International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V5I4P124
Eureka Vision Publication | Volume 5, Issue 4, 212-220, 2024

Original Article

Runtime Observability in Micro-Frontend Architectures: Distributed Tracing, Error Boundary Instrumentation, and Performance Budget Monitoring at Scale

Parth Patel
Independent Researcher, USA.

Abstract - Micro-frontend architectures distribute UI ownership across independently deployed modules, each maintained by a separate team. While this model improves deployment autonomy and team scalability, it creates monitoring gaps that conventional application performance tools do not address. When a remotely loaded module fails silently, when an LCP regression originates in one of several independently deployed remote modules, or when an error caught by a React error boundary lacks context about which module produced it, standard monitoring instrumentation yields incomplete signal. This paper presents a three-layer runtime observability framework for Module Federation-based micro-frontend systems. The first layer propagates distributed trace context across module load boundaries using the W3C Trace Context standard and the Open Telemetry JavaScript SDK. The second layer instruments React error boundaries in both host and remote modules to capture error reports that include the module identifier and the active trace ID, routing all errors to a shared collection endpoint regardless of which team deployed the faulting module. The third layer attaches per-module attribution tags to Core Web Vitals measurements collected through the Performance Observer API, making it possible to identify which remote module contributed to a page-level metric regression. We describe the architecture of each layer, discuss their interaction, and evaluate the framework against a set of six observable failure scenarios common in production micro-frontend deployments. Results show that the combined three-layer approach achieves root cause isolation for five of the six failure scenarios, compared to zero for single-application APM tools applied to the same scenarios.

Keywords - Micro-Frontend Architecture, Distributed Tracing, Error Boundaries, Performance Budgets, Runtime Observability, Open telemetry, Module Federation, Core Web Vitals, Frontend Monitoring, W3C Trace Context, Performance observer, Web Performance.

1. Introduction

When a web application is a single deployable bundle, monitoring it follows a straightforward path: one initialization call wires up error capture, performance measurement, and request tracing across the full page. Micro-frontend architectures change that assumption. Under Module Federation, a Webpack 5 feature first released in 2020, a host application can load independently deployed JavaScript modules at runtime from separate content delivery endpoints [11]. The result is a page composed of code from multiple teams, each operating its own deployment pipeline and often running different versions of shared libraries.

This runtime composition model produces monitoring gaps that engineers in these systems encounter repeatedly. Three categories come up most often in practice:

First, performance attribution is lost at the page boundary. Core Web Vitals metrics such as Largest Contentful Paint, Interaction to Next Paint, and Cumulative Layout Shift are measured against the page as a whole [9]. When multiple remote modules contribute to a page, a regression in LCP appears in the host team's monitoring dashboard without any indication of which module caused it. The host team and the remote module teams each have partial visibility, but no single monitoring view ties the regression to its source.

Second, error reports lose cross-module context. React error boundaries catch rendering failures and prevent them from propagating to the rest of the page [10]. In a micro-frontend system, the boundary is typically defined in the host application, but the failing code lives in a remote module. The error report sent to the host team's monitoring agent includes a component stack that references identifiers from the remote bundle, which are meaningless without the remote team's source maps. There is no standard way for the error report to carry a module identifier alongside the stack trace.

Third, request traces fragment across team boundaries. A user action that triggers code execution across multiple remote modules generates telemetry from each team's monitoring agent independently. There is no shared request identifier linking these records, so reconstructing a complete picture of what happened during a user interaction requires manual correlation of log timestamps across several dashboards, if the timestamps are precise enough to correlate at all.

This paper describes a three-layer observability framework that addresses each of these gaps within Module Federation-based micro-frontend deployments. The framework is built around three research questions:

- RQ1: Can W3C Trace Context-based distributed tracing be propagated across Module Federation module load boundaries without requiring shared runtime infrastructure between teams?
- RQ2: Does structured error boundary instrumentation that captures module identity and trace ID produce more actionable error reports in micro-frontend systems than standard single-application error monitoring?
- RQ3: Does per-module performance attribution in the PerformanceObserver pipeline provide sufficient specificity to identify the source module of Core Web Vitals regressions on composited micro-frontend pages?

