Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

React's Architectural Limitations in Distributed UI Systems: A Critical Analysis

Swaraj Guduru

Independent Researcher, USA

ARTICLE INFO

Received:01 Sept 2025

Revised:05 Oct 2025

Accepted:15 Oct 2025

ABSTRACT

This article critically examines React's architectural limitations when applied to distributed user interface systems, particularly in micro-frontend implementations. It explores the fundamental tension between React's component model, which assumes a unified runtime with shared context and rendering cycles, and the distributed nature of modern frontend architectures that emphasize team autonomy and independent deployment. The article progresses through React's core design assumptions, identifies specific challenges in cross-boundary state management and server-side rendering coordination, evaluates current industry mitigation strategies, including isolation techniques and module federation, and explores emerging paradigms that reimagine frontend architectures for distributed contexts. By identifying the architectural mismatches between React's monolithic design and distributed UI requirements, the article provides insights for organizations navigating these competing paradigms while seeking to maintain both system cohesion and team independence.

Keywords: Micro-Frontends, React Architecture, Distributed User Interfaces, Frontend Composition, Cross-Boundary State Management

1. Introduction and Background

Frontend development has undergone a deep transformation over the course of the 2010s. Codebases that were dominated by jQuery began to be supplanted by architectures that were organized in terms of components. Then in 2013, React became available, providing a new and different way of building interfaces that is based on the declarative style of programming and rendering via a virtual DOM. React quickly became popular in development communities because of the various technical benefits its architecture afforded. This library grew even faster in the development industry, thanks to its many technical affordances.

The component-based structure advocated by React enables decomposition of complex interfaces into modular, reusable units. Each component maintains an internal state while rendering predictably based on input properties. [1] This marked a significant departure from imperative DOM manipulation techniques prevalent in earlier frameworks. Traditional direct DOM operations incurred substantial performance penalties during re-rendering cycles. React's virtual DOM addressed these inefficiencies through the implementation of differential reconciliation algorithms, selectively updating only necessary DOM nodes rather than reconstructing entire interface trees. Combined with unidirectional data flow patterns, this architecture substantially improved performance characteristics for complex interface implementations.

React's evolution brought sophisticated features—hooks for state management, context API for prop-drilling prevention, concurrent rendering capabilities—while parallel shifts occurred in broader web architecture paradigms. Service-oriented systems have already been adopted by backend systems. Microservices have become a standard practice for designing scalable systems. Frontend architectures have followed suit, and monolithic applications are breaking apart into micro-frontends, or smaller applications that are independently deployable, owned by development teams, and built into the application flow. This approach is simply microservices for the UI, allowing sections of connected interfaces to be built and deployed without redeploying the entire app, the work of possibly multiple

Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ Research Article

teams. Architectural boundaries drawn along business domain lines rather than technical concerns facilitate organizational alignment with software structure. [2] Development teams gain the capability to deploy interface sections independently, potentially leveraging different technology stacks while maintaining adherence to shared design systems and integration contracts.

The fundamental question emerges regarding compatibility between React's core design assumptions and distributed UI requirements. React fundamentally presupposes unified runtime environments—shared memory space, coherent component hierarchies, synchronized rendering cycles. These architectural assumptions create tension points when applications fragment across team boundaries, service domains, or deployment units. Projects implementing distributed React architectures frequently encounter these limitations.

React's architectural constraints affect at least three disparate dimensions of creating an application: the pace at which features can be delivered, the characteristics of system reliability, and how the product scales with the organization. Organizations creating development initiatives using React experience friction between the inherent, centralized component model in React and micro-frontend's decentralized architecture. The misalignment between declarative programming in React and the decentralized nature of micro-frontends presents a formidable barrier to implementation.

