International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(23s) 2026 pp. 1664-1677
https://doi.org/10.70917/ijcisim-2026-5719
Article

A Performance Engineering Framework for Large-Scale Customer-Facing Web Applications: Optimizing Core Web Vitals in Micro-Frontend Architectures

Parth Patel

Independent Researcher, India
Email: parth.uiengineer@gmail.com

Abstract: Large customer-facing web applications get slow through accumulation rather than through a single bad decision, and micro-frontend architecture makes that accumulation structural rather than incidental. The architecture's defining virtue is that independent teams ship without coordinating, and its unavoidable consequence is that page performance stops having an owner: each team measures its own fragment, every fragment looks acceptable, and nobody measures the composite the customer actually loads. Duplicate dependencies arrive because no team's dependency graph contains any other's; independent data fetches serialise because sequencing emerges from mount order rather than from a decision; components initialise off-screen because their owning team cannot see what is above the fold in the assembled page. Every decision is locally defensible and the sum is not. This paper proposes a framework whose contribution is governance rather than technique. A page-level interaction budget is fixed from the business requirement and then divided among fragment owners as an allocation they draw against, so that one team needing more must obtain it from another and the trade becomes negotiated rather than invisible. Enforcement runs against the assembled page in continuous integration rather than against fragments in isolation. On a production billing page in a large consumer telecom application, applying these techniques reduced full interactive load from approximately twenty-one seconds to approximately eight, a reduction of roughly twelve to thirteen seconds. That figure is measured and shipped. The paper is explicit about which of its other figures are projections rather than results.

Keywords: web performance optimization, micro-frontend architecture, performance budgets, time to interactive, Core Web Vitals, interaction to next paint, cumulative layout shift, frontend architecture, performance governance, critical path, module federation, self-service digital channels

1. INTRODUCTION

1.1 Slowness is not what users report

Slowness rarely announces itself as a page that fails to load. It appears as a screen that looks finished and is not: a layout that has painted while the controls underneath it are still inert. The user taps, nothing happens, so they tap again. Or the page shifts under their thumb as a late component claims its space, and the tap meant for one control lands on another.

That gap between visually ready and actually usable is where trust breaks. Users do not diagnose it as latency. They conclude the application is broken, or that they did something wrong.

1.2 What it costs, and why the cost is invisible

The immediate cost is abandonment. On a task the user chose to do, such as browsing or comparing, they leave.

On a customer-facing application the more expensive failure is different, because the user frequently cannot leave. Someone paying a bill or changing a plan has to finish. So instead of abandoning, they call.

That is the conversion of a frontend problem into an operational one. Every user who gives up on a slow self-service flow and picks up the phone moves a transaction from a channel that costs almost nothing to one that costs real money per contact. The call centre sees volume with no clear cause. The engineering team sees metrics that look acceptable in aggregate. Neither team is looking at the number that connects them, and there is usually no number that does.

There is a compounding effect. A user defeated once by a slow flow often skips the web entirely next time. The cost is not a single call; it is a permanently more expensive customer.

1.3 What micro-frontends add

Micro-frontends solve an organisational problem genuinely well. Independent teams ship independently, without coordinating releases or contending over a shared codebase.

The trade is that page performance stops having an owner. Each team measures its own fragment and each fragment looks fine. What no one measures is the composite. Three teams each shipping a reasonable bundle produce an unreasonable page. Duplicate copies of the same library arrive because no team's dependency graph includes anyone else's. Independent data fetches fire against the same backend, each individually justified. Components not visible on first paint initialise anyway, because their owning team has no visibility into what is above the fold in the assembled page.

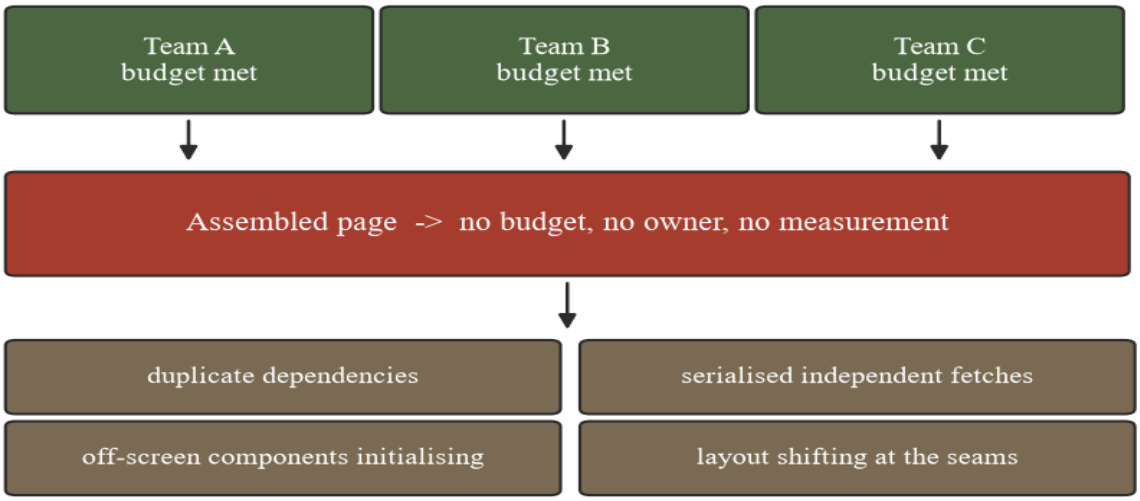

Figure 1. Why the composite goes unowned. Each fragment meets its own budget and the assembled page meets none, because no budget exists at that level.

