Architecture Case Study – Client-Side UI Integration Across Web Applications

Intended Audience

- Senior engineers transitioning into architecture
- Architects designing modular frontend systems
- Platform and frontend leads evaluating micro-frontend trade-offs

1. Context

Intent

The intent of this case study is to clearly establish the architectural context and constraints under which the system was designed, so that subsequent analysis of architectural decisions, trade-offs, and possible alternatives is grounded in the realities of the time. The focus is not on evaluating the correctness of the implementation, but on understanding why certain architectural choices made sense then, and how evolving tools, practices, and organizational models would influence those choices today.

Background & Problem Context

The system under study is an evaluation prototype created for an insurance domain use case. Its goal was to explore how a traditionally integrated enterprise application could be decomposed into loosely coupled web applications while still presenting a cohesive user experience.

From a business perspective, the application aimed to support customer-facing insurance workflows such as:

- Filing and tracking damage claims
- Communicating with the insurance team via email

To support these workflows, insurance employees needed the ability to:

- Search for and view customer and contract information
- Assist customers in filing damage claims
- View and respond to customer communications

Historical & Architectural Context

This prototype was developed approximately ten years ago by INNOQ engineers **Lucas Dohmen** and **Marc Jansing**, during an early phase of industry experimentation with microservices and modern frontend architectures.

At the time, frontend integration patterns were still evolving, and teams were beginning to explore how independently deployed services could be composed into user-facing applications. The project was intentionally positioned as an evaluation exercise to demonstrate concepts such as:

- Single Page Applications (SPA)
- Resource-Oriented Client Architecture (ROCA)
- Client-side integration across multiple independently deployed web applications

Rather than a production-grade backend, the system relied on a shared backend simulator that exposed RESTful HTTP endpoints returning customer, contract, and claims data. This allowed the focus to remain on frontend composition and integration rather than backend complexity.

Architectural Experimentation Goals

A deliberate architectural decision was made to combine:

- A shared, monolithic backend simulator
- Multiple independently deployed frontend applications

This approach enabled the team to explore:

- Independent deployment and evolution of frontend applications
- The use of different technologies across frontend components
- Client-side integration strategies in the absence of mature micro-frontend tooling

Key Constraints

- Frontend teams needed independent deployment autonomy
- No mature micro-frontend frameworks or module federation
- Browser-based integration as the only viable composition layer
- Shared backend used to reduce variables during evaluation

These constraints and choices form the foundation for the architectural analysis that follows.

2. System Overview

The web application consisted of the following components:

- **Main Application (Landing Page)**

  An independently deployed frontend application responsible for searching customers and displaying basic customer information. Implemented using Node.js, HTML, and JavaScript.

- **Damage Claims**

  An independently deployed frontend application that allowed insurance employees to log and manage damage claims. Implemented using Node.js, HTML, and JavaScript.

- **Letters**

  An independently deployed frontend application that allowed users to view customer letters and correspondence. Implemented using Node.js, HTML, and JavaScript.

- **PostBox**

  A frontend application responsible for displaying lists of customer mails and claim-related correspondence. Implemented using Java, Spring Boot, Maven, HTML, and JavaScript, demonstrating that frontend applications could be built and deployed using heterogeneous technology stacks.

- **Backend Simulator**

  A shared monolithic backend responsible for handling REST calls and returning customer, contract, and claims data. Implemented using Node.js.

- **Assets**

  A shared set of frontend artifacts (CSS, fonts, images, and JavaScript) used across applications to provide a consistent look and feel. This shared assets library is intended to act as an implicit coupling mechanism, trading visual consistency for tighter coordination across independently deployed applications.

Key Note: This case study focuses on architectural structure, integration patterns, and trade-offs rather than implementation details. The analysis is based on system boundaries, deployment models, integration mechanisms, and documented design intent, rather than a *line-by-line review* of source code.

However, if the reader is interested in looking at the actual source code repositories (designed and developed by German-speaking development teams), then the code can be viewed and analysed as needed by accessing them in GitHub from the links given below:

1. Main application: GitHub - ewolff/crimson-portal: The entry point for our demo
2. Common Assets: GitHub - ewolff/crimson-styleguide: Shared Assets for the Crimson Apps
3. Damage Component: GitHub - ewolff/crimson-damage
4. Letter Component: GitHub - ewolff/crimson-letter
5. Postbox Component: GitHub - ewolff/crimson-postbox: A Postbox application
6. Backend Simulator Component: GitHub - ewolff/crimson-backend: A backend for the other systems

3. Integration Mechanisms

This evaluation prototype has been used to explore frontend integration strategies in a system that is composed of independently deployed frontend applications backed by a shared monolithic backend component. Given the absence of mature micro-frontend frameworks at the time this prototype was developed, integration was primarily focused on browser-native mechanisms rather than shared runtime containers or server-side composition.

The primary navigation-based composition or integration mechanisms used were:

1. HTTP redirects
2. Link-based navigation with passing of required parameters
3. Client-side transclusion

All three mechanisms rely on **navigation-based composition**, where integration of different components occurs at page boundaries rather than through shared runtime state or containers.

Integration via Redirects

Certain workflows required users to transition from one frontend application back to another upon completion of an action.

For example: After a damage claim was submitted in the Damage application, the user is redirected back to the **Main Portal** for viewing updated customer or contract overview. This integration has been implemented using HTTP redirects, where the Damage application issues a redirect response pointing back to an URL in **Main Portal**.

Key architectural characteristics to consider:

1. The Damage application has to only be aware of target URL.
2. Sharing of any sort of frontend state is not needed.
3. Redirect target can be provided dynamically, thereby further reducing/delaying coupling.

This approach helps to represent a very low-coupling integration mechanism, relying entirely on standard web semantics. Similar patterns are commonly used in OAuth-based authentication flows (e.g., redirecting back from a third-party identity provider).

Integration via Links (Parameter-Based Navigation)

Most of the frontend integration for the system considered is achieved through links containing essential context like URL parameters.

For example, a link to Letter application has identifiers like:

- Contract ID
- Partner ID

These parameters allow the Letter application to:

- Retrieve all required information from shared backend simulator component.
- Render its UI independently.
- Avoid any runtime dependency on **Main Portal**

Architectural Implications

- **Main Portal** does not need to know how Letter application is implemented.
- The Letter application can modify its UI and any associated internal logic without impacting **Main Portal**
- Integration contracts are limited to URL structure and associated parameter semantics. This makes the URL itself the primary integration contract between applications.

This helps in accomplishing loose frontend coupling, with integration occurring at navigation boundary rather than through shared runtime components.

Integration via Client-side Transclusion (JavaScript-Driven)

In addition to redirects and link-based navigation, the system also employs Client-side transclusion for selected UI elements such as the postbox preview. In this approach, HTML fragments are dynamically fetched from another frontend application and embedded into the current page using browser-executed JavaScript.

This technique preserves ownership of presentation logic within the source application (Postbox in this case), while allowing its content to appear within the main portal. Integration remains link-based at its core, with JavaScript interpreting link metadata and rendering embedded content at runtime.

Notably, the system is designed to degrade gracefully: if transclusion fails or JavaScript is unavailable, the embedded view is replaced with a simple navigational link, ensuring continued usability and resilience.

Backend Coupling Considerations

While the frontend applications are loosely coupled via redirects and links, all applications rely on a shared backend simulator and database schema.

This leads to:

- Frontend applications that are tightly coupled to common backend data contracts.
- Changes to backend data structures impacting all frontend applications.
- Backend evolution is restricted without coordinated changes across the frontend ecosystem. This effectively shifts coordination complexity from runtime integration to data contract governance.