Technical teams encounter numerous integration challenges: dependency collision issues (React version conflicts coexisting in single-page contexts), runtime boundary complications, cross-fragment state synchronization requirements, and disjointed rendering lifecycles. Architectural considerations extend beyond purely technical domains into organizational territories—team autonomy boundaries, deployment independence requirements, and governance frameworks. DOM hydration inconsistencies frequently emerge when server-rendered content spans multiple independent applications. Performance degradation often results from duplicated bundle content and initialization logic across fragment boundaries.

Ultimately, if enterprises are going to support micro-frontend architectures, they will generally need to ensure their teams understand the inherent bounds of React outside of a single page's context. The conceptual understanding of the in-framework limitations will provide the basis for reasoned architectural decisions and impactful mitigation in ongoing debates between developer experience, runtime performance, and scaling at the organization level.

2. Theoretical Foundations of React's Component Model

The React component model catalyzed a paradigm shift in thinking about frontend architecture. React incorporates a handful of core technical principles that drive its construction of user interfaces; some of these principles can be regarded as strengths for monolithic applications, while others are more likely to create limitations in distributed settings.

The performance story hinges on React's virtual DOM. Instead of using traditional libraries that perform direct updates to the DOM, React keeps a lightweight tree of JavaScript objects that closely mirrors the actual DOM in the browser. This abstraction layer is where it will be able to stage rendering work. State mutations trigger the creation of a new virtual tree, which React then compares against the previous version. The comparison process—reconciliation—employs several heuristic shortcuts rather than computationally expensive exact tree diffing. Type-based element comparison forms the primary heuristic; React assumes elements of different types produce entirely different trees, while same-type elements merely require attribute updates. Key props function as persistent identifiers during collection rendering, preserving component state during reordering operations and preventing unnecessary recreations.

Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

React Assumption	Micro-Frontend Requirement	Architectural Tension
Single Virtual DOM Tree	Independent Rendering Trees	React assumes a unified tree structure while micro-frontends require autonomous rendering processes across boundaries
Shared Runtime Environment	Isolated Execution Contexts	React optimizes for shared memory and dependencies, while micro-frontends need clear runtime isolation
Unified Component Lifecycle	Independent Deployment Cycles	React expects synchronized component updates while micro-frontends demand independent release schedules

Table 1: React's Core Architectural Assumptions vs. Micro-Frontend Requirements. [3, 4]

Performance optimization techniques include batched DOM updates and selective subtree re-rendering. The entire architecture assumes a singular reconciliation context—one virtual tree within one JavaScript runtime. This architectural assumption becomes problematic for distributed UI systems where separate teams independently develop and deploy distinct interface sections on asynchronous release schedules [3]. Multiple reconciliation contexts existing simultaneously on a single page create boundary issues that React wasn't designed to handle smoothly.

Unidirectional data flow constitutes another foundational React principle affecting distributed implementation. Traditional React apps pass data downward through component hierarchies via props, with state changes triggering cascading re-renders through affected component subtrees. Parent components maintain state requiring modification from children through callback prop passing. This model creates predictable debug patterns—components function deterministically based solely on current props/state. FBJS DevTools relies on this predictability for time-travel debugging.

The architecture presupposes shared memory access between components. Distributed UI architectures fragment this assumption when components span runtime boundaries, forcing alternative communication patterns. Cross-boundary state synchronization typically requires event-driven architectures, pub/sub patterns, or backend-mediated state sharing. Many orgs implement Redux middleware layers with custom serialization/deserialization for cross-boundary events. Others employ BFFs (Backend-for-Frontend) proxies maintaining unified state sources. Both approaches diverge significantly from React's original mental model and introduce complexity absent in monolithic implementations [4]. Dev teams frequently maintain hybrid approaches—React-native state management within boundaries, custom cross-boundary protocols between them.

Component hierarchy coupling creates additional distributed system challenges. React's composition model builds deeply nested component trees where parent components control children's props and lifecycle. Context API extends this coupling beyond direct parent-child relationships, enabling any component to consume values from providers anywhere above in the tree. While solving prop-drilling headaches elegantly in monolithic apps, Context utterly breaks at micro-frontend boundaries since providers and consumers must share a single React tree instance. Similar limitations affect React's newer composition mechanisms—hooks like useContext, useReducer, and custom hooks combining multiple stateful behaviors all assume runtime proximity.

