
RESEARCH ARTICLE

A Cross-Layer Decision Framework for High-Performance Cross-Platform Front-End Applications

Netra Nandkumar Awate

Northeastern University, Boston, Massachusetts, USA

Corresponding Author: Netra Nandkumar Awate, **E-mail:** netra.awate1@gmail.com

ABSTRACT

Front-end performance often involves cross-cutting decisions where trade-offs must be made across layers of architecture. However, the justification for a trade-off is often made in isolation. The cost of one optimization is pushed to the server, the network, the device, or the monitoring pipeline and is often not measured. The paper identifies when and where performance cost moves between rendering, server-assisted delivery, state, caching and prefetching, observability, and platforms. The paper also asks how to observe cost movement in practice and what measurable signals indicate it. A structured search of peer-reviewed journal and conference literature was conducted, supplemented by backward citation checking for foundational work. Each source was verified against the publisher's bibliographic record and assigned to one of four evidence classes. A cost-transfer tracing exercise, using a hypothetical media-rich interactive commerce application as a recurring illustration, synthesizes five distinct repeated patterns related to pushing costs around layers: rendering pushes costs between the client tier and the server tier, aggregation pushes round trips into the server tier, and prefetching pushes costs into the speculative future. Caching reduces repeated latency but transfers risk to freshness and invalidation; telemetry improves visibility but consumes runtime and network resources. The performance cost of state management matters more concerning the update scope and subscription granularity than the library choice. Cross-platform overhead is dependent on metric, engine, and execution model, not framework name. The strongest evidence (learned preloading short video feeds) shows wasted bandwidth is a first-class metric alongside quality of experience. From the evidence gathered, a decision framework is developed that relates performance goals to measurable signals, architectural levers, and the cost each lever would most likely transfer. Consequently, it is believed that front-end performance is a conditional, cross-layer property of the whole system, and an improvement can be trusted only when the layer that pays the cost is likewise measured.

KEYWORDS

Cross-layer performance; front-end architecture; server-side rendering; prefetching; observability; cross-platform applications

ARTICLE INFORMATION

ACCEPTED: 19 September 2026

PUBLISHED: 21 October 2026

DOI: 10.32996/jcsts.2026.8.11.2

1. Introduction

1.1 Performance as a Cross-Layer Property

Modern front-end applications have evolved from document-oriented pages into distributed, stateful software systems that operate across heterogeneous runtimes. The same product runs in the user's desktop browser, a mobile browser, an embedded webview, a hybrid shell, and a cross-platform runtime for desktop and mobile apps, with their own constraints. The prediction of website performance is itself difficult. A systematic review of website performance prediction found no agreement on an evaluation procedure [1]. Subjective experiments add to this complication, as Core Web Vitals were found to not predict the perceived web quality of experience as well as expected, with the page loading time being the dominant influence [2]. Therefore, no single score stands in for what users experience.

Things get harder architecturally. While rendering raises the question of where to perform the computation that generates the display, on the server there is the added complexity of where the data is coming from, where it is being grouped, and how it is being shaped. State management would control what that change causes. This could include caching or prefetching, or even skipping work, or rescheduling it. Observability may determine whether the engineers may directly view production results, but an

optimization that improves one layer can increase cost elsewhere in the system. For instance, server-rendered pages may lower client time to interactive but increase time to first byte. An aggregation tier may reduce bandwidth but increase server load. Prefetching may reduce inter-page latency at the cost of bandwidth to other pages. While this approach provides dense telemetry that may expose device-specific regressions in a given change, it does involve some additional runtime overhead.

1.2 Application Context

These trade-offs become particularly visible in media-rich, high-interaction workloads. To make the cross-layer trade-offs concrete, this review uses a hypothetical media-rich interactive commerce application as a recurring illustration: a content feed that must load fast on mid-range phones, prefetch upcoming items before the user navigates away, guarantee product and price freshness, and report its health dynamically under traffic spikes. Editing-oriented tooling within such an application may carry heavy interactive state, many server round trips, and a requirement for low-latency interactivity. The output of a recommendation system can directly affect what item the user sees next, and the confidence of that ranking is treated here as an input to a prefetch policy rather than as an implementation detail of the ranking service itself. As a result, computing all layers at once is problematic for any account of front-end latency. The scenario is used only as an illustrative context; the synthesis and conclusions presented in this review are derived from the broader evidence base, not from this scenario, and the same illustrative approach recurs in Section 5.3.

The scenario is illustrative rather than a case study of any specific organization; no proprietary production data, employer-specific architecture, internal measurements, or confidential implementation details are used.

1.3 Research Questions and Contributions

While each layer is evidenced, this evidence is spread across web systems, software engineering, multimedia, mobile computing, and distributed systems research. The three following research questions are answered in Section 4 and Section 5 by combining evidence gathered.

RQ1. When optimizing a single front-end layer, where does the performance cost move?

RQ2. What signals are observable when the cost is passed on instead of removed?

RQ3. What performance principles hold across browser, webview, hybrid, and cross-platform implementations, and what do not?

The review answers these questions and contributes in four ways to the front-end performance literature.

(i) An evidence synthesis of performance trade-offs in rendering, server-assisted content delivery, state management, caching and prefetching, observability, and cross-platform code execution.

(ii) A cost-transfer model that shifts the performance critical path from the API data path through rendering and client state to production telemetry.

(iii) A cross-platform interpretation, dividing principles of transferable resources from behaviors dependent on a particular framework.

(iv) Decision-making framework with links between performance objectives, measurable metrics, architectural levers, the likely cost of each lever, and signals they send.

The most novel takeaway is the cross-layer synthesis, as rendering techniques, state libraries, caching techniques, and monitoring methods are all studied independently, but the survey does not provide another longitudinal comparison of frameworks and libraries in this area.