This trade-off appears to have been intentionally accepted so that there can be:

- Reduced complexity during architectural evaluation.
- Focused experimentation on frontend decomposition rather than backend refactoring

Architectural Summary

Architectural Concern	Approach Adopted
Frontend Components Integration	Redirects, Links, and Client-side transclusion
Level of coupling between Frontend Components	Low
Level of Coupling with Backend Simulator	High
Composition Layer	Browser
Application State Management	Reconstructed as per navigation
Deployment Independence Scope	Frontend Applications only

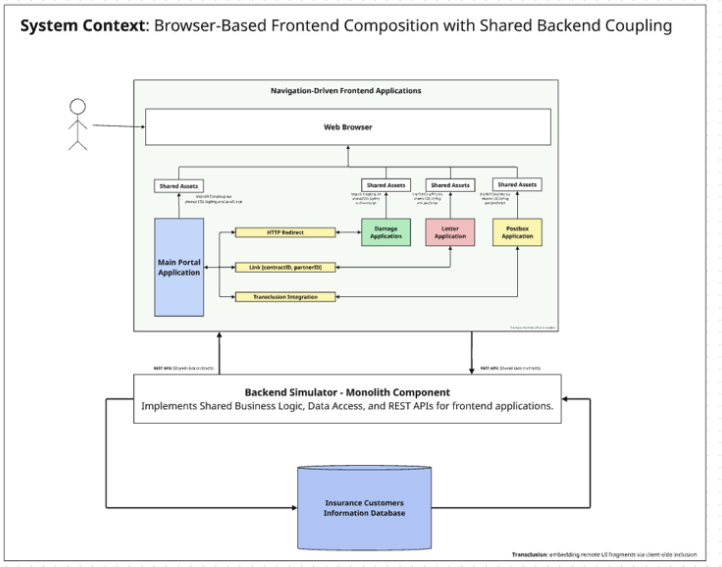
Key Points to Remember

This evaluation system represents a textbook example of:

- Frontend modernisation without backend ownership
- Efforts toward incremental improvements within organizational constraints.
- Optimising for local autonomy rather than architectural purity in isolation.

While effective for evaluation and early-stage decomposition, this approach places increasing pressure on navigation contracts, shared backend schemas, and cross-team coordination as the system scales.

Figure: Navigation-driven frontend composition with implicit asset coupling and shared backend dependency



Key Point to remember

> *Navigation-based composition optimizes for cognitive simplicity and deployment autonomy at the cost of interaction richness*

4. Trade-offs and Consequences

The architectural choices made in this evaluation prototype reflect a deliberate attempt to balance autonomy, simplicity, and organizational constraints. While the system successfully demonstrates browser-based frontend decomposition and loose coupling at the UI and navigation layer, it also introduces trade-offs that become increasingly visible as the system attempts to scale.

This section captures what worked well, the limitations that emerged, and the longer-term consequences of the decisions made.

What Worked Well

1. **High Frontend Autonomy:** Each frontend application could be developed, deployed, and evolved independently. Teams working on these applications had the freedom to:

- Choose technologies and frameworks based on local needs
- Release changes without coordinating deployments with other frontend teams
- Experiment with UI patterns in isolation, with confidence

This level of autonomy was particularly valuable during a period of rapid frontend innovation.

2. **Extremely Simple Frontend Integration Model:** By relying on browser-native mechanisms such as links, redirects, and transclusions, the system avoided:

- Shared runtime containers
- Complex orchestration layers
- Tight compile-time or runtime coupling between frontend applications

Integration was easy to understand, inspect, and debug using standard browser-based tools.

3. **Low Cognitive Overhead for Navigation:** Navigation-based composition aligns naturally with how browsers and users already behave. Each application:

- Owned its own page lifecycle
- Reconstructed state on load
- Avoided long-lived shared frontend state

This reduced frontend integration complexity and made runtime failures easier to isolate, fix, and redeploy.

4. **Incremental Frontend Modernization:** The architecture enabled frontend improvements without requiring backend ownership or refactoring. This made it feasible in organizations where:

- Backend systems were owned by different teams
- Backend modernization was out of scope in early phases
- Frontend modernization was the primary objective

Trade-offs and Limitations

1. **Backend-Centric Coupling:** Despite frontend autonomy, all applications depended on a shared backend simulator and database schema. This introduced:

- Tight coupling through shared data models
- Coordinated change requirements across frontend applications
- Limited ability to evolve backend APIs independently

Over time, the backend simulator becomes the primary constraint on system evolution.

2. **URL-Based Integration as a Contract:** URLs and query parameters effectively became the integration contracts between frontend applications. While simple, this led to:

- Fragile coupling through implicit parameter semantics
- Increased coordination for future parameter changes
- Limited discoverability and validation of integration contracts

As the number of applications grows, managing these contracts becomes increasingly difficult.

3. **Reconstructed Application State:** Because integration occurs at page boundaries:

- Application state is rebuilt on each navigation
- Cross-application workflows rely on repeated backend calls
- User experience may degrade due to reloads and latency

This model works well for simple workflows but struggles with richer, highly interactive user journeys.

4. **Shared Assets as Implicit Coupling:** The shared assets library ensured visual consistency but also introduced:

- Implicit coordination between teams
- Risk of breaking changes in shared CSS or JavaScript impacting all applications
- Challenges in versioning and rolling out visual changes

This coupling is further reinforced when shared JavaScript libraries (e.g., jQuery-based transclusion logic) becomes a runtime dependency across applications, increasing the impact radius of asset-level changes.

This represents a trade-off between visual consistency and independent evolution.

5. **Limited Observability and Cross-Application Tracing:** With navigation-driven composition:

- End-to-end user journeys span multiple applications
- Correlating logs, metrics, and errors becomes more difficult
- Troubleshooting requires stitching together multiple execution contexts

Client-side Transclusion introduces additional complexity in tracing UI failures, as rendering issues may originate from remote applications, but surface in a different execution context. Without explicit observability design, operational complexity increases significantly.

Long-Term Consequences

As systems evolve beyond evaluation and experimentation, several pressures emerge:

- Navigation contracts become increasingly brittle
- Backend schemas constrain frontend innovation
- Cross-team coordination increases, reducing autonomy
- User experience expectations outgrow page-based composition capabilities

While effective for early-stage decomposition and experimentation, this architecture naturally reaches limits in scalability, complexity, and user experience as systems grow.

Architectural Takeaway

This evaluation system represents a pragmatic and historically appropriate approach to frontend modernization under real-world constraints. It optimizes for:

- Local autonomy
- Incremental frontend improvement
- Organizational feasibility

In doing so, it consciously accepts:

- Centralized backend coupling
- Navigation-driven state reconstruction
- Implicit contracts at the URL and asset level

These trade-offs are not flaws. They are intentional decisions shaped by the tools, practices, and organizational realities of the time.

These limitations provide a useful baseline for further evaluation of how modern frontend and platform patterns can address the same challenges today.

5. How would we approach this Architecture today?

Overview

This section examines what has fundamentally changed since the prototype was created - almost a decade ago, and equally importantly, what has not changed. The goal is to evaluate how modern architectural patterns, platform capabilities, and organizational practices would influence the same problem statement and space without losing sight of original constraints and lessons learned and improvements enabled.

In this section, we will focus on exploring how modern approaches can help rebalance trade-offs that we identified in previous section.

What Has Changed Since This Architecture Was Designed?

There have been several foundational shifts since this evaluation prototype was designed, including:

1. **Frontend Architecture Maturity**:
    a. Micro-frontend patterns are now well understood and documented.
    b. Standardized mechanisms now exist for:
        i. Runtime composition
        ii. Dependency isolation
        iii. Versioning and deployment rollouts.
    c. Frameworks and tooling have reduced the cost of safe frontend decomposition
2. **Enhanced Platform and Infrastructure Capabilities**:
    a. Cloud-native platforms can now provide:
        i. Managed load balancing
        ii. Native health checks
        iii. Observability primitives
        iv. Secure service-to-service communication (Identity, mTLS, policy enforcement)
    b. Content Distribution Network (CDN) and edge capabilities are now mainstream rather than being optional.
3. **Organizational and Team Models**:
    a. Teams can now increasingly own vertical slices (UI + API + Data)
    b. DevOps and platform teams can provide paved roads rather than bespoke infrastructure
    c. Cross-Team contracts are more explicitly defined, versioned, and governed.
4. These shifts do not invalidate the original architecture, but they do change the place where complexity lives and how it gets managed.

Revising Frontend Composition: Then vs. Now

Then [Evaluation Prototype]:

1. Composition via:
    a. Links
    b. Redirects
    c. Client-side transclusions
2. Integration at Navigation Boundaries
3. Browser as primary composition layer
4. Loose UI coupling, tight backend coupling.

Now [Modern Options]

Modern architectures now offer additional composition strategies, including:

1. **Micro-Frontends with Explicit Contracts**
    a. Composition via:
        i. Module Federation
        ii. Web Components
        iii. SPA Shell with federated routes
    b. Shared contracts defined via:
        i. Versioned APIs
        ii. Typed Interfaces
        iii. Runtime Compatibility checks
    c. Ownership boundaries enforced at runtime.
2. **Backend-for-Frontend [BFF]**
    a. A thin, purpose-built backend per UI
    b. Shields frontend teams from backend schema churn and enables:
        i. Tailored APIs
        ii. Controlled aggregation
        iii. Independent evolution.
3. **Hybrid Composition Models**
    a. Navigation-based composition for coarse flows
    b. Runtime composition for high-interaction areas
    c. Server-side rendering for critical pages
    d. Client-side hydration for responsiveness

Key Shift:

The modern systems tend to make integration contracts more explicit, rather than implicit in URLs and shared assets.

