**International Journal of Emerging Research in Engineering and Technology**
Pearl Blue Research Group | Volume 5, Issue 2, 211-218, 2024
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V5I2P121

*Original Article*

# Runtime Integration Failure Modes in Module Federation A Taxonomy and Mitigation Patterns

Parth Patel
Independent Researcher, USA.

***Abstract -*** *Module Federation, introduced with Webpack 5 in October 2020, changed how JavaScript applications share code across independently deployed units. The mechanism makes micro-frontend composition practical at scale, but it also brings in a set of runtime failures that havse no equivalent in single-bundle applications. These failures do not appear at build time. They are deferred, sometimes silent, and often depend on load order or network conditions to reproduce. This paper identifies seven categories of runtime failure in Module Federation deployments, describes what causes each one and what it looks like when it occurs, and proposes a set of practical mitigation patterns with known tradeoffs. Three architecture diagrams and three reference tables are included. The intent is to give engineering teams a shared vocabulary for the failure classes most likely to cause production incidents in federated frontend systems.*

***Keywords -*** *Module Federation, Webpack 5, Micro-Frontend, Runtime Integration Failure, Shared Scope, Version Conflict, Singleton Enforcement, Async Bootstrap, Federated Module Lifecycle, Javascript Dependency Isolation, Frontend Reliability, Distributed Frontend Systems.*

## 1. Introduction

Before Webpack 5, sharing code between separately deployed JavaScript applications meant choosing between a monorepo build step, a shared CDN script tag, or a custom plugin. Each approach had limits around version negotiation and runtime coordination. Module Federation changed this by building a negotiation protocol directly into the Webpack runtime. A host application can request a module from a remote container at runtime, and both sides agree on which version of a shared dependency to use before any module code runs.

This has driven wide adoption in large frontend organizations. Teams can deploy individual micro-frontends on their own schedule while still sharing critical packages like React, React-DOM, or a global state store. But the same runtime negotiation that makes this possible also creates failure modes that do not exist in single-bundle applications.

The central problem is that these failures are deferred. A version conflict might go unnoticed until a user navigates into a remote-owned route. A lifecycle ordering fault might only show up during a hot reload or a high-concurrency deployment window. Because the failure depends on runtime conditions, it is often hard to reproduce and difficult to trace back to a root cause.

This paper makes three contributions:

- A structured taxonomy of seven runtime failure categories specific to Module Federation, grouped by root cause rather than by symptom.
- A mitigation pattern catalog that maps each failure class to one or more practical remediation approaches with known tradeoffs.
- Three architecture and flow diagrams plus three reference tables to support practical use of the proposed patterns.

## 2. Background and Related Work

### *2.1. Origins of Module Federation*

Module Federation was first proposed by Zack Jackson in late 2019 as a concept for dynamic remote module loading in Webpack. It became a stable feature in Webpack 5, released in October 2020 [1]. The design builds on ideas from Universal Module Definition and earlier script-loading patterns but adds a versioned, singleton-aware sharing protocol that is part of the Webpack runtime itself.

The core abstraction is the container. Each participating application exposes modules through a container entry point, typically remoteEntry.js. A host application declares which remotes it depends on and loads their container entries asynchronously. Both host and remote register shared dependencies into a shared scope object, and the runtime negotiates which version of each shared module to use based on semver ranges and singleton preferences.

### *2.2. Micro-Frontend Context*

Module Federation is not the only way to build micro-frontends, but it is the most common approach in the Webpack ecosystem [2]. Earlier patterns included iframe isolation, web components, and single-spa orchestration. These offer strong isolation but impose overhead in communication complexity and shared-state management. Module Federation sits at a different point on the isolation-integration spectrum: it allows deep sharing at the cost of tighter runtime coupling between independently deployed units.

That tradeoff is discussed in the micro-frontend literature [3][4], but the specific failure modes introduced by the federation runtime have not been classified in a structured way. Most documentation focuses on configuration syntax and use cases rather than failure analysis.