2. Literature Review

2.1 Measuring Front-End Performance

Identifying metrics for study is a historical challenge in performance studies. In a 2023 systematic review, the authors included 30 high-quality studies from 6657 initial records [1]. The most common metrics in these studies were traditional (HTTP) performance metrics, such as load time, page size, and response time. In comparison, machine learning methods were rarely used [1]. Both types of reliance are important because traditional indicators describe delivery better than they do interaction. The second source of uncertainty is perceived quality. The same subjective study that questioned Core Web Vitals found that consent banners correctly belong in any account of page load time as users experience it [2]. Engineering implications of this finding are that lab scores, field scores, and perceived quality should be considered together: one may move while the other two remain constant.

Interaction is a separate dimension to delivery metrics because a page can load quickly and yet be slow to respond to user input (taps, scrolls, and swipes) if the main thread is blocked after the load event doing scripts, layout, and third parties. Since most time in a session for feed-based apps and editing is after the first screen, responsiveness in use is as important as speed of arrival. The

literature surveyed below reflects this imbalance, as most results report on load-phase performance rather than sustained interaction behavior.

2.2 Architectural Fragmentation of the Front End

This has also happened in frontend development. Micro-frontends, for example, are a popular example of the frontend web being broken up into pieces and owned by different development teams. Advantages and disadvantages were found by reviewing a multivocal literature review on 173 peer-reviewed primary studies, with 43 discussing both advantages and disadvantages. The advantages are team autonomy and incremental improvement. The disadvantages are larger payloads, duplicated code, coupled components, and more complex performance level monitoring due to each shipped artifact having its own dependencies, initialization time at runtime, and telemetry [3]. Thus, organizational boundaries are performance boundaries. This effect is well known in architectural literature, but is rarely studied quantitatively in practice.

2.3 Secondary Studies Within Single Layers

The most developed are caching, and in particular web caching, which discards objects to avoid exceeding a storage limit, web prefetching, which predicts which objects will be requested next, and application-level caching, which caches computation results within an application [4]. A more thorough survey of application-level caching addresses mainly adaptive solutions to avoid the gradual performance loss of static configurations [5]. Both review papers treat the reduction of latency as a common goal, but the approaches they cover involve differing amounts of risk.

Cross-platform development has attracted its own stream of secondary research; one recent comparative analysis of market and developer trends showed that developer satisfaction, job opportunities, and ecosystem maturity are now as important as runtime behavior [6]. Within the reviewed corpus, few controlled studies compare cross-platform frameworks to a native baseline, but one of the best-designed remains influential [7]. Finally, there has been work on monitoring studies that characterize the differences between end-to-end synthetic and real-user monitoring and combine these to detect global, partial, and periodic problems on a high-traffic site [8].

2.4 Positioning

Within the reviewed corpus, no study jointly examines the effect of changing the path upstream of a rendering decision, the interaction of cache and state propagation decisions, and the interaction of telemetry with the budget that is to be protected. While these interactions are common discussion items among practitioners, the reviewed evidence provides limited coverage of their strength. The intent of this review is to partially close this gap by treating cross-layer cost transfer as the unit of analysis rather than the performance of a single technique.

3. Methodology

3.1 Review Design

This study used a structured evidence review rather than a systematic literature review. The latter would require screening logs, removing duplicates, and enabling another researcher to reproduce each decision about what to include from all relevant databases. An interpretive synthesis across heterogeneous evidence was used with transparent decisions another researcher could replicate. The review therefore does not claim to census every publication on front-end performance but claims that every source that was included was chosen, checked, and classified according to stated criteria.

3.2 Search Concepts and Sources

The search strings combined the three concept groups, and each was covered by a set of commonly used synonyms. Application terms included front end, web application, mobile web, hybrid application, cross-platform application, micro-frontend, and short video. Architectural terms were client-side rendering, server-side rendering, hydration, backend for frontend, API aggregation, REST, GraphQL, gRPC, state management, caching, prefetching, preloading, observability, and real-user monitoring. Outcome terms included latency, responsiveness, first contentful paint, CPU use, memory, energy, bandwidth waste, and instrumentation overhead. IEEE Xplore, the ACM Digital Library, ScienceDirect, SpringerLink, Wiley Online Library, and Google Scholar with backward citation checking are searched. The cutoff was 25 September 2026. The search was originally limited to publications from 2021 to 2026. Older works were only preserved when they formed the basis for caching, monitoring, or cross-platform benchmarking.

3.3 Inclusion, Exclusion, and Verification

Source was included if it described architectural layers or their interaction, a selected workload, a qualified performance metric, a comparison between systems, or an explicit review method, excluding tutorials, vendor documentation, and framework documentation without an assessment of their performance. Backend studies with no observable user impact were excluded, as were unpublished candidates in preprint format and one comparative rendering study, which was not published in an indexed venue. The remaining records were checked against either a publisher page, the DOI resolver of the candidate, or Crossref. Where

possible, quoted values were compared against the source paper abstract or publisher record. A total of 25 sources were included: 21 journal articles and 4 peer-reviewed conference/proceedings papers, 20 of which had been published in 2021 or later.

3.4 Evidence Classification

Due to differences in methods, each source was classified into one of four classes of evidence (see Table 1 for classification) to indicate their generalizability, and these classes are used throughout the results to qualify claims. These classes describe study design and the scope of inference supported by that design; they are not quality scores or rankings of individual publications.

Table 1. Evidence classes used in the review

Class	Definition	Use in the synthesis	Sources
C1 Controlled experiment	Comparison under a defined workload with measured outcomes	Supports directional claims within the tested workload	[2], [7], [9], [12], [13], [14], [15], [18], [19], [20], [22], [23], [24], [25] (14)
C2 Large-scale measurement or industrial study	Field measurement, production case or qualitative study of real projects	Supports claims about prevalence and production behaviour	[8], [10], [11], [17], [21] (5)
C3 Systematic review or survey	Secondary study with a stated review method	Frames the state of knowledge within one layer	[1], [3], [4], [5] (4)
C4 Comparative or contextual analysis	Characterization, market or developer-trend comparison without controlled performance measurement	Provides context only; not used for causal claims	[6], [16] (2)

3.5 Synthesis Procedure

The approach applied the cost-transfer tracing process to every source that was present. For every source reported as an optimization, the layer and metric were specified first, followed by the transferred or pushed cost. It then identified the layer absorbing that resource and whether the source measured the absorbing layer. This analysis exposes an inconvenient absence: many papers report a metric that they improved but do not state which layer paid the cost of gaining the one given metric. As workloads, hardware, networks, and metrics of the reviewed studies differed, the effect sizes from these studies were not pooled.