How Modern Approaches Can Help Rebalance the Trade-Offs

Concern	Evaluation Prototype	Modern Approach
Frontend autonomy	High	High [With Guardrails]
Integration simplicity	Very High	Moderate
Backend Coupling	High	Reduced via BFFs / APIs
Observability	Minimal	Built-in, Standardized
UX Continuity	Page-based	SPA + partial reloads

Failure Isolation	Browser-Level	Platform + Browser [Defense in Depth]
Contract Governance	Implicit	Explicit and Version Controlled
Cognitive Load	Low	Higher [managed via platforms]

Key Insight:

Modern architectures trade **simplicity of integration** for **predictability, observability, and scale**.

What Would Likely Stay the Same?

Not everything needs to change and some principles from the original architecture design remain highly relevant, including:

1. Prefer loose coupling over shared runtime state.
2. Favor resilience over perfect consistency.
3. Degrade gracefully when dependencies fail
4. Avoid premature backend refactoring when frontend modernization is the goal.

The original architecture's emphasis on organizational reality over theoretical purity still remains a valuable lesson.

New Questions that Modern Architecture can Answer

Modern architecture approaches introduce new responsibilities, including:

1. How would we version and govern frontend contracts?
2. How do we prevent micro-frontend sprawl?
3. Who would own shared runtime dependencies?
4. How do we enforce security boundaries across UI components?
5. How can we observe end-to-end user journeys across composed frontends?

These questions reflect how modern architectures shift complexity upward in the stack, rather than eliminating it.

Positioning this Case Study in a Modern Context

This evaluation prototype should not be seen as:

1. An outdated pattern
2. A failed approach
3. A stepping stone that must be discarded.

Instead, we need to keep in mind that it actually represents:

1. A baseline architecture
2. A constraint-driven design
3. A clear reference point for understanding how and why modern frontend architectures evolved.

This helps to provide a valuable contrast for evaluating modern solutions - not as replacement, but as responses to known limitations.

Closing Thoughts:

The most important takeaway is the evolution of architectural thinking and not the tools available at our disposal today.

From minimizing coupling through navigation

→ managing coupling through explicit contracts

→ governing complexity through platforms and standards

This sets the stage for us to look deeper into aspects like:

1. Security & Identity

2. Observability & Metrics
3. Testing Strategy
4. Scalability & Resilience
5. Cloud Deployment (AWS)
6. Delivery & Governance (IaC, CI/CD, SecOps)
7. Cost and FinOps Considerations

Point to remember:

> This case study intentionally avoids prescribing a single “correct” modern architecture. The goal here is to illuminate how architectural trade-offs shift when constraints change.

6. Security, Identity, and Trust Boundaries

Overview

As frontend systems evolve from monolithic applications into compositions of independently deployed components, **Security and Identity concerns** shift fundamentally.

What was once seen as a large internal concern - handled implicitly within a single application - now becomes a first-class architectural dimension.

In the evaluation prototype we discussed earlier, the security considerations were largely implicit and delegated to the backend simulator module. This was appropriate given the prototype’s goals and constraints.

However, as frontend composition increases, the integration boundaries multiply, and the following become unavoidable:

1. Explicit Trust Models
2. Identity Propagation
3. Security Controls

This section examines the following:

1. How security was implicitly handled in the original architecture?
2. What risks and assumptions were embedded in that approach?
3. How modern architectures would explicitly define trust boundaries, identity flows, and security responsibilities?

As frontend systems evolve from monoliths into composed applications, every new integration boundary becomes a potential security boundary.

Security Characteristics of the Evaluation Prototype

The original system exhibits the following security characteristics:

1. **Implicit Trust Model**:
    a. All frontend applications implicitly trust:
        i. Each other
        ii. Shared backend simulator
    b. There is no explicit authentication or authorization boundary between frontend applications.
    c. URLs and parameters are assumed to be valid and trustworthy. This can go wrong in many ways when one or more the following happens:
        i. Parameter tampering
        ii. Privilege escalation
        iii. Cross-application impersonation

This model works only under controlled conditions, like:

- Internal evaluation environments
- Trusted Networks

- Non-hostile user assumptions

2. **Backend-Centric Security Responsibility**
   a. Security concerns (authentication, authorization, data access) are assumed to be enforced by backend simulator
   b. Frontend applications act primarily as presentation layers with minimal security logic.

This simplifies frontend development, but creates a single point of trust and failure.

3. **Minimal Frontend Security Surface Awareness**

Due to the reliance of frontend integration on Links, Redirects, and Client-side transclusion, there is:

- No explicit protection against parameter tampering.
- No frontend-level access control enforcement.
- No isolation between frontend execution contexts beyond browser defaults.

For a prototype, this is acceptable, but in case of a production system, it would become a liability.

Trust Boundaries in Composed Frontends

Modern frontend architectures force architects to answer the critical question of "**Where do trust boundaries actually exist?**". In a composed frontend system, the trust boundaries may exist at multiple levels, including:

1. **Browser Boundary:** The browser enforces basic isolation (same-origin policy, CORS). This is necessary, but insufficient as a primary security model.
2. **Frontend Application Boundary:** Each independently deployed frontend can represent a separate trust domain. The shared assets or shared runtime libraries will weaken this boundary.
3. **Backend/API Boundary:** APIs often remain the strongest enforcement point for authorization, however, relying exclusively on backend enforcement increases coupling.
4. **Identity Boundary**: Identity must flow consistently across frontend components, backend services, as well as observability and audit systems. The evaluation prototype largely collapses these boundaries into a single implicit trust zone.

Unlike backend services, frontend components execute in a shared, user-controlled runtime, making strict isolation harder and trust assumptions more fragile.

Identity Propagation: Then vs. Now

Then [Evaluation Prototype]

- No explicit user identity propagation across frontend applications.
- User context is assumed to be globally valid.
- Authorization decisions are opaque and backend-driven.

This would work as designed when:

- Users are internal
- Threat models are minimal
- Security is not under active scrutiny

Now [Modern Architectures]

Modern systems treat identity as an **explicit architectural primitive**.

- Users authenticate once via a centralised or federated identity provider.
- Identity is represented and propagated as:
  - Tokens [JWT, opaque tokens]
  - Backend APIs
  - BFF Layers

This enabled:

- Fine-grained authorization
- Auditing and compliance

- Controlled delegation of access

Frontend Security in a Composed Architecture

Modern frontend architectures introduce new attack surfaces that must be explicitly addressed.

URL and Parameter Integrity

In navigation-based composition:

- URLs act as integration contracts
- Parameters often carry sensitive context

Modern mitigations can include:

- Signed or encrypted parameters
- Server-side validation via BFFs
- Minimising sensitive data in URLs

Client-side Transclusion risks

Client-side transclusion introduces specific risks like:

- Cross-site scripting [XSS]
- Injection of untrusted HTML
- Dependency on shared JavaScript Libraries.

Modern approaches mitigate these risks by implementing:

- Strict Content Security Policies [CSP]
- Sanitisation of transcluded content
- Runtime isolation via Web Components or sandboxed iFrames.

Shared Asset and Runtime Dependencies

Shared asset libraries increase the blast radius of:

- Vulnerabilities
- Breaking changes
- Dependency compromises

Modern practices can help in mitigating them by including:

- Versioned asset delivery
- Dependency scanning [SCA]
- Controlled deployment rollout and rollback mechanisms

Backend-for-Frontend [BFF] as a Security Boundary

One of the most impactful modern shifts is the introduction of **Backend-for-Frontend [BFF]** layers.

Security Benefits of BFFs include ability to:

- Terminate frontend authentication
- Enforce authorization policies
- Shield frontend components from backend complexity
- Act as a policy enforcement point

This eventually allows:

- Frontend teams to remain autonomous
- Backend schemas to evolve independently
- Security logic to remain centralised and auditable

In contrast, the evaluation prototype lacks any such intermediary layer.