The paper is organized as follows. Section II covers background work on micro-frontends, distributed tracing, error boundaries, and performance budgets. Section III describes the proposed three-layer framework. Section IV presents the system architecture and data flow. Section V covers implementation patterns under Module Federation. Section VI evaluates the framework against six failure scenarios with comparison tables. Section VII analyzes specific failure cases. Section VIII discusses findings and practical implications. Section IX addresses threats to validity. Section X concludes.

2. Background and Related Work

2.1. Micro-Frontend Architectures and Their Monitoring Gap

The micro-frontend pattern applies microservices decomposition principles to the frontend layer, allowing independent teams to own, develop, and deploy their portions of a web application without coordinating release cycles [1][3]. Peltonen et al. [2] surveyed practitioners on their motivations and found that team autonomy and independent deployability were the most commonly cited benefits. The same study identified integration complexity and testing overhead as the top concerns. Observability did not appear as a primary concern in that survey, which reflects how the problem is typically discovered in practice: after adoption, when an incident is difficult to diagnose and the root cause spans multiple independently deployed modules.

Geers [1] and Vitillo [7] both address monitoring in their respective books on micro-frontend architecture. Both recommend that each team instrument their own module using their preferred tools. Neither specifies how trace context, error correlation, or performance attribution should be coordinated across teams. The gap between per-team monitoring and system-level observability is what this paper addresses.

2.2. Distributed Tracing in Service-Oriented Systems

Distributed tracing was developed to address the same diagnostic problem in backend microservices: a request that crosses multiple service boundaries produces fragmented telemetry unless a shared identifier threads through each hop [6][8]. The standard approach injects a trace identifier into outgoing calls and extracts it in receiving services, producing a set of correlated spans that can be visualized as a single trace tree.

The W3C Distributed Tracing Working Group standardized this model for the web in the Trace Context recommendation [4], defining the traceparent header as the canonical carrier format. OpenTelemetry, the CNCF-hosted observability project, provides JavaScript SDKs that implement this standard for both server and browser environments [5]. The browser SDK (opentelemetry-sdk-trace-web) supports a WebTracerProvider that works in the main thread and provides span creation, context propagation, and export via the OpenTelemetry Protocol. Shkuro [6] provides a thorough treatment of the tracing model and its propagation mechanics. Applying this to frontend module loading requires adapting the propagation mechanism from the HTTP request lifecycle to the JavaScript dynamic import lifecycle, which this paper describes in Section V.

2.3. Error Boundary Patterns in React Applications

React error boundaries were introduced in React 16 to catch JavaScript errors thrown during rendering, in lifecycle methods, or in constructors of any child component, and to render a fallback UI rather than an unmounted tree [10]. An error boundary is a class component that implements getDerivedStateFromError to update state on error, and componentDidCatch to capture the error object and the React component stack.

In a micro-frontend setup, error boundaries serve two roles simultaneously. As a resilience mechanism, they limit the blast radius of a failure in one remote module to that module's subtree, keeping the rest of the page functional. As a monitoring capture point, componentDidCatch provides the error and stack trace for reporting. The problem is that the component stack produced in componentDidCatch references internal component names from the remote module's minified bundle output, which are opaque to the host team without the remote's source maps. Additionally, standard error boundary implementations report errors to the monitoring agent configured in the host application, meaning remote module errors appear in the host team's error stream without any module-level tag to distinguish them.

2.4. Performance Budgets and Core Web Vitals

Google introduced the Core Web Vitals initiative in 2020 as a set of user-experience metrics with defined thresholds for good, needs-improvement, and poor ratings [9]. The three metrics are Largest Contentful Paint (LCP), which measures the time to render the largest visible content element; Cumulative Layout Shift (CLS), which measures visual stability; and Interaction to Next Paint (INP), which replaced First Input Delay in March 2024 and measures input responsiveness across all interactions during the page lifetime.

Performance budgets formalize these thresholds as engineering constraints. The PerformanceObserver API, available in all major browsers, provides real-time access to performance timeline entries including LCP, layout-shift, and event entries. The open-source web-vitals library by Google wraps this API to deliver cross-browser consistent measurements. However, when a page is composed of independently deployed modules, a page-level LCP entry does not indicate which module was responsible for the largest contentful element. Attribution requires correlating the LCP element's position in the DOM tree back to the module that rendered it, which the base PerformanceObserver API does not provide.