3.6 Mapping to the Application Context

Each finding was also reviewed against the application context given in Section 1.2 regarding whether a finding bore on feed startup, navigation between feed items, or the freshness of commerce data such as price and availability. The analysis also examined whether the result could apply to editing-oriented tooling with heavy interactive state or operation during traffic peaks. The application scenario did not determine source inclusion and is not treated as empirical validation; it is used only, after synthesis, to illustrate how the synthesized framework can guide architectural reasoning. This question shaped the interpretation in Section 5.3, but not whether to include a source in the review, as a more general evidence base than a particular domain is needed. Where appropriate, findings in other domains such as enterprise web applications or server-side instrumentation are denoted in the context of the application domain.

4. Results and Findings

Table 2 summarizes the important synthesis sources in terms of the layer they inform. Each subsequent subsection follows a critical path from the rendering boundary to the execution context.

Table 2. Core evidence by layer

Layer	Source	Class	Contribution to the synthesis
Measurement	[1]	C3	Traditional indicators dominate; no standardized evaluation protocol
Measurement	[2]	C1	Core Web Vitals predict perceived quality less well than page load time
Rendering	[9]	C1	Server-side and hybrid rendering bring first content forward; hybrid consumes more server resources
Client execution	[10]	C2	Unused JavaScript is widespread, and removing it speeds mobile page loads
Client execution	[11]	C2	Third-party advertising adds page-load workload dominated by JavaScript
Data delivery	[12]	C1	A BFF reduces latency at larger data volumes; the monolith uses fewer resources
Data delivery	[13]	C1	Protocol choice trades response time against server CPU utilization

Layer	Source	Class	Contribution to the synthesis
Composition	[3], [14]	C3, C1	Micro-frontends support autonomy and can perform well, but risk payload growth
State	[15], [16]	C1, C4	Footprint and update cost follow library design; no library wins everywhere
Caching	[17]	C2	Application-level caching is error-prone and a common source of defects
Prefetching	[18], [19], [20]	C1	Prediction and learned preloading trade quality of experience against wasted bandwidth
Observability	[21], [22], [23]	C2, C1, C1	Overhead can be low in production but depends on workload; telemetry cost may escape headline metrics
Cross-platform	[7], [24], [25]	C1	Abstraction cost varies by metric, engine and execution model

4.1 Rendering and Client Execution Cost

This conclusion does not imply that server rendering is always better than client rendering. One comparison of comparable front ends found that server-side rendering and hybrid rendering greatly improved the performance of the initial load and search engine optimization, "reducing First Contentful Paint by approximately 65% on average" [9]. Another observation from client-side rendering was that it decreased server load but delayed the time to first render. A mixed approach of both gave the most optimal results with more resource use [9]. In an important sense, the result is a cost shift: the server is doing work that the client would do to deliver the content early.

Even if you render your client-side app, the performance cost is high. One analysis found that in a sample of 15000 web sites, "half of the 300,000 JavaScript files used in these pages have at least 70% of unused functions, accounting for 55% of the files' sizes" [10]. Controlled mobile studies, based on a 200-page set, found dead code elimination "speeds up page loads by 25-30%" because of reduced CPU and bandwidth. Hence, it is required to look beyond rendering when measuring performance and speed. Instead, it is more useful to ask how much client work a particular action requires and not where rendering actually occurs.

Now, first-party code is competing in the browser for the same client budget as third-party code. Some characterization done at the level of the browser found that online advertising accounts for "more than 15% of browser page loading workload, and approximately 88% of that is spent on JavaScript" [11]. For mobile devices, the overhead was less impactful due to better optimizations and smaller advertisement sizes and frequency. Commerce feeds that contain advertising, attribution tags, and tracking scripts are still affected, since third-party code requests are executed on the main thread alongside product rendering and gesture processing.

This suggests that when thinking about rendering, the focus should be on the dominant bottleneck in the request along the interaction path. While hydration may delay the visibility of interactive content, streaming reduces the time until users can perceive progress and increases the number of partial states that need to be synchronized. The best decision for rendering is therefore based on several signals on either side of the boundary. For example, the server-side signal includes the time to the first byte and the upstream time. These include first content timing, transferred JavaScript and data, interaction latency, main-thread time, and the duration of any window that is not interactive on the client.

4.2 Server-Assisted Data Delivery

Server-assisted rendering changes the data path as well as where the component executes. Server-rendered routes remain slow when they fan out serially to several services, retrieve oversized payloads, or wait on a general-purpose API that was never shaped for the view or the consumer. In one study that controlled for implementation, researchers ran a monolith, micro-frontend design, and microservices fronted by a BFF layer on a containerized platform. In the evaluated implementation, the BFF-based configuration reduced latency and the impact of large data volumes on interface responsiveness, while the monolithic configuration consumed fewer resources but behaved more sequentially and showed weaker scalability under the tested conditions [12]. Aggregation shifts the cost, but does not eliminate it, to the server tier and its operations.

The choice of communication protocol has its own trade-off between performance and load on the server. In one benchmark for containerized microservices, this flat and nested data was read from the data source with "several requests ranging from 100 to 500 requests" [13]. Under the retrieval workloads evaluated by Niswar et al., gRPC produced the lowest response times, while GraphQL showed higher response time and CPU utilization than the REST and gRPC implementations; these results should not be generalized beyond the tested retrieval patterns [13]. The CPU consumption is for the benefit that the client has the flexibility to request exactly what fields a view requires. The cost of this benefit to the client, however, does not appear in any of the browser's metrics.