Zero Trust Thinking Applied to Frontend Composition

Modern architectures increasingly adopt **Zero Trust Policies**, even with frontend ecosystems, including aspects like:

- Do not assume frontend components trust each other
- Authenticate and authorize every meaningful interaction, regardless of its origin.
- Minimize shared secrets and implicit contracts
- Assume partial compromise is possible.

Applied to frontend composition, this would mean:

- Explicit identity propagation
- Clear ownership of security responsibilities
- Observable enforcement points.

Architectural Takeaway

The evaluation prototype reflects a time when:

- Frontend security was largely backend-centric.
- Trust was implicit and environment-bound
- Identity was not a first-class frontend concern.

Modern architectures however require:

- Explicit Trust Boundaries
- Consistent identity propagation
- Defense-in-depth across browser, frontend, and backend layers.

Security is no longer something that can be handled elsewhere, but is something that must be designed into the architecture, just like integration and deployment.

Transition to Next Sections

With Security and Identity established as explicit architectural concerns, the next logical questions become:

- How do we test such a system effectively?
- Where do different kinds of tests provide maximum value?
- How do we prevent regression across independently deployed components?

This leads naturally the next section where we will be discussing more on Testing Strategy and Quality Boundaries.

7. Testing Strategy and Quality Boundaries

Overview

As systems evolve from monolithic applications into compositions of independently deployed frontend and backend components, testing strategy must evolve alongside architecture. In such systems, quality can no longer be ensured by a single, centralised test suite. Instead, it must emerge from clearly defined quality boundaries, ownership models, and layered testing responsibilities.

In the evaluation prototype discussed earlier, testing concerns were largely implicit and secondary to architecture exploration. This was appropriate given its experimental nature. However, when similar architectures are adopted for production systems, testing becomes a first-class architectural concern, tightly coupled to deployment independence, integration mechanisms, and organizational structure.

In this section, we will examine:

- How testing was implicit approached in the evaluation prototype?
- Why traditional end-to-end testing does not scale in composed frontend systems?
- How modern architectures redefine testing responsibilities across boundaries?
- Where different types of tests deliver maximum architectural value?

Testing Characteristics of the Evaluation Prototype

The original evaluation system exhibits the following testing characteristics:

Implicit and Minimal Testing Strategy

- Frontend applications were primarily tested in isolation.
- Integration correctness was validated manually through browser navigation
- Cross-application workflows were assumed to work if links and redirects resolved correctly.

This approach was acceptable at the time because:

- The system was an evaluation prototype, not a production platform.
- The focus was on architectural feasibility rather than long-term operability.
- The number of components and workflows was limited and well understood.

Navigation-Based Integration Reduced Test Complexity

Since integration occurred at page boundaries, many failure modes were naturally isolated, including:

- A broken frontend application did not prevent others from loading.
- Failures manifested as broken links rather than cascading runtime errors.
- Manual testing was sufficient to validate most integration paths.

However, this simplicity does not scale once:

- User journeys span multiple frontend application.
- Transclusion and runtime composition increase coupling.
- Independent deployments happen frequently.

Core Testing Challenge in Composed Frontends

In a composed frontend architecture, the central challenge is not test coverage, but test responsibility. Some of the key questions that emerge include:

- Who owns correctness of cross-application workflows?
- Where should integration behaviour be verified?
- How do we prevent regressions without coupling deployments?

Traditional approaches - especially large end-to-end [E2E] test suites - quickly become become problematic because:

- E2E tests are slow, brittle, and expensive to maintain
- Failures are hard to diagnose across multiple independently deployed systems.
- Introduction of simplicity coordination between teams reduces autonomy.

Modern testing strategies therefore tend to shift focus from testing everything centrally to testing the right things at the right boundaries.

Quality Boundaries as Architectural Boundaries

In modern systems, quality boundaries must align with architectural and ownership boundaries.

Frontend Application Boundary

Each independently deployed frontend application should own:

- Unit tests for internal logic and state handling
- Component and UI tests for rendering and interaction
- Contract tests for interfaces it consumes [APIs, BFFs]

This will help to ensure that:

- Teams can release independently with confidence
- Failures ar localized and diagnosable
- Quality responsibility is clearly assigned.

Integration Boundary

Integration testing should focus on contracts, not implementations. Instead of testing full workflows end-to-end, the teams should validate:

- URL contracts and navigation assumptions
- Expected parameters and schemas
- Error handling and degradation behaviour.

For example:

- Does a frontend correctly handle missing or malformed URL parameters?
- Does a transcluded component fail gracefully when unavailable?
- Are redirects resilient to partial outages?

These tests are lightweight, fast, and stable - and provide more architectural value than deep E2E tests.

Backend and BFF Boundary

Backend APIs and BFF layers are natural enforcement points for:

- Authentication and authorization rules
- Data validation and aggregation
- Backward compatibility guarantees

Testing at this boundary should emphasise:

- API contract tests
- Authorization and access control tests
- Backward compatibility and versioning tests

This will help to reduce the level of frontend coupling with backend evolution and will enable safer independent deployments.

Rethinking the Test Pyramid

In composed systems, the traditional test pyramid evolves into a test distribution model. The emphasis shifts toward:

- Strong unit and component tests at the edges.
- Contract tests at integration boundaries.
- Minimal, but intentional end-to-end tests

Please Note: Use end-to-end tests sparingly. These tests add value ony when:

- Validating critical user journeys
- Exercising security and identity flows
- Verifying platform-level integrations

The end-to-end tests should be:

- Few in number
- Highly stable
- Treated as smoke tests, not exhaustive validation.

Over-reliance on E2E tests often indicates missing contracts or unclear boundaries, not insufficient testing.

Testing Transclusion and Runtime Composition

Client-side transclusion introduces unique testing concerns, including:

- Rendering correctness depends on remote HTML fragments
- Failures may originate in another application
- Shared assets and scripts can affect multiple consumers.

Effective mitigation strategies include:

- Contract tests for transcluded HTML structure

- Snapshot or visual regression tests for embedded components.
- Failure-mode tests to ensure graceful degradation

Critically, the owning application of transcluded content must own its tests, even if the content appears elsewhere.

Observability as a Testing Complement

In composed systems, testing alone is not sufficient. Observability becomes a runtime extension of the testing strategy. Well-instrumented systems allow development teams to:

- Detect integration failures quickly
- Correlate errors across frontend applications
- Validate assumptions made during testing in real usage

Metrics, logs, and traces help answer questions that tests cannot, including:

- How often do transcluded components fail?
- Which navigation paths are most error-prone?
- Where do users abandon workflows?

This feedback loop allows development teams to continuously refine both architecture and associated tests/test suites.

Architectural Takeaway

In composed frontend architectures, testing strategy is architecture. The evaluation prototype succeeded with minimal testing because:

- Integration was simple
- Workflows were limited
- Organizational coordination was implicit

Modern systems, however, will require:

- Explicit quality boundaries
- Contract-first integration testing
- Clear ownership of test responsibilities
- Minimal reliance on brittle end-to-end tests

When testing aligns with architectural boundaries, teams can scale independently without sacrificing quality.

Transition to Next Section

With quality boundaries and testing responsibilities clearly defined, the next set of natural questions will be:

- How do we operate and observe such systems at scale?
- What metrics matter?
- How do we detect failures before users do?

This leads into the next section on Observability, Metrics, and Operational Readiness.

8. Observability, Metrics, and Operational Readiness

Overview

As systems evolve into compositions of independently deployed frontend and backend components, observability becomes the primary mechanism by which architects and development teams can understand system behaviour in production. In such architectures, failures are rarely binary in nature. Instead, they are partial, localised, and context-dependent.

In the evaluation prototype discussed earlier, operational visibility was minimal and largely unnecessary. The system was designed for demonstration and learning rather than sustained production use. However, once similar architectural patterns are adopted in real systems, observability must be treated as a first-class architectural concern, not as an afterthought.

In this section, we will explore:

- Why traditional monitoring approaches break down in composed frontend systems?

- How observability responsibilities align with architectural boundaries?
- What kinds of signal matter most in frontend-heavy architectures?
- How metrics, logs, and traces together enable operational confidence?

Observability Characteristics of Evaluation Prototype

The original evaluation system exhibits the following observability characteristics:

- No explicit metrics for frontend behaviour.
- No end-to-end tracing across frontend applications
- Limited logging, primarily backend-focused

This was acceptable because:

- The system had a small number of components
- Failures were visually obvious in the browser
- Manual inspection was sufficient for debugging

However, this approach had the following assumptions:

- Low deployment frequency
- Limited user base
- Known and predictable workflows

These assumptions tend to break down very quickly in production systems.