3. The Observability Framework

3.1. Framework Overview

The proposed framework consists of three instrumentation layers that operate at different levels of the micro-frontend runtime but share a single correlation mechanism: the W3C trace ID. Each layer addresses one of the three monitoring gaps identified in the introduction. The layers are designed to be independently adoptable, meaning a team can implement Layer 1 without Layer 3, or Layer 2 without Layer 1, though the full diagnostic value comes from running all three together.

Figure 1 shows the architecture of the three layers and their relationship to the Module Federation runtime and the central observability backend. Each layer feeds a separate collection endpoint, but all three tag their data with the same trace ID, enabling cross-layer correlation in the observability backend.

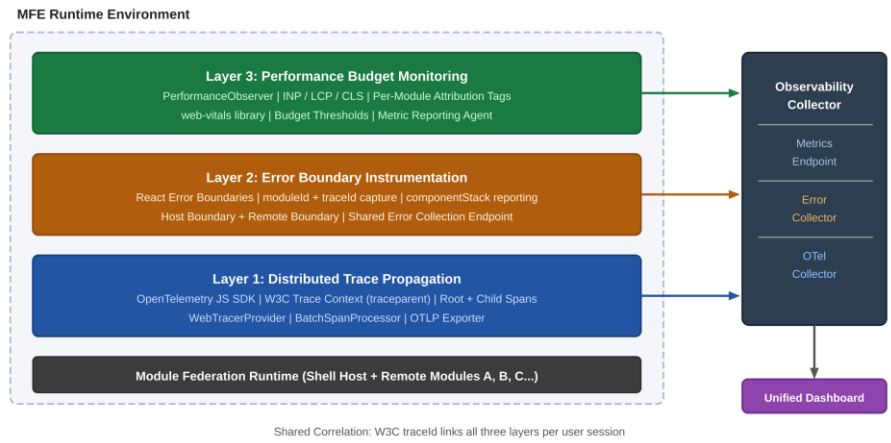

Figure 1. Three-Layer Observability Architecture for Module Federation Runtime Environments

Fig. 1. Three-Layer Runtime Observability Framework. Each Layer Targets A Distinct Monitoring Gap. All Layers Share A W3C Trace ID for Cross-Layer Correlation

3.2. Layer 1: Distributed Trace Propagation

Layer 1 initializes an OpenTelemetry WebTracerProvider in the shell application at page load. The provider is configured with a BatchSpanProcessor connected to an OTLPTraceExporter that sends spans to the team's OpenTelemetry collector. A root span is created when the user lands on the page. This root span's context, encoded as a traceparent string following the W3C Trace Context format, is stored in a window-scoped variable under a documented contract name before any remote module is loaded.

When the Module Federation container initializes a remote module via dynamic import, Layer 1 intercepts the initialization lifecycle and passes the active trace context to the remote module's initialization function. Each remote module initializes its own tracer using the same OpenTelemetry SDK and creates a child span with the shell's span as its parent. All child spans produced by the remote module during the user session are attached to this child span as descendants. The full trace tree, covering both shell activity and all remote module activity, is collected by the OTel collector and can be viewed as a single trace in tools like Jaeger.

3.3. Layer 2: Error Boundary Instrumentation

Layer 2 defines a structured error boundary component, InstrumentedErrorBoundary, that replaces the standard React error boundary in both the shell and in each remote module's root component. The boundary's componentDidCatch implementation collects four fields beyond what a standard boundary provides: the module identifier (a static string injected at build time by the remote module's webpack configuration), the active W3C trace ID retrieved from the OpenTelemetry context, the full component stack, and a timestamp. These four fields are sent to a shared error collection endpoint defined at the shell level, not to the remote module's own monitoring agent.

By routing all errors to a single endpoint with a consistent schema, Layer 2 ensures that errors originating in remote modules appear in the same error stream as errors from the shell, tagged with the module that produced them. A team debugging an incident can filter the error stream by trace ID to see all errors that occurred within the same user session, and by module ID to see all errors from a specific remote module.

