Decoding Frontend Evolution: From Monoliths to Micro-Frontend

Santhosh Podduturi

santhosh.podduturi@gmail.com

Abstract:
Frontend architecture has undergone significant transformations over the years, driven by the need for scalability, maintainability, and agility. In the rapidly evolving world of web development, the transition from monolithic architectures to fully distributed systems can be overwhelming. Many organizations attempt to leap directly from one to the other without considering the intermediary stages that could address their growing needs more effectively. This paper explores the evolution of frontend architecture, tracing its path from simple static pages to sophisticated full-stack monolithic systems, and eventually to distributed architectures. We will analyze the key trends, challenges, and future directions in frontend development. This helps guide organizations in navigating the various architectural patterns and challenges before committing to a fully distributed setup. By understanding the progression and intermediate alternatives, a clearer path to scaling web applications can be identified, helping to avoid the pitfalls of premature complexity.

Keywords: Micro-Frontend, Runtime Composition, Independent Deployments, Dynamic Integration, Frontend Modularity, Single SPA, Module Federation, Frontend Composition, Cross-Origin Resource Sharing (CORS), Dynamic Loading, Frontend Scalability, Agile Development, Continuous Delivery, Frontend Flexibility, Distributed Teams, Cross-Component Communication, Frontend Automation, Modular Web Applications, Seamless Integration, Deployment Independence, Code Reusability, Performance Overhead.

INTRODUCTION

In the journey of scaling web applications, many organizations face a critical decision: how to evolve from a monolithic architecture to something more scalable and flexible. The common mistake is rushing to a fully distributed system, skipping over the intermediate architectural options that might offer better solutions. A fully distributed architecture—often including micro-frontends—can provide significant scalability benefits, but it also introduces new complexities that require careful consideration [7].

This paper aims to provide insight into the **progressive decoupling spectrum**, a term used to describe the range of architectural options between the traditional monolith and fully distributed systems. The goal is to explore the available choices and help organizations make better decisions about how to scale their applications without falling into the trap of over-engineering [6].

The evolution of frontend architecture will be traced, from static pages to monolithic systems, and eventually to distributed architectures. Along the way, we will explore the pros and cons of each step in the spectrum, highlighting the alternatives that lie between monolithic and distributed architectures. By understanding these different options, businesses can avoid the pitfalls of jumping straight into complexity and instead select the path that best suits their current needs and future growth [3].

Early frontend architecture: Static and Server-Rendered Pages

In the early days of web development, frontends were primarily static HTML pages with minimal styling. JavaScript was limited in functionality, and server-side rendering (SSR) was the primary mechanism for generating dynamic content.

Static HTML and CSS

- Websites were built using plain HTML, CSS, and minimal JavaScript.
- Styling was applied using inline styles or external stylesheets.
- Every interaction required a full-page reload.

Server-Side Rendering (SSR)
- Technologies like PHP, ASP, and JSP allowed dynamic content generation on the server.
- Each user request led to a server computation, generating a new HTML page.
- This approach involved tightly coupling frontend and backend logic on the same server. It also had performance limitations due to server load and latency.

USE CASES:
Personal portfolios, marketing landing pages, documentation sites, temporary product promotions, and single-page scrolling websites.

Figure 1: Full Stack Monolith

The Monolith: The Beginning of the Journey
The journey of web development typically begins with the monolith. For most organizations, the monolith is the default starting point. If the architecture is unclear, it's likely that a monolithic system is being used. A monolith is defined as a **single unit of deployment**—one integrated codebase containing the frontend, backend, and database, all deployed together as one cohesive unit. This is the traditional "monolithic" approach.

Despite its reputation, the monolith is not inherently problematic. In fact, many successful companies rely on monolithic systems that scale to serve millions of users. However, the monolith is often misunderstood. It's frequently associated with legacy systems and outdated patterns, but this reputation is often undeserved. The main issue arises as the application grows. Once the complexity and scale increase, the monolith may become harder to manage, maintain, and scale to meet the evolving needs of the business. When this happens, it becomes necessary to explore new architectural approaches.

