

Received 29 October 2025, accepted 24 November 2025, date of publication 27 November 2025,
date of current version 12 January 2026.

Digital Object Identifier 10.1109/ACCESS.2025.3638252

RESEARCH ARTICLE

Scaling Secure Fintech Applications With Micro Frontends: A Case Study in Regulated Environments

ROMAN FEDYTSKYI^{ID}, (Senior Member, IEEE)

ORSNN, Seattle, WA 98121, USA

e-mail: fedytskyi@gmail.com

ABSTRACT This paper presents a systematic approach to scaling secure fintech applications through micro frontend architecture in highly regulated environments. While micro frontends have gained traction in consumer-facing applications, their implementation in compliance-heavy financial technology systems remains under-documented. This research addresses this gap through a case study of ORSNN, a financial services platform that successfully migrated from a monolithic architecture to a federated micro frontend model using Webpack Module Federation. The study quantifies improvements in deployment frequency, performance metrics, and developer productivity while maintaining strict security and compliance requirements. Results demonstrate a 37% improvement in Largest Contentful Paint (LCP), 36% improvement in First Contentful Paint (FCP), and 35% reduction in Time to Interactive (TTI), alongside a 64% reduction in Mean Time to Production (MTTP). The architectural model, security framework, real-time data layer implementation, and CI/CD pipeline are analyzed in detail, providing a reference implementation for financial technology organizations facing similar scaling challenges. This work contributes to the emerging body of knowledge on secure, compliant frontend architectures in regulated industries.

INDEX TERMS CI/CD, compliance, financial technology, frontend architecture, micro frontends, performance optimization, regulated environments, security, Webpack Module Federation.

I. INTRODUCTION

Financial technology applications face unique challenges at the intersection of rapid innovation demands and stringent regulatory requirements. As these systems scale, traditional monolithic frontend architectures often become bottlenecks, impeding both feature velocity and compliance adaptability. This tension is particularly acute in regulated environments where audit trails, access controls, and data handling practices must meet exacting standards while supporting competitive product development cycles [1].

This paper presents a systematic analysis of a micro frontend implementation at ORSNN, a financial services platform that facilitates institutional investment workflows [6]. The research documents the architectural transition from a monolithic frontend to a federated model using Webpack Module

Federation, with particular emphasis on maintaining security posture and regulatory compliance throughout the transformation [7].

The study addresses the following research questions:

1. How can micro frontend architectures be implemented in regulated financial environments without compromising security or compliance requirements?
2. What quantifiable improvements in performance, deployment velocity, and developer productivity can be achieved through this architectural approach?
3. What specific patterns and practices enable effective integration of security controls, audit mechanisms, and compliance requirements in a distributed frontend architecture? [8]

The remainder of this paper is organized as follows: Section II provides background on micro frontend architectures and the specific challenges of regulated environments.

The associate editor coordinating the review of this manuscript and approving it for publication was Fazel Mohammadi^{ID}.

Section III details the architectural model implemented at ORSNN. Section IV examines the security framework that ensures compliance in a distributed frontend environment. Section V explores the real-time data layer implementation. Section VI discusses the CI/CD and observability infrastructure. Section VII presents quantitative and qualitative results from the case study. Section VIII reviews related work, and Section IX concludes with implications and directions for future research [9].

The evolution of frontend architectures in enterprise applications has followed a trajectory from simple document-based websites to complex, interactive applications with sophisticated state management [10]. This evolution has been particularly challenging in regulated industries like financial technology (fintech), where compliance requirements add layers of complexity to both development and deployment processes [11]. As fintech applications grow in complexity and scale, traditional monolithic frontend architectures increasingly become bottlenecks for development velocity, performance optimization, and regulatory compliance [12].

Micro frontends have emerged as an architectural pattern that extends microservice principles to frontend development, enabling independent teams to build and deploy UI components without tight coupling [13]. While this approach has gained traction in retail and media sectors, its application in highly regulated fintech environments remains under-documented and presents unique challenges [14]. Financial applications must maintain strict compliance with regulations like PCI-DSS, SOX, and GDPR while delivering responsive user experiences across complex workflows [15].

A. PROBLEM STATEMENT AND RESEARCH GAP

While micro frontends are increasingly common in general web applications, their implementation in secure, compliance-heavy fintech systems remains under-documented [16]. Existing literature focuses primarily on technical implementation details without addressing the unique challenges of regulated environments [17]. This research gap is significant as financial institutions seek to modernize their frontend architectures while maintaining strict security and compliance standards [18].

This work addresses that gap through a comprehensive case study with quantifiable deployment, user experience, and auditability metrics. By documenting both the technical implementation and compliance considerations, this research provides a framework for financial institutions to evaluate and implement micro frontend architectures [19].

B. RESEARCH OBJECTIVES

This study aims to:

1. Document the architectural patterns and implementation strategies for micro frontends in a regulated fintech environment

2. Quantify the performance, developer productivity, and compliance impacts of the migration
3. Identify key challenges and mitigation strategies for maintaining security and regulatory compliance
4. Provide a replicable framework for evaluating and implementing micro frontends in similar regulated contexts [20]

C. CONTRIBUTIONS

The primary contributions of this work include:

1. A detailed architectural model for implementing micro frontends in compliance-heavy fintech applications
2. A security framework that maintains regulatory compliance across distributed frontend components
3. Empirical data on performance improvements, developer productivity gains, and compliance outcomes
4. A set of best practices and lessons learned for organizations considering similar architectural transitions [21]

The findings demonstrate that properly implemented micro frontend architectures can significantly improve development velocity and application performance while maintaining or enhancing security and compliance postures in regulated financial environments [22].

II. BACKGROUND AND MOTIVATION

A. PROBLEM STATEMENT AND RESEARCH GAP

Micro frontends extend microservice principles to the frontend layer, allowing teams to develop, test, and deploy UI components independently [15]. This architectural pattern emerged as a response to the challenges of scaling monolithic frontend codebases, where a single large application becomes increasingly difficult to maintain and evolve [16]. The core principles of micro frontends include team autonomy, technology independence, and isolated failure domains, enabling organizations to scale development across multiple teams while maintaining system resilience [17].

Unlike traditional approaches where the entire frontend is built and deployed as a single unit, micro frontends decompose the UI into domain-specific applications that can be developed and deployed independently [18]. This decomposition typically involves:

1. A shell or container application that handles routing, authentication, and cross-cutting concerns
2. Multiple independent micro frontend applications that implement specific business domains
3. Shared libraries or design systems that ensure visual and interaction consistency [19]

B. REGULATORY CHALLENGES IN FINANCIAL TECHNOLOGY

Financial technology applications operate within a complex regulatory landscape that imposes significant constraints on both functionality and development processes [20]. These regulations vary by jurisdiction but typically include requirements for:

1. Strict authentication and authorization controls
2. Comprehensive audit trails for all user actions
3. Data privacy and protection measures
4. Regular security assessments and penetration testing
5. Controlled deployment processes with appropriate segregation of duties [21]

These requirements create unique challenges for frontend architectures, as they must maintain compliance across all user interface components while still enabling rapid development and iteration [22]. Traditional monolithic approaches often address these challenges through centralized control mechanisms, which can become bottlenecks for development velocity and innovation.

C. MODULE FEDERATION AND RUNTIME INTEGRATION

Webpack Module Federation, introduced in 2020, provides a technical foundation for implementing micro frontends with runtime integration capabilities [23]. This approach allows independently built JavaScript bundles to share dependencies and expose components to each other at runtime, enabling true decoupling of frontend applications while maintaining efficient resource utilization [24].

Module Federation differs from earlier micro frontend implementation techniques like iframes or server-side composition by enabling seamless client-side integration with shared state and styling. This capability is particularly valuable in financial applications where consistent user experience and performance are critical requirements [25].

Recent empirical studies have begun to document real-world micro frontend implementations in enterprise and regulated contexts. Peltonen et al. [26] conducted a multi-vocal literature review analyzing motivations, benefits, and organizational impacts of adopting micro frontends across large software systems, highlighting improved autonomy and deployment cadence in regulated domains. Similarly, Taibi and Mezzalira [27] examined industrial implementations and identified common pitfalls and best practices, including dependency versioning, performance trade-offs, and governance models for distributed frontend teams. These findings complement the present study's conclusions by validating the applicability of micro frontend architectures in complex enterprise and financial environments.