Why Observability is Harder in Composed Frontends

In composed frontend architectures, failures are rarely manifested as complete outages. Instead:

- One frontend may be unavailable while others continue to work as designed
- A transcluded component may fail silently
- Navigation may succeed but render incomplete data
- Performance degradation may occur only on specific paths

This creates three core challenges:

Fragmented Execution Contexts

- Each frontend application has its own runtime, logs, and deployment lifecycle
- Browser-based composition spans multiple origins and services
- Errors may surface far from their root cause

User Journeys Spanning Multiple Systems

- A single user action can traverse:
  - Multiple frontend applications
  - One or more BFFs
  - Several backend services
- Without correlation, the failures appear to be disconnected

Graceful Degradation Masks Failures

- Navigation-based integration allows systems to continue functioning
- Failures may degrade UX rather than break functionality
- Silent failures can go unnoticed without explicit signals

As a result, observability becomes the only reliable way to understand system health at scale.

Observability as an Architectural Responsibility

In modern systems, observability must be designed into the architecture, not bolted on. this means answering key architectural questions like:

- What does “healthy” mean for each component?
- Which failures are acceptable, and which are not?
- How do we correlate events across boundaries?
- Who owns which signals?

Observability responsibilities should align with architectural ownership.

Observability Boundaries in a Composed Architecture

Frontend Application Observability

Each frontend application should own visibility into:

- Page load performance
- Client-side errors and exceptions
- User interaction failures
- Degraded states (e.g., missing transcluded content)

Key signals can include:

- Error rates (JavaScript errors, failed API calls)
- Performance metrics (LCP, TTFB, Interaction latency)
- Availability of dependent coronets (transclusion success/failure)

Eventually, Frontend observability must answer the question: “**Is the user experience working as intended?**”

Integration and Navigation Observability

Since integration happens at navigation and composition boundaries, visibility must include:

- Redirect success and failure rates
- Broken or malformed links
- Missing or invalid URL parameters
- Transclusion load failures

These signals can then help to detect:

- Contract drift between applications
- Partial outages
- Hidden integration regressions

Critically, successful navigation does not imply successful experience - observability must distinguish between the two.

Backend and BFF Observability

Backend services and BFFs remain central enforcement and aggregation points. Observability at this layer should focus on:

- API latency and error rates
- Authorization failures
- Schema compatibility issues
- Downstream dependency health

Since BFFs sit between frontend and backend systems, they are ideal places to:

- Correlate frontend requests with backend calls
- Enrich logs with user and request context
- Enforce consistent telemetry standards.

Tracing User Journeys Across Frontends

One of the most critical observability challenges is end-to-end visibility. In composed systems:

- A single user journey spans multiple frontend and backend components
- Traditional backend-only tracing will be insufficient

- Browser-to-backend correlation is essential

Modern approaches rely on:

- Propagated correlation IDs
- Consistent requests context across layers
- Explicit tracking of navigation events

this will enable development teams to answer questions like:

- Where did a user journey slow down?
- Which component caused a failure?
- Did a failure originate in frontend, BFF, or backend?

Without this visibility, troubleshooting devolves into guesswork.

Metrics That Matter in Frontend-Centric Systems

Not all metrics are equally valuable. I composed frontend architectures, metrics should reflect architectural intent, not just infrastructure health.

Experience-Oriented Metrics

- Page load success rate
- Time to meaningful interaction
- Error-free session percentage
- Degradation frequency (e.g., fallback rendering)

Integration Health Metrics

- Redirect failure rates
- Transclusion success/failure ratios
- Contract violation counts
- Dependency availability from browser's perspective

Deployment and Change Metrics

- Error rate changes post-deployment
- Performance regressions correlated with releases
- Rollback frequency

Together, these metrics provide a system-level view, not just component-level status.

Observability and Organizational Readiness

Observability is not just a technical capacity - it is an organizational enabler. Well-designed observability:

- Reduces mean time to detect [MTTD]
- Reduces mean time to recover [MTTR]
- Enables independent deployments with confidence
- Supports continuous architectural learning

Without it:

- Teams rely on anecdotal reports
- Failure are discovered by end users
- Autonomy degrades under operational uncertainty

In this sense, observability is a prerequisite for scaling both systems and teams.

Architectural Takeaway

The evaluation prototype succeeded without explicit observability because:

- Its scope was limited
- Its purpose was exploratory in nature
- Failures were easy to detect manually.

Modern composed frontend systems, however, require:

- Explicit observability at every integration boundary
- User-centric metrics, not just service health
- Correlated signals across frontend and backend layers
- Clear ownership of operational signals

Observability is no longer optional infrastructure- it is part of the architecture itself.

Transition to Next Section

With observability in place, the next natural questions become:

- How do we scale such a system reliably?
- How do we design for availability and resilience?
- What changes when we move this architecture to the cloud?

This leads directly into the next section on Scalability, Resilience and Cloud Deployment on AWS.

9. Scalability, Resilience, and Availability

Overview

As systems grow in user base, functionality, and organizational footprint, scalability and resilience stop being non-functional concerns and become core architectural drivers. In composed frontend architectures, this shift is particularly pronounced because failure modes are no longer centralised or binary in nature.

In the evaluation prototype, scalability and resilience were not the primary design goals. The system was intended to demonstrate frontend decomposition and integration patterns rather than operating under real-world load or failure conditions. That said, several architectural choices made in the prototype implicitly influenced how the system would behave under stress.

In this section, we will be examining:

- How scalability and resilience were implicitly handled in the evaluation prototype?
- What limitations emerge when similar patterns are applied at scale?
- How modern architectures explicitly design for growth, failure, and availability?
- How responsibilities shift across frontend, platform, and backend layers?

Scalability characteristics of the Evaluation Prototype

The evaluation prototype exhibits the following scalability characteristics:

Horizontal Scalability by Construction (Frontend only)

Since each frontend application is being deployed independently:

- Frontend components can be scaled independently.
- Traffic spikes affecting one application do not directly overload others
- Stateless frontend services can be replicated easily

This is a natural benefit of decomposition, even without explicit scalability planning. However, this type of scalability is uneven at best.

Centralized Backend as a Scaling Bottleneck

All frontend applications rely on a shared backend simulator and database schema. This introduces:

- A single scaling choke point
- Shared resource contention
- Limited ability to scale read/write paths independently.

Even if frontend components scale horizontally, the backend scalability ultimately constraints system throughput. this creates an illusion of scalability at UI layer that does not hod under sustained load.

Resilience Characteristics of the Evaluation Prototype

Navigation-based Resilience

One of the most interesting properties of the prototype is its inherent resilience through navigation-based composition. Since integration relies on:

- Links
- Redirects
- Optional client-side transclusion

The system exhibits the following behaviours:

- If a frontend application is unavailable, other applications still continue to function
- Broken integrations degrade to navigation links rather than hard failures
- Users can often continue working, albeit with reduced functionality

This is a form of graceful degradation, achieved without complex infrastructure

Failure Isolation at the UI Boundary

Each frontend application fails independently:

- A failure in the Postbox application does not break the Main Portal application.
- A failed transclusion does not block page rendering.
- Browser defaults provide basic isolation

This aligns well with the principle of "*Fail independently, recover independently*". However, this resilience is accidental in nature rather than designed.

Where the Prototype Reaches its Limits

While the prototype handles simple failures well, several limitations emerge as scaling increases.

Backend-Centric Failure Amplification

Because all applications depend on the same backend module:

- Backend outages ultimately affects all frontend applications simultaneously
- Partial frontend failures become system-wide failures
- Graceful frontend degradation cannot compensate for backend availability

Resilience at the UI layer cannot offset fragility at the business logic and data layers.

Lack of Load Awareness and Backpressure

The prototype des not address:

- Rate Limiting
- Backpressure handling
- Load shedding

As a result:

- Sudden traffic spikes propagate directly to the backend module
- Slowdowns cascade across frontend applications
- Users will end up experiencing latency rather than controlled degradation

No Explicit Availability Targets

The system does not define:

- Availability SLAs

- Degraded-mode expectations
- Critical vs. non-critical workflows

Without explicit targets, resilience remains incidental rather than intentional

Modern Approaches to Scalability and Resilience

Modern architectures treat scalability and resilience as explicit design goals, not emergent properties.