The coupling creates practical problems for standard UI concerns spanning micro-frontend boundaries. Global theming systems traditionally implemented via ThemeProviders must be duplicated across boundaries or reimplemented using alternative mechanisms. Auth state

Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

traditionally flowing through context requires alternative sharing approaches—often via cookie/localStorage with redundant hydration or federated login services. Design system component libraries, depending on contextual configuration, face difficult implementation decisions at boundaries.

React's runtime assumptions further highlight architectural limitations in distributed environments. The library expects a unified JavaScript context where all components access identical React instances, reconciler implementations, and event systems. This unity enables synthetic events to normalize browser differences and implement efficient delegation patterns. Features like concurrent rendering, suspense boundaries, and prioritized transitions depend entirely on a single scheduler coordinating the entire component tree.

Distributed architectures break these assumptions when multiple React versions coexist on a single page. Version conflicts create subtle and difficult debugging challenges—especially when shared component libraries inadvertently reference multiple React instances. Synthetic events behave unpredictably across boundaries. Error boundaries fail to catch exceptions across reconciler contexts.

Building robust distributed React systems requires explicit strategies addressing these limitations. Teams commonly implement runtime isolation through module federation with shared singletons, namespace sandboxing techniques, or custom runtime bridges mediating between React instances. Standardized versioning policies, explicit interface contracts, and comprehensive integration testing become mandatory rather than optional practices.

React's architectural decisions, optimized perfectly for cohesive monolithic applications, become significant liabilities in distributed contexts without additional coordination mechanisms. The fundamental tension between React's unified runtime expectations and distributed system principles creates unavoidable complexity that must be explicitly managed.

3. Fundamental Challenges in Distributed UI Implementation

Distributed UI architectures built with React face substantial technical hurdles originating from architectural incompatibilities. Micro-frontends fragment application interfaces into distinct chunks maintained by separate teams. This concept extends backend microservice principles into frontend territory. Key drivers behind micro-frontend adoption include domain-aligned team boundaries, release independence, technology flexibility, and integration resilience.

These architectural principles fundamentally clash with React's design assumptions. React expects components to exist within a unified memory space sharing a single rendering context. Micro-frontends create vertical application slices that encapsulate functionality from the database through UI layers with minimal cross-team dependencies during development cycles. This contrasts with React's inherent horizontal composition structure, where components freely share context, props, and rendering lifecycle hooks within a unified application boundary.

Approach	Implementation Method	Key Limitations
Backend-Mediated State	Shared API endpoints with WebSockets or polling	Network latency impacts responsiveness; requires additional backend infrastructure
Client-Side Event Bus	Custom event dispatchers with serialized payloads	Complex synchronization logic; potential race conditions across boundaries
Federated State Stores	Coordinated Redux/MobX stores with bridge adapters	Version compatibility issues; increased bundle size from duplicate state libraries

Table 2: State Management Approaches in Distributed React Architectures. [5, 6]

Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

Integration techniques vary substantially across implementations. Some organizations employ iframe-based strategies, providing maximum runtime isolation at the cost of seamless visual integration. Others leverage Web Components as standardized interface boundaries. Build-time approaches use module bundlers for composition during compilation phases. Server-side strategies utilize fragment-based composition through ESI or template assembly. Runtime orchestration dynamically loads fragment code into shell applications. Each strategy introduces specific trade-offs regarding isolation strength, performance characteristics, and developer experience [5].

Technical conflicts emerge across multiple fronts. Dependency collisions occur when different React versions must coexist on a single page - particularly problematic given React's internal reconciler implementation details. CSS isolation remains notoriously difficult - global styling from one fragment frequently disrupts another's layout unexpectedly. Runtime composition demands sophisticated orchestration mechanisms handling asynchronous loading, graceful degradation during failures, and proper initialization sequencing.