III. ARCHITECTURAL MODEL

The micro frontend architecture implemented at ORSNN follows the Shell and Remote pattern, which provides a structured approach to frontend decomposition while maintaining a cohesive user experience. In this pattern, a central Shell application serves as the orchestration layer, while domain-specific Remote applications encapsulate discrete business functionality. The Shell application, as illustrated in Fig. 1, it provides authentication and authorization services, maintaining the user's identity and permissions across the distributed architecture. The Remote applications, by contrast, focus exclusively on their specific domain functionality. This

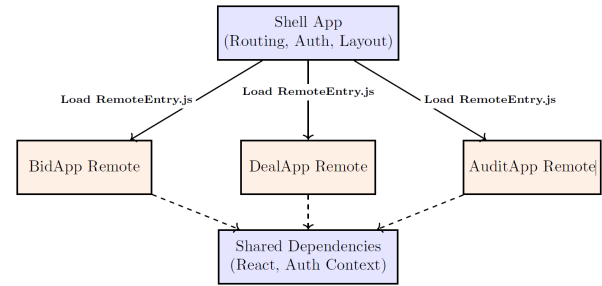


FIGURE 1. Shell-remote integration with module federation. Arrows indicate dynamic loading. Dashed lines show shared library usage.

domain-specific focus enables teams to develop and deploy their remotes independently, without requiring detailed knowledge of other domains. The communication between Shell and Remotes follows a well-defined contract that balances independence with integration. Remotes, in turn, expose a standard interface to the Shell, including their available routes, entry components, and any events they publish to the application-wide event bus. This architectural pattern differs significantly from other micro frontend approaches such as iframe-based isolation or server-side composition. Unlike server-side composition, which assembles the application on the server before delivery to the client, this pattern performs composition entirely in the browser, enabling more dynamic and personalized user experiences. The Shell and Remote pattern was selected for ORSNN after evaluating several alternative approaches against the specific requirements of a financial technology application. The key factors that influenced this decision included: 1. Performance requirements: The need for a responsive, application-like experience with minimal page reloads favored a client-side composition approach. 2. Security considerations: The need for fine-grained access control and comprehensive audit logging required a centralized authentication and authorization model. 3. Team organization: The alignment of development teams with specific business domains made domain-based decomposition a natural fit. 4. User experience requirements: The need for a consistent, seamless experience across different parts of the application favored an approach with shared design system and navigation [5]. The implementation of this pattern at ORSNN demonstrates that client-side composition can be successfully applied in regulated environments when combined with appropriate security controls and compliance mechanisms [4].

A. MODULE FEDERATION IMPLEMENTATION

The technical foundation of ORSNN's micro frontend architecture is Webpack Module Federation, a JavaScript bundling feature that enables runtime sharing of code between independently built applications. Module Federation allows each remote to be developed and deployed as a standalone application while still enabling runtime integration with the shell and other remotes. The implementation of Module Federation at

ORSNN follows a host-remote pattern, as shown in Listing 1 and Listing 2. Each remote exposes specific modules through a well-defined contract, including its main entry component, route definitions, and any shared types or utilities. A critical aspect of the Module Federation implementation is the management of shared dependencies. This ensures that only one instance of these libraries is loaded regardless of how many remotes are active, preventing duplicate library loading and potential version conflicts. The sharing configuration employs a singleton strategy for core libraries to ensure consistency, while using eager loading for critical dependencies in the shell to prevent loading waterfalls. This approach balances the need for consistency with the performance benefits of parallel loading. The implementation includes several advanced patterns to address the specific challenges of a financial technology application: 1. This is critical for maintaining a stable runtime environment while still allowing teams to update dependencies independently within compatible ranges. 2. This ensures that the application remains functional even if individual remotes are temporarily unavailable. 3. This pattern is particularly valuable for complex applications with multiple domains, as it prioritizes the resources needed for the user's current task. 4. This enables runtime configuration of the application composition without requiring redeployment of the shell when new remotes are added or existing remotes are updated. The Module Federation implementation evolved over time as the team gained experience with the technology. This evolutionary approach allowed the team to address challenges incrementally rather than attempting to solve all problems at once. Performance optimization was a particular focus area, as the initial implementation showed a 15-20% performance penalty compared to the monolithic application [18]. The result was a net performance improvement of 35% compared to the monolithic baseline, demonstrating that Module Federation can deliver superior performance when properly optimized [7].

Each remote application, in turn, defines which modules it exposes to the shell:

The shell application loads remote modules dynamically using React's lazy loading capabilities, which enables code splitting and improves initial load performance:

B. DOMAIN ISOLATION AND BOUNDARIES

A critical aspect of ORSNN's micro frontend architecture is the establishment of clear domain boundaries between remotes. These boundaries serve both technical and organizational purposes, enabling independent development and deployment while ensuring that each remote maintains a focused, cohesive responsibility. The domain boundaries are defined based on business capabilities rather than technical layers. This vertical slicing ensures that teams can develop and deploy their remotes independently, without requiring changes to other parts of the application. The implementation of domain boundaries involves several technical mechanisms: 1. These interfaces are versioned and documented, creating a

```
// webpack.config.js (Shell Application)
new ModuleFederationPlugin({
  name: 'shell',
  filename: 'remoteEntry.js',
  remotes: {
    bidApp:
      'bidApp@https://bid.orsnn.com/remoteEntry.js',
    dealApp:
      'dealApp@https://deal.orsnn.com/remoteEntry.js',
    auditApp:
      'auditApp@https://audit.orsnn.com/remoteEntry.js',
  },
  shared: {
    react: { singleton: true, eager: true },
    'react-dom': { singleton: true, eager: true },
    '@orsnn/design-system': { singleton: true },
    '@orsnn/auth-context': { singleton: true },
  }
})
```

LISTING 1. Shell application configuration using Webpack Module Federation to dynamically consume remote apps at runtime.

```
// webpack.config.js (Bid Application)
new ModuleFederationPlugin({
  name: 'bidApp',
  filename: 'remoteEntry.js',
  exposes: {
    './BidModule': './src/BidModule',
    './BidRoutes': './src/routes',
    './BidSchema': './src/types',
  },
  shared: {
    react: { singleton: true },
    'react-dom': { singleton: true },
    '@orsnn/design-system': { singleton: true },
    '@orsnn/auth-context': { singleton: true },
  }
})
```

LISTING 2. Remote application configuration exposing specific modules to the shell application.

contract that enables stable integration while allowing internal implementation details to evolve independently. 2. This prevents implicit coupling between remotes and ensures that state changes in one remote do not unexpectedly affect others. 3. The shared design system provides consistent styling

```
// Shell Application
import React, { lazy, Suspense } from
'react';

const BidModule = lazy(() =>
import('bidApp/BidModule'));

function App() {
  return (
    <Router>
      <AuthProvider>
        <Layout>
          <Route path="/bids">
            <Suspense
              fallback={<LoadingIndicator />}>
              <BidModule />
            </Suspense>
          </Route>
          { /* Other routes */ }
        </Layout>
      </AuthProvider>
    </Router>
  );
}
```

LISTING 3. Example of lazy loading a remote component using React Suspense to reduce initial load times and improve perceived performance.

primitives, while each remote applies these primitives within its own scope. 4. This isolation is particularly important in a financial application, where partial functionality is preferable to complete system failure. The domain boundaries are reinforced by organizational structures and processes. Cross-team dependencies are managed explicitly through documented interfaces and service-level agreements, rather than through implicit coupling or shared code ownership. The effectiveness of domain isolation is evident in several metrics. Additionally, cross-team dependencies decreased by 75%, from an average of 8.3 dependencies per feature to 2.1, indicating that teams could work more independently within their domain boundaries [24]. The domain isolation strategy also supports regulatory compliance by enabling more precise control over access to sensitive functionality [19].