The benefits and drawbacks of micro-frontends also yield a marginal gain vs. risk. One piece of empirical research found performance enhancements in apps with micro-frontends and microservices compared to apps with monolithic front ends [14]. Evidence from the review, however, supports that the architecture makes matters worse through its payload bloat and duplicate

code [3]. After all, composition discipline is determinative; shared dependencies, route-level loading, and one telemetry pipeline preserve the benefits, while fragments shipping their own runtime erode benefits.

The contract transforms request topology from a back-end detail into a performance contract, with important properties like the number of upstream requests a view requires, if they can be run in parallel, how much unused data has to be passed through the boundary, who owns cache freshness, and what happens to a partially rendered surface when one of its sources is slow. In another example, one contract for a short video feed may just define one view-shaped response per swipe window, instead of a separate call for each item, product card, and creator badge.

4.3 State Management and Update Propagation

State management research is predominantly library comparison, which is a mature subfield. A benchmark comparing six popular state management libraries in a React e-commerce application includes code size, Core Web Vitals, memory overhead, and update behavior [15]. Excluding differences between library designs in footprint and update costs, a mature centralized store is heavier than a light-weight store in the tested workload. This is concluded from a study that compared Context API for small to medium applications, Redux and MobX for large and complex applications, and Zustand for applications of any size. Neither study identifies a library that is best for every workload and scale.

The more general inference concerns update scope: the state should be placed and subscribed to at the most local level consistent with correctness and coordination. Centralized stores can be used when a global shared state for higher-order features is desired across components that are distant from each other. Such features have wide subscriptions and thus propagate to distant components. By contrast, local state restricts propagations but requires coordination to ensure that features agree on their states. Server-state libraries are also of interest since they combine ownership with caching, deduplication, and background refresh, and their design needs to consider the freshness model, which is discussed next.

Signals like update time are not enough. The following signals can reveal the underlying issues: re-renders caused by the update, subscription fan-out, memory usage, and derived computation triggered by the update. These ideas are illustrated with a creator editing tool: a timeline that changes many times per second needs fine-grained subscriptions, whereas a catalogue configuration that is read by all screens but only infrequently changes can be stored in a broad store with little cost penalty. Fine granularity is less critical when layout or painting dominates latency; however, it is most important when it prevents expensive work later.

Server-originated data should generally be modeled separately from ephemeral interface state when its ownership, freshness, and update lifecycle differ. Feed items, product records, usage analytics, and similar data originate on the server, change without user command, and are often shared across views at the same time. They cannot be treated like local client state, because you'd end up with copies diverging and updates firing in multiple places. Centralizing ownership of server-originated data can reduce duplicate requests and inconsistent local copies, although the performance benefit depends on cache policy, update frequency, and subscriber behavior. This centralization implies an additional caching policy, with a need to apply the freshness classes discussed below to it.

4.4 Caching, Prefetching, and Freshness

The tradeoff in caching is latency, but this tradeoff is often harder to master than hit rates indicate. A qualitative study of ten software projects found application-level caching to be time-consuming and error-prone work that frequently introduces defects [17]. A fast cache that returns stale or semantically invalid data improves latency but reduces correctness. Each cached item has a freshness class, specifying whether it is immutable, reusable within an upper bound on staleness, reusable only after revalidation, or never reusable. Because HTTP caches, service workers, client data caches, and memoized computations operate at different levels of semantic recognition, invalidation recommendations rarely translate directly from one layer to another.

Caching can create cache conflicts at multiple levels. A service worker can deliver responses to a client data cache that seem fresh, or a memoized computation can pollute its cache however long after its inputs change. These conflicts usually show up in e-commerce scenarios. A cached product description can be stale by several hours, while a price, a stock count, or a promotion window may be stale by a few seconds or not at all. Because assigning a single freshness policy to an entire response leads to wasted latency or incorrectness, view-shaped responses separate volatile and stable fields.

Prefetching and lazy loading can be thought of as scheduling policies: prefetching brings work early, and lazy loading defers work that may prove unnecessary or not worth performing. The effectiveness of a prefetching policy depends highly on the quality of its predictions. A study of an unstable mobile network used path-aware embedding and supervised learning to improve prefetching performance [18]. The improvement in performance, however, depends on the availability of sufficient behavioral data, which drives the algorithm towards high-traffic applications and away from new or less visited paths.

For the short video segment preloading scenario, learning-based buffer-aware preloading achieves "QoE improvement by 28.7% and 51.4% compared to state-of-the-art algorithms, while mitigating bandwidth wastage by 40.7% and 83.2% without sacrificing video quality" [19]. Deep reinforcement learning with domain knowledge evaluation is introduced and retains an improvement in QoE score of 3.73%-11.28% compared to state-of-the-art algorithms, while achieving an average bandwidth waste of only 10.27%-12.07% when evaluated against 4G, WiFi, and 5G traces. Both of these studies consider wasted bandwidth as a first-class outcome alongside quality of experience, which is a perspective that could benefit general web prefetching research.

For recommendation-based feeds, ranking and delivery might be coupled. Swipe-based statistics and the confidence of a recommendation may suggest how far down the feed the client prefetches. Prefetching further into the feed is well-suited when the next item is predictable and prefetched less when it is not. In aggressive preloading, the cost is upfront: bandwidth, radio energy, egress from the delivery network, and device memory. Lazy loading defers this cost to the first time the user interacts. The approach will be worthwhile if the decrease in visible latency is greater than the cost of doing the work early.

4.5 Observability and Its Performance Budget

Performance evidence is only as useful as the measurement context in which it is gathered. Therefore, the authors designed and evaluated an automated observability approach based on an industrial case study on the automated instrumentation of open-source telemetry agents in "four large-scale web applications" [21]. This method was able to collect enough interaction and runtime data for experience analysis while using acceptable overhead and performance during production [21]. Despite the promising result, the published abstract characterizes the overhead qualitatively rather than quantitatively.