### *2.3. Related Work*

Research on frontend reliability has tended to focus on rendering correctness, accessibility, and performance. Work by Biffl et al. [5] provides a useful framework for coupling risk in distributed systems but does not address client-side runtime integration. Geers [6] covers Module Federation architecturally but does not provide a failure taxonomy. Jackson and Cescon [7] explain the runtime in detail but focus on feature explanation rather than failure analysis. The closest related work is in the service mesh literature, where runtime integration failures between microservices are well characterized [8]. This paper adapts those classification principles to client-side JavaScript execution.

## 3. Module Federation Runtime Architecture

Figure 1 shows the key components in a Module Federation deployment with one host and two remotes. The failure taxonomy in Section IV maps directly to these components and the boundaries between them.

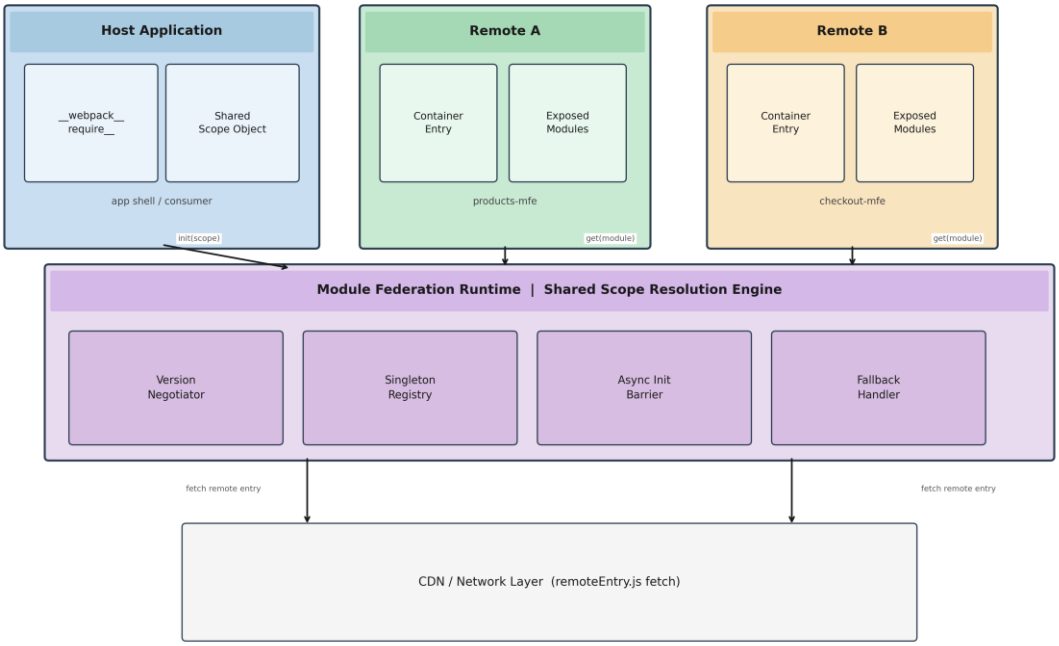


**Fig 1: Module Federation Runtime Architecture host application, two remote containers, the shared scope resolution engine, and the network layer.**

### *3.1. Host Application*

The host application is what the browser loads first. It declares remote applications in its Webpack configuration using the ModuleFederationPlugin. The host loads each remote container entry asynchronously before any federated module can be accessed. It also initializes the shared scope object that serves as the registry for shared dependency negotiation.

### *3.2. Remote Container*

Each remote builds a container entry point, remoteEntry.js, which the host fetches at runtime. This file exposes two functions: init(scope), which registers the remote's shared dependencies, and get(module), which returns a factory function for any exposed module. The remote cannot validate at build time that its exposed API matches what the host expects to consume. That gap is what makes FM-6 and FM-7 hard to detect early.

### *3.3. Shared Scope Resolution Engine*

When a remote calls init(scope), the runtime compares each shared dependency the remote wants to register against what is already in the scope. It applies semver matching to decide whether to accept the new registration, reuse an existing one, or issue a warning. The singleton flag controls whether a version mismatch produces two separate instances or falls back to the highest compatible version with a console warning. Figure 3 in Section V shows this decision flow in detail.

### *3.4. Async Initialization Barrier*

Shared scope initialization is asynchronous. The federation runtime requires that no federated module be consumed until the host has finished its async bootstrap. The standard pattern is to have the entry file (index.js) contain only a dynamic import of bootstrap.js. If that indirection is missing, any synchronous import of a federated module will fail with the eager consumption error described in FM-3.