Exploring the Progressive Decoupled Spectrum
As the need for scalability and flexibility grows, many organizations consider moving away from the monolithic architecture. The problem is that most organizations jump straight to a fully distributed architecture without fully understanding the options that lie between the monolith and distributed systems. This is where

the **progressive decoupled spectrum** comes into play as shown in the picture —it represents the various options that organizations can explore before making the leap to a fully distributed architecture.

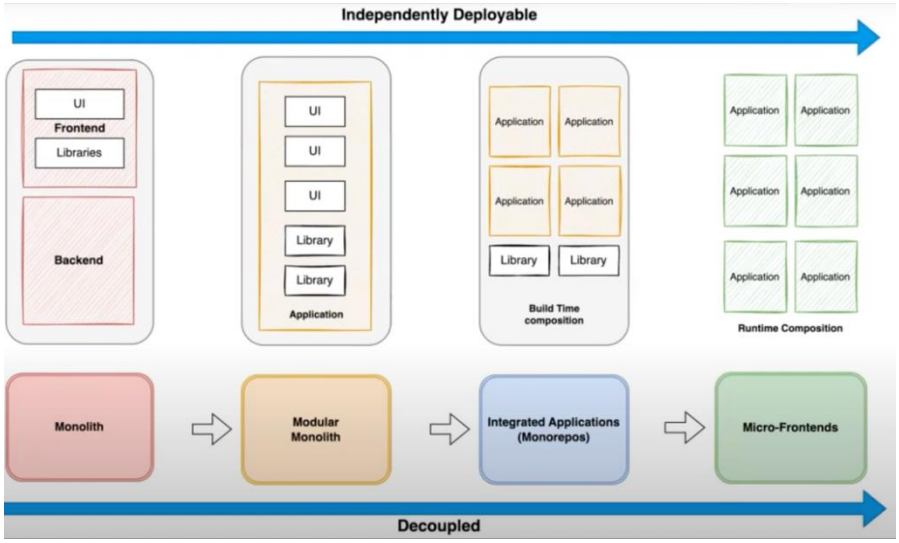

Figure 2: Progressive Decoupled Spectrum

Rather than jumping from 0 to 100 and moving directly to complex systems like micro-frontends, organizations should consider the alternatives that exist in between. These alternatives provide solutions for gradually decoupling parts of the application without the complexity of full distribution.

The Frontend Monolith

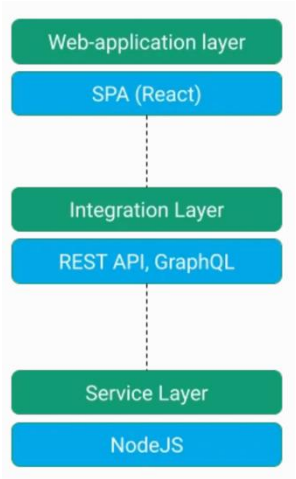

Figure 3: Frontend Monolith

The frontend monolith emerged as a result of the rise of microservices in backend development. While backend teams adopted the microservices approach, deciding to separate their services into independent units, the frontend largely remained unified, functioning as a monolithic system. As companies began to separate backend and frontend concerns, the frontend did not evolve at the same pace. Instead of following the trend of breaking things into smaller services, it retained its monolithic structure.

This led to what is now known as the frontend monolith, a particular flavor of monolithic architecture where Single Page Applications (SPAs) like Angular, React and Vue.js interact with backend services through APIs. The popularity of SPAs further reinforced the frontend monolith, as the backend could remain decoupled, communicating via APIs, while the frontend continued as a single, cohesive unit. [8]

Challenges with Monolithic Frontend Architectures

1. Code Maintainability Issues

- In monolithic architectures, frontend codebases become **large and difficult to manage** over time.
- **Example:** A large e-commerce platform like **Amazon** has thousands of UI components. In a monolithic frontend, even a small UI change (e.g., modifying the cart widget) requires modifying a **centralized, tightly coupled codebase**, leading to increased risk of breaking other parts of the application.
- Refactoring and debugging become time-consuming since changes can have **unintended side effects** across different modules.

2. Scalability Limitations

- A single, monolithic frontend makes **scaling development teams difficult**.
- **Example:** In a **video streaming service** like **Netflix**, different teams manage features like user profiles, recommendations, video playback, and payment processing. A monolithic structure forces all teams to work in a shared codebase, making **independent feature development and releases impossible**.
- Load balancing and performance optimizations also become challenging since scaling an entire frontend instance is **less efficient than scaling only specific features**.