Adjacent evidence from server-side instrumentation similarly shows that observability overhead can become significant under some workloads. A server-side experiment was conducted over 5,000 experiments on 70 microservice APIs [22]. Its overall "throughput decreased by up to 8.40%, with some individual cases experiencing up to a 30% reduction compared to the baseline" [22]. These are containerized backend services, so do not assume these numbers will translate directly to browser JavaScript. The study, regardless, can be relevant for instrumentalized methods where overhead depends on sampling, serialization, the number of events, and instrumentation interfaces.

In mobile browsers, advertising also had a large effect size, increasing the energy consumption of mobile web apps. Analytics affected only one browser [23]. The first contentful paint did not change considerably. Both browsers had a statistically meaningful effect on full page load time, but with a small effect size [23]. Further, telemetry cost may not be visible in the headline metric but may still be present in energy consumption and completion time.

This review therefore proposes an explicit telemetry budget. The telemetry (interaction traces, long task reports, error payloads, resource timing, and custom events) should be sampled and aggregated based on information value, and the monitoring layer should be treated like any other client code. The recommendation of a two-loop approach further strengthens this proposal. Controlled experiments can characterize performance and detect regressions, while real-user monitoring can explore the effect across the actual device and network space [8].

4.6 Cross-Platform Execution Contexts

Cross-platform evidence generally supports the main result: abstraction has costs that are not necessarily equal or large enough to uniquely identify or differentiate any framework. A controlled benchmark compared five cross-platform development frameworks against an Android native baseline [7]. In this benchmark, several cross-platform frameworks incurred overhead relative to native Android on some metrics, while individual frameworks matched or exceeded native performance on others [7].

The web is not a single execution context, and so a study comparing native Android apps for the same services to web apps found that the native apps consumed orders of magnitude less energy [24]. Web apps also used more CPU and memory [24], and the study could not determine their frame time [24]. Another comparison of progressive web applications determined that native development has the lowest energy impact and that the energy impact of progressive web applications is mainly affected by the browser engine [25].

In relation to RQ3, cross-platform evidence should additionally report back the engine, bridge, and rendering pipeline rather than just the name of the framework used. Second, the principles seem to apply to resources rather than APIs: performing less work, keeping the updates to the sub-resource proportional to changed data, and reusing data only under explicit freshness rules. The other two principles are deferring low-priority work and measuring on reasonably representative devices. Webviews pay the cost of parsing, executing, and laying out pages. They also pay for the overhead of integrating with the host environment and for bridging, serialization, or custom rendering in cross-platform runtimes.

Embedded web views are web browsers that an application shows as part of itself (instead of a stand-alone web browser). For pages hosted in an embedded webview (product pages, checkout flows, promotional surfaces), the browser engine version,

caching, and process model are effectively dictated by the hosting app. Optimizations validated in desktop browsers may not transfer directly to embedded webviews because the browser engine, process model, host integration, and caching behavior can differ. Measurement plans that include the embedded context (rather than a proxy for it such as a mobile browser) are more likely to catch these.

5. Discussion: An Integrated Cross-Layer Decision Framework

5.1 Cost-Transfer Model

The path for these results is consistent with a model where performance cost travels down the critical path of five layers inside the execution context envelope. This starts with an API data access layer, rendering and hydration, client state management, caching and prefetch schedule, and finally production telemetry, as summarized in Figure 1, which also shows the usual cost direction on edges.

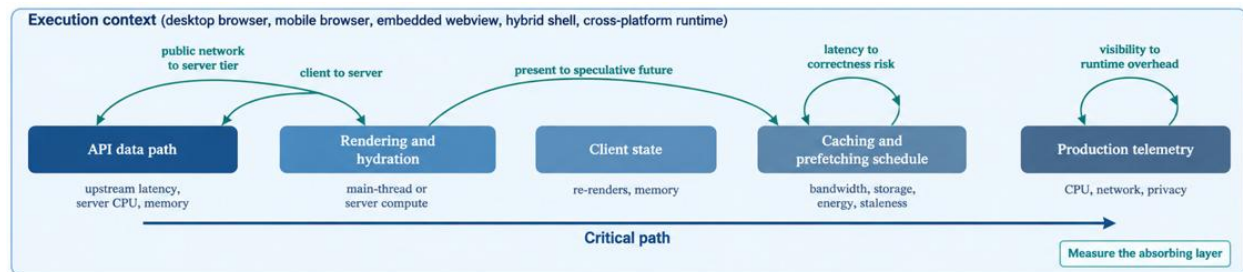


Figure 1. Cost-transfer model: five layers on the front-end critical path within an execution context, with the recurring directions in which optimization relocates cost.

Each model layer consumes a budget. The data path consumes upstream latency, server CPU, and server memory. These steps (render/hydrate, client state) require main-thread and compute time, respectively, on either side. The latter in particular requires re-render work and memory, which depend on subscription fan-out. Thus, the caching and prefetching schedule ends up using bandwidth, storage, energy, and a correctness budget (tolerable staleness). Telemetry uses CPU, network bandwidth, and privacy. The cost of a budget depends on the context in which it is used; for example, running the same operation on a mid-range phone in a webview is pricier than running it in a desktop web browser.

The answer to RQ1 is derived from the tracing process. Across the corpus of evidence, cost moved in five orientations. Technical choices to avoid future work include rendering choice, which moves work across the client-server boundary; aggregation, which moves round trips from the public network to the server tier [9], [12], [13]; prefetching, which moves work from the present into a speculative future [19], [20]; and caching, which sacrifices latency for a correctness risk [17]. Each move trades visibility for runtime overhead [21], [22]. As with the others, none is harmful in itself, but all become harmful if the absorbing layer is unmeasured.