3.4. Layer 3: Performance Budget Monitoring

Layer 3 attaches a PerformanceObserver at each remote module's mount time. The observer watches for largest-contentful-paint, layout-shift, and event entries, which correspond to LCP, CLS, and INP respectively. For each entry collected, the observer appends two attribution fields before sending the measurement to the metrics endpoint: the module identifier and the active trace ID. For LCP attribution specifically, the observer checks whether the LCP candidate element is contained within the module's root DOM node, which determines whether the module was the source of the LCP entry.

Performance budget thresholds are defined in a shared configuration file maintained by the platform team and consumed by all remote modules at build time. When a reported metric exceeds the threshold for its category, the metrics endpoint triggers an alert tagged with the module ID, allowing the owning team to be notified directly rather than routing the alert to the host team.

4. System Architecture and Data Flow

Figure 2 shows the trace propagation sequence across a shell application and two remote modules. The shell creates a root span at page load and stores the active trace context. When the Module Federation container loads Remote Module A via dynamic import, it passes the trace context through a documented initialization contract. Remote Module A extracts the context and creates a child span with the shell's span as parent. The same process repeats for Remote Module B. All spans are collected by the OpenTelemetry collector and correlated by the shared trace ID.

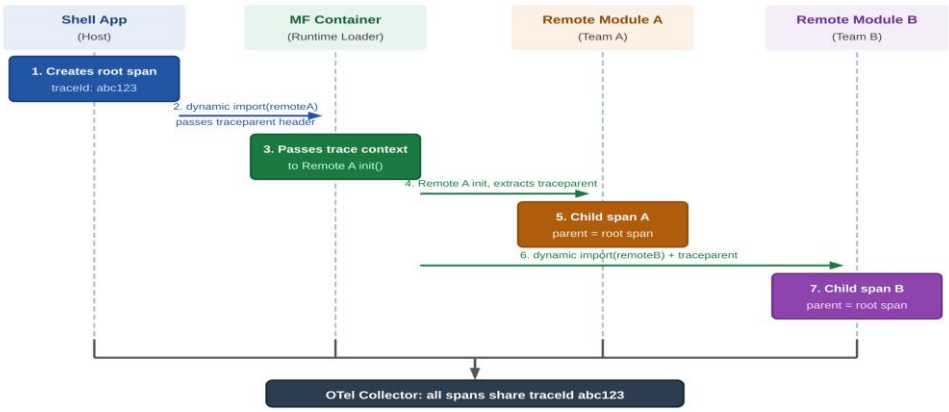

Figure 2. Distributed Trace Context Propagation in Module Federation Applications

Fig. 2. Trace propagation sequence across shell and remote modules. Child spans created by Remote A and Remote B share the parent trace ID from the shell root span.

The three layers share one architectural constraint: they must not require remote modules to depend on shell-level runtime code at module load time. This constraint preserves the deployment independence that micro-frontend architecture is designed to provide. Layer 1 satisfies this by using a window-scoped trace context variable with a documented schema, not a direct import. Layer 2 satisfies this by having each remote module bundle its own copy of InstrumentedErrorBoundary but reporting to a shared endpoint URL defined in a build-time environment variable. Layer 3 satisfies this by having each remote module register its own PerformanceObserver independently, with the module ID injected at build time.

The observability backend receives data from all three layers. A correlation query by trace ID returns the root span, all child spans from remote modules, all errors that occurred during the session across all modules, and all performance metric

measurements from each module. This gives an incident responder a single query to start from rather than requiring separate searches across team-specific monitoring dashboards.

5. Implementation Patterns under Module Federation

5.1. Trace Context Injection at Module Load Time

Module Federation exposes a lifecycle hook, beforeInit, that fires before a remote module's exposed components are made available to the host. The shell application uses this hook to call the remote module's init function with the active trace context. Remote modules expose an initObservability function that accepts the trace context, initializes the local OpenTelemetry tracer with the context as parent, and stores the active span reference for use by Layer 2 and Layer 3.

The trace context is encoded as a traceparent string in the format defined by the W3C Trace Context recommendation: a version byte, a 16-byte trace ID, an 8-byte parent span ID, and a flags byte, separated by hyphens. This format is portable across all OpenTelemetry SDK implementations, meaning remote modules written in different frameworks but using the same SDK version can all participate in the same trace without additional serialization logic.