3. Team Collaboration Bottlenecks

- When multiple teams work on the same monolithic frontend, **code conflicts and dependencies** slow down development.
- **Example:** A banking application with separate teams for customer dashboards, transactions, and loan management might face **coordination issues** when making changes, leading to **delayed releases**.
- **Parallel development** is hindered, making it harder to **experiment with new features** or **test alternative UI approaches**.

4. Technological Stagnation

- Monolithic architectures make **technology migration difficult** because the entire system is built on a single stack.
- **Example:** A social media platform initially built with **AngularJS** would struggle to migrate to **React or Vue.js**without rewriting the entire frontend.
- Developers are often forced to work with **outdated tools** since upgrading a monolithic application **requires extensive testing and rewrites**.

5. Deployment and Release Complexities

- **Any small UI change requires redeploying the entire application**, increasing deployment risks.
- **Example:** An **online marketplace** adding a new feature to the seller dashboard would have to go through **full regression testing** before deployment, slowing down time-to-market.
- CI/CD pipelines become inefficient because even minor updates **trigger a full build and test cycle**.

From Monolith to Modular Monolith: A Step Toward Scalability

As applications grow in size and complexity, maintaining a single, tightly coupled codebase becomes difficult. The modular monolith addresses these challenges by breaking down the monolithic structure into loosely coupled, independently developed modules, while still maintaining a single unit of deployment. This approach allows teams to organize their code more effectively, enhancing maintainability and scalability without introducing the overhead of a fully distributed system.

A modular monolith retains the **simplicity** of a monolithic application—one codebase, one deployment unit—but adds the **modularity** needed to better structure and scale the system. It's a stepping stone between the monolith and more complex, fully distributed architectures, like micro-frontends.

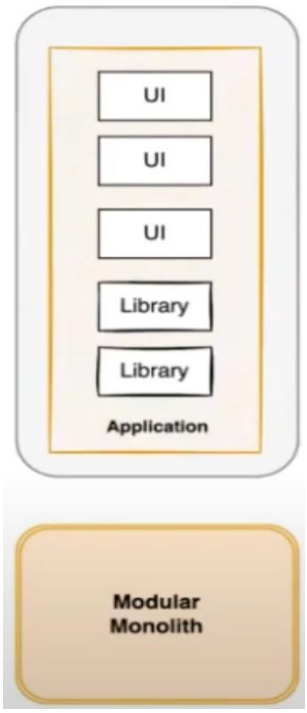

Figure 4: Modular Monolith

Key Features of a Modular Monolith

1. **Single Codebase**

Like the traditional monolith, all the frontend code resides in one place. However, in a modular monolith, it is organized into well-defined, independent modules. This setup improves the structure of the codebase and helps developers focus on specific features or areas of the application.

2. **Loose Coupling**

Each module in a modular monolith has distinct functionalities and minimized dependencies. By reducing the reliance on other modules, changes to one module have a minimal impact on others, improving flexibility and making the system easier to maintain.

3. **Improved Maintainability**

With clearly defined modules, a modular monolith enhances code organization and maintainability. The modular approach makes the codebase easier to understand, debug, and modify, as developers can work on isolated sections without fear of breaking unrelated features.

4. **Faster Development**

Compared to fully distributed systems (like micro-frontends), modular monoliths often allow for quicker development cycles. Since all the code resides in a single codebase, development, testing, and deployment processes are simpler and more streamlined.

Benefits of a Modular Monolith

1. **Simplified Deployment**

Since the entire application remains a single unit, deployment is simpler than managing multiple micro-frontends. There's no need to handle complex integration points between frontend modules.

2. **Better Initial Performance**

With a single codebase and minimal inter-module communication, modular monoliths often deliver better initial performance than micro-frontends. The lack of network overhead between modules enables faster rendering and response times.

3. **Reduced Complexity for Smaller Projects**

For small to medium-sized projects, a modular monolith strikes a balance between scalability and simplicity. It avoids the overhead of managing distributed systems while providing enough modularity to maintain clean code organization as the project grows.