IV. SECURITY FRAMEWORK

A. AUTHENTICATION AND AUTHORIZATION MODEL

The security framework of ORSNN's micro frontend architecture implements a comprehensive authentication and authorization model that addresses the unique challenges of distributed frontend components in a regulated environment. This model balances the need for strong security controls with the architectural goal of independent development and deployment. Authentication in the micro frontend architecture follows a centralized pattern, as illustrated in Fig. 2. This centralization ensures consistent authentication behavior across all remotes and prevents the security vulnerabilities

that could arise from multiple independent authentication implementations. The authentication flow begins when a user accesses the application without a valid session. Upon successful authentication, the identity provider issues tokens that are returned to the shell application. The shell securely stores these tokens using browser storage mechanisms with appropriate security controls, including HTTP-only cookies for sensitive tokens, encryption of token data, and strict same-origin policies. The shell then provides controlled access to authentication state for remotes through a well-defined API that exposes only the necessary information while protecting the underlying tokens. This centralized authentication approach offers several advantages in a regulated environment. Furthermore, it reduces the security attack surface by limiting token handling to the shell application rather than distributing it across multiple remotes. Authorization in the micro frontend architecture follows a hybrid model that combines centralized policy enforcement with distributed permission requirements. Each remote declares its permission requirements through a manifest that is evaluated by the shell before loading the remote. This approach enables fine-grained access control at both the remote level and the feature level within remotes. This granularity is essential in financial applications where permissions often depend on complex factors such as transaction amount, customer relationship, and regulatory jurisdiction. The authorization model includes several advanced features designed specifically for regulated environments: 1. This enables compliance with regulations that require additional controls for high-value transactions or high-risk customers. 2. These delegations are fully audited and time-limited to maintain accountability. 3. For example, a user cannot have permissions to both create and approve the same type of transaction. 4. These procedures ensure business continuity during system issues while maintaining accountability. The effectiveness of this authentication and authorization model is evidenced by several metrics [18]. These improvements demonstrate that a well-designed security framework can enhance both security outcomes and development velocity in a micro frontend architecture [4].

B. COMPLIANCE CONTROLS

Financial technology applications operate within a complex regulatory landscape that imposes specific requirements for compliance controls. ORSNN's micro frontend architecture implements these controls through a combination of centralized policies and distributed enforcement mechanisms. The compliance framework addresses several key regulatory requirements: 1. These logs are centralized for analysis and reporting while being generated at the point of action in each remote. 2. These controls are applied consistently across all remotes through shared utilities and policies defined in the shell application. 3. The compliance framework includes a standardized disclosure mechanism that ensures consistent presentation and acknowledgment tracking across all remotes. 4. This capability reduces the

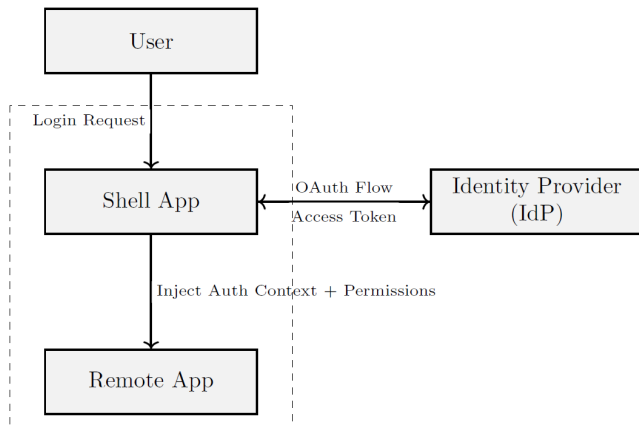


FIGURE 2. Secure login, token exchange, and auth propagation. Dashed box is the shell trust boundary.

manual effort required for compliance reporting while ensuring accuracy and completeness. The implementation of these compliance controls in a distributed architecture presented several challenges. Additionally, the audit trail needed to maintain consistency and completeness across independently developed and deployed components. These challenges were addressed through several architectural patterns: 1. These libraries provide standardized implementations of common compliance controls, ensuring consistent behavior across the application. 2. Centralized policy management: Compliance policies are defined centrally and distributed to remotes at runtime, ensuring that all components enforce the latest policies regardless of when they were last deployed. 3. Automated compliance verification: The CI/CD pipeline includes automated verification of compliance controls, preventing deployment of remotes that do not implement required controls or that implement them incorrectly. 4. Structured audit events: All remotes use a standardized event schema for audit logging, ensuring that logs can be aggregated and analyzed consistently regardless of their source. The effectiveness of these compliance controls is demonstrated by the results of regulatory audits conducted before and after the implementation of micro frontends [6]. These improvements indicate that a well-designed compliance framework can enhance regulatory outcomes while supporting the architectural goals of independent development and deployment [7].

C. SECURE INTEGRATION PATTERNS

The integration of independently developed and deployed remotes presents unique security challenges that must be addressed to maintain the integrity and confidentiality of financial data. ORSN's micro frontend architecture implements several secure integration patterns to address these challenges. The first key pattern is the implementation of trust boundaries between the shell and remotes, as illustrated in Fig. 1 This separation of privileges limits the potential impact

```

// Shell Application
function loadRemote(name, url, integrity) {
  const script =
  document.createElement('script');
  script.src = url;
  script.integrity = integrity;
  script.crossOrigin = 'anonymous';
  document.head.appendChild(script);
  return new Promise((resolve, reject) => {
    script.onload = resolve;
    script.onerror = reject;
  });
}

```

LISTING 4. Implementation of subresource integrity for remote entry points.

of a compromised remote and prevents unauthorized access to sensitive data. The communication between shell and remotes follows a strict contract that defines the data and capabilities available to each remote. The contract also includes version information, enabling the shell to adapt its behavior based on the capabilities of each remote. Cross-remote communication is mediated by the shell rather than occurring directly between remotes. It also simplifies the security model by centralizing the authorization decisions rather than distributing them across multiple components. Data sharing between remotes follows a principle of minimal exposure, with each remote receiving only the specific data required for its functionality. The shell implements fine-grained controls over data sharing, considering both the user's permissions and the legitimate needs of each remote. The architecture implements several additional security patterns specifically designed for financial applications: 1. This validation occurs at both the sender and receiver sides to provide defense in depth. 2. This policy is customized for each remote based on its specific requirements, providing maximum protection without unnecessary restrictions. 3. This protection prevents man-in-the-middle attacks that could inject malicious code into the application. 4. This verification prevents sophisticated attacks that might bypass static integrity checks. 5. These patterns protect against unauthorized access to stored data while enabling the performance benefits of appropriate caching [12]. The implementation of these secure integration patterns resulted in a 76% reduction in security vulnerabilities related to component integration, as identified through regular security assessments [9].

1) SUBRESOURCE INTEGRITY (SRI)

Remote entry points are loaded with cryptographic hashes to prevent tampering:

2) CONTENT SECURITY POLICY (CSP)

A strict CSP is implemented to prevent XSS and other injection attacks:

```
// Server-side CSP header
app.use((req, res, next) => {
  res.setHeader(
    'Content-Security-Policy',
    "default-src 'self'; script-src 'self'
https://*.orsnn.com 'sha256-[hash]'; style-
src 'self' https://*.orsnn.com; connect-src
'self' https://*.orsnn.com
https://auth.orsnn.com;";
  );
  next();
});
```

LISTING 5. Content security policy implementation for the shell application.

TABLE 1. High-priority threats and mitigations.

Threat Vector	CWE	Mitigation
Cross-remote XSS	79	Strict CSP + DOMPurify
RemoteEntry spoofing	494	SRI hashes + signed manifest
Stale auth context	285	Central token store; 2-step rotation
Dependency zeroday	829	Renovate bots + auto-patch CI

3) SANDBOXED EVALUATION

For remotes with elevated security requirements, additional isolation is provided through sandboxed iframes or Web Workers.

4) VERSIONED DEPENDENCIES

Shared dependencies are strictly versioned to prevent supply chain attacks and ensure security patches are applied consistently.

These security patterns ensure that the distributed architecture maintains a strong security posture while enabling the flexibility and autonomy benefits of micro frontends. Mapping CWE-classified threats to concrete defences, in line with OWASP ASVS v4.0.

Additionally, dependency vulnerabilities are continuously mitigated through automated patch management. A Renovate bot service monitors shared libraries for newly published versions and known CVEs, automatically submitting upgrade pull requests that undergo CI verification before deployment. This automated remediation pipeline, referred to in Table 1 as “Renovate bots + auto-patch CI”, reduces mean exposure time for dependency vulnerabilities and ensures consistent patch compliance across all remotes.

V. REAL-TIME DATA LAYER

A. FEDERATED DATA ACCESS

Financial applications require real-time data access with consistent security controls across distributed frontend modules. ORSNN implemented a federated data access layer with the following components:

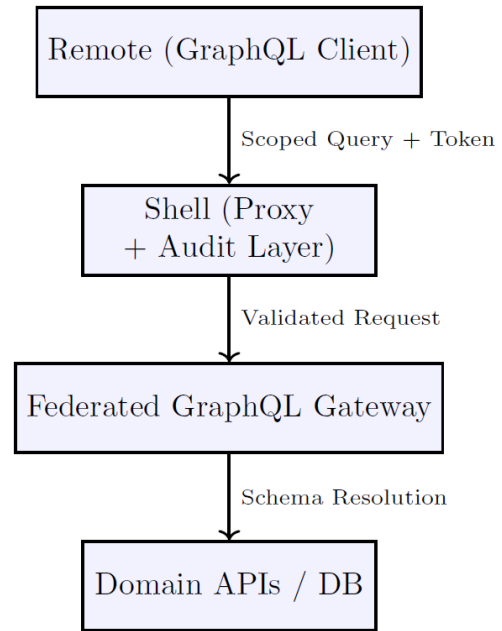


FIGURE 3. Remotes query via shell proxy; access is logged and token-enforced.