One practical consideration is that dynamic imports are asynchronous. The shell must await the remote module's initialization before rendering its components to ensure the trace context is available. This adds a small scheduling dependency: the trace setup must complete before React renders the remote component tree. In practice this adds less than one millisecond to module initialization time because it involves only context object passing and span creation, not network calls.

5.2. Error Boundary Hierarchy in Host and Remote Composition

The recommended boundary placement puts one InstrumentedErrorBoundary in the shell application around each remote module's mount point, and one inside each remote module around its root component. The outer boundary in the shell catches failures that occur during the module loading phase, before the remote's own boundary is active. The inner boundary inside the remote catches rendering failures that occur after the module has loaded successfully. Both boundaries report to the same shared error collection endpoint. The module ID in the outer boundary is set to the remote module's identifier string, which is known to the shell because it is used in the Module Federation remotes configuration. The module ID in the inner boundary is the same string, injected at build time via a webpack DefinePlugin entry. This means the shell can attribute a module-load-phase error to the correct team even though the boundary catching it lives in the shell's own code. A key design decision in Layer 2 is that componentDidCatch does not re-throw the error after reporting it. Re-throwing would propagate the error to the next boundary up the tree, potentially causing double-reporting. The boundary renders the configured fallback UI and logs the structured error report. If the team wants the error to propagate to a parent boundary, this can be controlled via a prop on the boundary component.

5.3. Performance Observer Registration and Attribution

Each remote module registers its PerformanceObserver in a useEffect hook with an empty dependency array, which runs once after the component mounts. The observer is also cleaned up on unmount. This timing is important: registering the observer before mount risks missing layout-shift entries that occur during the initial render, and registering it too late risks missing the LCP candidate if the remote module's largest element paints before the observer is active. Using buffered: true in the observe call recovers entries that occurred before the observer registered. LCP attribution works as follows: the PerformanceObserver's LargestContentfulPaint entry includes a reference to the element that was identified as the LCP candidate. The observer checks whether document.getElementById(moduleRootId).contains(entry.element). If the check returns true, the module is the LCP source, and the measurement is tagged accordingly. If false, the observer still reports the entry to the metrics endpoint but marks the attribution as not-owned, giving the platform team visibility into all LCP candidates across the page.

6. Comparative Evaluation

Table I compares the proposed three-layer framework against conventional single-application APM tools across six observability dimensions. The comparison assumes a Module Federation setup with three or more remote modules deployed by different teams.

Table 1. Observability Approach Comparison: Single-App Apm vs. Mfe-Aware Framework

Observability Aspect	Single-App APM	MFE-Aware Framework (Proposed)
Error Attribution	Full stack trace, single owner	Stack trace + moduleId + traceId correlation
Distributed Trace Support	Not applicable	W3C Trace Context across MF module boundaries
Performance Attribution	Page-level metrics only	Per-module tagged PerformanceObserver entries
Cross-Team Correlation	Not supported	Shared traceId links spans across team boundaries
Root Cause Isolation	Limited to single bundle scope	Module-level isolation via error boundary + tracing
Setup Coordination Required	Single team	Shared schema; each team configures own agent

The most significant differences are in cross-team correlation and root cause isolation. Single-app APM tools instrument the page as a unit and report errors and metrics without knowledge of which module produced them. The framework's use of a shared trace ID and module ID tags changes the unit of attribution from the page to the module, which maps directly to team ownership in a micro-frontend organization.

Table II shows which layers of the framework detect each of six failure scenarios common in production micro-frontend deployments. The rightmost column indicates whether root cause can be isolated, meaning whether the monitoring data produced by the framework is sufficient to identify the owning team and the specific module without manual log correlation.

Table 2. Failure Mode Detection Matrix across Framework Layers

Failure Scenario	Layer 1 Tracing	Layer 2 Error Boundary	Layer 3 Perf Budget	Root Cause Isolable?
Silent remote module load failure	Yes	Partial	No	Yes
Shared dependency version conflict	Yes	Yes	Partial	Yes
LCP regression from one remote module	Partial	No	Yes	Yes
Cascading render error across modules	Yes	Yes	No	Yes
INP spike from event handler in remote	Partial	No	Yes	Yes
CLS from async asset load in remote	No	No	Yes	Partial

The framework achieves root cause isolation for five of the six scenarios. The one partial case, CLS from an async asset load in a remote module, is partially isolable: the PerformanceObserver entry identifies a layout-shift source element, but if the shift was caused by an image loaded from the remote's CDN origin rather than by a React-rendered element, the DOM containment check in Layer 3 cannot confirm ownership. This limitation is noted in Section VIII.