State management across fragment boundaries presents particularly thorny challenges. Traditional React patterns depend entirely on component proximity within a shared tree structure. Context API becomes useless across micro-frontend boundaries since providers and consumers require the same React instance. Redux stores cannot directly span separate applications without significant adaptation. Even basic component communication via props breaks down completely at runtime boundaries.

Organizations adopt various workarounds for cross-boundary state coordination. Many implement BFF proxies that maintain a centralized state server-side while synchronizing to multiple frontend fragments. Custom event mechanisms using browser storage events or postMessage APIs create communication channels between isolated fragments. Message broker architectures sometimes mediate between fragments using WebSocket connections to central state services. Establishing clear state ownership rules becomes critical - defining which fragment controls specific data and how changes propagate [6].

Complex workflows spanning multiple domain boundaries create additional integration challenges. Shopping functionality split between product catalog and checkout fragments requires meticulous state synchronization. User authentication must maintain consistency across all application sections. Notifications need reliable delivery to relevant UI components regardless of which team owns the receiving fragment.

Server rendering compounds these difficulties substantially. Monolithic React applications follow straightforward SSR patterns: the server produces complete HTML markup, the client receives it alongside JavaScript bundles, and React hydrates the static DOM by attaching event handlers and reconstructing virtual component trees. Distributed architectures fragment this unified process across team boundaries.

Each micro-frontend team implements SSR independently while ensuring a compatible output. Numerous coordination problems emerge: fragments might depend on others loading first, nested hydration occurs when fragments contain components from multiple teams, and duplicate resource loading happens without careful dependency management. Organizations must select appropriate composition strategies based on specific requirements - pure server composition achieves fast initial rendering but complicates interactive elements, while client-side approaches offer flexibility at the cost of potential hydration inconsistencies.

Organizational impacts often match or exceed technical challenges. Though micro-frontends promise team autonomy, React's composition model creates unavoidable interdependencies. Teams frequently discover their supposedly independent technical decisions cascade unexpectedly across fragment

Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ Research Article

boundaries. Choices about state management approaches, styling methodologies, or dependency versions create integration conflicts requiring cross-team coordination.

Development workflows grow increasingly complex. Local environments must simulate integration with fragments from other teams. Testing strategies require both isolated component validation and cross-boundary integration verification. Deployment pipelines need sophisticated orchestration to prevent broken user experiences during partial updates.

Many enterprises establish dedicated platform teams responsible for integration infrastructure, shared libraries, and cross-team governance. These groups maintain standardized communication patterns between fragments, ensure consistent design system implementation, and establish versioning policies for shared dependencies. This governance layer represents substantial overhead absent from monolithic applications, highlighting fundamental tensions between React's inherent architectural assumptions and micro-frontend organizational objectives.

4. Current Industry Mitigation Strategies

The software industry has devised numerous techniques addressing React's limitations in distributed interface scenarios. Each technique presents unique balances between boundary strength, rendering efficiency, and developer accessibility. Technical teams typically adopt approaches that match specific organizational constraints and technical needs. iFrame-based separation stands as the simplest isolation method. This approach embeds separate React applications within distinct browsing contexts, creating natural DOM partitioning and runtime separation. The method resolves several persistent challenges: style interference disappears since CSS remains confined to each frame's scope; module conflicts vanish through complete runtime separation; security boundaries strengthen as browser protections prevent cross-fragment scripting attacks.

Nevertheless, frame-based architectures present substantial operational difficulties. Fragment interaction requires complex messaging protocols using serialized data through browser messaging channels. Common interface components like navigation elements or modals cannot naturally traverse boundaries, forcing teams toward either component duplication or elaborate synchronization mechanisms. System resources face significant strain from redundant asset loading, memory inefficiency, and duplicate JavaScript processing across contexts. Interface coherence frequently suffers from perceptible discontinuities and interaction constraints [7]. The Custom Elements specification offers alternative boundary mechanisms utilizing platform-native standards. This approach packages React applications inside defined custom elements with Shadow DOM, providing style isolation. The method enables tighter page integration compared to frames while preserving necessary boundaries. Interface fragments expose formalized contracts through element attributes and event interfaces, establishing cleaner integration points than direct component coupling. Shadow DOM boundaries prevent most styling conflicts while supporting theme inheritance through custom CSS properties.