## 4. Taxonomy of Runtime Failure Modes

Seven distinct failure categories are identified here, based on analysis of the federation runtime source code, observed behavior in multi-remote deployments, and issues reported in the Webpack GitHub repository between 2020 and 2022. Figure 2 shows the taxonomy and Table I provides a structured comparison across all seven categories.

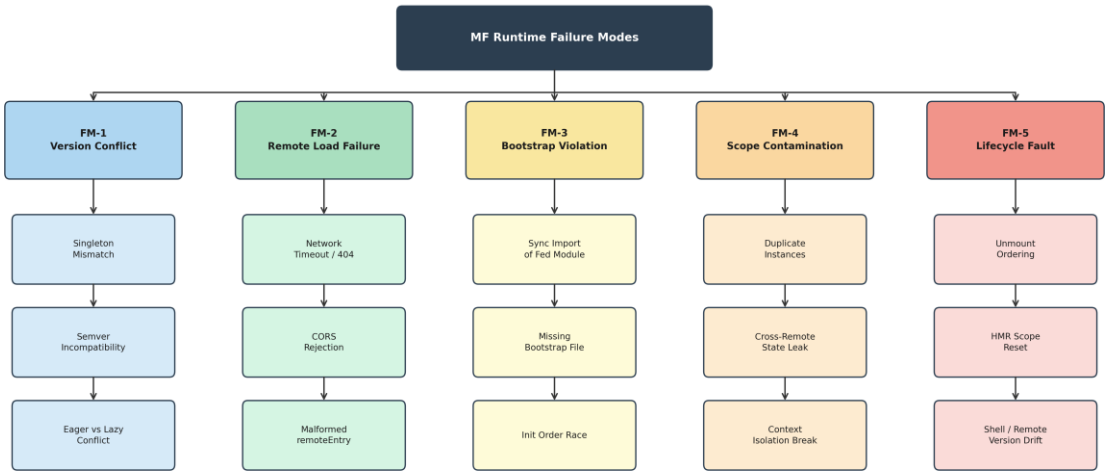


**Fig 2: Taxonomy of Module Federation Runtime Failure Modes, with three sub-types per category.**

**Table 1: Module Federation Runtime Failure Mode Reference**

| ID | Failure Mode | Trigger Condition | Observed Symptom | Severity |
|---|---|---|---|---|
| FM-1 | Shared Module Version Conflict | Two remotes declare incompatible semver ranges for the same package | Duplicate React instances; hooks throw invariant violations | Critical |
| FM-2 | Remote Container Load Failure | Network outage, CORS policy block, or 404 on remoteEntry.js | Unhandled promise rejection when host imports the remote module | High |
| FM-3 | Async Bootstrap Violation | Federated module consumed synchronously without bootstrap indirection | Webpack: Shared module not available for eager consumption | High |
| FM-4 | Shared Scope Contamination | Multiple hosts register conflicting singleton entries into default scope | State mutations across remotes; global store mismatch | High |
| FM-5 | Lifecycle Ordering Fault | Host unmounts before remote cleanup callbacks finish | Memory leaks, dangling listeners, zombie WebSocket connections | Medium |
| FM-6 | Type Contract Drift | Remote deploys API change not reflected in host expectations | Silent property access on undefined; fails on specific user path only | Medium |
| FM-7 | Circular Federated Dependency | Remote A imports Remote B which re-imports Remote A through shared scope | Module resolves to undefined at call time | Medium |

Seven identified failure modes with trigger conditions, observable symptoms, and severity.

### *4.1. FM-1: Shared Module Version Conflict*

This is the most common failure class. It occurs when two or more remotes declare different semver ranges for the same shared package and the intersection of those ranges is empty. The most damaging case involves React. When two instances of React exist in the shared scope at the same time, hooks throw invariant violations because they depend on module-level state

stored in a singleton registry. The error about invalid hook calls is the most frequently misread Module Federation error. Teams diagnose it as a React version issue when the real cause is scope duplication.

