redeploying the whole system. If one feature fails, such as the cart or login system, the entire application can stop working. This high dependency between modules makes the system fragile and less scalable.

Micro-Frontend Architecture (MFA) was introduced as a solution to these

problems. The idea is to break down a large frontend into smaller, self-contained applications called **Micro-Frontends (MFEs)**. Each micro-frontend focuses on a

single business feature, such as Catalog, Cart, or Payment. These MFEs are developed and maintained independently by different teams. Later, they are integrated into a single seamless

user interface that looks like one complete application to the end user.

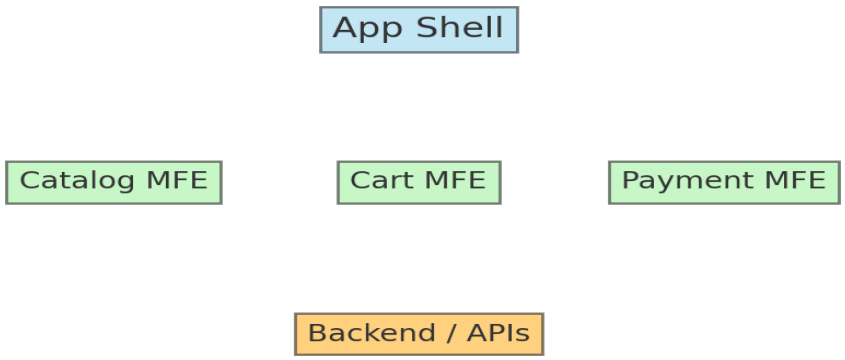

Figure 1: Micro-Frontend Architecture.

Literature Review & Background

Monolithic Frontend Architecture:

For many years, frontend development has been dominated by the **monolithic model**, where all features, UI components, and logic are stored in a **single large codebase**..

One of the biggest challenges is **maintainability**. With thousands of lines of code and multiple features tightly coupled together, even a small change in one part of the codebase can unintentionally break another part of the system.

Another drawback is **risk of failure**. In a monolithic architecture, if one module (for example, the checkout process) experiences a bug, it can affect the entire system. This lack of isolation increases downtime and impacts the end-user

experience. Additionally, monolithic frontends often enforce a **single technology stack**, which limits flexibility and slows down innovation.

Micro-Frontend Architecture:

The frontend is divided into smaller, self-contained applications called micro- frontends. Each MFE handles one specific business domain such as product catalog, shopping cart, payment system, or user profile.

The most important characteristic of micro-frontends is **independent builds and deployments**. Each micro-frontend can be built, tested, and deployed separately, without requiring a full rebuild of the entire application.

Another benefit is **team autonomy**. In traditional monoliths, all developers contribute to the same codebase, often leading to conflicts and slower progress.

The concept also promotes **fault isolation**. Since each micro-frontend runs independently, if

one module fails, it does not bring down the entire system. Figure 1: Micro-Frontend Architecture.