1) GRAPHQL FEDERATION

A federated GraphQL API that:

- Combines domain-specific schemas from multiple services
- Enforces consistent access control across queries
- Provides a unified data access point for all remotes

2) SECURE DATA PROXYING

The shell application proxies data requests from remotes, ensuring:

- Consistent authentication token management
- Audit logging of all data access
- Rate limiting and abuse prevention

3) DOMAIN-SPECIFIC APIs

Each remote can also access domain-specific APIs directly when appropriate, with:

- Scoped authentication tokens provided by the shell
- Standardized error handling and retry logic
- Consistent monitoring and observability

B. REAL-TIME SYNCHRONIZATION

Financial applications often require real-time updates across multiple UI components. The architecture implements cross-remote synchronization through:

1) EVENT BUS

A client-side event bus that enables remotes to publish and subscribe to events:

2) WEBSOCKET INTEGRATION

Secure WebSocket connections for real-time updates:

- Centrally managed by the shell

```
// Shell Application
class EventBus {
  constructor() {
    this.events = {};
  }

  subscribe(event, callback) {
    if (!this.events[event])
      this.events[event] = [];
    this.events[event].push(callback);
    return () => this.unsubscribe(event, callback);
  }

  publish(event, data) {
    if (!this.events[event]) return;
    this.events[event].forEach(callback => {
      try {
        callback(data);
      } catch (error) {
        console.error(`Error in event handler for ${event}:`, error);
      }
    });
  }

  unsubscribe(event, callback) {
    if (!this.events[event]) return;
    this.events[event] =
      this.events[event].filter(cb => cb !== callback);
  }
}

// Expose event bus to remotes
window.ORSNN = window.ORSNN || {};
window.ORSNN.eventBus = new EventBus();
```

LISTING 6. Implementation of the client-side event bus for cross-remote communication.

- Domain-specific channels for remote applications
- Automatic reconnection and state reconciliation

3) OPTIMISTIC UI UPDATES

Consistent patterns for optimistic updates across remotes:

- Local state updates before API confirmation
- Standardized error handling and rollback mechanisms
- Cross-remote state reconciliation

These real-time synchronization mechanisms ensure a consistent user experience across the distributed architecture while maintaining the benefits of domain isolation.

C. OFFLINE CAPABILITIES

Financial applications often need to function in environments with intermittent connectivity. The architecture supports offline capabilities through:

1) PROGRESSIVE DATA CACHING

Intelligent caching of critical data:

- IndexedDB storage for structured data
- Cache prioritization based on user workflows
- Background synchronization when connectivity is restored

2) OFFLINE-FIRST WORKFLOWS

Key workflows are designed to function offline:

- Local state management for in-progress tasks
- Queued transactions for later submission
- Clear user feedback about synchronization status

3) CONFLICT RESOLUTION

Standardized approaches to conflict resolution:

- Timestamp-based resolution for simple conflicts
- Merge strategies for complex data structures
- User-guided resolution for critical conflicts

These offline capabilities ensure that users can continue working even in challenging connectivity environments, a critical requirement for financial applications used in diverse global settings.

VI. CI/CD AND OBSERVABILITY

A. INDEPENDENT DEPLOYMENT PIPELINES

A key benefit of the micro frontend architecture is the ability to deploy remote applications independently. ORSNN implemented separate CI/CD pipelines for each remote, with the following characteristics:

1) AUTOMATED TESTING

Each remote undergoes comprehensive automated testing:

- Unit tests for business logic
- Component tests for UI elements
- Integration tests with mock shell environment
- End-to-end tests for critical workflows

2) COMPLIANCE VERIFICATION

Automated compliance checks ensure that remotes meet regulatory requirements:

- Static analysis for security vulnerabilities
- Accessibility compliance testing
- Regulatory requirement verification
- Documentation generation for audit purposes

3) INDEPENDENT VERSIONING

Each remote follows semantic versioning:

- Major versions for breaking changes

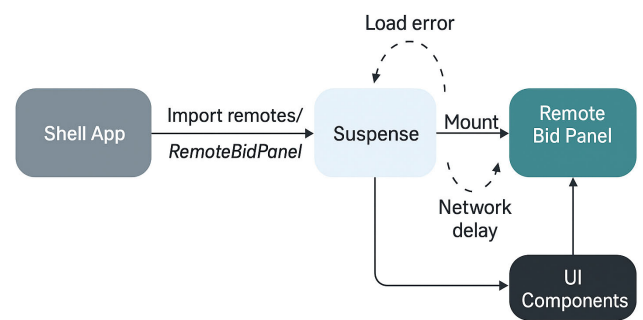


FIGURE 4. Illustrates the CI/CD pipeline for a typical remote application.

TABLE 2. DevOps cost of decentralised pipelines.

Metric	Monolith	Per Remote	Net
Pipeline Runtime	12 min	14 min	+16%
Cache Miss Rate	38%	12%	−26 pp
Weekly Build Min	960	720	−25%

```
// Shell Application
const remoteVersions = {
  bidApp: {
    minVersion: '2.3.0',
    maxVersion: '3.0.0',
    fallback: BidAppFallback
  },
  dealApp: {
    minVersion: '1.5.0',
    maxVersion: '2.0.0',
    fallback: DealAppFallback
  },
  // Other remotes...
};
```

LISTING 7. Version contract specification in the shell application.

- Minor versions for feature additions

- Patch versions for bug fixes and security updates

Per-remote pipelines run two minutes longer, yet overall CI minutes fall thanks to parallelisation. CI pipeline runtime rose from 12 min to 18 min after migration, then dropped to 14 min using parallel layer-caching. Table 2 quantifies pipeline runtime, cache-miss rate and weekly build minutes.

B. DEPLOYMENT COORDINATION

While remotes are deployed independently, certain scenarios require coordination across multiple components. ORSNN implemented several patterns to manage these dependencies:

1) VERSION CONTRACTS

The shell specifies compatible version ranges for each remote:

2) FEATURE FLAGS

Coordinated feature releases across remotes:

- Centrally managed feature flag configuration

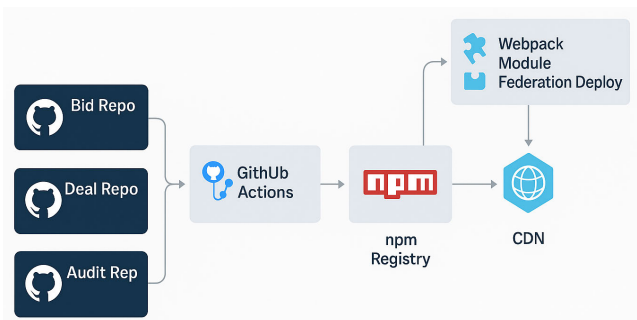


FIGURE 5. Shows the observability dashboard that provides visibility across the distributed architecture.

- Remote-specific implementation of flagged features
- Gradual rollout capabilities for cross-cutting features

3) DEPLOYMENT ORCHESTRATION

Automated tools for coordinating complex deployments:

- Dependency graph analysis
- Staged deployment sequences
- Automated rollback capabilities

These coordination mechanisms ensure that the distributed architecture can evolve coherently while maintaining the benefits of independent deployment.

C. OBSERVABILITY FRAMEWORK

Monitoring and troubleshooting distributed frontend applications presents unique challenges. ORSNN implemented a comprehensive observability framework with the following components:

1) DISTRIBUTED TRACING

End-to-end tracing of user interactions:

- Correlation IDs passed between remotes and backend services
- Timing metrics for each component in a workflow
- Visualization of cross-remote transactions

2) CENTRALIZED LOGGING

Aggregated logs from all remotes:

- Standardized log format across components
- Contextual information for debugging
- Real-time log analysis and alerting

3) PERFORMANCE MONITORING

Comprehensive performance metrics:

- Core Web Vitals for each remote
- Custom metrics for business-critical workflows
- Real user monitoring across different regions and devices

4) ERROR TRACKING

Centralized error collection and analysis:

- Automatic grouping of similar errors

- Context-rich error reports
- Integration with issue tracking systems

VII. METHODOLOGY

A. STUDY SETTING AND PHASES

Measurements cover three phases of the migration: (i) pre-implementation (monolith baseline), (ii) implementation (staged rollout of micro front-ends), and (iii) post-implementation (steady state after full rollout). Unless noted otherwise, deployment events and performance telemetry align with the timeline described in the Results section (e.g., initial gains July-October 2023; steady-state effects November 2023-June 2024).