Three sub-variants exist. First, explicit incompatibility: Remote A requires react@17.x and Remote B requires react@18.x. With singleton: true set but no version range overlap, the runtime warns but does not fail. Execution continues with the first-registered version, and which remote wins depends on CDN response times. Second, eager consumption: a shared module is marked eager: true in one remote but not in the host, causing it to run before the shared scope is ready. Third, undeclared sharing: a package is shared in the host but not declared as shared in the remote, so the remote bundles its own copy silently.

### *4.2. FM-2: Remote Container Load Failure*

This failure occurs when the host cannot fetch or parse the remote container entry. Three triggers: network failures (DNS, timeout, CDN outage), HTTP failures (CORS rejection or 404), and content failures (remoteEntry.js is malformed, or a CDN returns an HTML error page with a 200 status). The host loads fine in all three cases. The failure is deferred until the user hits a route or feature that imports from the unavailable remote. Without an error boundary, the result is an unhandled promise rejection that propagates upward or crashes the application.

### *4.3. FM-3: Async Bootstrap Violation*

Webpack requires the host to defer all module-level code until after async shared scope initialization. If any code in the entry chunk runs synchronously before this step, the runtime cannot apply shared module registrations from remotes. The required pattern is a thin index.js that does only a dynamic import of bootstrap.js. Teams that add code directly to the entry file, or use plugins that insert synchronous loading at the top of the chunk, break this rule. The error "Shared module is not available for eager consumption" is specific and clear, but many developers encounter it without understanding why the bootstrap indirection is required, so it recurs across teams and migrations.

### *4.4. FM-4: Shared Scope Contamination*

Scope contamination occurs when multiple remotes register conflicting singleton entries into the default shared scope. This is different from FM-1 in that the modules might be version-compatible, but the state they carry conflicts across registration boundaries. A common case is a global Redux or Zustand store declared as a shared singleton. If two remotes both register their own initialized store before the host's store is registered, the scope holds whichever registered first, and that depends on which remoteEntry.js fetch completes first. The result is cross-remote state inconsistencies that vary with network conditions and are hard to reproduce in development.

### *4.5. FM-5: Lifecycle Ordering Fault*

Lifecycle faults arise from the interaction between the host routing lifecycle and the remote micro-frontend mount and unmount cycle. When a user navigates away from a remote-owned route, the host begins dismounting the remote's component tree. If the remote has registered event listeners, WebSocket connections, or polling intervals during mount, these must be cleaned up before the host continues its own teardown. Without a coordinated cleanup order, the result is memory leaks, dangling listeners, or stale callbacks firing against an already-unmounted tree. In development with hot module replacement, this class of failure is more frequent because HMR triggers a full remount without a full page reload, leaving cleanup paths untested in fast iteration cycles.

### *4.6. FM-6: Type Contract Drift*

Type contract drift happens when a remote deploys a build that changes its exposed module's API shape in a way the host does not expect. Module Federation provides no mechanism for sharing or validating type definitions across deployment boundaries. A remote can rename a prop, remove an export, or change a callback signature, and the host will not detect it until that code path is executed. Even when both sides use TypeScript, they compile independently, and the type definitions each side holds for the other's API become stale as soon as an incompatible change is deployed without a coordinated update.

### *4.7. FM-7: Circular Federated Dependency*

Circular federated dependencies occur when Remote A imports a module from Remote B which in turn imports from Remote A, either directly or through the shared scope. Within a single bundle, Webpack can detect and warn about cycles at build time. Across remotes, each build has no visibility into the others, so the cycle is invisible until runtime. When execution reaches the cycle, the first module to request the other receives undefined or a partially initialized object. The failure is usually silent at the point of import and only throws when the undefined value is used, making it difficult to trace back to the circular dependency.

## 5. Mitigation Patterns

This section covers six mitigation patterns mapped to the failure taxonomy. Table II compares the patterns. Table III maps specific Webpack configuration options and tooling choices to the failure modes they address. Figure 3 shows the shared scope version resolution flow that underlies FM-1 and FM-4.