Nobody is behaving badly. The architecture's core virtue, that teams need not know about each other, is also what guarantees the composite goes unowned, because the system has no place where total cost is accounted for.

1.4 Contribution and scope

This is shipped production work rather than a proposed framework. The author has worked on performance optimisation continuously since moving into a technical lead role in 2019, across essentially every page of a large

consumer telecom application. The clearest single case, and the one this paper uses throughout, is a billing page where the work landed in the second quarter of 2021.

The framework generalises what was actually done. The pattern came out of the work rather than preceding it.

The contribution is three shifts, none of which is a new technique:

1. The budget is allocated at the composite rather than asserted per fragment. A page-level interaction budget is fixed first, then divided among fragment owners as an allocation they draw against.

2. Enforcement runs against the assembled page in continuous integration, not against fragments in isolation. A fragment passing its own gate is necessary and not sufficient.

3. The governing metric is time to interactive on the assembled page, tied to a downstream business measure, rather than a synthetic score no one outside engineering can interpret.

What this paper does not do. It reports one strong measured result and is explicit about the rest. Section 9 separates the measured figures from the projected ones, and Section 10 records what was not instrumented.

2. Background and Related Work

Current practice falls into four categories, all of them sound and none of them sufficient at composite scale.

A note on what the user actually experiences. The metrics in common use measure the page. The failure they are proxies for is a person tapping a control that does not respond, and the two are not the same thing. A page can post acceptable numbers while producing the specific experience described in Section 1.1, because a metric averaged across a session does not capture the moment the user first tried to act. Every measurement discussed below inherits that limitation, which is why Section 3 fixes the governing metric as interaction time on the assembled page rather than a composite score.

Measurement and monitoring. Synthetic auditing and real-user monitoring feed Core Web Vitals [1], [2]. These establish that a page is slow and roughly where, after it has already shipped.

Build-time and delivery optimisation. Code splitting, tree shaking, bundle analysis, delivery network and caching strategy, image optimisation, and compression. This area is mature, well-tooled, and largely automatable.

Runtime technique. Lazy loading, resource hints for prefetch and preload [3], virtualisation, memoisation, request batching, and field-selective query languages that let a client request only what it consumes [4].

Governance. Performance budgets asserting a ceiling on bundle size or a metric threshold, enforced in continuous integration and failing the build when exceeded.

The techniques are sound. What breaks at scale is the assumption underneath them: that some team owns the page.

Budgets in their standard form are enforced per artifact. A team asserts that its bundle stays under a threshold and the pipeline holds the line. Under a monolithic frontend the sum of the artifacts is the page, so per-artifact enforcement approximates composite enforcement. Under micro-frontends it does not. Ten fragments each passing their own budget produce a composite that passes no budget at all, because no budget exists at that level.

The same holds for measurement. Each team's real-user data is scoped to its fragment. Aggregate dashboards report averages across an application no single user ever experiences as an average. The composite page, which is the artifact the customer actually loads, is measured by no one and owned by no one.

The gap this paper addresses is therefore not technical. Every technique it applies is established. What is missing from practice is a place where the composite is owned, budgeted, and gated, and the architecture guarantees that no team will provide one by default.

3. Layer 1: The Composite Budget and Its Allocation

Before any technique is applied, the page is assigned a single interaction budget expressed as ninetieth-percentile user-interactive time.

The budget is set from the business requirement, not from current performance. This is the load-bearing detail. A budget derived from what the page does today ratifies whatever it does today, and the exercise becomes documentation rather than governance.

That budget is then divided among fragment owners as an allocation. Each owner receives a share of transferred bytes, main-thread time, and critical-path requests. The allocation is the unit of governance for everything that follows: a team wanting more must obtain it from another team, which makes the trade a negotiation rather than an invisible externality