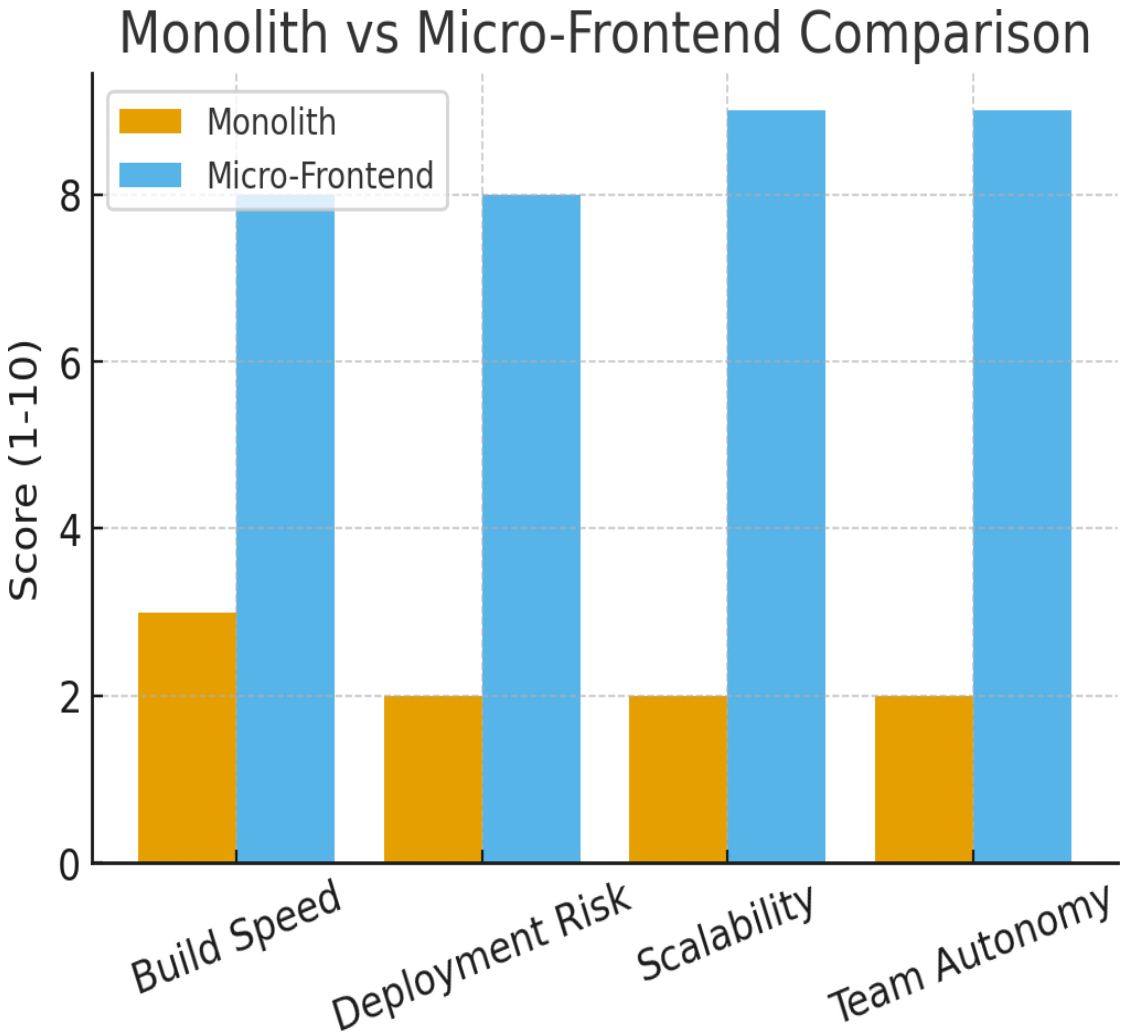

Figure 2: Monolith vs Micro-Frontend Comparison Chart.

Proposed Architecture & Tools

A micro-frontend system typically has an App Shell that handles global navigation, authentication, and layout. Individual MFEs (like Catalog, Cart, and Payments) plug into this shell. Each MFE communicates with its own backend or a Backend-for-Frontend (BFF). A GraphQL gateway or REST APIs connect the frontends with underlying services and databases.

A modern micro-frontend architecture not only focuses on splitting the user interface into smaller applications but also emphasizes **seamless integration, security, and scalability**. The **App Shell** acts as the backbone, ensuring consistent navigation, authentication, and shared layout elements. This shell can also manage **shared libraries** (for example, a design system or utility functions) to maintain a consistent user experience across multiple MFEs.

Communication between MFEs and backend services is handled through **API gateways**, either REST or GraphQL. In some cases, a **Backend-for-Frontend (BFF)** layer is introduced to tailor APIs specifically for each MFE, reducing complexity and network overhead.

Languages and Tools Used:

- Frontend: JavaScript, TypeScript, HTML, CSS
- Frameworks: React, Angular, Vue
- Backend: Node.js, Java, Python
- Databases: MySQL, MongoDB
- DevOps: Docker, Kubernetes, GitHub Actions, Jenkins
- Monitoring: Prometheus, Grafana, Sentry

Benefits of Micro-Frontends

Independent Deployments Reduce Risk

Each MFE can be deployed separately, minimizing the chances of breaking the entire system.

Teams Work Autonomously with Their Own Tech Stacks

Different teams can choose the framework or library that best fits their module (React, Angular, Vue, etc.).

Faster Recovery in Case of Failure

Fault isolation ensures that even if one module crashes, the rest of the application continues running.

Scalability as Features Grow

Applications can expand smoothly since MFEs scale independently.

Easier Adoption of New Frameworks

Individual MFEs can be upgraded to new frameworks without rewriting the whole app.

Parallel Development

Multiple teams can build and release features simultaneously, speeding up time- to-market.