Scaling Along Architectural Boundaries

In modern systems:

- Frontend applications scale independently
- BFFs scale based on UI-specific traffic patterns
- Backend services scale by capability, not by consumer

This enables:

- Targeted scaling
- Cost-efficient resource allocation
- Reduced blast radius under load

Resilience Through Layered Responsibility

Rather than relying solely on browser behaviour, resilience is distributed across layers:

- **Frontend Layer**
  - Graceful degradation
  - Cached views
  - Fallback rendering
- **BFF Layer**
  - Circuit breakers
  - Request aggregation
  - Controlled failure responses
- **Backend Layer**
  - Bulkheads
  - Retries with backoff
  - Data partitioning

This creates defense in depth, rather than single-point resilience.

Availability in Composed Frontend Systems

Availability in modern systems is not absolute - it is contextual.

Partial Availability as a Feature

Modern architectures explicitly design for:

- Partial feature availability
- Read-only odes
- Progressive enhancement

Rather than asking "*Is the system up?*", the development teams ask "*Which capabilities are available right now?*". This mindset aligns naturally with frontend decomposition

CDN and Edge as Availability Enablers

While not present in the evaluation prototype, modern systems commonly use:

- CDN-backed static assets
- Edge caching for frontend content

- Regional traffic routing

This enables:

- Fast global access
- Reduced backend load
- Improved resilience to regional failures

Scaling Teams Along with Systems

Scalability is not only technical in nature - it is organizational as well. Modern approaches enable:

- Teams to own scalable units [Frontend + BFF + APIs]
- Independent release cycles
- Localized failure ownership

Without this alignment:

- Technical scalability is undermined by coordination overhead
- Operational risk increases despite horizontal scaling

The evaluation prototype hits at this alignment, but does not fully realise it.

Architectural Takeaway

The evaluation prototype demonstrates that:

- Frontend decomposition naturally improves failure isolation
- Navigation-based integration provides surprising resilience
- Independent deployments are a prerequisite for scalability

However, it also reveals that:

- Backend centralisation limits scalability
- Resilience must be designed, not assumed
- Availability requires explicit architectural intent

Modern architectures respond to this requirement by:

- Scaling systems along ownership boundaries
- Designing for partial failure and partial availability
- Treating resilience as a shared responsibility across layers

Transition to Next Section

With scalability and resilience established as architectural concerns, the next logical step is looking into the deployment context and ask:

- How would this architecture be hosted in the cloud?
- What changes when we move to a cloud provider like AWS?
- Where do infrastructure, networking, and delivery pipelines fit?

This leads directly into the Section 10: Cloud Deployment Strategy [AWS].

10. Cloud Deployment Strategy [AWS]

Overview

Moving this architecture into the cloud is not merely a hosting decision - it is an architectural amplification. Cloud platforms like AWS make certain architectural choices easier, safer, and more scalable, while also exposing weaknesses that were previous hidden, as in the case of evaluation prototype or on-premise setups.

In this section, we will explore how the evaluated architecture- and its modernized variants - would be deployed on a cloud setup like AWS, focusing on:

- Deployment boundaries and ownership

- Infrastructure responsibilities across layers
- How cloud-native primitives rebalance scalability, resilience, and security
- What should not be over-engineered prematurely.

The intent here is not to prescribe a single “correct” AWS architecture approach, but to show how AWS can enable intentional trade-offs that are aligned with the system’s evolution.

Cloud Readiness of the Evaluation Prototype

The original evaluation prototype is surprisingly cloud-friendly, even though it was not explicitly designed for the cloud environment.

What Translates Well

- Independently deployable frontend applications
- Stateless frontend services
- Clear HTTP-based integration
- Shared asset delivery via HTTP
- Container-friendly workloads

These characteristics naturally map to modern cloud primitives.

What Becomes a Constraint

- Shared monolithic backend component
- Tight coupling via shared database schema
- Lack of explicit identity and trust boundaries
- No separation between control plane and data plane concerns

Was does not fix these concerns automatically, but it makes them visible sooner.

Logical Deployment Units on AWS

A modern AWS deployment would typically separate the system into the following logical units:

Frontend Applications

Each frontend application becomes an independently deployable unit:

- Main Portal
- Damage Claims
- Letters
- Postbox

Key characteristics include:

- Independently scalable
- Independently deployable
- Independently observable
- Independently owned

Shared Assets

Assets [CSS, JavaScript, Fonts, Images] become:

- Versioned
- Immutable
- CDN-backed

This helps to reduce blast radius and improve performance of the system.

Backend & Integration Layer

Depending on the level of modernization we aim for:

- Legacy monolithic backend [initial state]
- Backend-for-Frontend [BFF] services [modernized state]
- Shared or decomposed APIs

This layer becomes the primary enforcement point for security, policy, and data access.

Frontend Hosting Model on AWS

Static vs. Dynamic Frontends

Modern AWS deployments typically split frontend concerns into:

Static Content

- HTML shells
- JavaScript bundles
- CSS and Assets

These can be hosted via:

- Object storage + CDN [e.g., S3 + CloudFront]
- Immutable, versioned deployments

Dynamic Behavior

- API calls
- Authenticated flows
- Personalization

These can be handled via:

- BFFs
- APIs
- Edge logic [selectively]

This separation improves performance, availability, and cost efficiency

API and Backend Hosting Considerations

Backend-for-Frontend [BFF] Pattern on AWS

Each frontend application may have a corresponding BFF and the responsibilities would include:

- Authentication termination
- Authorization enforcement
- API aggregation
- Data shaping
- Rate limiting

The benefits of this approach include:

- Reduced frontend-backend coupling
- Clear security boundary
- Better observability

Legacy Backend Hosting

If the backend refactoring is our of scope, then:

- Backend can be hosted as-is
- Frontend modernization can still proceed
- Cloud can act as an adapter, not a forcing function

This would then mirror organizational realities reflected in the original prototype.

Networking and Traffic Flow

AWS enables explicit control over traffic paths, including:

- Public entry points for frontend traffic
- Private networks for backend communication
- Controlled ingress and egress

This helps in enabling architects to define:

- Trust zones
- Blast radius boundaries
- Network-level isolation

Importantly, network boundaries should reinforce - not replace - application-level security

Availability and Fault Tolerance on AWS

AWS provides primitives that align well with composed frontend:

Multi-AZ Deployments

- Frontend services can run across availability zones
- Failures are isolated without user-visible downtime

CDN and Edge Caching

- Assets remain available even during backend outages
- Performance improves globally

Health Checks and Routing

- Unhealthy services can be removed from traffic.
- Degraded modes become easier to implement intentionally

Observability as a First-Class Cloud Concern

On AWS, observability is no longer option. Aspects like:

- Logs
- Metrics
- Traces

All become native signals, not just add-ons. This enables:

- End-to-end request tracing across frontends and BFFs
- Faster root cause analysis
- Clear ownership of failures

This directly addresses limitations identified in Sections 8 and 9.

Cost and Scalability Considerations

Cloud deployment introduces economic feedback loops:

- Independently scalable components enable cost isolation
- Over-coupled systems surface cost inefficiencies quickly
- Traffic patterns influence architectural decisions

This encourages:

- Better domain boundaries
- Conscious scaling decisions
- Avoidance of “scale everything” defaults

What Should Be Avoided Early

A key architectural discipling is knowing what not to do, including:

- Avoid premature service mesh adoption
- Avoid over-abstracting infrastructure
- Avoid coupling all frontends into a single runtime
- Avoid centralizing deployment pipelines too early

AWS enables complexity - it does not justify it

Architectural Takeaway

Deploying above discussed architecture on AWS will help to:

- Reinforce the value of frontend decomposition
- Expose backend coupling as a limiting factor
- Enable explicit security, scalability, and availability controls
- Align system architecture and team ownership

Most importantly, AWS shifts architecture from possibility to accountability

Transition to Next Section

With the deployment context established, the next logical question would be "*How do we manage this infrastructure safely, repeatably, and at scale?*". this leads directly into **Section 11: Infrastructure as Code, CI/CD, and Governance**, where we will address:

- How environments are defined
- How changes flow to production
- How guardrails replace manual coordination

11. Infrastructure as Code, CI/CD, and Governance

Overview

As systems evolve into collections of independently deployed components, manual coordination does not scale. Infrastructure, deployment, and operational controls must become:

- Repeatable
- Auditable
- Enforceable by default

In the evaluation prototype, these concerns were intentionally out of scope. However, in a modern cloud-based architecture - especially one built around independently deployed frontend applications - Infrastructure as Code [IaC], CI/CD pipelines, and governance mechanisms are not optional. They are the foundation that allows autonomy without chaos.

