IJIT

INTERNATIONAL JOURNAL OF INFORMATION TECHNOLOGY

Publishing Refereed Research Article, Survey Articles and Technical Notes.

Journal ID: 4971-6785

IAEME Publication
Chennai, India
editor@iaeme.com/ iaemedu@gmail.com
https://iaeme.com/Home/journal/IJIT

International Journal of Information Technology (IJIT)
Volume 6, Issue 1, January-June 2025, pp. 123-135, Article ID: IJIT_06_01_010
Available online at https://iaeme.com/Home/issue/IJIT?Volume=6&Issue=1
ISSN Online: 2251-2809; Journal ID: 4971-6785
Impact Factor (2025): 16.10 (Based on Google Scholar Citation)
DOI: https://doi.org/10.34218/IJIT_06_01_010

BUILDING MODULAR WEB PLATFORMS WITH MICRO-FRONTENDS AND DATA LAYER ABSTRACTION: A CASE STUDY IN ENTERPRISE MODERNIZATION

Chandra Shekar Chennamsetty
Principal Software Engineer, Autodesk Inc, USA.

ABSTRACT

In the era of rapid digital transformation, enterprises are increasingly seeking scalable and maintainable web architectures to support evolving business requirements. Traditional monolithic web platforms often hinder agility and innovation due to tightly coupled components and complex deployment pipelines. This research explores the design and implementation of a modular web platform leveraging micro-frontends and data layer abstraction, demonstrated through a real-world enterprise modernization case study. By decomposing the frontend into independently deployable units and introducing a unified data access layer, the proposed architecture enables parallel development, enhances scalability, and streamlines integration with legacy and cloud-native systems. The case study also incorporates a custom internal data-sharing framework built on Redux with a publish-subscribe model, as well as performance tooling through SpeedCurve, Dynatrace, and Splunk. A 2-second page load time goal was met by leveraging CDN-based micro-frontend delivery and optimized data-fetching techniques. The architecture further utilizes AWS Lambda and ECS Fargate for scalable backend service deployment. Quantitative metrics—including deployment *frequency, frontend load performance, and build time—are used to evaluate the impact. The findings underscore the practical benefits and architectural considerations of adopting micro-frontend patterns and data abstraction layers in large-scale enterprise web modernization initiatives.*

Keywords: Micro-Frontends, Data Layer Abstraction, Modular Web Architecture, Enterprise Modernization, Frontend Decomposition, Web Platform Scalability, API Integration, Legacy System Modernization, Case Study, Digital Transformation, CDN Deployment, Redux Pub-Sub.

Cite this Article: Chandra Shekar Chennamsetty. (2025). Building Modular Web Platforms with Micro-Frontends and Data Layer Abstraction: A Case Study in Enterprise Modernization. *International Journal of Information Technology (IJIT)*, 6(1), 123-135.
DOI: https://doi.org/10.34218/IJIT_06_01_010

1. Introduction

As enterprises strive for agility and scalability in their digital platforms, legacy monolithic web architectures are proving increasingly inadequate. These systems, characterized by tight coupling and centralized deployment pipelines, hinder innovation and responsiveness to business needs.

Modern approaches like **micro-frontends** and **data layer abstraction** offer a solution by breaking down the frontend into independently deployable components and decoupling the frontend from backend complexities. Micro-frontends (MFEs) apply microservices principles to the user interface by enabling modular and domain-oriented development. This decomposition allows teams to build and deploy UI modules autonomously while maintaining independent release cycles.

A unified data access layer simplifies integration with diverse services and systems, ensuring that frontend modules fetch only the data they require.

This paper presents a case study of an enterprise modernization effort where a monolithic web platform was re-architected using these principles. The new modular design enabled faster development, improved maintainability, and seamless integration with both legacy and cloud-native services.

We outline the architecture, implementation strategy, and results, supported by performance metrics and system benchmarks. Key contributions include practical insights into adopting micro-frontends, an internal data-sharing framework using Redux and publish-subscribe patterns, and a robust data abstraction layer for large-scale enterprise environments.

2. Technical Foundations and Literature Review

The architectural evolution from monolithic to modular systems has been a focal point in modern software engineering. Monolithic applications, although simple to develop initially, become increasingly difficult to scale and maintain as complexity grows. These limitations have driven the adoption of modular approaches such as microservices and, more recently, micro-frontends.

Micro-frontends enable the decomposition of a frontend monolith into smaller, independent units that can be developed, tested, and deployed separately by different teams. This approach promotes parallel development, independent releases, technology diversity, and reduces the risk of system-wide regressions.