Important Considerations for Modular Frontend Monoliths

1. **Module Boundaries**

Clear boundaries between modules are essential to maintain loose coupling. Defining these boundaries ensures that developers can work independently on different modules without affecting the overall application. Without well-defined boundaries, the system can become tightly coupled, defeating the purpose of modularity.

2. **Code Organization**

Organizing the code into logical, independent modules is critical for long-term maintainability. Developers should follow best practices for structuring modules and ensuring that they are easy to understand, modify, and extend.

3. **Potential for Scaling Issues**

While modular monoliths are suitable for smaller applications, they can face scaling challenges as the project grows. Managing a large, single codebase becomes difficult, and as new features are added, the risk of the application becoming too complex to maintain increases. In such cases, it may be necessary to consider transitioning to a fully distributed system.

4. **Technological Stagnation**

A modular frontend monolith still has the challenge of being tightly coupled to one technology stack, which can make it difficult to migrate to newer frameworks or tools. This leads to slower adoption of newer technologies since migrating a modular monolith is still resource-intensive. Teams may be forced to continue using older technologies to avoid the disruption of a large-scale refactor.

5. **Potential for Over-Modularization**

While modularity can improve organization, there is a risk of over-modularizing the frontend application, leading to excessive complexity without significant benefits. Over-modularization can lead to fragmentation of the codebase, making it harder to navigate and maintain. Developers might spend more time managing dependencies between excessively small modules rather than focusing on delivering new features.

Monorepo: Integrated applications

A monorepo is not the same as a monolith. It is a software development strategy where an entire codebase for a company or a set of micro-applications is stored in a single repository. However, there are several trade-offs to consider when adopting a monorepo approach.

Monorepos allow for the organization of multiple applications and their dependencies within a single repository. Applications in a monorepo can share and reuse components, libraries, and configurations. This setup fosters a collaborative organizational culture, as code sharing within the monorepo means that the challenges faced by different teams are also shared. This makes it easier to address and solve problems collectively. Additionally, maintaining coding standards and ensuring code quality becomes more straightforward in a monorepo environment.