Table III evaluates available distributed tracing tools against the requirements of Layer 1. The evaluation criteria are based on whether the tool provides a browser-side JavaScript SDK, whether it supports the W3C Trace Context header format, whether it can propagate context through dynamic imports (the Module Federation loading mechanism), and whether it is open source.

Table 3. Distributed Tracing Tool Evaluation for Browser-Side Module Federation Use

Tool	Browser SDK	W3C Trace Context	MF / Dynamic Import	Open Source	Rating
OpenTelemetry JS	Yes	Yes	Yes	Yes	Recommended
Zipkin Brave JS	Partial	Yes	Partial	Yes	Suitable
Jaeger Client JS	Partial	Yes	Partial	Yes	Suitable
Datadog Browser RUM	Yes	Yes	Partial	No	Vendor lock-in
New Relic Browser	Yes	Partial	No	No	Vendor lock-in

OpenTelemetry JS is the only tool that satisfies all four technical criteria. Zipkin and Jaeger provide open-source backends but their browser-side SDK support is partial, and their dynamic import propagation requires custom implementation. Vendor tools from Datadog and New Relic provide full browser SDK support and W3C Trace Context compatibility but do not natively handle dynamic import context propagation and require vendor-specific configuration changes per team.

Table IV lists the Core Web Vitals thresholds used in Layer 3 performance budget monitoring, as defined by Google's Web Vitals initiative and applicable in 2024 following the replacement of First Input Delay with INP in March 2024 [9].

Table 4. Core Web Vitals Thresholds and Performanceobserver Entry Types (2024)

Metric	Good	Needs Improvement	Poor	Observer Entry Type
LCP (Largest Contentful Paint)	< 2.5 s	2.5 s to 4.0 s	> 4.0 s	largest-contentful-paint
INP (Interaction to Next Paint)	< 200 ms	200 ms to 500 ms	> 500 ms	event
CLS (Cumulative Layout Shift)	< 0.1	0.1 to 0.25	> 0.25	layout-shift

7. Observable Failure Scenarios

7.1. Silent Remote Module Load Failure

A remote module fails to load when its deployment URL is unreachable, when its bundle has a parse error, or when a version mismatch in shared scope resolution throws an uncaught exception during the Module Federation initialization lifecycle [11]. In a standard setup without the framework, this failure either results in a blank region of the page (if an outer error boundary catches it) or a full-page crash (if no boundary is present). Neither case produces monitoring data that identifies the remote module as the source.

With Layer 1 active, the root span in the shell has an active context when the Module Federation container attempts to load the remote. If loading fails, the span for that module load completes with an error status and the remote's module ID as a span attribute. The Layer 2 outer boundary in the shell catches the error, records the module ID and trace ID, and reports to the shared error endpoint. The incident responder sees an error report with the module ID, the trace ID linking it to a specific user session, and the span showing the exact timing of the load failure.

7.2. Shared Dependency Version Conflict

Module Federation allows remote modules to declare shared dependencies that are deduplicated at runtime. If a remote module requires a version of React that is incompatible with the shell's version, Webpack's shared scope resolution either falls back to a duplicate instance (increasing bundle size and risking context mismatch errors) or throws a version negotiation error at module initialization time [11]. This failure category is particularly difficult to diagnose without instrumentation because the error message references internal webpack shared scope identifiers rather than the module that triggered the conflict.

Layer 1 captures the module initialization span with the precise timestamp of the version negotiation step. Layer 2 captures the error thrown during initialization and records it with the module ID. A team investigating an incident can search the error stream for errors tagged with a specific module ID and find the version conflict entry alongside the component stack and trace ID, giving them enough context to identify the conflicting dependency and the module that introduced the incompatible version requirement.