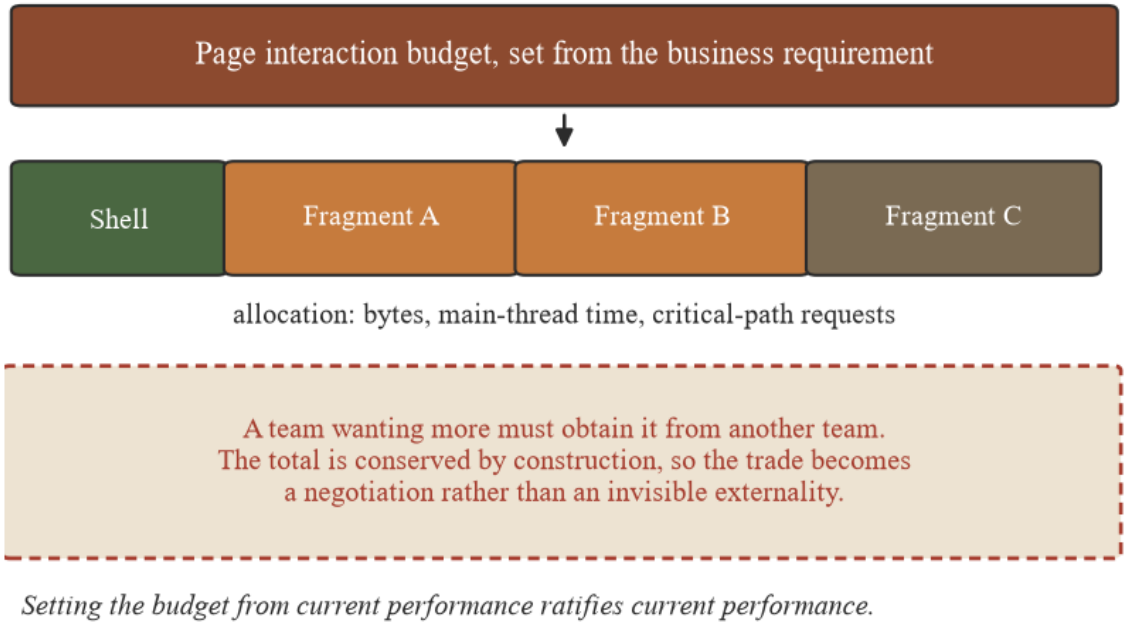

Figure 2. Budget allocation at the composite. The total is conserved by construction, so a team needing more must obtain it from another.

This is the framework's central move, and it is worth being precise about what changes. Under per-fragment budgets, a team that needs more bytes takes them, and the cost lands on the composite where no one is watching. Under allocation, the same team must find another team willing to give up headroom. The total is conserved by construction, and the conversation that was previously not happening now has to happen.

3.1 Why the allocation has to be denominated in three quantities

An allocation expressed only in transferred bytes is the common form and it is insufficient, because bytes are the cheapest of the three constraints to satisfy and the least predictive of interaction time. A fragment can stay well inside a byte budget and still monopolise the main thread, and it can make few requests that each sit on the critical path.

The three quantities are therefore allocated together: transferred bytes, main-thread time, and critical-path requests. A team spending its main-thread allocation has spent it regardless of how few bytes it shipped, and this is

what prevents the most common form of budget gaming, which is compressing harder and doing the same amount of work on arrival.

Main-thread time is the hardest of the three to attribute and the most important to allocate, for the reason developed in Section 7.1: it is the one genuinely shared resource, and a fragment's consumption of it is felt by every other fragment rather than by itself.

4. Layer 2: Delivery and Bundle Optimisation

Standard practice, applied per fragment and audited at the composite.

Code splitting along route and interaction boundaries, so a fragment ships only what its initial view requires.

Tree shaking and dependency deduplication, audited across fragments rather than within them. This qualification is the whole point: duplicate copies of a shared library are invisible to any single team's bundle analysis, because each team's analysis is correct within its own boundary.

Module federation or an equivalent shared-dependency mechanism, so common runtime libraries resolve to one instance.

Compression, delivery network placement, and cache policy, including deliberate review of time-to-live settings. In the billing case a cache window raised from ten minutes to one hour materially improved hit rate on assets that changed far less often than they were being revalidated. This is the least sophisticated item in the framework and one of the more effective, which is a recurring pattern in performance work.

5. Layer 3: Critical Path Sequencing

This is where most of the recovered time lives, and where fragment-local reasoning fails hardest.

Parallelise what is independent. The dominant failure pattern in the billing analysis was assets and interfaces loading in series that had no data dependency on one another. Third-party component assets were downloading after the host's primary data call rather than alongside it. Moving those assets into parallel with the billing component and its data call was worth several seconds on its own.

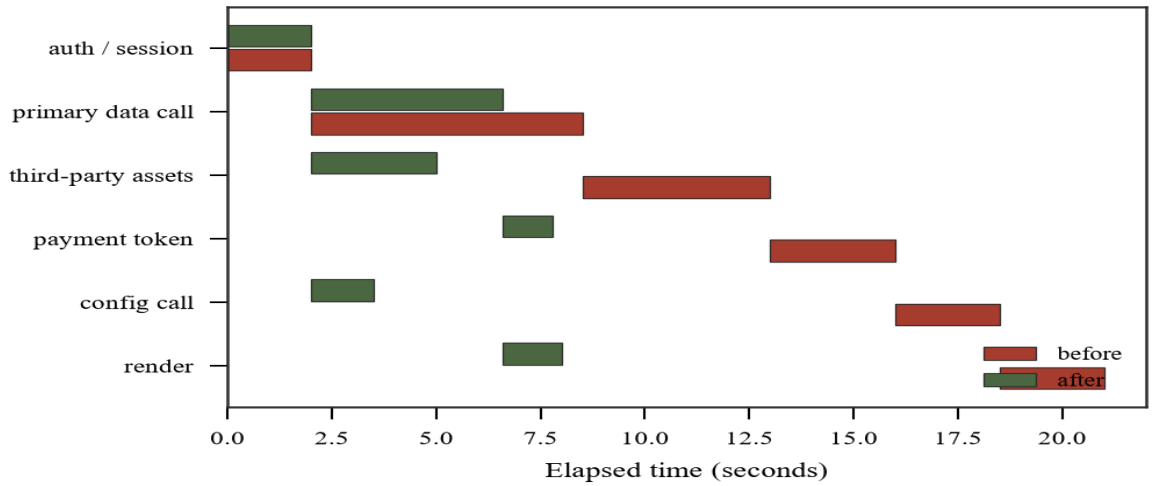

Figure 3. Critical path before and after sequencing work. Independent requests that ran in series had no data dependency on one another. Illustrative of the pattern found in the billing analysis.