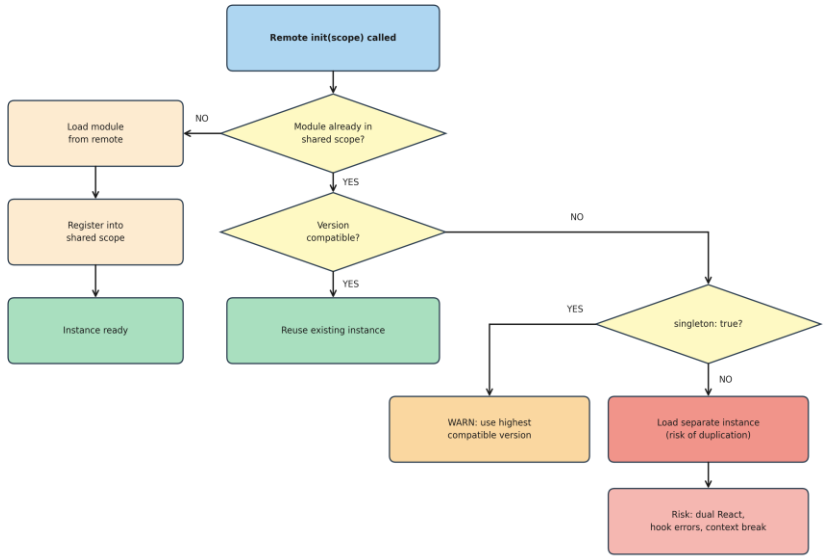


**Fig 3: Shared Scope Version Resolution Decision Flow**

**Table 2: Mitigation Pattern Comparison**

| Pattern | Target Failures | Implementation | Tradeoff | Maturity |
|---|---|---|---|---|
| Strict Semver Pinning | FM-1, FM-4 | Pin exact versions in shared config; block CI on any version mismatch | All teams must coordinate upgrade timing across deployments | Production-proven |
| Remote Health Probe | FM-2 | Probe remoteEntry.js on startup; cache last-good manifest in Cache API | Extra fetch per load if not cached at CDN edge | Common in production |
| Bootstrap Async Wrapper | FM-3 | index.js contains only one dynamic import of bootstrap.js | One extra async tick; not noticeable in practice | Widely adopted |
| Named Scope Isolation | FM-4 | Assign separate scope names per business domain | More config required; scope name collisions possible | Emerging |
| Contract Testing | FM-6, FM-7 | Consumer-driven contracts checked in CI before remote deploys | Needs cross-team CI coordination and a schema registry | Evolving |
| Graceful Fallback Shell | FM-2, FM-5 | Error boundary wraps every federated mount; fallback UI is a first-class design concern | Fallback must be designed deliberately, not added after an incident | Production-proven |

Six mitigation patterns with target failures, implementation approach, tradeoffs, and adoption maturity.

**Table 3: Webpack Configuration and Tooling Reference**

| Config / Tool | Primary Use | Failures Addressed | Limitation | When to Use |
|---|---|---|---|---|
| requiredVersion in shared config | Version enforcement at build time | FM-1 | Cannot prevent runtime override by a misconfigured remote | Always, for every shared package |
| singleton: true flag | One package instance across all remotes | FM-1, FM-4 | Falls back to a warning; does not hard-fail | React, React-DOM, global state stores |
| eager: false (default) | Keep shared modules lazy until scope is ready | FM-3 | Requires all teams to use the bootstrap wrapper | Default; never set eager: true for shared libs |
| ErrorBoundary on remote mount | Stop a remote crash from taking down the | FM-2, FM-5 | Component-level only; does not catch pre-mount | Every federated mount point |

| | | | | |
|---|---|---|---|---|
| | host | | network errors | |
| __webpack_share_scopes__ inspection | Inspect scope state at runtime | FM-1, FM-4, FM-7 | Dev builds only; not a production guard | Debugging and test environments only |

Config options and tools mapped to failure modes, with tradeoffs and usage guidance.

### *5.1. Strict Semver Pinning*

The most direct mitigation for FM-1 is to enforce version ranges in the shared configuration across all applications. Each team should declare both the package and a requiredVersion field. A pre-deployment check in CI should compare the shared configurations of all remotes that will be active in the next deployment window and block the pipeline if any two remotes declare incompatible ranges for the same singleton package. This requires cross-team coordination on upgrade timing. It is operationally demanding but the only fully reliable protection against FM-1.