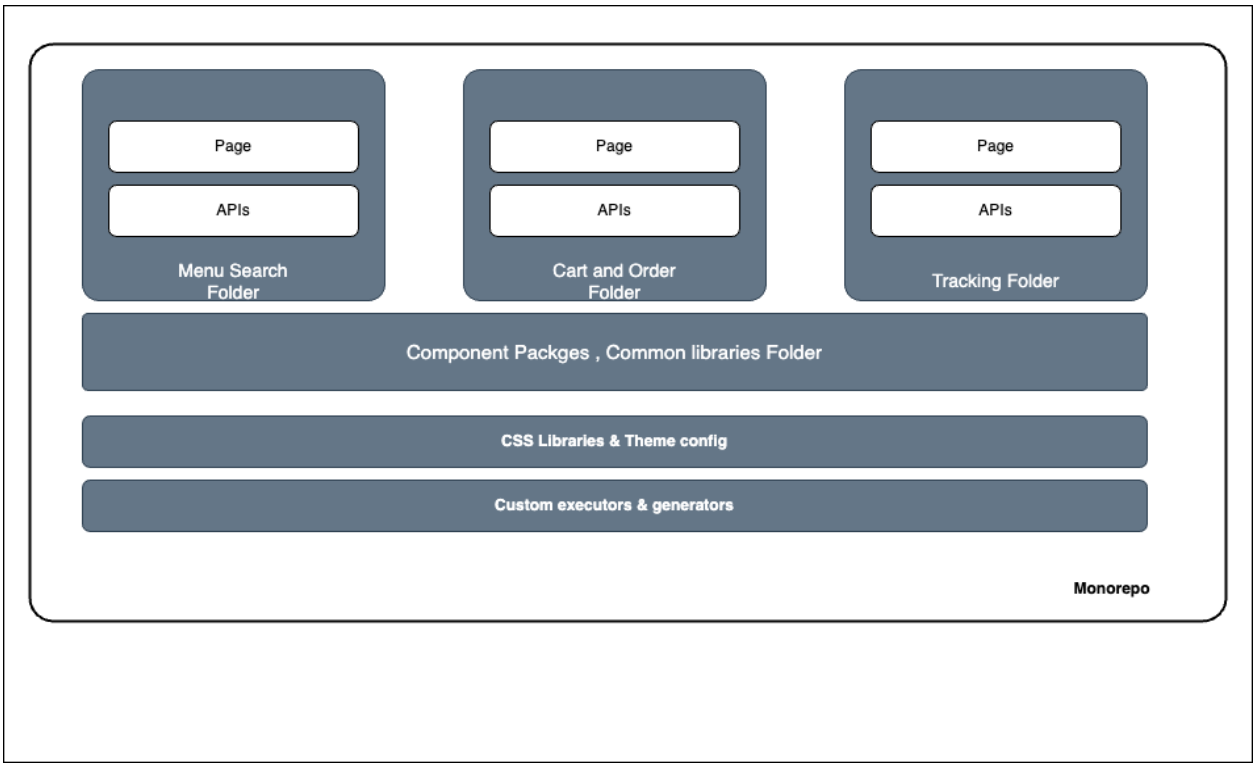

Figure 5: Frontend Monorepo

Monorepos provide many advantages in terms of code reuse, simplified dependency management, and improved collaboration [6]. However, they come with trade-offs related to build complexity, performance at scale, and the challenges of managing large repositories. Choosing whether or not to use a monorepo depends on the size of the organization, the complexity of the projects, and the tools available for managing the repository.

Pros of a frontend monorepo:

1. **Code Reusability:**

Easily share components, utilities, and logic across different frontend applications within the project, promoting code consistency and reducing duplication.

2. **Improved Collaboration:**

All developers have access to the entire codebase, facilitating easier communication and understanding of how different parts of the system interact.

3. **Simplified Dependency Management:**

Centralized management of shared dependencies, ensuring all projects are using the same versions and avoiding dependency conflicts.

4. **Consistent Development Workflow:**

Standardized build and testing processes across all projects within the monorepo.

5. **Easy Code Search and Navigation:**

Searching for code across the entire project becomes more efficient with everything in one place.

Cons of a frontend monorepo:

1. **Build Complexity:**

Large monorepos can lead to longer build times and increased complexity in managing the build process, requiring robust tooling to optimize.

2. **Performance Issues:**

Large repositories can impact Git operations like cloning and pulling, potentially slowing down developer workflow.

3. **Security Concerns:**
Giving all developers access to the entire codebase might raise security concerns, requiring careful access control management.
4. **Scalability Challenges:**
As the project grows, managing the monorepo can become increasingly difficult, especially with large teams.
5. **Less Isolation:**
Changes in one project within the monorepo can potentially impact other projects more readily, requiring careful testing and coordination.

Important Considerations:
1. **Team Size and Project Scope:**
Smaller teams with simpler projects may not need the complexity of a monorepo.
2. **Tooling:**
Utilizing appropriate tools like Yarn Workspaces, Lerna, or TurboRepo is essential for managing a monorepo effectively.
3. **Clear Development Practices:**
Establishing clear guidelines for code organization, branching strategies, and commit messages is vital for a successful monorepo.

Micro-Frontend Architecture: A Fully Distributed Approach
Micro-frontends represent the pinnacle of modularity and independence in frontend development. Positioned at the far end of the architectural spectrum, they are fully distributed and self-contained units of the user interface. Each micro-frontend is a separate application that is developed, deployed, and updated independently. They do not rely on the monolith or any central container. In fact, micro-frontends can be deployed as individual, isolated applications, making the deployment process more granular and efficient.
A key feature of micro-frontends is their modularity. The frontend application is divided into smaller, independent modules, each serving a specific business domain or feature. This division enables teams to work autonomously, with each team owning and managing its own micro-frontend.