Defer what is not render-blocking. A call whose result affects nothing on first paint is loaded asynchronously. The canonical case in the billing page was a token retrieval for a payment provider: it gates a payment option, not the page, and it was sitting on the critical path.

Prefetch across the flow, not within the page. If a user reaching billing arrived from a known prior step, that step is where billing's assets should be fetched. This is the one technique that requires cross-fragment cooperation by construction, because the fragment doing the prefetching is not the one that benefits. No per-fragment incentive produces it.

Eliminate redundancy. The billing analysis surfaced the same payment token call issued twice, a duplicated payment configuration call, and a redundant upsert against a shared service. None of these is a defect in any team's code. They are the same request issued by two owners who cannot see each other.

Reduce payload at the source. Where the interface consumes a fraction of what an endpoint returns, moving to a field-selective query [4] cuts transfer and parse cost. This requires backend cooperation, and performance work that stops at the frontend boundary leaves it on the table. In the billing case the migration from a full-payload endpoint to a field-selective one was one of the four changes that produced the result in Section 9.

6. Layer 4: Enforcement

Techniques decay without a gate. Three mechanisms hold the line.

The composite continuous-integration gate. The assembled page, not fragments in isolation, is built and measured pre-merge against the current production versions of its peers. A fragment passing its own budget while pushing the composite over the page budget fails the merge. This is the mechanism that distinguishes the framework from standard budget practice, and it is the one most likely to be resisted, because it makes a team's merge depend on a build they do not control.

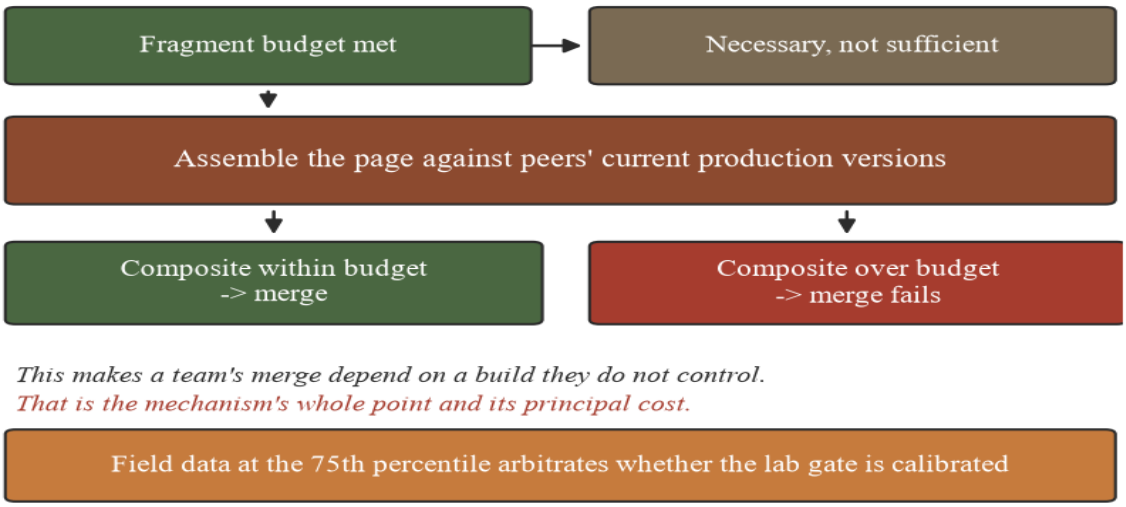

Figure 4. The composite gate. A fragment passing its own budget is necessary and not sufficient, which is what distinguishes this from standard budget practice.

Per-component render budgets at runtime. Individual components are given render-time ceilings enforced in development, so a regression is caught by the team that introduced it rather than surfacing weeks later in aggregate field data. The author has built a working implementation of this against the rendering framework's profiler interface.

Field data as the arbiter. Lab gates block merges; seventy-fifth-percentile field data over a twenty-eight-day window determines whether the lab thresholds are calibrated correctly. When the two disagree, the lab configuration is what is wrong.

7. What Is Specifically Hard About Micro-Frontends

Six problems exist here that do not exist in a single application. Each is a direct consequence of teams not needing to know about each other.

7.1 The shared runtime is a contended resource with no owner

A monolithic application has one main thread and one team reasoning about it. Micro-frontends have one main thread and many teams, none of whom can see the others' work.

This matters most for responsiveness. Interaction latency is a composite property by construction: a long task from one fragment delays an interaction in another, and the second fragment's team sees the regression in their metrics while having no access to its cause. Each team profiles its own fragment in isolation, where it performs fine. The contention exists only in the assembled page, which is the one configuration nobody profiles.