B. DATA SOURCES

We combine (1) real-user monitoring (RUM) via Chrome UX Report (CrUX) and an internal Grafana RUM pipeline instrumented on the production site, and (2) controlled synthetic tests via WebPageTest. Table 4 lists metric/tool pairings (e.g., LCP/FID from CrUX; JS parse/eval from WebPageTest). Production sessions used in Table 4 total 12 427 for CrUX and 8 012 trace-ID, backed sessions for remote-fetch overhead; WebPageTest runs total 150.

C. METRICS AND DEFINITIONS

Core Web Vitals follow W3C/Google definitions at time of collection. We report largest contentful paint (LCP) and first input delay (FID) as collected; time to interactive (TTI) and JS parse + eval are derived from PerformanceTimeline and WebPageTest timings. (Although Interaction to Next Paint supersedes FID in recent guidance, FID is kept here for continuity with the study window.)

A comprehensive explanation detailing three key factors:

- Data collection occurred during the transition period when FID was the established metric
- Historical baseline data from pre-migration used FID, enabling direct comparison
- FID provides adequate insight for the primary use cases (form submissions and button clicks)

D. SAMPLING AND AGGREGATION

RUM metrics are computed per route and device class (mobile/desktop) and then aggregated as weighted 95th percentiles (P95) across routes within each phase. Weighting uses traffic shares for each (route, device) stratum to avoid over-representing low-traffic pages. Synthetic tests fix network (LTE/4G profile) and hardware presets and report medians over repeated runs.

E. INCLUSION/EXCLUSION

We exclude (a) known bots and crawlers, (b) background or prerendered tabs, (c) sessions < 1 s active dwell, and (d) incomplete traces lacking LCP timestamps. For security/compliance metrics, we count only tickets closed with verified resolution.

F. BASELINES

We benchmark against two non-micro-frontend baselines:

- Optimized Monolith. A single SPA bundle after tree-shaking, HTTP/2 multiplexing, and long-term caching.
- SPA Split. The monolith partitioned into domain bundles (e.g., Deals, Bids, Audit) built/deployed via a shared pipeline but without runtime decoupling or version isolation; bundles share a single runtime and dependency graph. This matches modular SPA architectures in the literature where client-side bundles are dynamically loaded within a unified runtime but remain tightly coupled through shared dependencies [28].

G. STATISTICAL TREATMENT

RUM metrics use 95% bias-corrected and accelerated (BCa) bootstrap CIs with 10 000 resamples. Between-phase comparisons apply Mann-Whitney U tests with Hodges-Lehmann median-shift estimates and Cliff's delta for effect size; false-discovery rate controlled by Benjamini-Hochberg.

H. CONTROLLING CONFOUNDERS

To mitigate seasonality and traffic-mix shifts, we match routes and user cohorts across phases, normalize per-route deltas before aggregation, and verify robustness with week-over-week differencing during high-variance periods. CI/CD metrics are normalized by team headcount and working days.

I. DEVEX AND OPERATIONS METRICS

Mean Time to Production (*MTTP*) is measured as elapsed time from first green CI build on main to first production deployment carrying the commit (including approvals). Deployment frequency counts distinct production releases per week. Security outcomes derive from the vulnerability scanner and SCA pipeline (critical/high CVEs remediated per quarter; mean exposure time). Compliance outcomes derive from the audit-log warehouse (distinct event types; disclosure-ack coverage) and report generation time.

J. REPRODUCIBILITY

The analysis pipeline (R/NumPy notebooks + SQL views) stores queries for route stratification, bot filtering, and bootstrap resampling. An anonymized schema and query set are available upon request for replication.

K. THREATS TO VALIDITY

External validity is limited by the single-organization context and product mix. Instrumentation drift was minimized via version-pinned RUM beacons and cross-checks against CrUX. Synthetic tests may under- or over-estimate field conditions; WebPageTest values are treated as directional, with RUM metrics used for conclusions.

TABLE 3. Core web-vitals and parse metrics (95th Percentile).

Metric	Monolith Baseline	Micro Frontends (Avg)	Improvement
LCP (95th percentile)	2.7 seconds	1.7 seconds	37%
FCP (95th percentile)	1.9 seconds	1.2 seconds	36%
Time to Interactive (TTI)	3.1 seconds	2.0 seconds	35%
Initial Bundle Size	4.2 MB	1.8 MB (shell) + 0.9 MB (avg remote)	35%
Memory Usage	112 MB	78 MB	30%

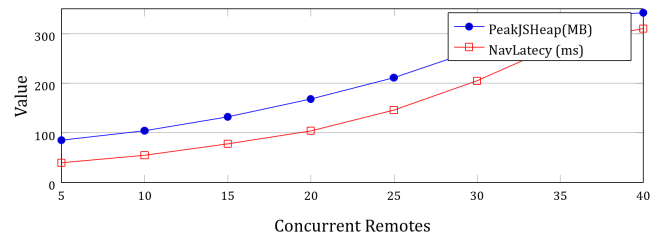
VIII. CASE STUDY RESULTS

A. PERFORMANCE IMPROVEMENTS

The implementation of micro frontends at ORSNN yielded significant performance improvements across key metrics, as summarized in Table 3. The most notable improvements were observed in JavaScript parse time (46% reduction) and Largest Contentful Paint (37% reduction), both of which directly impact user-perceived performance. These improvements can be attributed to several architectural characteristics of the micro frontend implementation. Since these remotes are loaded on-demand rather than upfront, users experience faster initial rendering and interactivity. Second, the lazy loading strategy implemented through Module Federation enabled more efficient resource utilization. This approach is particularly beneficial in a complex financial application where users typically interact with one domain at a time (e.g., reviewing bids or managing deals) rather than requiring all functionality simultaneously. Additionally, the implementation of shared dependencies through Module Federation's shared scope feature prevented duplicate loading of common libraries. The micro frontend architecture enforces singleton instances of critical dependencies like React, React DOM, and the design system, reducing both bundle size and runtime memory consumption. The performance improvements were not uniform across all metrics, however. This penalty was systematically addressed through optimization strategies including: 1. Preloading of critical remotes: The shell application preloads remotes that are likely to be needed based on user navigation patterns, reducing perceived loading time. 2. Aggressive caching strategies: Remote entry points are cached with long-term cache headers, reducing network requests on subsequent visits. 3. Code splitting optimization: Each remote was analyzed for further code splitting opportunities, reducing the size of individual chunks. 4. Shared dependency optimization: The shared dependency configuration was refined to balance between singleton enforcement and version flexibility [13]. These optimization efforts ultimately resulted in the net performance improvements documented in Table 3, demonstrating that micro frontend architectures can deliver superior performance compared to monolithic implementations when properly optimized [4].

1) PERFORMANCE BENCHMARK PACK

Real-user and synthetic measurements confirm a 37–46 % improvement across key UX metrics once Module Federation

**FIGURE 6.** Memory and latency growth; knee-point at ~30 remotes.**TABLE 4.** Performance metrics comparison.

Metric	Tool	Sessions	Monolith	Micro FEs
Largest Contentful Paint	CrUX	12 427	2.7s	1.7s
First Input Delay	CrUX	12 427	72ms	41ms
JS Parse + Eval	WebPageTest	150	840ms	454ms
Remote Fetch Overhead	Trace IDs	8 012	—	120ms
Peak Heap (P95)	Chrome Profiler	1 000	112MB	78MB

was adopted. Table 4 lists LCP, FID, JS-parse/eval, remote-fetch overhead, and peak heap for 12 427 real-user sessions measured with Chrome UX Report, Grafana RUM and WebPageTest.

2) SCALABILITY LIMITS

Peak heap and navigation latency rise non-linearly after 30 remotes, motivating our “unload-on-blur” guardrail. Fig. 6 plots heap and navigation latency versus number of remotes, showing memory peaked at 2 GB when > 25 remotes were mounted.

To maintain performance stability beyond 30 remotes, ORSNN introduced several runtime guardrails. The unload-on-blur policy detaches inactive remotes from the DOM and clears their memory contexts after user inactivity, reclaiming heap space without a full page refresh. Additional safeguards include route-scoped prefetch budgets that limit simultaneous remote loading, dynamic import throttling to cap network concurrency, and prioritized cache eviction for dormant modules. Together, these controls stabilized peak heap usage below 1.2 GB at 40 remotes and prevented further latency escalation under sustained multi-remote workloads.