Strategy	Integration Fidelity	Team Autonomy Impact
iFrames	Low - Significant UX and styling discontinuities	High - Complete technical independence with minimal coordination
Web Components	Medium - Native DOM integration with style isolation	Moderate - Framework-agnostic interfaces with shared styling standards
Module Federation	High - Seamless React component composition	Lower - Requires coordination on shared dependencies and versions

Table 3: Comparison of Micro-Frontend Integration Strategies. [7]

Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ **Research Article**

React implementations inside Shadow DOM encounter several technical obstacles. Event delegation systems in React operate inconsistently through shadow boundaries, requiring additional handling layers. Programmatic DOM access using reference objects fails across boundary edges without specialized adapters. Portal functionality needs significant modification to function properly with shadow root containers.

Webpack Module Federation represents a sophisticated dependency-sharing approach addressing numerous isolation limitations. This mechanism allows discrete build pipelines to share JavaScript modules during runtime execution without code duplication. Unlike basic separation techniques, Federation establishes cross-application dependency relationships while maintaining deployment autonomy.

The architecture depends on several critical components: container applications managing the federated environment; exposed remote modules providing functionality; shared dependency definitions preventing duplication; and asynchronous interface modules handling cross-application interaction. Implementation patterns vary considerably: shell designs where lightweight containers orchestrate remote functionality; component approaches assembling interface elements from distributed sources; and route-based systems loading complete application sections per navigation path [8].

Despite significant advantages, Federation introduces substantial configuration complexity. Build systems require precise coordination preventing version incompatibilities. Common dependencies demand strict version management preventing subtle runtime failures. Loading sequence becomes critically important when applications share initialization state. Error recovery grows exponentially more complex when remote modules encounter loading or execution problems.

Proprietary composition frameworks have emerged addressing unique organizational requirements beyond standardized solutions. These custom implementations typically provide specialized management layers controlling fragment loading, initialization sequencing, and cross-boundary communication. Solutions range from basic route handlers dynamically importing bundles to comprehensive runtime containers implementing standardized lifecycle protocols and messaging systems.

Sophisticated custom implementations frequently include specialized capabilities missing from standard approaches: failure isolation, preventing cascade effects across fragment boundaries; unified performance monitoring spanning distributed components; compliance verification, ensuring adherence to architectural standards. While effectively addressing organization-specific requirements, these custom frameworks represent substantial engineering investments requiring dedicated maintenance teams. This creates potential organizational bottlenecks, potentially undermining the very team independence micro-frontends aim to establish.

Strategy selection requires detailed trade-off analysis considering technical requirements, organizational capabilities, and sustainability concerns. Basic isolation minimizes implementation overhead but degrades user experience through performance limitations. Federation enables seamless composition but requires sophisticated build infrastructure and governance frameworks that smaller teams struggle to support. Custom frameworks provide targeted functionality but create ongoing maintenance obligations and potential organizational dependencies.

Beyond technical factors, organizations must consider broader implications: team structure requirements when dedicated platform groups become necessary; security exposure differences between approaches; scalability limitations for organizations managing numerous micro-frontends; and maintainability challenges as underlying technologies evolve.

Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ Research Article

5. Emerging Paradigms and Future Directions

React's architectural constraints in distributed environments have sparked significant innovation across frontend architecture domains. Several emerging paradigms now challenge traditional rendering models while addressing fundamental distribution challenges.

Resumability-based frameworks mark a decisive shift from traditional hydration approaches. Server rendering has evolved through distinct phases since React's introduction. Early SSR simply generated static markup for initial display before completely reinitializing applications client-side—improving perceived performance but introducing substantial JavaScript overhead and potential hydration inconsistencies. Later advances brought streaming HTML delivery, allowing browsers to process content chunks progressively rather than waiting for complete server responses.