Handling. Long-task attribution instrumented at the composite and tagged by originating fragment, so the regression lands on the team that caused it rather than the team that observed it. Per-component render budgets catch the ceiling breach at authoring time. Beyond that, a main-thread yielding discipline applied as a shared convention rather than a per-team preference, since one team's unbroken task defeats every other team's yielding.

7.2 Dependency duplication is invisible to every participant

In a monolith, two copies of the same library is a bundler warning. Under independent deployment it is the default outcome: each team's dependency graph is complete and correct within its own boundary, and none of them contains the others.

The cost compounds three ways. Bytes, in the form of multiple copies of a date library or icon set. Runtime, in the form of multiple framework instances, which breaks context across fragment boundaries and forces defensive workarounds costing more than the duplication. And version skew, where one fragment ships against one major version of a shared design system and another ships against the next, so the shared component renders differently depending on which fragment mounted it first.

Handling. Module federation with explicitly shared singletons for the runtime-critical set: framework, router, state library, design system. A version policy with a supported range rather than a pin, so teams retain upgrade autonomy within bounds. A composite-level dependency audit in continuous integration that fails on unexpected duplication of anything in the shared set.

The honest trade. Shared singletons reintroduce exactly the coupling micro-frontends were adopted to remove. Two teams now cannot upgrade the framework independently. This is a real cost, and the framework's position is that it is worth paying for the runtime-critical set and not worth paying for anything else. It should be stated as a deliberate concession rather than presented as free.

7.3 Independent release cadence leaves the composite without a baseline

A monolith has one deploy, one baseline, one before and after. With ten fragments deploying independently, the assembled page changes several times a day and never in a way any single team initiated.

Three consequences follow. Regressions become unattributable after the fact, because between the good measurement and the bad one, several teams shipped. Field data compounds this: a twenty-eight-day window contains dozens of deploys, so by the time a metric moves the cause is buried. And there is no natural gate, because each team's pipeline validates their fragment against their own build, which is the only configuration in which their change is guaranteed to look fine.

Handling. The composite gate is the load-bearing mechanism here. Paired with it, continuous synthetic measurement of the composite on a fixed cadence independent of any team's deploy, so regressions are timestamped against a deploy log and attribution becomes mechanical rather than forensic.

7.4 Data fetching duplicates across boundaries by construction

This is the failure mode with the clearest evidence in the billing analysis. Two fragments needing the same data will each request it, because neither knows the other exists.

Worse than duplication is serialisation. Independent requests execute in series because the sequencing emerged from mount order rather than from anyone's decision. In the billing case, third-party component assets downloaded after the host's primary data call despite having no dependency on it: several seconds of pure sequencing cost, invisible to both owners.

Handling. A shared request layer with in-flight deduplication and a normalised cache, so an identical request from a second fragment resolves against the first rather than issuing. A declared data-dependency graph per fragment, which makes the independence explicit enough to parallelise. And critical-path review at the composite, since the sequencing problem is visible only in the assembled waterfall. Waterfall analysis of the composite, rather than per-fragment profiling, is the diagnostic of record.

7.5 Third-party dependencies multiply per team

Each fragment owner independently evaluates and adopts vendors: analytics, payment providers, tag managers, experiment platforms. Each adoption is locally reasonable. The composite accumulates a set no one approved, executing on the shared main thread, often loaded synchronously by vendor snippets that assume they are the only script on the page.

The billing page showed this directly: an external provider's assets and initialisation sat on the critical path of a page whose owning team did not choose that provider.

Handling. Third-party additions consume the adopting team's byte and main-thread allocation, which prices the decision at the point it is made. A vendor registry at the composite so duplicate-purpose vendors surface before adoption rather than after. Facade loading for anything not required for first interaction.

7.6 Layout stability degrades at the seams

Fragments hydrate independently and on their own timelines. Each reserves space correctly within itself; the composite shifts as fragments arrive in nondeterministic order. Layout shift is a page-level metric measuring a page-level property that no fragment owner can control alone.

Handling. The shell reserves fixed dimensions for every fragment slot before any fragment loads, making arrival order irrelevant to layout. Slot dimensions are part of the composite contract rather than a fragment's internal decision.

7.7 What the Techniques Cost to Keep

Techniques are applied once. Governance is what determines whether the improvement survives, and the survival problem has a specific shape under independent deployment.

Improvements decay by accretion, not by reversal. No team reverses the optimisation. What happens is that each subsequent change adds a small amount of bytes, main-thread time, or critical-path depth, every addition is individually defensible, and the composite drifts back toward where it started. This is the same mechanism described in Section 1.3 operating in the opposite direction, and it is why the framework's contribution is the allocation and the gate rather than the techniques.

The prefetch technique is the least stable. Section 5 notes that cross-flow prefetching requires the fragment doing the work to be a different fragment from the one that benefits. That arrangement has no owner under normal incentives: the prefetching team carries cost and receives no metric improvement, so the code is a candidate for removal at every subsequent refactor unless the allocation model explicitly credits it. This is the clearest case in the framework where a technique depends on the governance rather than the reverse.