A micro-frontend typically corresponds to a bounded business domain (e.g., user profile or reporting) and is composed at runtime via frameworks such as Webpack Module Federation or Single-SPA.

In parallel, **data layer abstraction** provides a unified interface between frontend modules and backend services. By introducing an abstraction layer—often implemented using GraphQL, API gateways, or Backend-for-Frontend (BFF) patterns—teams can isolate data concerns from presentation logic. This allows for better maintainability, security enforcement, and easier adaptation to backend changes without impacting the frontend.

Several frameworks and tools have emerged to support these patterns:

- **Single-SPA** and **Webpack Module Federation** for micro-frontend composition
- **GraphQL** and **Apollo Federation** for unified data access
- **API gateways** like **AWS API Gateway**, **Kong**, or **Apigee** for service mediation
- **SpeedCurve**, **Dynatrace**, and **Splunk** for performance monitoring and diagnostics

Recent research and industry case studies have demonstrated the benefits of modular architectures in improving deployment frequency, reducing time-to-market, and enabling distributed team collaboration. However, challenges remain in dependency management, shared state coordination, performance optimization, and governance at scale.

3. Architectural Design and Methodology

The modernized platform architecture was designed with two core principles: modularity and independence. The goal was to decouple both the frontend and backend components to allow autonomous development, faster deployment, and better scalability. This was achieved using a combination of micro-frontend architecture and a data layer abstraction strategy.

3.1 Micro-Frontend Composition

The frontend was decomposed into discrete, independently deployable modules aligned with business domains. Each micro-frontend was developed by a dedicated team and integrated at runtime using Webpack Module Federation and Single-SPA.

Key architectural features:

- Independent Deployment Pipelines
- Shared Component Libraries for consistency
- CDN Publication of MFEs for faster delivery and broader availability
- Runtime Integration via dynamic loading to ensure flexibility and low coupling

3.2 Data Layer Abstraction

To isolate the frontend from backend complexity, a Backend-for-Frontend (BFF) pattern was implemented using GraphQL and REST services, mediated through AWS API Gateway.

- GraphQL enabled flexible, tailored queries and reduced over-fetching
- Legacy REST services were abstracted and proxied behind the gateway
- Centralized security and request throttling were enforced

3.3 Internal Data Sharing Framework

A proprietary data-binding framework was built as a wrapper around Redux, designed to facilitate inter-MFE communication using a **publish-subscribe model**.

- Each MFE used dedicated **data providers**—modular JavaScript components that dynamically load, fetch data, and prepare it for use
- These modules run independently and publish structured data to the Redux store
- MFEs subscribe to the appropriate Redux slices, ensuring seamless state synchronization

This internal framework enabled loosely coupled, on-demand data exchange while maintaining high performance and testability.

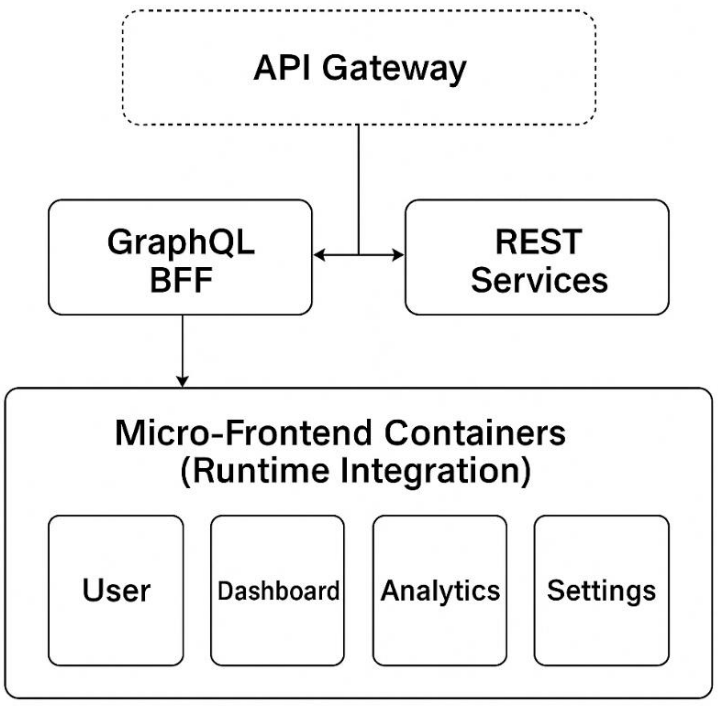
3.4 Deployment and CI/CD Pipeline

Deployment was automated using GitHub Actions and Docker-based pipelines, with services deployed to **AWS ECS Fargate** and **AWS Lambda**.