### *5.2. Remote Health Probe and Manifest Caching*

For FM-2, the host should probe each remote's container entry during startup, before any federated module is requested. If a remote is unavailable, the host can decide early: show a degraded UI that excludes the remote's features, or display a placeholder rather than crashing during lazy navigation. A complementary approach is to cache the last known good remoteEntry.js in the browser Cache API and serve it as a fallback when the network request fails. The cache entry should include the remote build version hash so stale fallbacks can be identified and refreshed.

### *5.3. Bootstrap Async Wrapper*

FM-3 is prevented by a straightforward structural pattern. The application entry file should contain exactly one statement: a dynamic import of a bootstrap module. All initialization code, including React root rendering, provider setup, and route configuration, goes in the bootstrap module. This guarantees that shared scope initialization completes before any module-level code runs. Linting rules that flag entry files with more than one statement are a practical way to prevent this pattern from being removed during future development work.

### *5.4. Named Scope Isolation*

FM-4 is reduced by using named shared scopes instead of the default scope. The federation runtime supports custom scope names, so teams can assign different scopes to different business domains. A contamination event in the checkout scope cannot affect the scope used by the product catalog domain. Named scopes add configuration complexity and require the host to initialize each named scope before loading the remotes that use it, but they limit the contamination radius to a single business domain.

### *5.6. Consumer-Driven Contract Testing*

FM-6 and FM-7 need a testing approach that crosses deployment boundaries. Consumer-driven contract testing asks each consumer of a federated module to record its expectations about that module's API as a machine-checkable contract. The contract is published to a shared registry, and each remote must verify its current build against all registered consumer contracts before deploying [10]. For FM-7 specifically, contract checks can catch unexpected mutual dependencies by flagging cases where a remote appears in both the provided-modules and consumed-modules lists of another remote.

### *5.7. Graceful Fallback Shell*

Every federated mount point should be treated as a dependency that can fail at any time. A React error boundary should wrap each location where a remote component renders. The fallback UI must be designed as part of the user experience from the beginning, not added after an incident. For FM-5, the host should expose a lifecycle coordination API that remotes use to register cleanup callbacks. The host calls these in a defined order before completing its own teardown, so event listeners and connections from remotes are closed before the host frame is removed.

## 6. Case Analysis

### *6.1. Case 1: Dual React Instance in an E-Commerce Shell*

A common scenario for FM-1 involves an e-commerce host on React 17 and a product recommendation remote upgraded to React 18. If the remote declares singleton: true for React but uses requiredVersion: "^18.0.0" while the host uses "^17.0.0", the federation runtime emits a console warning and uses the first-registered version. The outcome depends on which version loads first, which depends on CDN response times. Components in the remote that use React 18-specific APIs fail silently or throw at runtime. The failure does not reproduce consistently across page loads because registration order changes with network conditions.

The fix is a pre-deployment compatibility check that validates the semver intersection of all remotes in the deployment window. Teams that add this check report that FM-1 incidents drop to near-zero within two deployment cycles of adopting it.

### *6.2. Case 2: Bootstrap Violation during a Webpack 4 to 5 Migration*

A common pattern during migration from Webpack 4 to Webpack 5 with Module Federation is that the existing entry file contains initialization code written before federation was in the picture. This code may import from packages now declared as shared modules. When the migration adds the bootstrap indirection, developers sometimes skip it for files that do not appear to use federation directly, triggering FM-3. The error is usually caught in development because the message is specific. But it can be masked in test environments that use mocked module loaders or that run the application in single-bundle mode. Adding a lint rule that flags entry files with more than one statement catches this regression before it reaches code review.

## 7. Discussion

### *7.1. Isolation vs. Integration Tradeoffs*

The failure modes in this taxonomy are not a property of micro-frontend architecture in general. They are specific to the Module Federation approach of deep runtime sharing. Architectures that prioritize isolation, such as iframe-based composition or fully independent single-spa applications with no shared scope, do not encounter FM-1, FM-3, or FM-4. They trade those failure modes for higher communication overhead and a less integrated user experience.