Deferral decays when ownership changes. A call deferred because it does not affect first paint is correct only while that remains true. A later change that makes the deferred result render-blocking reintroduces the cost silently,

and the team making that change has no reason to know the call was deliberately deferred. Recording deferral decisions as part of the fragment's declared data-dependency graph, rather than as an implementation detail, is what makes the assumption visible to whoever changes it next.

Third-party additions accumulate fastest. As Section 7.5 notes, each vendor adoption is locally reasonable. Without allocation, the cost lands on the composite and the adopting team's incentives are entirely satisfied. Pricing the addition against the adopting team's own budget is the only mechanism in the framework that acts at the moment the decision is made rather than after.

The general observation is that every technique in Sections 4 and 5 has a decay path, and every decay path runs through a boundary where the team making a change cannot see the assumption they are invalidating. That is the architecture's defining property restated as a maintenance problem, and it is why enforcement is a layer rather than a step.

8. RESULTS

8.1 The measured result

On the production billing page, full interactive load fell from approximately twenty-one seconds to approximately eight. That is a reduction of roughly twelve to thirteen seconds, or fifty-seven to sixty-two percent.

Four changes produced it, and all are running in production:

1. Lazy-loading components not needed for the initial render.

2. Prefetching markup, styles, and scripts during earlier steps in the user flow, so assets were warm by the time the user reached billing.

3. Migrating the backend call from a full-payload endpoint to a field-selective query, so the interface requested only the fields it consumed.

4. Reshaping some of the data served, in cooperation with the backend team, rather than treating this as a frontend-only problem.

The fourth is worth noting because it is the one most often skipped. Performance work confined to the frontend boundary cannot address a payload whose shape is decided elsewhere.

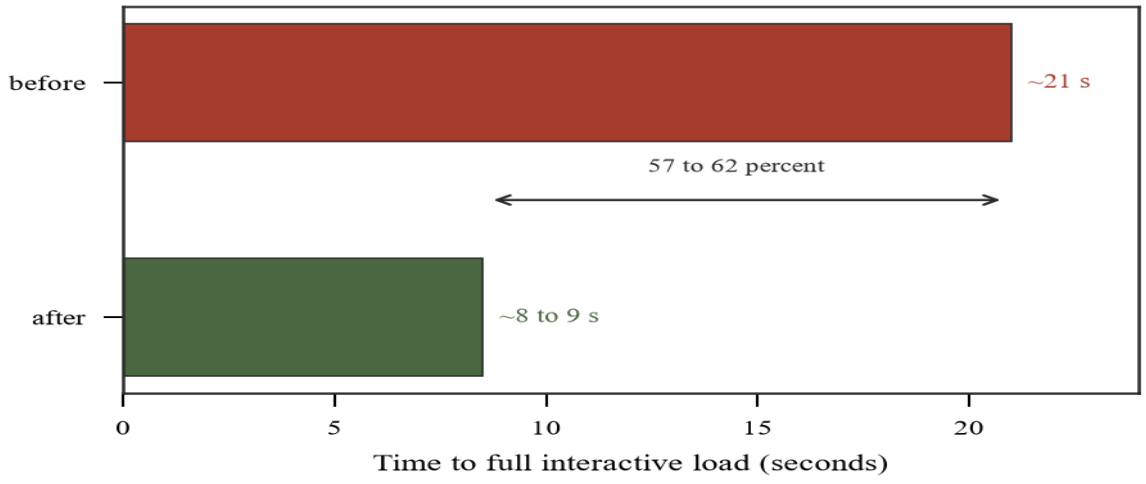

Figure 5. The measured result on the production billing page, second quarter 2021.

8.2 Qualitative outcome

Measurement was based on ninetieth-percentile data supplied by the performance function. The observed customer response was that users engaged with the page and completed payment more often than under the previous experience, where some proportion abandoned the flow. This is reported as an observation rather than as a measured conversion delta, because no controlled measurement of conversion was run.

8.3 What is projected rather than measured

A bundle reduction from approximately two megabytes to approximately eight hundred kilobytes is a projection of what the micro-frontend approach should deliver, not a measured outcome. It is stated here separately from Section 8.1 deliberately, because it is a design expectation and the interactive-load figure is a result. The two should not be cited together as though they had the same status.

Table 1. Evidence status of the reported figures.

Figure	**Status**
Interactive load approximately 21 s to approximately 8 s	**Measured**, shipped to production, second quarter 2021
Reduction of approximately 12 to 13 s, 57 to 62 percent	**Measured**, derived from the above
Cache window raised from 10 minutes to 1 hour improving hit rate	Measured qualitatively as an improvement. Magnitude not reported
Several seconds recovered from parallelising third-party assets	Measured within the billing analysis as a component of the total
Improved payment completion and engagement	Observed. Not a controlled conversion measurement
Bundle approximately 2 MB to approximately 800 KB	**Projection.** Not a measured result
Core Web Vitals before and after by individual metric	Not reported. Measurement was on interactive time at the ninetieth percentile
Call deflection or assisted-channel volume change	Not measured. This is the framework's stated premise and its least evidenced part

9. What This Costs an Organisation to Adopt

The framework is a technical mechanism resting on an organisational precondition, and adopting it has costs that fall on people rather than on systems. Naming them is more useful than presenting the mechanism as free.