5.2 Decision Framework

Table 3 matches general performance (that is, transferred-cost signals) to evidence-supported levers and the costs those levers transfer. The table also constitutes the response to the second research question (RQ2), about the signals indicating whether costs have been transferred. The framework's unique feature is the transferred cost column, naming the signal the team must add to its dashboards before being allowed to adopt the lever.

The column sequence from left to right is target, signal, lever, and transferred cost. Success is defined by target and signal. The lever is the most likely architectural change, and the transferred cost is the most likely place to estimate the price. A team designing server aggregation to eliminate client waterfalls, for example, would add server CPU, memory, and upstream fan-out to its monitoring before the rollout, so that any relocation of cost stays visible in the same review cycle as the gain. The transferred-cost column identifies the signal that should be monitored alongside the lever so that relocated cost remains observable.

Table 3. Cross-layer decision framework

Goal	Signals to monitor	Primary lever	Likely benefit	Transferred cost	Evidence
Faster first content	First content timing; time to first byte; transferred JavaScript	Server-side or hybrid rendering with server aggregation	Earlier visible content	Server resource use; hydration gap; upstream fan-out	Moderate [9], [12]

Goal	Signals to monitor	Primary lever	Likely benefit	Transferred cost	Evidence
Faster interaction	Interaction latency; main-thread time; long tasks	Remove unused and third-party JavaScript; defer non-critical work	Less main-thread contention	Deferred work surfaces later; more loading boundaries	Moderate [10], [11]
Fewer client-server waterfalls	Request count; fan-out; payload size	View-shaped aggregation (BFF); parallel upstream calls	Fewer public network round trips	Server CPU and memory; coupling of view and adapter	Moderate [12], [13]
Lower update cost	Re-render count; subscription fan-out; memory	Localized state; selectors; fine-grained stores	Smaller update scope	Coordination complexity	Moderate [15], [16]
Lower repeated latency	Cache hit rate; staleness incidents; revalidation rate	Freshness-classed caching at the appropriate layer	Less network and computation	Stale data; invalidation defects	Moderate [4], [5], [17]
Smooth feed navigation	Startup delay; rebuffering; wasted bytes	Confidence-conditioned preloading	Fewer stalls on swipe	Bandwidth, energy and memory spent on unviewed items	Strong [19], [20]
Production diagnosis	Field metrics by device and network; error and trace correlation	Real-user monitoring with synthetic baselines; sampled telemetry	Detection of field regressions	Instrumentation overhead; privacy exposure	Moderate [8], [21], [22]
Cross-platform consistency	Startup; CPU; memory; energy; frame time	Runtime choice and tuning informed by the execution model	Code reuse at bounded cost	Bridge, engine or rendering overhead	Strong [7], [24], [25]

The strength of evidence codes in Table 3 matches those in Table 1. Strong evidence is defined as more than one independent source showing the effect in controlled experimental conditions; thus, moderate evidence is a single controlled study, a review of evidence, or both.

5.3 Interpretation for a Media-Rich Interactive Commerce Application

Here are four examples of how the framework can help design the illustrative application described in Section 1.2. The first example is the startup of the content feed. A server-assisted first frame would likely help constrained devices, but only if the server can assemble the first feed window in a single parallel pass. The dashboard should also measure upstream fan-out and server time along with first content timing. Otherwise the improvement may merely be relocated.

The proportions of preloading between feed items could vary based on recommendation confidence, navigation behavior, and wasted bytes. Recording these wasted bytes would be a main outcome, analogous to the short-video studies [19], [20]. Where confidence is low or the network is metered, the policy may fall back to shallow preloading and may accept a small delay at the moment of interaction.

Lastly, editing-oriented tooling, which has narrow subscriptions and view-shaped data flows on its editing surfaces, represents a high-frequency state that can be written and read often, especially during traffic peaks. In short, the telemetry on peaks should bias toward diagnostic signals rather than the status quo, since the cost of observation competes with what you're watching.

The fourth is freshness of commerce data; the image and description can be cached aggressively. Because price, stock, and promotion fields generally require revalidation at short intervals or each time the view is accessed, these fields are separated out of the view and included in the feed response. This allows the bulk of the data to be reused while maintaining business and trust implications.

5.4 Applying the Framework

The framework is not a recipe or set of rules. A team could take an externally visible goal (like making the first content appear faster or using the feed more smoothly) and determine what their signal would be. Next, find the full critical path, i.e., from upstream data access through rendering, state, and caching to telemetry, of the affected views, and which of these layers are

already measured. The target device and network population used in the R&D are therefore taken from field data, because they are the ones expected to affect performance the most, even though they are not the same bottlenecks as would be found with lab testing on a mid-range phone. Only then is a lever from Table 3 selected with the transferred cost signal from that line. Validation can then close the feedback loop with controlled tests of the metric in question and measurements of that absorbed layer. When that change persists in the field, a change that improves the metric in question but degrades the absorbing layer is seen more as a relocation of cost than as a win.

6. Research Gaps and Future Directions

Finally, Table 4 provides a summary of the gaps, the layers affected by each, the level of current evidence, and suggested study designs for filling the gaps. Figure 2 shows how unevenly the current evidence base covers the layers.