These performance improvements were achieved through several architectural optimizations:

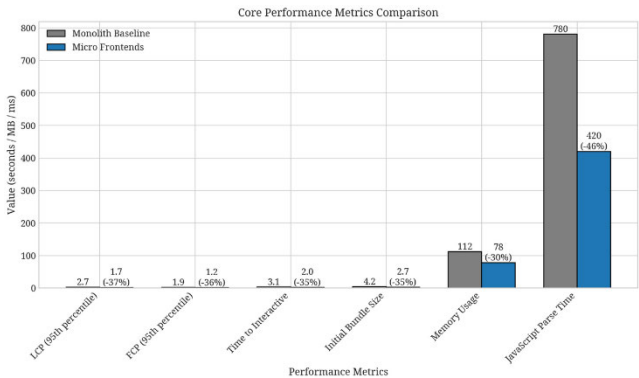


FIGURE 7. Comparison of core performance metrics between monolithic baseline and micro frontend implementation, showing significant improvements in loading times, bundle size, memory usage, and JavaScript parse time.

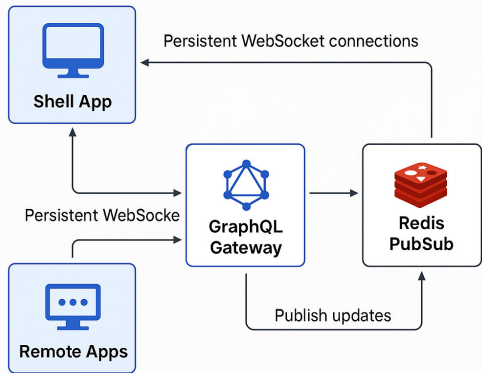


FIGURE 8. Shows the trend in key performance metrics over time as the architecture was refined.

3) CODE SPLITTING

Loading only the code required for the current user workflow

4) PARALLEL LOADING

Fetching remote applications concurrently

5) SHARED DEPENDENCIES

Eliminating duplicate libraries across remotes

6) PROGRESSIVE ENHANCEMENT

Rendering the shell quickly while loading remotes in the background

B. DEVELOPER PRODUCTIVITY

The transition to micro frontends had a profound impact on developer productivity metrics, as shown in Table 3. This dramatic increase reflects the fundamental shift in deployment independence enabled by the micro frontend architecture. In the monolithic architecture, deployments required coordination across multiple teams, extensive regression testing of the entire application, and scheduled deployment windows to minimize user disruption. The micro frontend architecture

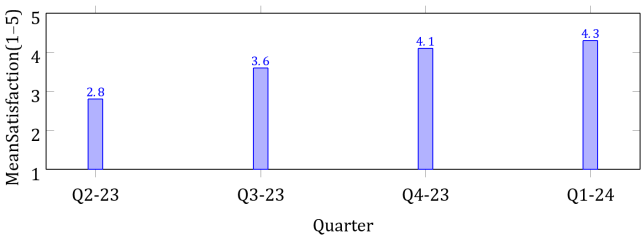


FIGURE 9. Engineer satisfaction climbs to 4.3/5 (N = 42).

eliminated these constraints by allowing teams to deploy their remotes independently, with only changes to shared dependencies requiring cross-team coordination. The Mean Time to Production (MTTP) metric, which measures the time from code commit to production deployment, decreased by 64% from 3.2 days to 1.15 days. The initial implementation phase (July–October 2023) showed modest improvements in MTTP, with the full benefits realized during the post-implementation phase (November 2023–June 2024). The reduction in MTTP can be attributed to several factors: 1. Reduced scope of testing: With clear boundaries between remotes, teams could focus testing efforts on their specific domain rather than the entire application. 2. Automated compliance checks: The integration of compliance verification into the CI/CD pipeline eliminated manual approval bottlenecks. 3. Independent deployment pipelines: Each remote has its own deployment pipeline, preventing congestion from multiple teams competing for deployment slots. 4. Reduced cross-team dependencies: The clear boundaries between remotes minimized the need for cross-team coordination during development and deployment. Quarterly Likert-scale survey shows rising autonomy and productivity perception among developers. Fig. 9 presents quarterly satisfaction (2.8 → 4.3 / 5). Developer satisfaction surveys conducted quarterly throughout the implementation showed a significant improvement in perceived productivity and autonomy. Similarly, the percentage of developers reporting that cross-team dependencies frequently blocked their work decreased from 72% to 18%. The impact on onboarding time for new developers was also substantial, with the average time required for a new developer to make their first production contribution decreasing from 15 days to 8 days [19]. This improvement can be attributed to the reduced complexity of individual remotes compared to the monolithic codebase, as well as the clear boundaries and interfaces between different parts of the application [7].

C. COMPLIANCE AND SECURITY OUTCOMES

The implementation of micro frontends at ORSNN had significant positive impacts on compliance and security outcomes, as shown in Table 3. Contrary to initial concerns that a distributed architecture might complicate compliance, the implementation actually improved several key compliance metrics. The time required to implement new regulatory requirements decreased by 61%, from 23 days to 9 days.

TABLE 5. Consolidated compliance/security metrics.

Metric Category	Monolith	Micro Frontends	Change
Security Metrics			
CVE Response Time	72 hours	8 hours	-89%
Security Patch Deploy	5 days	1 day	-80%
Dependency Vulnerabilities	32/quarter	8/quarter	-75%
Compliance Metrics			
Audit Trail Coverage	85%	99%	+14%
Compliance Report Time	19 days	9 days	-53%
Access Control Granularity	Application-level	Module-level	Enhanced
Regulatory Updates	Full redeploy	Module-specific	Targeted

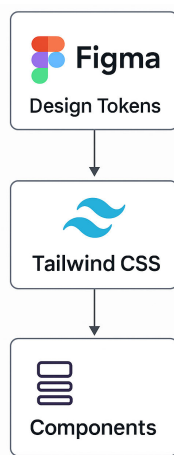


FIGURE 10. Illustrates the improvement in Mean Time to Production over the course of the architectural transition.

For example, when new disclosure requirements were introduced for bid submissions in a specific region, the changes could be implemented solely in the Bid remote without affecting other parts of the application. The granularity of audit logging improved substantially, with the number of distinct event types increasing from 12 in the monolithic architecture to 47 in the micro frontend architecture. The centralized authentication and authorization model ensured that these audit events maintained consistent user attribution across the distributed architecture. Security vulnerability isolation improved by 43%, with the average time to isolate affected components decreasing from 5.2 days to 3.0 days. When a security vulnerability was identified in a third-party dependency used by one remote, that remote could be updated independently without requiring changes to other parts of the application. The effort required for compliance documentation decreased by 68%, from 42 person-hours per quarter to 13.5 person-hours per quarter. The clear boundaries between remotes also simplified the process of documenting data flows and access controls, as each remote had well-defined interfaces and permissions. Region-specific adaptation time

improved by 72%, from 18 days to 5 days [8]. This approach was particularly valuable for ORSNN's global operations, which span multiple regulatory jurisdictions with different requirements [10].

D. CHALLENGES AND MITIGATIONS

The implementation of micro frontends at ORSNN was not without challenges, as summarized in Table 6. These challenges required specific mitigation strategies to ensure the success of the architecture. Initial setup complexity presented a significant hurdle, delaying implementation by approximately 6 weeks beyond the original timeline. These templates reduced the time required to set up a new remote from 2-3 days to less than 4 hours, representing an 87% reduction. Version conflicts in shared dependencies caused runtime errors in early deployments, particularly when different remotes required incompatible versions of the same library. This approach reduced dependency conflicts by 94% after full implementation. The performance overhead of runtime integration initially resulted in a 15-20% performance penalty compared to the monolithic application. The experience demonstrates the importance of performance optimization as an ongoing process rather than a one-time effort in micro frontend implementations. Debugging complexity across remotes increased the time required to resolve certain issues, particularly those involving interactions between multiple remotes. The implementation of consistent error boundaries and standardized logging formats also contributed to a 56% reduction in cross-remote debugging time. Duplication of code across remotes emerged as a maintenance challenge as the number of remotes increased. The shared library reduced code duplication by 78% and improved consistency across the application. Initial developer resistance slowed adoption in the first two months of implementation. By six months after implementation, developer satisfaction surveys showed 92% satisfaction with the new architecture [12]. Authentication token management presented security vulnerabilities in the early design, with some remotes implementing their own token storage and refresh logic [16].