Recent innovations focus on partial hydration techniques where only interactive elements require JavaScript initialization while static content remains untouched. This evolutionary path culminates in resumability patterns that fundamentally redefine server-client relationships. Unlike conventional hydration, which essentially duplicates rendering work client-side, resumability mechanisms serialize application state and execution context, enabling browsers to continue rather than restart application execution. These frameworks employ fine-grained reactivity systems with progressive enhancement principles, drastically reducing JavaScript requirements while preserving rich interactions where necessary.

Server components represent another architectural advancement, shifting rendering responsibilities dynamically between server and client based on data access patterns, interactivity requirements, and performance considerations. These approaches create cleaner boundaries between static and interactive interface regions, aligning naturally with micro-frontend architectural principles [9]. System resilience improves substantially as component failures remain isolated without compromising entire application experiences.

Distribution-native rendering addresses fundamental misalignments between React's monolithic model and distributed interface requirements. Edge computing emerges as particularly transformative, relocating rendering logic to network periphery nodes rather than centralized servers or client devices. This architectural shift harnesses globally distributed infrastructure to process requests at physically proximate points, substantially reducing response latency while enhancing scalability and reliability characteristics.

Edge rendering implementations typically deploy lightweight JavaScript processors across numerous geographic locations, enabling dynamic content generation with performance profiles approaching static asset delivery. This approach delivers several critical advantages for distributed interfaces: drastically improved time-to-first-byte measurements with content generated physically nearer users; consistent global performance metrics eliminating regional variations common in centralized architectures; enhanced fault tolerance through geographic distribution of rendering capabilities.

This paradigm aligns naturally with micro-frontend principles—independent interface fragments render and compose at edge locations without requiring origin server communication. Advanced implementations incorporate contextual rendering decisions based on device capabilities, network conditions, or user attributes, optimizing delivery strategies per request. The approach typically employs streaming response patterns, delivering static elements immediately while dynamic or personalized components follow progressively. This enables sophisticated multi-level caching strategies with static content stored aggressively while dynamic elements utilize validation patterns balancing freshness against performance [10].

Cross-framework composition models address organizational realities where multiple frontend technologies coexist, particularly during migration periods or following corporate acquisitions. These

Journal of Information Systems Engineering and Management
2025, 10(61s)
e-ISSN: 2468-4376
https://www.jisem-journal.com/ Research Article

approaches acknowledge practical limitations of framework standardization across teams, focusing instead on interoperability layers enabling heterogeneous technologies to function cooperatively.

Technical implementation challenges extend into governance considerations, requiring organizations to balance the standardization necessary for cohesive experiences against team autonomy. Cross-framework systems establish explicit contracts through standardized communication interfaces, frequently leveraging browser-native mechanisms like Custom Elements with Shadow DOM for encapsulation. These interfaces define consistent property passing, event handling, and lifecycle management patterns regardless of underlying implementation details.

Sophisticated implementations develop adaptation layers translating between fundamentally different programming models—bridging React's declarative approach with Angular's change detection systems or Vue's reactivity mechanisms. These translation layers enable seamless component composition across frameworks while preserving internal implementation specifics. Enterprise component registries frequently supplement these technical approaches, providing discovery mechanisms for available components while establishing governance around dependency sharing, version management, and compatibility requirements.

Resulting architectures support gradual technology migration, allowing teams to adopt newer frameworks without requiring organization-wide coordination. This flexibility creates sustainable enterprise architectures accommodating both technological advancement and organizational evolution while maintaining system cohesion and experience consistency.

Research indicates several promising directions potentially transform distributed interface construction fundamentally. Compiler-centric approaches gain increasing attention, applying build-time analysis techniques for runtime optimization by eliminating unnecessary abstractions, optimizing rendering paths, and generating specialized code variants. These approaches identify optimization opportunities impossible during runtime execution—removing unused components, inlining critical paths, or generating deployment-specific variants.