Table 4. Research gap matrix

Gap	Layers affected	Evidence status	Proposed study design
Cross-layer benchmarks	All layers	Not identified in the reviewed corpus: layers are evaluated separately	Vary one layer, hold the others constant, and measure the absorbing layers
Browser-server performance contract	Data delivery; rendering; caching	Limited in the reviewed corpus: aggregation and protocol studies lack front-end metrics	Equivalent interface workloads across direct access, general APIs and view-specific aggregation
Front-end telemetry overhead	Observability; execution context	Thin within included studies: industrial evidence is qualitative and numeric evidence is server-side	Browser benchmark across device classes under fixed event volumes
Cross-platform reproducibility	Execution context	Thin within included studies: applications, versions and metrics differ	Published executable workloads with runtime versions and raw data
State and cache interaction	State; caching	Thin within included studies: comparisons use single applications	Offline and reconnect scenarios under controlled update rates
Resource efficiency as a primary outcome	All layers, prefetching in particular	Emerging within the reviewed corpus: strongest in published short-video preloading studies	Report energy, memory and wasted bytes alongside latency
Recommendation and delivery coupling	Prefetching; data delivery	Not identified in the reviewed corpus: ranking and delivery evaluated separately	Couple recommendation confidence to preload depth; report engagement and waste

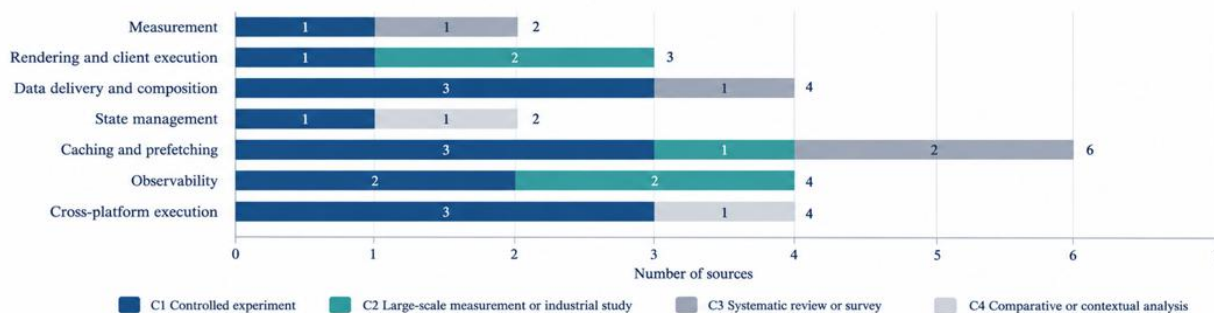


Figure 2. Distribution of the 25 included sources by architectural layer and evidence class, showing thin coverage of state management and measurement.

Cross-layer benchmarks are the main gap. Rendering, data topology, state, caching, and observability are evaluated independently, while production applications combine them across layers. A benchmark varying a single layer while fixing the others could help

clarify whether the benefits are true to the technique or just cost shifted elsewhere. A server rendering study could describe upstream request count, fan-out, data volume, cache behavior, server resource use, and various metrics from the browser side.

The browser-server performance contract is also only under-integrated; although aggregation and protocol work have shown that response shaping and communication choice can change latency and resource utilization [12], [13], these are rarely integrated with hydration, client caching, state propagation, or field measurements in performance research and design. The proposed framework, by comparing interface workloads for direct service access, general-purpose APIs, and view-specific aggregation, would be empirically testable.

Front-end observability overhead has not been studied as extensively as backend overhead. There is no current cross-ecosystem benchmark of telemetry overhead for low-end phones, desktop browsers, webviews, and cross-platform runtimes at the same observed event throughput. Cross-platform reproducibility is poor; applications, device types, framework runtime versions, workloads, and metrics vary. Future work should publish concrete workloads, runtime versions, hardware/network profiles, and raw measurements to enable reproducibility across platforms and studies. However, evidence is lacking on how to handle state management and client data caching in situations with rapid updates and offline/reconnects.

Resource efficiency is another important dimension. Most front-end performance work, whether optimizing for latency or throughput, treats energy, memory pressure, accessibility, and privacy as second-class resources, if they are considered at all. The short-video preloading literature, where wasted bandwidth is a first-class design consideration, might be considered a better model for the larger front-end performance literature.

Another gap is that recommendation-based interfaces largely treat the ranking and delivery problems as orthogonal. However, a recommendation's confidence has an impact on how much content clients need to pre-fetch. However, studies of pre-fetch depth based on recommendation confidence, which report on both user engagement and wasted resources, would fill a gap between studies in disparate venues.

7. Limitations

This study employed an evidence review, which is not a systematic literature review, because it only reviews articles that are strictly relevant and interpretable. We may have missed work in subscription-only databases or work that used different terminology. The review is heterogeneous in discipline and methodology; some works compare frameworks, some take a close look at a single framework, and some consider back-end infrastructure that does not impact the critical path on the front end. This heterogeneity may arise from the fact that these studies did not pool the effect sizes.

Source selection and evidence classification were conducted by one reviewer using the predefined criteria in Table 1. This introduces the possibility of reviewer judgment bias, which is acknowledged as a limitation. The classes themselves, such as whether a study had a controlled comparison, are defined directly from study design dimensions, which leaves comparatively little room for disagreement, but no independent second coding was performed to confirm this.

The front-end ecosystem moves faster than publications. Consequently, metrics on hydration, scheduling heuristics, state propagation, and rendering are likely outdated by new framework and browser releases. Some evidence comes from server-side instrumentation research, which is not directly applicable to the browser environment. The results in this section were characterized per their published abstracts and should be read directionally. This section includes the state management comparison and the micro-frontend performance result. Although such a measurement was not taken newly, in Section 5.3, the context of a short-video commerce application is relevant to such an interpretation.

8. Conclusion

For RQ1, this review identified five recurring directions in which performance cost moves when a single front-end layer is optimized: rendering choices trade work between client and server, aggregation moves round trips to the server tier, prefetching moves work to a speculative future, caching trades latency for a risk to correctness, and telemetry trades visibility for runtime overhead. These choices are not harmful in themselves, but they become so when the absorbing layer is not measured.

For RQ2, transferred cost becomes visible through complementary signals: upstream fan-out and server resource use for server-assisted rendering and aggregation, re-render counts and subscription fan-out for the cost of broad state, wasted bytes for speculative preloading, and instrumentation overhead for observability.

For RQ3, the principles that hold across execution contexts concern resources rather than framework identity: performing less work, scoping updates to the data that actually changed, reusing data only under explicit freshness rules, deferring low-priority work, and measuring on representative devices and execution contexts.