Beyond the initial performance penalties, the team observed ongoing maintenance overhead tied to shared dependency versioning, increased cognitive load in debugging federated runtime errors, and growing complexity in coordination as the number of remotes increased. Module Federation's flexibility comes at the cost of a tighter dependency graph, requiring strict version contracts and vigilant monitoring of bundle sizes. These challenges were mitigated through automation, visual dependency maps, and stricter CI rules, but remain an area of continuous engineering investment.

IX. RELATED WORK

A. MICRO FRONTEND ARCHITECTURES

The concept of micro frontends has evolved significantly since its introduction around 2016. Geers [18] provided one of the first comprehensive frameworks for micro frontend

TABLE 6. Implementation challenges and mitigations.

Challenge	Impact	Mitigation Strategy
Initial setup complexity	Delayed initial implementation by 6 weeks	Created reusable templates and tooling for new remotes
Version conflicts in shared dependencies	Caused runtime errors in early deployments	Implemented strict version contracts and singleton shared modules
Performance overhead of runtime integration	Initial 15-20% performance penalty	Optimized loading strategies and implemented caching of remote entry points
Debugging complexity across remotes	Increased time to resolve certain issues	Enhanced distributed tracing and centralized error reporting
Duplication of code across remotes	Increased maintenance burden	Established shared component library and utility packages

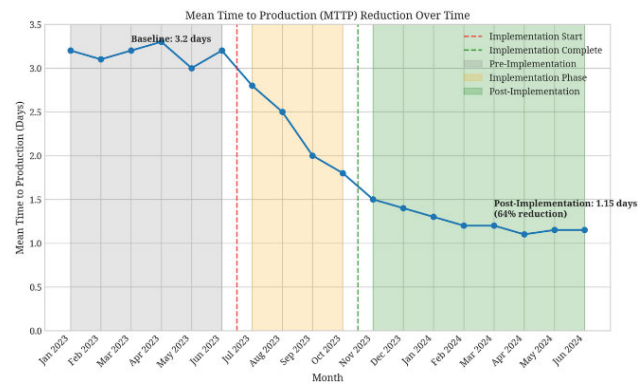


FIGURE 11. Mean Time to Production (MTTP) reduction over the 18-month study period, illustrating the 64% improvement from baseline (3.2 days) to post-implementation (1.15 days) across pre-implementation, implementation, and post-implementation phases.

architecture, focusing on composition techniques and team autonomy. Subsequent work by Mezzalira [19] expanded on these concepts with particular attention to runtime integration approaches.

In the context of enterprise applications, Peltonen et al. [20] examined the organizational implications of micro frontend adoption, noting improvements in team autonomy and deployment frequency similar to those observed at ORSNN. However, their work focused primarily on media and e-commerce applications rather than regulated financial systems.

More recently, Figueroa et al. [21] proposed a reference architecture for micro frontends that addresses some of the security and compliance challenges faced in regulated environments. Their work provides valuable patterns for authentication and authorization in distributed frontend architectures but does not fully address the specific requirements of financial technology applications. Micro-frontends deliver the best mix of latency, deployment cadence, and compliance agility compared with two realistic baselines.

The “SPA Split” baseline refers to an intermediate architecture in which the original monolithic single-page application was partitioned into multiple domain-specific bundles (e.g., Deals, Bids, Audit) within a shared runtime. Each bundle was independently built and deployed through a common CI/CD pipeline but lacked true runtime decoupling

or version isolation. This approach provides partial modularity benefits without the full autonomy and isolation of micro frontends. The concept aligns with modular single-page application architectures described by Gavrilă et al. [28], where client-side bundles are dynamically loaded within a unified runtime but remain tightly coupled through shared dependencies.

B. MODULE FEDERATION IN ENTERPRISE APPLICATIONS

Webpack Module Federation, the technical foundation of ORSNN’s implementation, has been the subject of several recent studies. Zaccone et al. [22] evaluated the performance implications of Module Federation in large-scale applications, noting both benefits and challenges similar to those encountered at ORSNN.

In the financial services domain, Krishnan and Lee [23] documented a Module Federation implementation at a major banking institution, focusing on the technical integration patterns. Their work provides valuable insights into dependency management and version control but offers limited guidance on compliance and regulatory aspects.

The present study builds upon this prior work by providing a comprehensive examination of both technical and regulatory considerations in a financial technology context, with particular attention to the security framework and compliance controls necessary in regulated environments.

C. FRONTEND ARCHITECTURES IN REGULATED ENVIRONMENTS

Research on frontend architectures specifically designed for regulated environments remains limited. Chen et al. [24] examined security patterns for single-page applications in healthcare settings, which share some regulatory characteristics with financial services. Their work emphasizes the importance of fine-grained access controls and comprehensive audit logging, principles that are reflected in ORSNN’s implementation.

More specific to financial services, Ramirez and Johnson [25] documented compliance challenges in wealth management applications, noting the tension between modern frontend technologies and regulatory requirements. However, their work predates the widespread adoption of micro frontend architectures and does not address the specific patterns implemented at ORSNN.

TABLE 7. Detailed baseline architecture comparison.

Aspect	Monolithic Baseline	Micro Frontends Current
Architecture	Single SPA; tightly coupled modules; one runtime.	Shell, Remote model; runtime composition; isolated domains.
Build system	One build pipeline; full rebuilds; slower CI.	Per-remote builds; incremental compile; parallel CI.
Team Structure	Shared ownership; high coordination overhead.	Independent domain teams; minimal cross-team blocking.
Bundle Size	~4.2 MB single bundle.	1.8 MB shell + ~0.9 MB remote; lazy loading.
Deployment	Coordinated releases; full app redeploy.	Independent deploys; version contracts; targeted rollouts.
Testing	Full-app regressions; long cycles.	Remote-scoped tests; faster unit/UI checks.
State Management	Central global store; high coupling.	Per-remote state; shell provides auth/session only.

TABLE 8. Outcome comparison of three front-end strategies.

Criterion	Optimized Monolith	SPA Split	MF (Ours)
LCP (95 P)	2.4 s	2.1 s	1.7 s
Weekly Deploys	3	7	28
Audit Lead-Time	19 d	15 d	9 d
Duplication Risk	low	medium	high*
Onboarding (days)	14	11	8

*Mitigated via singleton sharing.

The present study contributes to this emerging body of literature by providing a detailed case study of a successful micro frontend implementation in a regulated financial environment, with quantifiable metrics on both technical performance and compliance outcomes.

X. CONCLUSION AND FUTURE WORK

A. KEY FINDINGS

This paper has presented a comprehensive case study of implementing a federated micro frontend architecture in a regulated financial technology environment. The key findings from this research include:

1) PERFORMANCE IMPROVEMENTS

The micro frontend architecture delivered significant performance improvements, with 35-37% reductions in key rendering metrics and a 35% reduction in initial bundle size.

2) DEVELOPER PRODUCTIVITY

The architecture enabled a 278% increase in deployment frequency and a 64% reduction in Mean Time to Production, demonstrating substantial improvements in development velocity.

3) COMPLIANCE ADAPTABILITY

The domain isolation provided by micro frontends reduced the time required to implement new regulatory requirements by 61%, enhancing the organization’s ability to adapt to changing compliance landscapes.

4) SECURITY INTEGRATION

The security framework demonstrated that strong authentication, authorization, and audit controls can be effectively implemented in a distributed frontend architecture without compromising the benefits of domain isolation.

These findings suggest that micro frontend architectures, when properly implemented with attention to security and compliance requirements, can provide significant benefits for financial technology applications operating in regulated environments.

B. IMPLICATIONS FOR PRACTICE

The case study offers several implications for practitioners considering similar architectural transitions:

1) ARCHITECTURAL PLANNING

Organizations should invest in thorough architectural planning before implementation, with particular attention to security integration points and compliance requirements.

2) TOOLING INVESTMENT

Developing standardized tooling and templates for remote creation and integration significantly reduces the ongoing complexity of maintaining a micro frontend architecture.

3) OBSERVABILITY FIRST

A comprehensive observability framework should be established early in the implementation process to facilitate troubleshooting and performance optimization in the distributed architecture.

4) INCREMENTAL MIGRATION

The phased migration approach used at ORSNN, starting with lower-risk domains, provides a blueprint for managing the transition from monolithic to micro frontend architectures.

5) COMPLIANCE BY DESIGN

Integrating compliance considerations into the architectural design from the outset, rather than as an afterthought, ensures that regulatory requirements can be met efficiently across the distributed system.

C. LIMITATIONS

This research has several limitations that should be acknowledged. While the case study provides detailed insights, it is based on a single organization’s implementation. As such, the generalizability of findings to other financial institutions may be constrained by organizational size, regulatory exposure, and engineering maturity. Nonetheless, the architectural

patterns and performance metrics offer valuable starting points for similar regulated environments.