- Each micro-frontend had its own pipeline
- Dockerized builds ensured parity across environments
- Integration testing was decoupled
- Lambda functions were used for lightweight backend tasks and API responses

3.5 System Architecture Diagram

Below is a simplified representation of the system architecture



4. Case Study – Enterprise Web Modernization

4.1 Organization Context and Challenges

The enterprise, operating in the financial services sector, relied on a monolithic web application. Key issues included:

- Long deployment cycles and update downtime
- Tight frontend-backend coupling
- Inhibited team parallelism
- Poor component scalability

4.2 Modernization Objectives

- Enable modular UI deployment
- Introduce data abstraction via BFF

- Preserve legacy integrations
- Achieve 2-second page load target
- Publish modules to CDN for fast delivery

4.3 Solution Architecture

- **Frontend**: MFEs built in React, composed with Webpack Module Federation
- **Backend**: GraphQL BFF layered over REST services
- **Infrastructure**: Deployed via Lambda and ECS Fargate
- **Observability**: Instrumented with SpeedCurve, Dynatrace, and Splunk

4.4 Implementation Timeline

Phase	Duration	Key Activities
Discovery	4 weeks	Platform audit, domain decomposition
MVP Development	8 weeks	Micro-frontend POC, GraphQL setup
Integration	6 weeks	API Gateway config, CI/CD
Full Rollout	10 weeks	Module migration, production deployment

4.5 Tools and Technologies Used

Category	Technology
UI Framework	React, Tailwind CSS
Micro-Frontend	Webpack Module Federation, Single-SPA
Data Layer	GraphQL (Apollo Server), REST
DevOps	Docker, GitHub Actions, AWS ECS, AWS Lambda
API Management	AWS API Gateway, JWT Auth
Observability	SpeedCurve, Dynatrace, Splunk

4.6 Integration Approach

- CI/CD per module
- Shared libraries maintained via semver
- E2E tests run independently
- Monitoring through Prometheus + Grafana

4.7 Organizational Impact

- Reduced inter-team dependencies
- Legacy preserved with GraphQL layer

- Enhanced developer ownership and delivery pace

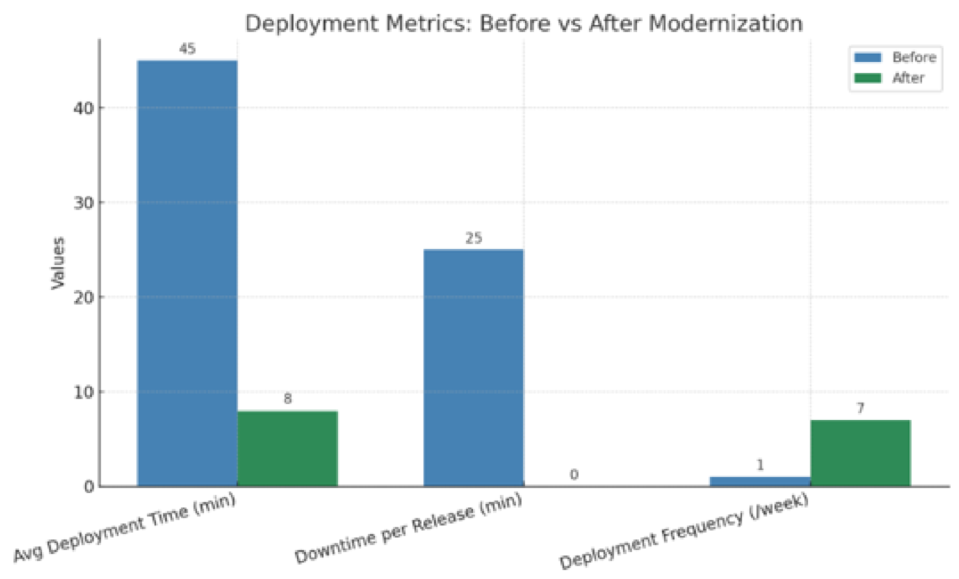
5. Quantitative Evaluation and Impact Analysis

To assess the effectiveness of the modular architecture, a comprehensive evaluation was conducted comparing the system's performance before and after modernization. The analysis focused on deployment efficiency, application responsiveness, development velocity, and operational scalability.

5.1 Deployment Efficiency

Metric	Pre-Modernization	Post-Modernization
Deployment Time	45 minutes	8 minutes/module
Downtime During Release	20–30 minutes	Zero
Deployment Frequency	Weekly	Daily (per module)

Graph: Deployment metrics before and after implementing micro-frontends and modular CI/CD pipelines.