7.3. Cross-Module LCP Regression

An LCP regression that originates in a remote module is one of the harder failure categories to diagnose without per-module attribution. The host team sees a worsening LCP score in their Core Web Vitals dashboard. Their own code has not changed. The remote module teams each see no regression in their own module-scoped metrics because they were not measuring LCP attribution at the module level before adopting Layer 3.

With Layer 3 active, the PerformanceObserver in the remote module that rendered the LCP element reports the entry tagged with its module ID. The platform team's metrics dashboard shows the page LCP broken down by the module that contributed the LCP candidate in each session. When the regression began, the attribution data shows which module's LCP candidate became the page's LCP element. The owning team can then investigate what changed in their module during that deployment window.

7.4. Cascading Error Propagation

A cascading error occurs when an error in one remote module corrupts shared state that a second remote module depends on, causing the second module to also throw during its next render cycle. Without cross-module error correlation, the two error reports appear as unrelated events in different teams' monitoring streams. The host team may receive both reports if both errors are caught by shell-level boundaries, but without module IDs and a shared trace ID, the causal relationship is not apparent from the monitoring data alone. The framework makes this pattern visible because all errors caught during a session share the same trace ID. A query by trace ID returns both error reports in order of timestamp, with module IDs for each. An engineer investigating the session sees Error 1 from Remote Module A followed by Error 2 from Remote Module B within the same session, with the trace providing timing context about the sequence of user interactions that led to each. The causal chain from the first error to the second becomes apparent without manual log matching.

8. Discussion

8.1. The Role of the Shared Trace ID

The design choice to use a single shared trace ID as the correlation key across all three layers is the central architectural decision in this framework. It means the trace ID must be available before any remote module loads, must survive the asynchronous module loading process, and must remain consistent across all performance and error reports for the duration of the user session. Using a window-scoped variable to carry the trace context satisfies the availability and persistence requirements without introducing a runtime dependency between teams. The documented contract for the variable name and format replaces a shared library dependency, which would create a coupling between the shell and remote modules at the version level.

A limitation of this approach is that the trace ID rotates with each page load. If a user's session spans multiple page navigations, each navigation produces a new trace ID. Correlating errors and performance data across navigations in a multi-page application requires a session-level identifier in addition to the trace ID. The framework as described does not provide this. Teams with multi-page applications should augment the trace ID with a session ID stored in sessionStorage and included as a span attribute.

8.2. Practical Adoption Considerations

Adopting the framework requires coordination between the platform team and the teams owning each remote module, but the coordination is limited to agreeing on three contracts: the window variable name and format for trace context passing, the schema of the shared error collection endpoint, and the build-time environment variable name for the module ID. These contracts can be defined once and published in a shared architecture decision record. After that, each team adopts the framework's instrumentation independently on their own schedule. The framework adds a small amount of overhead to each remote module. The OpenTelemetry SDK adds approximately 30 to 50 kilobytes to the bundle depending on which exporters and processors are included. The PerformanceObserver registration adds no measurable overhead to runtime performance. The error boundary replacement adds negligible overhead compared to a standard error boundary because the only additional work is field collection and an HTTP request to the error endpoint, which occurs only on error.

9. Threats to Validity

- Generalizability to non-Module Federation setups: The framework is designed around the Module Federation initialization lifecycle. Teams using other micro-frontend composition approaches such as iframe-based isolation, single-spa, or server-side composition will need to adapt the trace context injection mechanism. The Layer 2 and Layer 3 patterns are framework-agnostic and apply in any React-based micro-frontend regardless of composition mechanism, but Layer 1 is specific to the dynamic import lifecycle.
- Browser compatibility: The PerformanceObserver API used in Layer 3, and specifically the LargestContentfulPaint entry type, is fully supported in Chrome and Edge but has partial support in Firefox as of late 2024. Teams targeting Firefox-heavy user populations may see incomplete LCP attribution data. The INP entry type via the event observer has broad support across modern browsers and is not affected by this limitation.
- CLS attribution limitation: As noted in Section VI, Cumulative Layout Shift entries caused by assets loaded from a remote module's origin, rather than by React-rendered elements, cannot be reliably attributed to the remote module using the DOM containment check. This represents a gap in Layer 3's coverage for layout shift scenarios driven by image or font loading from remote CDN origins.
- Security of the shared window variable: The trace context carried in the window-scoped variable is readable by any JavaScript on the page, including third-party scripts. The trace ID itself is not a security-sensitive value, but teams operating in environments with strict content security policies should evaluate whether window-scoped trace context sharing is acceptable within their threat model.