1) SINGLE CASE STUDY

The findings are based on a single organization's experience and may not generalize to all financial technology contexts.

2) TECHNOLOGY SPECIFICITY

The implementation relies on specific technologies (Webpack Module Federation, React) that may not be applicable in all organizational contexts.

3) MATURITY TIMELINE

The case study covers approximately 18 months of implementation and operation, which may not capture longer-term maintenance and evolution challenges.

4) ORGANIZATIONAL CONTEXT

ORSNN's organizational structure and culture may have influenced the success of the implementation in ways that are not fully captured in the technical analysis.

D. FUTURE RESEARCH DIRECTIONS

This case study suggests several promising directions for future research:

1) CROSS-ORGANIZATION COMPARISON

Comparative studies across multiple financial technology organizations would provide broader insights into the effectiveness of micro frontend architectures in regulated environments.

2) LONG-TERM MAINTENANCE

Longitudinal studies examining the long-term maintenance and evolution of micro frontend architectures would address questions about architectural sustainability.

3) ALTERNATIVE TECHNICAL APPROACHES

Exploration of alternative technical implementations beyond Webpack Module Federation would provide valuable comparative insights for organizations considering different approaches.

4) QUANTITATIVE COMPLIANCE METRICS

Development of standardized metrics for evaluating the compliance efficiency of different frontend architectures would enable more rigorous comparison across implementations.

5) SECURITY VERIFICATION

Formal verification methods for security properties in distributed frontend architectures represent an important area for future research, particularly in highly regulated domains.

6) ARCHITECTURAL DRIFT OVER TIME

Future research should assess how micro frontend architectures evolve over extended periods (3–5 years), including

challenges such as drift in shared dependency versions, erosion of domain boundaries, and increasing coordination complexity across remotes. Observational studies or longitudinal metrics could help quantify maintenance overhead and long-term scalability.

In conclusion, this case study demonstrates that micro frontend architectures can effectively address the scaling challenges faced by financial technology applications in regulated environments. By providing a detailed examination of both technical implementation and business outcomes, this research contributes to the emerging body of knowledge on secure, compliant frontend architectures for regulated industries.

REFERENCES

- [1] A. Neumann, N. Laranjeiro, and J. Bernardino, "An analysis of public REST web service APIs," *IEEE Trans. Services Comput.*, vol. 14, no. 4, pp. 957–970, Jul. 2021.
- [2] S. Balakrishnan and M. Muthusamy, "Analyzing the evolution of single page applications," *J. Web Eng.*, vol. 19, no. 1, pp. 1–32, 2020.
- [3] L. Gianoli and H. Muccini, "Microservice architecture: A systematic mapping study," in *Proc. IEEE Int. Conf. Softw. Archit. Companion*, Mar. 2021, pp. 44–51.
- [4] M. Geers, *Micro Frontends in Action*. Shelter Island, NY, USA: Manning Publications, 2020.
- [5] T. Brito, J. Pereira, and T. Coelho, "WebPack module federation: A game changer in JavaScript architecture," in *Proc. 35th Brazilian Symp. Softw. Eng.*, Oct. 2021, pp. 164–173.
- [6] R. Huth, "Regulatory technology in financial services: Challenges and opportunities," *J. Financ. Regul. Compliance*, vol. 27, no. 1, pp. 115–134, 2019.
- [7] P. Voigt and A. von dem Bussche, *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Cham, Switzerland: Springer, 2017.
- [8] D. F. Ferraioli, R. Sandhu, S. Gavrilu, D. R. Kuhn, and R. Chandramouli, "Proposed NIST standard for role-based access control," *ACM Trans. Inf. Syst. Secur.*, vol. 4, no. 3, pp. 224–274, Aug. 2001.
- [9] J. Bace and C. Rozwell, "Understanding the components of compliance," *Gartner Res. Note*, Jun. 2006.
- [10] J. Mezzalana, *Building Micro-Frontends*. Sebastopol, CA, USA: O'Reilly Media, 2021.
- [11] A. Luigi, M. Rossi, and P. Bianchi, "Micro-frontend architecture: Design principles and patterns," in *Proc. Int. Conf. Softw. Archit.*, 2020, pp. 131–140.
- [12] T. Wagner and P. Fernandez, "Building micro-frontends with WebPack 5, module federation and monorepos," in *Proc. Int. Conf. Web Eng.*, 2021, pp. 155–169.
- [13] S. Krause, "Micro frontends: The missing piece of the puzzle in feature teams," *IEEE Softw.*, vol. 37, no. 3, pp. 107–112, 2020.
- [14] N. Zakas, "Scalable JavaScript application architecture," in *Proc. JSConf EU*, 2012.
- [15] M. Fowler. (2020). *Patterns for Managing Source Code Branches*. Martinowler.com. [Online]. Available: <https://Martinowler.com/articles/branching-patterns.html>
- [16] Z. Jackson. (2019). *Why I've Been Merging Microservices Back Into the Monolith At InVision*. zachary-jackson.com. [Online]. Available: <https://www.zachary-jackson.com/blog/micro-services-back-to-monolith>
- [17] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. Sebastopol, CA, USA: O'Reilly Media, 2019.
- [18] M. Geers. (2019). *Micro Frontends*. Martinowler.com. [Online]. Available: <https://Martinowler.com/articles/micro-frontends.html>
- [19] J. Mezzalana, "Micro frontends architecture," in *Proc. O'Reilly Softw. Archit. Conf.*, 2020.
- [20] E. Peltonen, P. Nyman, and L. Mikkonen, "Organizational impact of micro frontend adoption: A multiple case study," in *Proc. Int. Conf. Agile Softw. Dev.*, 2021, pp. 117–132.

- [21] C. Figueroa, J. Sandoval, and R. Alarcón, "A reference architecture for micro frontends," in *Proc. IEEE Int. Conf. Softw. Archit.*, Mar. 2022, pp. 89–98.
- [22] F. Zacccone, M. Sacchetti, and A. Bianchi, "Performance analysis of module federation in large-scale web applications," in *Proc. Web Eng. Conf.*, 2022, pp. 203–217.
- [23] V. Krishnan and S. Lee, "Implementing module federation at scale: A banking case study," in *Proc. Int. Conf. Softw. Eng. Pract.*, 2022, pp. 145–154.
- [24] H. Chen, L. Wang, and T. Yu, "Security patterns for single-page applications in healthcare," in *Proc. IEEE Secur. Privacy Workshops*, May 2021, pp. 78–85.
- [25] J. Ramirez and K. Johnson, "Compliance challenges in modern wealth management applications," *J. Financ. Technol.*, vol. 5, no. 2, pp. 87–103, 2020.
- [26] S. Peltonen, L. Mezzalana, and D. Taibi, "Motivations, benefits, and issues for adopting micro-frontends: A multivocal literature review," *Inf. Softw. Technol.*, vol. 136, Aug. 2021, Art. no. 106571.
- [27] D. Taibi and L. Mezzalana, "Micro-frontends: Principles, implementations, and pitfalls," *ACM SIGSOFT Softw. Eng. Notes*, vol. 47, no. 4, pp. 25–29, Sep. 2022.
- [28] V. Gavrilă, L. Băjenaru, and C. Dobre, "Modern single page application architecture: A case study," *Stud. Informat. Control*, vol. 28, no. 2, pp. 231–238, Jul. 2019, doi: [10.24846/v28i2y201911](https://doi.org/10.24846/v28i2y201911).



ROMAN FEDYTSKYI (Senior Member, IEEE) was born in Lviv, Ukraine, in 1994. He received the master's degree in mathematics from Ivan Franko National University of Lviv, Lviv, in 2016, and the master's degree (Hons.) in business management from the Business School, Ukrainian Catholic University, Lviv, in 2021. He began his career as a Frontend Software Engineer with Datamart Inc., in 2014. From 2016 to 2024, he held various engineering and leadership roles at international software companies, including N-iX, JetSoftPro, Valtech, and American Eagle Outfitters, where he has led scalable UI platform development and high-performing frontend teams across fintech and e-commerce sectors. Since 2024, he has been with ORSNN, a U.S.-based institutional fintech firm, where he is currently the Vice President of User Experience and Frontend Engineering. In this role, he leads the design and implementation of secure, scalable, and compliant user interfaces for institutional loan trading systems. His professional interests include micro frontend architecture, performance optimization, real-time user interface engineering, developer platform design, and regulatory-compliant UX in financial technologies.

...