Improved Maintainability

Smaller, modular codebases are easier to understand, debug, and maintain.

Enhanced User Experience

Updates and fixes can be rolled out faster with minimal downtime, improving reliability for end-users.

CHALLENGES & LIMITATIONS

Higher Initial Setup Cost

- Setting up multiple pipelines, repositories, and deployment environments requires more time and resources.
- Infrastructure and DevOps complexity is greater compared to monolithic apps.

Requires Strong Governance for Consistency

- Without proper design systems and coding standards, different teams may create inconsistent UI/UX patterns.
- Governance is needed to enforce common practices across teams.

Risk of Inconsistent User Experience

- If each team uses different frameworks, styling, or libraries, the application may look fragmented.
- Requires a shared design system to maintain uniformity.

Performance Issues

- Loading multiple MFEs in parallel can slow down initial page load.
- Requires optimization techniques like lazy loading, caching, and bundling.

Complex Communication Between MFEs

- Cross-module communication may become complicated.
- Event bus or shared state management tools are needed to avoid conflicts.

Study: E-Commerce Platform

E-commerce is one of the best domains to showcase the impact of micro- frontends because it contains multiple independent business functions. In a traditional monolithic frontend, modules like **Catalog, Cart, Checkout, and Payments** are tightly coupled. A small bug in the Cart could require
redeploying the entire application, leading to downtime and customer dissatisfaction.

By adopting a micro-frontend approach, the platform separated its interface into **Catalog, Cart, and Payment MFEs**. Each team managed its own MFE with separate build pipelines, deployments, and monitoring dashboards. For example, the **Catalog team** focused on product listings and search
optimization, the **Cart team** managed user sessions and discounts, and the
Payment team integrated multiple payment gateways.

This modular approach led to measurable improvements. If a bug occurred in the Cart, only the Cart MFE needed redeployment, leaving Catalog and
Payments unaffected. This reduced downtime significantly and improved customer trust. In fact, deployment frequency increased **3x**, allowing faster delivery of new features like personalized recommendations and one-click checkout. Moreover, the mean time to recovery (MTTR) reduced by **25%**, highlighting faster issue resolution.

Migration Roadmap

Step 1: Identify Business Domains
Break the application into clear business areas (e.g., Catalog, Cart, Payment, User Profile).

Step 2: Choose Composition Strategy & Tool
Decide how MFEs will integrate: Webpack Module Federation, single-spa, iframes, or Web

Components.

Step 3: Set Up Core Infrastructure

Prepare DevOps pipelines, containerization (Docker), and orchestration (Kubernetes) for independent scaling.

Step 4: Convert One Feature into a Micro-Frontend (Pilot)

Start with a low-risk feature (e.g., Reviews or Search) to validate the architecture.

Step 5: Establish CI/CD Pipelines for Independent Deployments Automate build, test, and deployment pipelines for each MFE.

Step 6: Standardize Authentication & Security

Implement SSO (Single Sign-On), OAuth 2.0 / JWT, and shared authentication services.

Step 7: Create a Unified Design System

Ensure consistent branding, components, and styling across all MFEs with a shared UI library.

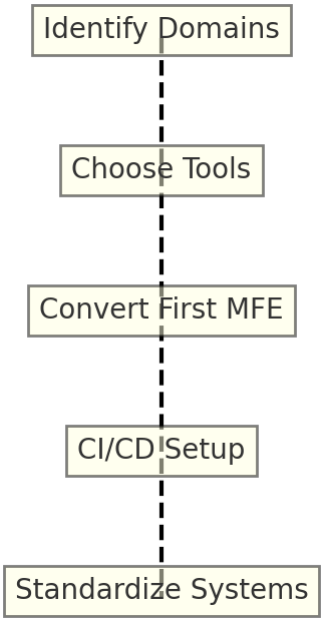

Figure 3: Migration Roadmap Flowchart.

CONCLUSION & FUTURE SCOPE

Micro-frontends offer a scalable and agile way to build large full-stack applications. They allow independent team ownership, reduce deployment risk, and improve performance. However, they require strong governance, performance optimization, and shared standards. Future research can explore AI-driven optimization of MFE performance and automated governance models

REFERENCES

1. Module Federation Documentation
2. single-spa Documentation
3. OWASP Frontend Security Guidelines
4. Prometheus and Grafana Monitoring Docs
5. Micro-Frontends.org