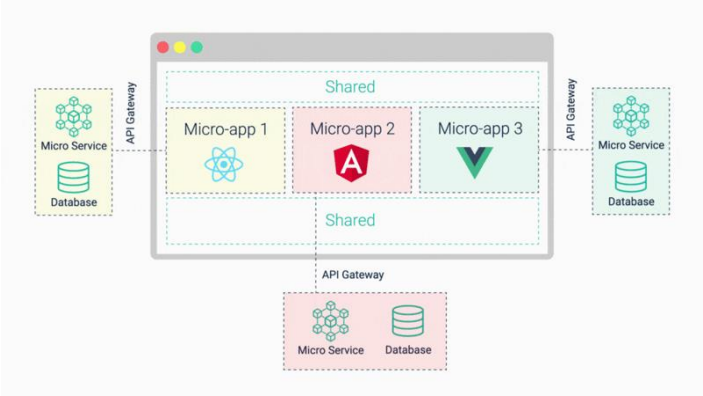

Figure 6: Micro-Frontend

The **independence** of micro-frontends is crucial, as they can be developed using different technologies or frameworks (e.g., React, Vue, Angular, or even HTMX). This flexibility allows teams to choose the best tool for their specific requirements. Each micro-frontend has its own release cycle and can be scaled independently to handle varying traffic loads, offering significant scalability benefits [2].
Micro-frontends are inspired by the **microservices architecture** used in backend development, where individual services are broken down into small, independently deployable units. The first mention of micro-frontends appeared in the **ThoughtWorks Technology Radar** in late 2016, introducing a new way to build complex, large-scale web applications.
While the concept of breaking down web applications into smaller, isolated modules isn't new—consider the use of to embed one web application into another—it has gained renewed popularity with microfrontends.

Large enterprises, such as Amazon, have long used this approach to compose multiple web applications into a seamless user experience.

Key Characteristics of Micro-Frontends:

1. **Full Independence**: Each micro-frontend is developed and deployed independently. Teams can use different tech stacks for different modules, creating a more flexible and scalable environment.
2. **Modular Composition**: The application is sliced into smaller, domain-driven components, each focused on a specific part of the user interface. This modularity improves maintainability and simplifies scaling.
3. **Team Autonomy**: Each team manages their own micro-frontend, giving them full control over development, deployment, and scaling. This autonomy allows teams to work without depending on other teams for updates or releases [1].
4. **Independent Release Cycles**: With each micro-frontend functioning as a standalone application, each module has its own release cycle, reducing the risk of single points of failure across the entire web application.
5. **Scalability**: Micro-frontends can be scaled independently based on user demand, optimizing resources and providing a more efficient way to handle traffic spikes.

There are couple of major approaches to build Microfrontends [5]

1. Build time composition
2. Run time composition

Build-Time Composition of Micro-Frontends

Refers to a method of integrating micro-frontends at the time of building the application, rather than at runtime. In this architecture, the micro-frontends are combined into a single application during the build process. When the user accesses the application, they are served the entire composed system, which has already been built and bundled together.

In a build-time composition setup, each micro-frontend is independently developed, tested, and deployed. However, instead of composing the individual micro-frontends dynamically at runtime, they are merged and packaged into a single application bundle during the build phase. This means that developers can focus on building and maintaining independent micro-frontends, and then compile them into one cohesive unit during the build process. Once built, the entire frontend is deployed together, and users interact with the fully composed system.

Key Characteristics of Build-Time Composition:

1. **Static Integration**: The micro-frontends are combined into a single, unified codebase before deployment, making it a more traditional build process where all components are included in the final bundle.
2. **Less Runtime Overhead**: Since the composition happens during the build phase, there is no need for runtime composition, reducing overhead and improving performance by avoiding runtime dependencies.
3. **Simplicity in Deployment**: Unlike runtime micro-frontends, which may require independent deployment cycles for each micro-frontend, build-time composition typically results in fewer, more streamlined deployment processes, as everything is bundled together before going live.
4. **Faster Initial Load**: Because the application is pre-compiled and bundled, users experience faster load times since the browser only needs to load one single application instead of multiple components being fetched individually at runtime.

Challenges of Build-Time Composition:

1. **Less Flexibility**: One of the trade-offs is that this approach is less flexible compared to runtime composition. Any updates or changes to a micro-frontend require a rebuild of the entire application, making it harder to update individual components independently without affecting the whole system.
2. **Tighter Coupling**: As micro-frontends are integrated into a single build process, they may lose some of the autonomy they would have in a runtime composition system. Each micro-frontend becomes part of a larger build, which can increase interdependencies between components.