In this section, we will explore:

- How infrastructure should be defined and managed?
- How changes safely flow from code to production?
- How guardrails replace centralized controls?
- How governance evolves without blocking teams?

Infrastructure as Code [IaC]

Why IaC Becomes Mandatory

In a composed frontend architecture:

- Each frontend application may have its own infrastructure needs
- Shared components [CDN, identity, networking] must be consistent
- Environments must be reproducible across teams and stages

Manual provisioning quickly becomes:

- Error-prone
- Non-auditable
- Non-repeatable

IaC addresses this by making infrastructure a versioned, reviewable artifact

Logica IaC Boundaries

A clean separation of concerns typically emerges:

Platform-Level Infrastructure [Shared]

Owned by platform or enablement teams:

- Networking primitives
- Identity providers
- CDN configuration
- Observability foundations
- Security baselines

These evolve slowly and are governed centrally

Application-Level Infrastructure [Per Frontend/BFF]

Owned by individual tams:

- Hosting configuration
- Scaling rules
- Service-specific permissions
- Deployment environments

this preserves team autonomy while remaining compliant

Key Architectural Principle*: Infrastructure definitions should mirror architectural boundaries -not organizational convenience*

CI/CD Pipelines for Composed Systems

Why CI/CD is Not Just Automation

In this architecture, CI/CD pipelines are **policy enforcement mechanisms**, not just deliver tools. They help to ensure:

- Only valid artifacts reach production
- Contracts are respected
- Security checks are non-negotiable
- Rollback are possible and fast

Pipeline Responsibilities by Layer

Frontend Pipelines

Typical responsibilities include:

- Build and bundle UI artifacts
- Run unit and integration tests
- Enforce linting and dependency checks
- Publish immutable artifacts
- Trigger controlled deployments

Each frontend application owns its own pipeline.

BFF / Backend Pipelines

Additional responsibilities include:

- API contract validation

- Authentication and Authorization tests
- Backward compatibility checks
- Database migration controls [if applicable]

Deployment Independence with Safety

Pipelines help to enable:

- Independent releases
- Controlled rollouts
- Canary or progressive deployments
- Fast rollback on failure

This directly supports the autonomy goals that was highlighted in earlier sections.

Governance Without Central Bottlenecks

The Governance Problem

As autonomy increases, so does the risk of:

- Inconsistent practices
- Security drift
- Tool sprawl
- Contract erosion

Traditional governance usually relies on elements that do not scale, including:

- Central approval boards
- Manual reviews
- Documentation enforcement

Guardrails Over Gatekeepers

Modern governance shifts from permission-based control to policy-based enablement. Examples include:

- Mandatory security scans in pipelines
- Enforced infrastructure templates
- Versioned contract definitions
- Automated compliance checks

This enables teams t move fast within clearly defined boundaries.

Governance as Code

Governance rules themselves become code, including:

- Security policies
- Resource limits
- Network restrictions
- Identity requirements

This ensure governance is:

- Transparent
- Testable
- Evolvable

Security Integration into Delivery Pipelines [SecOps]

Security is no longer a downstream concern. In modern architectures:

- Static analysis [SAST]

- Dependency scanning [SCA]
- Infrastructure security checks
- Runtime policy enforcement

are integrated directly into CI/CD pipelines. this aligns with the Zero Trust principles we discussed in Section 6

Managing Change at Scale

Versioning and Compatibility

To avoid systemic breakage:

- Frontend contracts must be versioned
- APIs must support backward compatibility
- Deprecation must be explicit and observable

CI/CD pipelines become the enforcement point for these rules

Environment Parity

IaC enabled:

- Consistent environments across dev, test, and prod
- Reduced “works on my machine” failures
- Predictable deployment behavior

This is critical for independently deployed components

Organizational Implications

The combination of IaC, CI/CD, and governances reshapes teams:

- Platform teams build paved roads
- Product teams focus on business value
- Architecture becomes executable, not descriptive

This reflects the same organizational realism emphasized throughout this case study

Architectural Takeaway

Infrastructure as Code, CI/CD pipelines, and governance mechanisms are not supporting concerns - they are architectural primitives. In a composed frontend system, they:

- Enable autonomy without fragmentation
- Enforce security without blocking delivery
- Replace manual coordination with executable policy
- Allow systems to evolve safely over time

The original evaluation prototype demonstrated what was possible. Modern delivery practices determine what is sustainable.

Transition to Final Sections

With architecture, security, testing, observability, deployment, and governance established, the remaining questions become:

- How do cost and scalability influence architectural choices?
- How do we measure architectural success over time?
- What signals can tell us when the architecture must evolve again?

These lead naturally into the concluding sections on **Cost, FinOps, and Architectural Evolution**

12. Cost, FinOps, and Scalability Economics

Overview

As systems transition from prototypes and evaluations into long-running production platforms, cost becomes an architectural constraint, not an operational afterthought. Every architectural decision - deployment mode, integration strategy, runtime composition, observability depth - carries an economic footprint.

In the evaluation prototype we discussed earlier, cost considerations were intentionally minimized. This was appropriate give its goals of experimentation, learning, and architectural exploration. However, in a modern cloud-hosted, independently deployed frontend ecosystem, cost efficiency and scalability economics must be explicitly designed.

In this section, we will examine:

- How architectural choices influence cost behavior?
- Where cost pressure emerges in composed frontend systems?
- How FinOps practices help balance speed, scale, and spend?
- How cost signals inform architectural evolution over time?

Cost Characteristics of the Evaluation Prototype

The original architecture exhibits a number of cost-friendly traits:

- Minimal infrastructure components
- Monolithic backend simulator
- Browser-based composition with no orchestration layer
- Static asset delivery

From a cost perspective, this meant:

- Low infrastructure overhead
- Predictable runtime behavior
- Limited operational tooling

However, these benefits came at the expense of:

- Scalability flexibility
- Observability depth
- Independent cost attribution

As the system scales or moves to production-grade hosting, these trade-offs become increasingly visible.

Cost Drivers in Modern Composed Frontend Architectures

Modern frontend architectures introduce new cost vectors that must be understood and managed.

Independent Deployments

Each frontend application may incur:

- Build and deployment costs
- Hosting and runtime costs
- Monitoring and logging costs

While autonomy increases, so does the need for cost attribution per team or product area.

Increased Observability

Modern observability stacks introduce cost related to:

- Metrics ingestion
- Log retention
- Distributed tracing
- Real-user monitoring [RUM]

Without guardrails, observability can quickly become one of the largest cost centers

Backend-for-Frontend [BFF] Layers

While BFFs reduce coupling and improve UX, they also:

- Introduce additional compute layers
- Increase request fan-out
- Require scaling and monitoring

Architecturally beneficial layers must still justify their economic footprint

Traffic Amplification

In composed systems:

- A single user interaction may trigger multiple requests
- Client-side transclusion increases cross-service calls
- Partial reloads can multiply backend interactions

This amplifies both compute and network costs

FinOps as an Architectural Discipline

FinOps is not about cost cutting - it is about cost-aware decision making. In a composed frontend system, FinOps principles help answer:

- Which architectural choices deliver proportional business value?
- Where does elasticity outperform fixed provisioning?
- When does simplicity beat theoretical scalability?