Someone must own the composite budget. This is the precondition without which nothing else functions. The owner needs the authority to refuse an allocation increase, which means the role has to sit somewhere that can say no to a delivery team under deadline pressure. Where fragment teams report into different organisations with different priorities, this authority frequently does not exist anywhere, and the gate degrades into a formality with a standing exemption list.

Build times increase for everyone. Every fragment's pipeline now assembles and measures the full page. That cost is paid on every merge by every team, including teams whose changes could not possibly affect the composite. Trading a slower pipeline for a protected composite is the right trade and it is a trade, and a team feeling it daily will ask whether it is.

An arbitration path has to exist. When a team legitimately needs more allocation, someone must decide whether they get it and from whom. That decision is commercial and political rather than technical, and there is no version of the allocation model that avoids it, because the whole point is that the total is conserved.

Blocked teams cannot always unblock themselves. A fragment team failing the composite gate because of another team's regression has neither the access nor the context to fix it. The framework converts an invisible

externality into a visible dependency, which is an improvement, and the improvement is felt as an obstruction by whoever is blocked.

Shared singletons remove upgrade autonomy. As Section 7.2 states, two teams can no longer upgrade the framework independently. This is the concession the architecture was adopted to avoid, accepted deliberately for the runtime-critical set.

Table 2. What the framework costs, and who pays it.

Cost	Who bears it	Avoidable
Composite budget ownership	An individual with cross-team authority	No; the framework does not work without it
Longer build times	Every fragment team, every merge	Partially, by assembling less often at some loss of protection
Allocation arbitration	A decision-maker above the teams	No; a conserved total requires arbitration
Being blocked by another team	The team that did not cause the regression	Partially, through fast attribution and a clear escalation
Reduced upgrade autonomy	Teams sharing the runtime-critical set	Scoped, by keeping the shared set minimal

The pattern across the table is that the framework redistributes costs that were previously paid invisibly by the composite onto teams that can see and object to them. That is the mechanism working as intended, and it is also why adoption is an organisational decision rather than a technical one.

11. Adopting the Framework Incrementally

The framework as described requires an organisational precondition many teams do not have. It can nonetheless be adopted in stages, and the stages have different prerequisites.

Stage one: measure the composite. Continuous synthetic measurement of the assembled page on a fixed cadence, independent of any team's deploy, requires no organisational change and no allocation model. It establishes what the customer actually experiences and provides the number every later argument depends on. Most organisations do not have this and can obtain it in days.

Stage two: attribute regressions. Deploy metadata in real-user monitoring, and long-task attribution tagged by originating fragment. This is still purely technical and it changes the conversation, because a regression stops being a shared mystery and becomes a specific team's change.

Stage three: publish the budget without enforcing it. A page-level interaction budget stated and reported against, with no gate. This surfaces which fragments are consuming what, and it does so before anyone's merge depends on it, which is the point at which resistance would otherwise begin.

Stage four: allocate. Dividing the budget among owners, still without a blocking gate. Teams can now see their share and their consumption.

Stage five: gate. The composite gate blocking merges. This is the stage requiring the organisational precondition, and attempting it before stages one to four have established the data and the norms is what produces the exemption lists described in Section 11.

The ordering matters because each stage makes the next easier to justify. A gate proposed without the measurement behind it is a process imposition; a gate proposed after two quarters of visible composite regression is a response to evidence.

10. LIMITATIONS

One page, one result. The measured outcome is a single page over a single optimisation effort. The billing page was chosen because it was the worst case and because its constraints are the most severe, which makes it a strong demonstration and an unrepresentative sample. Whether the same techniques yield comparable improvement on pages that were never as bad is not established, and the likely answer is that they yield less.

The 90th-percentile framing hides the distribution. Measurement was on ninetieth-percentile interactive time, which is a reasonable governing statistic and reports one point. How the whole distribution moved, and in particular whether the improvement was uniform or concentrated in a slow tail, is not reported and would change how the result should be read.

The floor is set by things the frontend does not control. Backend response time dominates. The waterfall showed individual responses in the three to seven second range. No amount of parallelisation, deferral, or payload reduction moves a number that belongs to the server. Field selection reduces transfer and parse cost but not the time the origin takes to produce a response. Once every independent request is parallel and every deferrable request is deferred, the remaining time is the slowest single dependency, and that dependency is usually not yours.

Third-party components resist the framework almost entirely. A payment provider's initialisation, another vendor's internal call sequencing, and vendor bundle size are outside the codebase. You can control when they load and whether they block; you cannot control what they do once running. Several phases of this work amounted to isolating a dependency rather than improving it, which caps the achievable gain at whatever the vendor's own performance permits.

Some work is irreducible. Authenticated transactional pages have a floor content pages do not. Authentication handshakes, session validation, entitlement and eligibility checks, and fraud and compliance calls must complete before the page is meaningfully usable, cannot be deferred past first interaction, and are often genuinely sequential. A billing page has more of this than almost any other surface. Personalisation is similarly resistant: content varying per customer cannot be cached at the edge or prerendered, so the techniques that most cheaply improve marketing pages are unavailable exactly where business impact is highest.