3. **Scaling Issues**: Because the entire system is built together, scaling individual micro-frontends independently can be more challenging. Unlike runtime composition, where micro-frontends can be deployed and scaled on their own, build-time composition results in the entire application being scaled as a whole.

Tools for Build-Time Composition: Several tools can facilitate the build-time composition of micro-frontends. These typically involve bundlers like **Webpack**, **Parcel**, or **Rollup**, which allow different micro-frontends to be bundled together into a single build output. With proper configuration, these tools can merge the micro-frontends, handling dependencies and ensuring the frontend is fully integrated at build time.

When to Use Build-Time Composition:

1. **Simpler Projects**: If the project doesn't require frequent updates to individual micro-frontends or has relatively static content, build-time composition can be more efficient.
2. **Performance-Sensitive Applications**: For applications where performance is critical and runtime overhead is a concern, build-time composition ensures that everything is bundled and optimized ahead of time, reducing the load time and resource usage at runtime.
3. **Less Frequent Updates**: If the individual components of the application don't change very often, and there isn't a strong need for independent deployment cycles for each micro-frontend, build-time composition is a practical approach.

Runtime Composition of Micro-Frontends

Refers to the technique of dynamically composing and integrating individual micro-frontends at the time the application is running in the browser, rather than at build time. In this architecture, each micro-frontend is independently deployed and loaded in the browser as needed, allowing for a more flexible and scalable approach compared to traditional monolithic frontends.

With runtime composition, multiple micro-frontends can be developed, deployed, and updated independently by different teams, and then stitched together dynamically when the user accesses the application. This allows for a high degree of autonomy for each micro-frontend, as well as greater flexibility in terms of deployment and updates. Since the composition occurs at runtime, each micro-frontend is served as a self-contained unit, typically through different deployment units, such as different URLs or separate services, and they are composed into a unified experience for the end user.

Key Characteristics of Runtime Composition:

1. **Dynamic Integration**: Micro-frontends are composed in the browser, rather than during the build process. This allows different parts of the UI to be loaded and rendered dynamically based on the user's interactions and the current application state.
2. **Independent Deployments**: Each micro-frontend is deployed independently, meaning teams can update, scale, and deploy their parts of the frontend without affecting other components. The micro-frontends communicate with each other via shared APIs or events, but they remain decoupled.
3. **Autonomy for Teams**: Teams have complete control over their micro-frontend, allowing them to choose the technology stack, deployment cycle, and updates independent of the rest of the application. For example, one team could build a micro-frontend using React, while another uses Vue.js, as long as they follow a common interface for integration.
4. **Modular User Experience**: Users can experience different versions of micro-frontends depending on which parts of the application are enabled for them. The composition can adapt in real-time, enabling features such as A/B testing, gradual feature rollouts, or personalization.

Benefits of Runtime Composition:

1. **Flexibility**: Since micro-frontends are composed at runtime, teams can deploy new versions of individual micro-frontends without requiring a full deployment of the entire application. This enables continuous delivery and makes the frontend architecture more agile and adaptable.
2. **Seamless User Experience**: Users can be served the most relevant version of each micro-frontend at runtime, and different parts of the UI can be updated or rolled back independently. This leads to a smoother user experience, as the whole application does not need to be rebuilt or re-deployed for every change.

3. **Improved Scalability**: Each micro-frontend can be scaled independently according to its own usage and traffic patterns. For example, if one part of the application sees heavy traffic (e.g., a feature-rich dashboard), it can be scaled up without affecting other parts of the system.
4. **Technology Agnostic**: Teams can use different technologies for different micro-frontends, making it easier to experiment with new frameworks, libraries, or languages without disrupting the entire frontend architecture. For example, one micro-frontend might be built using Angular, while another could be based on React.

Challenges of Runtime Composition:

1. **Increased Complexity**: Managing multiple micro-frontends at runtime can introduce complexity, especially when coordinating communication between them and ensuring they work seamlessly together. For instance, handling cross-component communication or shared state management can be challenging.
2. **Performance Overhead**: Dynamically loading micro-frontends at runtime may lead to additional loading times, especially if micro-frontends are large or if too many micro-frontends are loaded simultaneously. The browser has to load and execute code for each individual micro-frontend, which can affect overall performance.
3. **Cross-Origin Issues**: Since micro-frontends are often deployed independently, cross-origin issues may arise, particularly if the micro-frontends are hosted on different domains or subdomains. This requires careful handling of CORS (Cross-Origin Resource Sharing) and proper integration of micro-frontends from different sources.
4. **Coordination and Consistency**: When composing micro-frontends at runtime, it's crucial to ensure that all components stay in sync. For example, making sure the user experience is consistent across micro-frontends (such as maintaining a consistent theme or user authentication state) can be more difficult to manage.