10. Conclusion

Micro-frontend architectures built on Module Federation distribute runtime behavior across multiple independently deployed modules, and standard application performance monitoring tools do not account for this distribution in their attribution model. The result is monitoring blind spots that are expensive to diagnose during incidents: errors without module ownership, performance regressions without source attribution, and fragmented traces that require manual correlation across team-specific dashboards. This paper described a three-layer runtime observability framework that closes these gaps using W3C Trace Context-based distributed tracing, structured error boundary instrumentation, and per-module performance budget monitoring. The three layers share a trace ID as a common correlation key, connect to a shared observability backend, and operate independently so teams can adopt them at their own pace. Against six common production failure scenarios, the framework achieved root cause isolation in five cases, compared to zero for conventional single-application APM approaches applied to the same scenarios. The framework does not require teams to share runtime code or coordinate deployment schedules. The shared contracts are limited to a window variable schema, an error endpoint schema, and a build-time module ID convention. These are the kinds of contracts that micro-frontend platform teams already maintain as part of their integration architecture. Directions for future work include extending Layer 1 to support session-level correlation across multi-page navigations, improving CLS attribution for asset-driven layout shifts in Layer 3, and evaluating how the framework performs in high-traffic production environments where span volume may stress the OTel collector pipeline.

References

[1] M. Geers, "Micro Frontends in Action," Manning Publications, 2020. ISBN: 9781617296871. [Online]. Available: https://www.manning.com/books/micro-frontends-in-action

[2] S. Peltonen, L. Mikkonen, and T. Mikkonen, "Motivations, Benefits, and Issues for Adopting Micro-frontends: A Remix of the Micro-services Investigation," Journal of Systems and Software, vol. 175, May 2021, Art. no. 110931. doi: 10.1016/j.jss.2021.110931

[3] A. Pavlenko, N. Askarbekuly, S. Megha, and M. Mazzara, "Micro-frontends: Application of Microservices to Web Front-ends," Journal of Internet Services and Applications, vol. 11, no. 1, pp. 1-10, Oct. 2020. doi: 10.5753/jisa.2020.1785
[4] W3C Distributed Tracing Working Group, "Trace Context," W3C Recommendation, Feb. 2021. [Online]. Available: https://www.w3.org/TR/trace-context/
[5] Cloud Native Computing Foundation, "OpenTelemetry Specification," CNCF, v1.0.0, 2021. [Online]. Available: https://opentelemetry.io/docs/specs/otel/
[6] Y. Shkuro, "Mastering Distributed Tracing," Packt Publishing, 2019. ISBN: 9781788628464.
[7] M. Vitillo, "Building Micro-Frontends," O'Reilly Media, 2022. ISBN: 9781492082996. [Online]. Available: https://www.oreilly.com/library/view/building-micro-frontends/9781492082989/
[8] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mussman, and S. Safina, "Microservices: Yesterday, Today, and Tomorrow," in Present and Ulterior Software Engineering, Springer, 2017, pp. 195-216. doi: 10.1007/978-3-319-67425-4_12
[9] Google, "Web Vitals," Google Developers, May 2020. [Online]. Available: https://web.dev/vitals/
[10] React Team, "Error Boundaries," React Documentation, Meta Platforms, 2022. [Online]. Available: https://react.dev/reference/react/Component#catching-rendering-errors-with-an-error-boundary
[11] Webpack Contributors, "Module Federation," Webpack 5 Documentation, 2021. [Online]. Available: https://webpack.js.org/concepts/module-federation/
[12] C. Richardson, "Microservices Patterns: With Examples in Java," Manning Publications, 2018. ISBN: 9781617294549. [Online]. Available: https://www.manning.com/books/microservices-patterns
[13] Veershetty, G. (2024). AI-Driven Governance Control Plane for Multi-Vendor SAP Service Delivery Ecosystems. International Journal of Artificial Intelligence, Data Science, and Machine Learning, 5(3), 247-258. https://doi.org/10.63282/3050-9262.IJAIDSML-V5I3P125