5.2 Application Performance

Measured using Lighthouse, WebPageTest, SpeedCurve, and Dynatrace.

Metric	Before	After	Improvement
TTFB	1.5 sec	900 ms	~40%
FCP	3.2 sec	1.8 sec	~44%
LCP	4.8 sec	2.3 sec	~52%

5.3 Development Velocity

Metric	Monolithic	Modular
Feature Rollout Time	4–6 weeks	2–3 weeks
Concurrent Teams	1–2	4–5
Merge Conflicts	High	Low

5.4 Scalability and Fault Isolation

- Modules scaled horizontally
- Analytics failure didn't impact dashboard/user profile
- Fallback UIs ensured continuity during backend failures

5.5 Summary

Micro-frontend architecture combined with BFF and CDN deployment resulted in faster performance, better reliability, and improved delivery cadence.

6. Engineering Insights and Lessons Learned

The transition from a tightly coupled monolithic architecture to a modular, micro-frontend-based system required not just technical changes, but a strategic rethinking of development practices, team structures, and deployment governance. This section presents a deeper examination of the **key challenges**, **practices adopted**, and **lessons learned** across several dimensions of the transformation.

6.1 Micro-Frontend Complexity Management

While micro-frontends (MFEs) enabled teams to develop and deploy independently, the distributed nature of this architecture introduced new complexities that had to be carefully managed.

- **Shared State Isolation and Coordination**
  MFEs operated as self-contained units, but certain application-level states (e.g., user authentication, global preferences, feature flags) needed to be shared across modules. A lightweight internal **publish-subscribe framework built on Redux** was introduced. This framework acted as a bridge between MFEs using decoupled, event-driven communication, which avoided tight integration while still allowing data sharing.
- **Styling Conflicts and Namespace Collisions**
  CSS conflicts across MFEs were initially problematic due to shared DOM space. This was mitigated using:
  - **CSS Modules** for locally scoped class names
  - **Shadow DOM** for true encapsulation in custom elements where necessary
  - **Global design tokens** via a central design system to maintain consistent theming
- **Cross-Team UI Consistency**
  As different teams developed different MFEs, enforcing a uniform user experience was a challenge. A **centralized design system**, linters, and Storybook-driven UI development were adopted to ensure pixel-perfect consistency and code hygiene.
- **Performance Optimization Across MFEs**
  Runtime integration of MFEs resulted in increased initial bundle sizes. Techniques like **lazy loading**, **tree shaking**, and **async imports** were introduced to ensure that each MFE only loaded what it needed when it was needed, minimizing FCP and LCP timings.

6.2 Data Abstraction Layer Design

The data abstraction layer, implemented using GraphQL in a BFF pattern, was a cornerstone of frontend-backend decoupling. While powerful, it required thoughtful engineering to achieve performance and security goals.

- **Granular Access and Authorization**
  To prevent overexposure of data, **role-based access control (RBAC)** policies were enforced both at the **API Gateway layer** and within GraphQL resolvers. Auth tokens (JWT) were validated centrally, and permission scopes dictated data visibility at field-level granularity.

- **Query Overhead and Optimization**

  Early GraphQL usage led to over-fetching and inefficient queries. Through:

  - **Query usage analytics** from Apollo Studio
  - **Query whitelisting** for production environments
  - **Data loader patterns** and response caching

  the team significantly reduced redundant API calls and backend load.

- **Legacy System Integration**

  Wrapping legacy REST APIs with GraphQL resolvers allowed incremental backend modernization. This approach enabled gradual migration without breaking frontend contracts. Proxying through **AWS API Gateway** also allowed monitoring, throttling, and transformation of legacy responses into modern schemas.

6.3 DevOps and CI/CD Maturity

A distributed architecture required a more mature and scalable DevOps strategy, which evolved significantly during the project lifecycle.

- **Template-Driven Pipelines**

  With 10+ independently deployable MFEs, managing CI/CD pipelines became a bottleneck. The team adopted **templated GitHub Actions workflows** using shared YAML includes, reducing duplication and standardizing build logic.

- **Version Management**

  Shared component libraries were managed using **semantic versioning (semver)** with changelogs and automated release tagging. This ensured backward compatibility and allowed MFEs to opt-in to new features safely.

- **Safe Rollouts**

  Feature flags, **canary deployments**, and **blue-green environments** were used to validate changes before full-scale releases. Errors could be rolled back without affecting the entire platform.