Tools for Runtime Composition:

1. **Single SPA**: This JavaScript framework allows different micro-frontends (built with various frameworks) to be combined into a single, unified application at runtime. Single SPA handles routing, loading, and coordination of the micro-frontends, enabling seamless integration of independent applications.
2. **Module Federation (Webpack 5)**: Module Federation allows different micro-frontends to share code and dependencies at runtime. This feature of Webpack 5 enables dynamic loading and execution of micro-frontends, making it a powerful tool for building applications with runtime composition. [4]
3. **iFrames**: Although generally not recommended for modern micro-frontends, embedding micro-frontends inside <iframe> elements is one of the oldest ways to integrate different parts of an application independently at runtime. This approach is simple but can lead to challenges in terms of communication between micro-frontends.

When to Use Runtime Composition:

1. **Large, Distributed Teams**: If different teams need to work independently on different parts of the frontend with minimal coordination, runtime composition is an excellent choice. It allows for independent deployments and releases while ensuring the application remains cohesive at runtime.
2. **Frequent Updates**: When micro-frontends need to be updated regularly without impacting other parts of the application, runtime composition allows each component to be updated and deployed independently.
3. **Highly Modular Applications**: Applications that require modularity, such as large-scale web applications with varied features, can benefit from runtime composition. It ensures that each micro-frontend is only responsible for a small, self-contained part of the application.

CONCLUSION

Frontend architecture has evolved from **static pages** to **monolithic** and **modular** structures, ultimately leading to **Micro Frontends** for large-scale applications. Each architectural phase introduced **trade-offs** between **performance, scalability, and maintainability**. Organizations must **carefully evaluate their needs** before transitioning to a new model, ensuring they **balance complexity with business requirements**.

As technology advances, **WebAssembly, edge computing, and AI-driven frontends** will further revolutionize web development, making applications more **efficient and intelligent**.

REFERENCES:

[1] Luca Mezzalira, "Building Micro Frontends: Scaling Teams and Projects with Modular Frontend Architecture," O'Reilly Media, 2021.

[2] Cam Jackson, "Micro Frontends - The Future of Frontend Architectures?", ThoughtWorks, 2019.

[3] H. Patil, "Contemporary Front-end Architectures," *webf.zone*, 2019. [Online]. Available: https://blog.webf.zone/contemporary-front-end-architectures-fb5b500b0231

[4] Cam Jackson, "Micro Frontends - The Future of Frontend Architectures?," ThoughtWorks, 2019. https://www.thoughtworks.com/insights/blog/micro-frontends

[5] S. Peltonen, S. Brinkkemper, and E. Visser, "Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review," *arXiv*, 2020. [Online]. Available: https://arxiv.org/abs/2007.00293.

[6] S. Peltonen, L. Mezzalira, and D. Taibi, "Motivations, Benefits, and Issues for Adopting Micro-Frontends: A Multivocal Literature Review," *arXiv preprint arXiv:2007.00293*, 2020. [Online]. Available: https://arxiv.org/abs/2007.00293

[7] M. Van Hees and B. Dhoedt, "Quality Evaluation of Modern Front-End Web Frameworks," Ghent University, Faculty of Engineering and Architecture, 2019. [Online]. Available: https://libstore.ugent.be/fulltxt/RUG01/002/837/311/RUG01-002837311_2020_0001_AC.pdf.

[8] **M. Kroiss,** *"From Backend to Frontend: Case Study on Adopting Micro Frontends,"* Master's thesis, TU Wien, 2021. [Online]. Available: https://repositum.tuwien.at/bitstream/20.500.12708/17595/1/Kroiss%20Manuel%20-%202021%20-%20From%20backend%20to%20frontend%20Case%20study%20on%20adopting%20Mmcro...pdf