The governance model has a real coupling cost. The composite gate makes one team's merge depend on a build they do not control. That is the mechanism's whole point and also its principal cost. A fragment team can be blocked by a regression another team introduced, with neither the access nor the context to fix it. Build times increase, since every pipeline now assembles the full page. And the gate creates an escalation path that did not previously exist, because someone must arbitrate when a team legitimately needs more allocation.

Enforcement fails where the organisation is not aligned. None of this works without an owner for the composite budget. Where fragment teams report into different organisations with different priorities, a page-level budget has no one empowered to enforce it, and the gate becomes something teams route around through provisional exemptions and temporary allocation increases that never revert. The framework is a technical mechanism resting on an organisational precondition it cannot itself create.

The framework has not been evaluated against an alternative. The billing result establishes that the techniques applied produced a large improvement. It does not establish that the governance model was necessary to obtain it, because a sufficiently motivated single team could have made the same four changes without a composite budget or a composite gate. What the governance model claims to prevent is regression over time across independent teams, and that claim is about a period longer than the one measured.

Measurement gaps. Field-data latency is structural: a twenty-eight-day window means a regression can be live for weeks before it surfaces. Lab gates catch what lab environments can model, which excludes real device and network diversity. And attribution in a composite remains partly heuristic: long-task attribution to an originating

fragment is reliable, but attributing a field metric shift to a specific fragment deploy, when several teams shipped in the same window, is inference rather than measurement.

The business-metric link is asserted rather than measured. Section 1.2 argues that slow self-service converts into assisted-channel volume, and Section 3 proposes setting the budget from that cost. No instrumentation connecting latency to self-service completion or call volume was in place, so the framework's stated premise is currently its least evidenced part. This is the most significant gap in the paper.

11. Future Work

Automatic fragment attribution for field regressions. The clearest gap. Instrumenting deploy metadata into real-user monitoring so that a seventy-fifth-percentile shift can be mechanically tied to a specific fragment version, rather than reconstructed from a deploy log after the fact.

Budget-aware dependency tooling. A pre-merge check reporting a dependency upgrade's byte and main-thread cost against the team's remaining allocation, making drift visible at the moment it is introduced rather than in a quarterly audit.

Extending render budgets to the composite. The current implementation enforces per-component render ceilings within a single component tree. The natural extension is cross-fragment: budgets declared per fragment and enforced against an assembled page, which would make the composite gate a published tool rather than a described practice. This is the single change that would most strengthen the contribution's reproducibility.

The business-metric link. Instrumenting latency against self-service completion and assisted-channel volume, so the budget can be set from measured cost per second rather than from an engineering judgement. As Section 10 records, this is the framework's premise and its weakest evidence, and closing it would change what kind of claim the framework is able to make.

A note on sequencing. Fragment attribution comes first, because everything else in the list becomes measurable once a field regression can be tied to a deploy. Budget-aware dependency tooling is second because it prevents the drift the attribution would otherwise keep discovering. The composite render-budget extension is third and is the most valuable to publish, since it is what would let another organisation reproduce the mechanism rather than reimplement it from a description.

The business-metric link is listed last and is the most important. It is last because it requires the other three to be running before latency can be correlated with anything reliably, and it is most important because until it exists the page budget is set by engineering judgement presented as a business requirement, which is the one part of the framework that currently asserts more than it can show.

12. CONCLUSION

Micro-frontend architecture is defined by teams not needing to know about each other. Every failure this paper describes is a direct consequence of that property: duplicated calls, serialised independent requests, redundant dependencies, components initialising off-screen, layout shifting at the seams. None of them is anyone's mistake.

The techniques that fix them are conventional and were conventional before this work. What the framework contributes is a place where the composite is owned, budgeted, and gated, because the architecture guarantees no team will own it by default. Fixing the budget at the page and dividing it as an allocation converts an invisible externality into a negotiation. Enforcing against the assembled page rather than against fragments makes a locally passing change fail when it is globally wrong.

The result is real: interactive load on a production billing page fell from roughly twenty-one seconds to roughly eight, and the change is running. The bundle figure that sometimes accompanies this work is a projection rather than a result, and Section 8.3 keeps the two apart deliberately.

What the paper cannot yet show is the link it argues from. The case for setting a page budget from business cost rests on latency converting into assisted-channel volume, and that conversion was never instrumented. Until it is, the framework's premise is an argument rather than a measurement, and the budget is set from an engineering judgement dressed as a business requirement. Closing that gap is the most useful work remaining.

REFERENCES

1. Google, Web Vitals: Essential Metrics for a Healthy Site. [Online]. Available: https://web.dev/vitals/
2. World Wide Web Consortium, Event Timing API and Element Timing Specifications, W3C Web Performance Working Group.
3. World Wide Web Consortium, Resource Hints, W3C Working Draft, Web Performance Working Group.
4. GraphQL Foundation, GraphQL Specification. [Online]. Available: https://spec.graphql.org/
5. World Wide Web Consortium, Layout Instability API, W3C Web Performance Working Group.