- **Observability**

  SpeedCurve, Dynatrace, Splunk, and Prometheus were used for:

  - Monitoring app performance (TTFB, FCP, LCP)
  - Alerting on regressions
  - Tracking errors and user journeys

  This observability layer proved crucial in maintaining system health post-deployment.

6.4 Organizational Considerations

The architectural shift also necessitated a transformation in how teams were structured, how ownership was defined, and how alignment was maintained.

- **Domain-Oriented Team Ownership**

  Teams were restructured to align with business domains rather than technical layers. Each team owned the full lifecycle of a domain's MFE — from design and development to deployment and monitoring.

- **Decentralized Governance with Central Guardrails**

  While autonomy was encouraged, a central architecture review board provided guardrails around:

  - API schema changes
  - Shared library updates
  - Security and access patterns

- **Skill Development**

  Moving to this architecture required engineers to learn:

  - Module Federation
  - GraphQL schema design
  - Observability tooling
  - Docker and cloud-native deployment (Lambda, Fargate)

- **Cross-Functional Collaboration**

  Weekly cross-team demos, shared retrospectives, and architecture guilds helped prevent knowledge silos and fostered a culture of open experimentation and alignment.

6.5 Key Takeaways

- **Modularization requires more discipline, not less**

  While modular MFEs give teams autonomy, they demand stronger standards for integration, testing, and communication.

- **Data abstraction is a strategic enabler**

  Beyond performance improvements, abstracting data through GraphQL and BFF enabled faster backend evolution and reduced change ripple effects.

- **Observability must be baked in early**

  Instrumentation with tools like Splunk and Dynatrace allowed real-time feedback loops, making it easier to detect and resolve issues rapidly.

- **Architecture and culture evolve together**

  The technical success of micro-frontends was deeply tied to team autonomy, skill development, and governance mechanisms that encouraged safe innovation.

7. Conclusion

This case study demonstrated how transitioning from a monolithic web architecture to a modular system using micro-frontends and data layer abstraction can lead to significant improvements in scalability, development velocity, and maintainability. By enabling independently deployable UI modules and decoupling frontend logic from backend complexities, the architecture supported parallel development and faster time-to-market, while reducing deployment risks and inter-team dependencies.

Key innovations included the use of Webpack Module Federation for runtime composition, GraphQL-based Backend-for-Frontend (BFF) for streamlined data access, and a custom internal data-sharing framework built on Redux with a publish-subscribe model. Performance goals such as a 2-second page load time were met through optimized loading strategies, CDN distribution of micro-frontends, and observability tools like Splunk, Dynatrace, and SpeedCurve. Additionally, deployment via AWS Lambda and ECS Fargate ensured scalable, cloud-native service delivery.

Ultimately, the success of this modernization was not solely technical but organizational. The shift required new team structures, governance models, and DevOps practices that emphasized autonomy with alignment. The approach serves as a practical blueprint for large enterprises aiming to modernize digital platforms incrementally while maintaining stability and delivering continuous value to end users.

References

[1] M. Tillmann and J. Vogel, "Micro-Frontend Architectures: State of the Art, Challenges, and Opportunities," Proceedings of the IEEE International Conference on Software Architecture, 2022.

[2] Z. Liu, R. A. Maximo, and A. Garcia, "An Empirical Study on the Use of Module Federation in Frontend Applications," ACM SIGSOFT Software Engineering Notes, vol. 48, no. 2, 2023.

[3] Y. Hartono, "Micro-Frontend Design in Practice," in 2022 IEEE Conference on Web Engineering, pp. 92–101.

[4] Amazon Web Services, “Best Practices for Microservice Architectures.” [Online]. Available: https://aws.amazon.com/microservices/

[5] Splunk Inc., “Observability with Splunk: Real-Time Monitoring for Modern Applications.” [Online]. Available: https://www.splunk.com/en_us/observability.html

[6] SpeedCurve Ltd., “SpeedCurve: Monitor Frontend Performance and User Experience.” [Online]. Available: https://www.speedcurve.com/

[7] Dynatrace LLC, “Dynatrace: AI-Powered Observability Platform.” [Online]. Available: https://www.dynatrace.com/.

Citation: Chandra Shekar Chennamsetty. (2025). Building Modular Web Platforms with Micro-Frontends and Data Layer Abstraction: A Case Study in Enterprise Modernization. International Journal of Information Technology (IJIT), 6(1), 123-135.

Abstract Link: https://iaeme.com/Home/article_id/IJIT_06_01_010

Article Link:
https://iaeme.com/MasterAdmin/Journal_uploads/IJIT/VOLUME_6_ISSUE_1/IJIT_06_01_010.pdf