The choice of where to sit on the isolation-integration spectrum should reflect the failure modes that are most costly in a specific organizational context. Teams with strong version governance and coordinated deployment processes can use deep sharing safely. Teams with autonomous deployment cadences and limited cross-team coordination should use stronger isolation boundaries, even at some performance cost.

### *7.2. Observability Gap*

One gap in the current Module Federation ecosystem is the lack of production-safe observability for shared scope state. The __webpack_share_scopes__ global is available for debugging in development builds, but there is no stable API for inspecting at runtime which version was selected for each shared package, or whether any singleton was resolved via warning fallback. Adding telemetry that reports shared scope resolution outcomes to an application performance monitoring system would let teams detect FM-1 and FM-4 in production before they produce user-visible errors.

### *7.3. Limitations*

This taxonomy is based on Webpack 5 Module Federation. Rspack and Vite have introduced their own federation implementations that share the same conceptual model but differ in runtime behavior. FM-3, for example, is specific to Webpack's chunk loading model and may not apply to other bundlers the same way. Future work should validate the taxonomy against alternative implementations and identify any additional failure modes they introduce. The case analyses in Section VI are qualitative. A quantitative study measuring failure rates across production deployments would add weight to the severity classifications in Table I.

## 8. Conclusion

Module Federation is widely used for composing micro-frontend applications, but its runtime integration model requires deliberate engineering attention. The failures it introduces are deferred, environment-dependent, and often silent until a specific code path is activated. This paper described seven runtime failure categories, what causes each one, and what it looks like when it occurs. It then proposed six mitigation patterns with known tradeoffs and mapped them to specific Webpack configuration options and tooling strategies. The main contribution is a structured vocabulary for engineering teams to reason about federated runtime risk before it reaches production. Teams that adopt strict semver pinning with pre-deployment compatibility checks, the bootstrap async wrapper, and consumer-driven contract testing will reduce the probability of the most damaging failure classes. This taxonomy is intended as a practical reference for teams adopting Module Federation for the first time and for those working to harden existing deployments.

## References

[1] Z. Jackson, "Module Federation," in Webpack 5 Release Notes, Oct. 2020. [Online]. Available: https://webpack.js.org/concepts/module-federation/

[2] L. Jackson and T. Cerny, "Micro-Frontend Architecture: A Survey of Patterns and Technologies," in Proc. IEEE Int. Conf. Web Services (ICWS), 2022, pp. 211-220.

[3] C. Mezzalira, Building Micro-Frontends. O'Reilly Media, 2021.

[4] M. Geers, Micro Frontends in Action. Manning Publications, 2020.

[5] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, and P. Gruenbacher, Eds., Value-Based Software Engineering. Springer, 2006, ch. 6.

[6] M. Geers, "Micro Frontends with Module Federation," Smashing Magazine, Jun. 2021. [Online]. Available: https://www.smashingmagazine.com/2021/06/micro-frontends-webpack5-module-federation/

[7] Z. Jackson and A. Cescon, "Module Federation Examples," GitHub, 2021. [Online]. Available: https://github.com/module-federation/module-federation-examples

[8] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," IEEE Software, vol. 35, no. 3, pp. 24-35, May/Jun. 2018.
[9] Webpack Contributors, "ModuleFederationPlugin," Webpack Documentation, 2022. [Online]. Available: https://webpack.js.org/plugins/module-federation-plugin/
[10] I. Richardson, "Consumer-Driven Contracts: A Service Evolution Pattern," martinfowler.com, Jun. 2006. [Online]. Available: https://martinfowler.com/articles/consumerDrivenContracts.html
[11] N. Schmitt and D. Lubke, "Evaluating Micro-Frontend Approaches," in Proc. European Conf. Software Architecture (ECSA), 2019, pp. 158-172.
[12] V. Filkov and M. W. Godfrey, "Exploring the Relationship between Co-changes and Run-Time Dependencies," in Proc. IEEE Int. Conf. Software Maintenance (ICSM), 2004, pp. 408-412.
[13] Veershetty, G. (2023). Risk-adaptive transition and transformation (RATT): A predictive governance framework for SAP cloud migration programs. International Journal of Leading Research Publication, 4(12). https://doi.org/10.70528/IJLRP.v4.i12.2170