Key FinOps Principles Applied

Visibility

Costs must be:

- Attributed per application, team, or capability
- Visible alongside performance and reliability metrics

without visibility, optimization is guesswork

Accountability

teams that own services should also:

- Understand their cost impact
- Participate in optimization decisions
- Balance performance, reliability, and spend

this reinforces true end-to-end ownership

Optimization, not Minimization

The goal is not the lowest cost, but the best cost-to-value ratio. Examples include:

- Paying more for faster depolyments if it accelerates delivery
- Reducing observability granularity where it adds little insight
- Accepting higher CDN costs for significantly improved UX

Architectural Levers for Cost Control

CDN and Edge Optimization

Frontends benefit heavily from:

- Aggressive caching
- Static asset delivery via CDN

- Reduced origin traffic

This often delivers the highest ROI per architectural effort

Tiered Observability

No all signals need the same retention or resolution:

- High-cardinality traces only where needed
- Aggregated metrics for steady-state monitoring
- Sampling strategies for RUM and tracking

Observability becomes intentionally designed, not blindly enabled

Elastic Scaling

Modern platforms allow:

- Scale-to-zero for low-traffic components
- Event-driven scaling
- Demand-based resource allocation

this aligns cost with actual usage rather than peak assumptions

Contract-Aware Optimization

Explicit frontend and API contracts allow:

- Predictable request patterns
- Safer caching strategies
- Reduced over-fetching

Better contracts lead directly to better cost behavior

Scalability Economics: When Architecture Meets Growth

Scalability is not just about handling load - it is about handling load economically. A system that scales technically, but becomes prohibitively expensive is not sustainable. The key considerations include:

- Linear vs. exponential cost growth
- Diminishing returns for added layers
- Operational overhead per additional component

Architectures must be periodically re-evaluated against real usage patterns, not theoretical models

Architectural Takeaway

Cost is a first-class architectural signal. In composed frontend systems:

- Autonomy introduces economic responsibility
- Observability introduces cost as well as insight
- Scalability must be justified, not assumed
- FinOps bridges architecture and business reality

The evaluation prototype optimised for learning and simplicity. Modern architectures must optimise for sustained value over tie. The most resilient architectures are not the most complex ones - they are the ones that **balance autonomy, scalability, and economics with intent**.

Transition to Conclusion

With architecture, security, testing, observability, delivery, governance, and cost addressed, we have now reached the final question "*How do we know when an architecture has succeeded - and more importantly, when it must evolve again?*". This leads naturally into the **final section on architectural signals, evolution, and closing reflections**.

13. Architectural Signals, Evolution, and When to Change

Overview

Architectures are not static artifacts. They are living systems shaped by usage, organizational structures, platforms capabilities, and time. What begins as a well-reasoned design can gradually drift from its original assumptions as constraints change and new pressures emerge.

In this final section, we will reflect on:

- How to recognize when an architecture is working well?
- what signals indicate growing misalignment?
- When evolution is warranted - and when restraint is the better choice?
- Why architectural success is measured over time, not at design completion?

Architecture as a Set of Hypotheses

Every architecture encodes a set of assumptions about:

- User behavior
- Scale and growth
- Team Structure
- Technology maturity
- Risk tolerance

The evaluation prototype described in this case study represents a set of hypotenuses that were valid at the time, including:

- Frontend autonomy could be achieved through navigation-based composition
- Backend ownership constrains were real and non-negotiable
- Browser-native mechanisms were sufficient integration primitives
- Simplicity would outweigh theoretical scalability needs

the architecture did not fail when these assumptions changed. It simply reached the boundaries of what it was designed to optimize for

Positive Architectural Signals

Architectures rarely announce success explicitly - but they do emit healthy signals. Examples include:

- Teams deploy independently with confidence
- Incidents are localized rather than systemic
- Changes are predictable in scope and impact
- Developers understand the system without excessive documentation
- Operations efforts remain proportional to system size

The evaluation prototype demonstrated many of these signals in its intended context:

- Clear ownership of frontend components.
- Minimal coordination for change
- Low operational overhead
- High resilience through browser-level isolation

These are indicators of architectural alignment, not accidental success

Warning Signals That Demand Attention

Equally important is recognising when an architecture begins to emit stress signals. Common indicators include:

- Increasing coordination for simple changes
- Repeated workarounds around core constraints
- Growing divergence between intended and actual uage
- Rising operational or cognitive overhead

- Escalating cost without proportional value

In the evaluated site, such signals would emerge as:

- Fragile URL and navigation contracts
- Backend schema changes blocking frontend evolution
- UX degradation from repeated reloads
- Difficulty tracking end-to-end user journeys
- Increased reliance on shared assets as hidden coupling points

These are not failures - they are signals that the architecture is being asked to do more than what it was designed for.

Evolution vs. Replacement

One of the most common architectural mistakes is assuming that change requires replacement. In reality, successful systems evolve incrementally by:

- Introducing new patterns alongside new ones
- Strengthening contracts without breaking flows
- Isolating complexity rather than redistributing it
- Preserving work simplicity where possible

Modern patterns such as:

- Backend-for-Frontend laers
- Explicit frontend contracts
- Platform-provided observability
- Progressive micro-frontend adoption

Are best introduced in response to concrete signals, not as theoretical upgrades.

The Cost of Over-Engineering

Equally dangerous is evolving too early. Premature adoption of:

- Heavy orchestration
- Complex runtime composition
- Excessive abstraction layers
- Overly rigid governance models

Can erode the very benefits the original architecture delivered, including:

- Autonomy
- Speed
- Clarity
- Resiience

The evaluation prototype reminds us that simplicity is not naive - it is often the most appropriate response to real constraints.

Architecture as Organizational Memory

Architectures encode more than technical decisions - they preserve organizational learning. this case study captures:

- The state of frontend architecture at a specific moment in time.
- The constraints team were operating under
- The trade-offs that were consciously accepted
- The lessons that informed later evolution

Seen this way, architecture becomes a form of institutional memory - one that helps future teams understand why things are the way they are, not just what exists today.

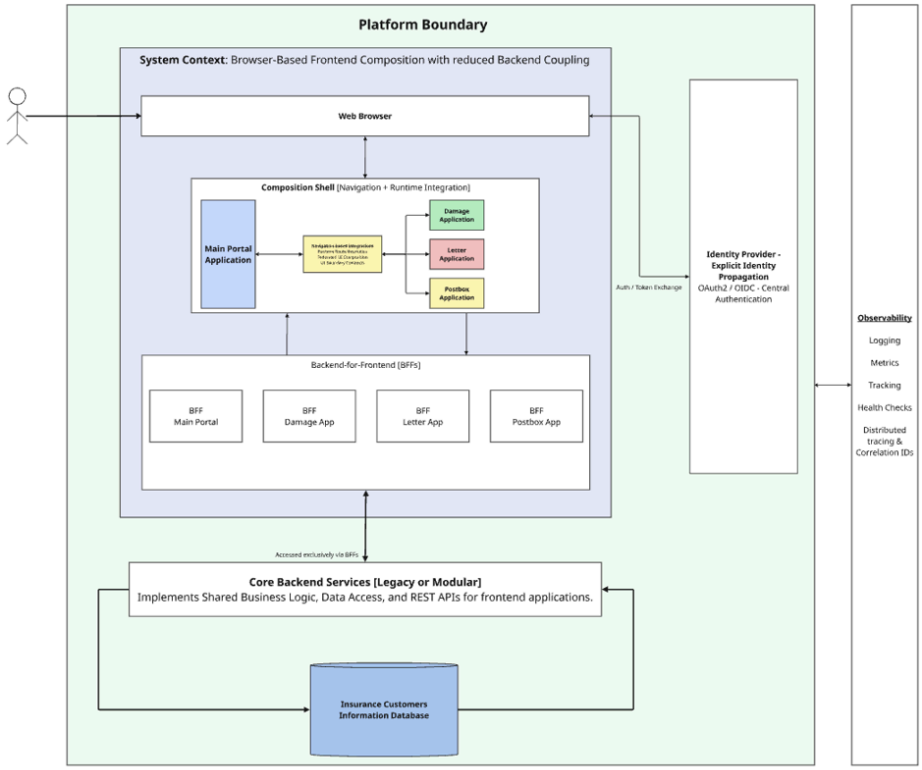
Final Takeaway

The most important lesson from this case study is not a specific pattern or technology. It is this:

> Good architectures are not defined by how advanced they are, but by how well they align with reality - and how gracefully they evolve when reality changes.

The evaluation prototype succeeded because it optimised from its moment in time. Modern architectures succeed when they respect that lineage rather than discard it.

Figure: Rebalanced frontend architecture with explicit trust boundaries, runtime composition, per-frontend backend isolation, and platform-level observability



Legend:
This architecture rebalances coupling by shifting identity, observability, and scalability concerns into the platform layer, while preserving frontend autonomy through explicit contracts and per-frontend backends. Testing and security responsibilities align with these boundaries rather than individual components.

Closing Reflection

Architectural maturity is not about having all the answers upfront. It is about:

- Listening to signals
- Responding proportionally
- Knowing when to hold steady - and when to change

This case study is ultimately about **managing coupling organizational constraints**, not frontend architecture per se.

It is also not an endpoint. It is a reflection point - one that continue to be value precisely because it was grounded in real constraints, real trade-offs, and real learning

Key Takeaways

- Navigation is a valid architectural boundary
- Implicit contracts scale poorly
- Modern platforms do not remove complexity - they relocate it
- Architecture is inseparable from organization