Content-aware rendering represents another emerging direction where systems dynamically select rendering strategies based on content characteristics, interaction patterns, and performance requirements. These frameworks might apply static generation for relatively stable content, server rendering for personalized non-interactive elements, and client-side rendering for highly interactive components—all within unified architectures optimizing each rendering decision independently rather than applying monolithic approaches across entire applications.

Paradigm	Key Innovation	Distributed UI Benefit
Resumability	State serialization without full hydration	Eliminates cross-boundary hydration mismatches while reducing client JavaScript
Partial Hydration	Selective interactivity for specific components	Enables independent activation of micro-frontend fragments with lower overhead
Edge Rendering	Computation moved to CDN edge locations	Improves global performance consistency while enabling distributed composition

Table 4: Emerging Rendering Paradigms for Distributed UIs. [9, 10]

Conclusion

The architectural limitations of React in distributed UI contexts represent a significant challenge for organizations scaling frontend development across multiple teams. While React excels in single-

application contexts through its unified component model and efficient reconciliation process, these same characteristics create friction when applied to distributed architectures that prioritize team autonomy and independent deployment. Current mitigation strategies offer viable but imperfect solutions: isolation approaches maintain strong boundaries at the cost of user experience cohesion; Module Federation enables more seamless composition but introduces complex dependency coordination requirements; and custom runtime solutions demand significant development investment. Emerging paradigms like resumability-based frameworks and edge-rendering approaches suggest a future direction where frontend architectures embrace distribution as a fundamental design principle rather than an afterthought. Organizations navigating this evolving landscape must carefully balance the developer experience benefits of React against the organizational scalability advantages of truly distributed architectures, potentially adopting hybrid approaches that leverage React within team boundaries while implementing distribution-aware patterns for cross-team integration. The continued evolution of these patterns will likely reshape frontend architecture practices as distributed UI systems become increasingly prevalent across the industry.

References

[1] Sanity, "React.js overview," 2023. [Online]. Available: https://www.sanity.io/glossary/react-js [2] Michael Geers, "Micro Frontends extending the microservice idea to frontend development," 2023. [Online]. Available: https://micro-frontends.org/

[3] GeeksforGeeks, "ReactJS Reconciliation," 2023. [Online]. Available: https://www.geeksforgeeks.org/reactjs/reactjs-reconciliation/

[4] Chad R. Stewart, "Build Resilient UIs: Frontend Architecture that doesn't suck!," Dev. to 2022. [Online]. Available: https://dev.to/chad_r_stewart/frontend-architecture-and-tooling-that-will-lead-to-a-more-resilient-codebase-7ib

[5] Cam Jackson, "Micro Frontends," Martin Fowler's Technology Radar, 2021. [Online]. Available: https://martinfowler.com/articles/micro-frontends.html

[6] Geeks for Geeks, "Handling State and State Management | System Design," GeeksforGeeks, 2025. [Online]. Available: https://www.geeksforgeeks.org/system-design/handling-state-and-state-management-system-design/

[7] Rahul Gupta, "How Microfrontends Work: From iframes to Module Federation," FreeCodeCamp, 2025. [Online]. Available: https://www.freecodecamp.org/news/how-microfrontends-work-iframes-to-module-federation/

[8] Webpack, "Module Federation," Webpack Documentation, 2023. [Online]. Available: https://webpack.js.org/concepts/module-federation/

[9] Aurora Scharff, "React Frameworks and Server-Side Features: Beyond Client-Side Rendering," Certificates.dev Blog, 2025. [Online]. Available: https://certificates.dev/blog/react-frameworks-and-server-side-features-beyond-client-side-rendering

[10] Esolz Technology Blog, "Edge-Rendered Websites: Next-Gen Static and Dynamic Delivery," [Online]. Available: https://esolz.net/edge-rendered-websites/