The paper's primary contribution is a cross-layer cost-transfer model and decision framework that links a performance objective to the architectural lever used, the layer likely to absorb the transferred cost, and the signals needed to observe that cost. Section

5.3 shows, using an illustrative media-rich interactive commerce application, how different combinations of these layers come into play around feed startup, preloading, editing-oriented tooling, and data freshness. The next step for researchers is reproducible cross-layer studies that report the absorbing layer alongside the augmented metric. Front-end performance is best understood as a conditional property of the whole system, and engineering decisions made at this level are likely to remain valid longer than decisions tied to a specific framework or software release.

Funding: This research received no external funding.

Conflicts of Interest: The author declares no conflict of interest.

Data Availability: No new data were generated. Every source in the evidence base is listed in the References.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers.

References

- Anas Shatnawi et al., "Observability of web applications: An industrial case study", *Journal of Systems and Software*, vol. 239, art. 112886, September 2026. Available: <https://doi.org/10.1016/j.jss.2026.112886>
- Andreas Bjørn-Hansen et al., "An empirical investigation of performance overhead in cross-platform mobile development frameworks", *Empirical Software Engineering*, vol. 25, no. 4, pp. 2997–3040, July 2020. Available: <https://doi.org/10.1007/s10664-020-09827-6>
- Behnam Pourghassemi et al., "adPerf: Characterizing the Performance of Third-party Ads", *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 1, pp. 1–26, February 2021. Available: <https://doi.org/10.1145/3447381>
- Biao Hou et al., "Gamora: Learning-Based Buffer-Aware Preloading for Adaptive Short Video Streaming", *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 11, pp. 2132–2146, November 2024. Available: <https://doi.org/10.1109/tpds.2024.3456567>
- Christos Petalotis et al., "An empirical study on the performance and energy costs of ads and analytics in mobile web apps", *Information and Software Technology*, vol. 166, art. 107370, February 2024. Available: <https://doi.org/10.1016/j.infsof.2023.107370>
- Gregor Jošt and Viktor Taneski, "State-of-the-Art Cross-Platform Mobile Application Development Frameworks: A Comparative Study of Market and Developer Trends", *Informatics*, vol. 12, no. 2, art. 45, April 2025. Available: <https://doi.org/10.3390/informatics12020045>
- Jesutofunmi Kupoluyi et al., "Muzeel: Assessing the Impact of JavaScript Dead Code Elimination on Mobile Web Performance", *Proceedings of the 22nd ACM Internet Measurement Conference*, pp. 335–348, October 2022. Available: <https://doi.org/10.1145/3517745.3561427>
- Jonhny Mertz and Ingrid Nunes, "A Qualitative Study of Application-Level Caching", *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 798–816, September 2017. Available: <https://doi.org/10.1109/TSE.2016.2633992>
- Jonhny Mertz and Ingrid Nunes, "Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches", *ACM Computing Surveys*, vol. 50, no. 6, art. 98, November 2017. Available: <https://doi.org/10.1145/3145813>
- Jürgen Cito et al., "Identifying Web Performance Degradations through Synthetic and Real-User Monitoring", *Journal of Web Engineering*, vol. 14, no. 5–6, pp. 414–442, December 2015. Available: <https://journals.riverpublishers.com/index.php/JWE/article/view/3845>
- Michał Droń and Tomasz Szandała, "Web Application State Management: A Review of Leading React Frameworks", *IEEE Software*, vol. 43, no. 2, pp. 112–118, March 2026. Available: <https://doi.org/10.1109/MS.2025.3621269>
- Miloš Bardáč and Martin Bobák, "Efficient Frontend for Microservice-Based Applications", *Modern Management based on Big Data VII, Frontiers in Artificial Intelligence and Applications*, July 2026. Available: <https://doi.org/10.3233/FAIA260627>
- Mohammad Ghattas et al., "Predicting Website Performance: A Systematic Review of Metrics, Methods, and Research Gaps (2010–2024)", *Computers*, vol. 14, no. 10, art. 446, October 2025. Available: <https://doi.org/10.3390/computers14100446>
- Muhammad Niswar et al., "Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC", *International Journal of Electronics and Telecommunications*, vol. 70, no. 2, pp. 429–436, June 2024. Available: <https://doi.org/10.24425/ijet.2024.149562>
- Mulki Indana Zulfa et al., "Caching strategy for Web application – a systematic literature review", *International Journal of Web Information Systems*, vol. 16, no. 5, pp. 545–569, October 2020. Available: <https://doi.org/10.1108/IJWIS-06-2020-0032>

- Neha Kaushik et al., "Micro Frontend Based Performance Improvement and Prediction for Microservices Using Machine Learning", *Journal of Grid Computing*, vol. 22, art. 44, April 2024. Available: <https://doi.org/10.1007/s10723-024-09760-8>
- Nikolas Wehner et al., "Do you agree? Contrasting Google's Core Web Vitals and the impact of cookie consent banners with actual web QoE", *Quality and User Experience*, vol. 8, art. 5, June 2023. Available: <https://doi.org/10.1007/s41233-023-00058-3>
- Ruben Horn et al., "Native vs Web Apps: Comparing the Energy Consumption and Performance of Android Apps and their Web Counterparts", 2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 44–54, May 2023. Available: <https://doi.org/10.1109/MOBILSoft59058.2023.00013>
- Sanja Brekalo et al., "Characteristics and comparison of global state management tools in React applications", *Acta Graphica*, vol. 33, no. 1, pp. 23–33, June 2025. Available: <https://doi.org/10.25027/agj2017.28.33.1.3>
- Semra Kermo and Adnan Dželihodžić, "Comparative Analysis of Server-Side, Client-Side, and Hybrid Rendering Approaches in Modern Web Applications", 2026 25th International Symposium INFOTEH-JAHORINA, pp. 1–6, March 2026. Available: <https://doi.org/10.1109/infoteh68759.2026.11477683>
- Severi Peltonen et al., "Motivations, benefits, and issues for adopting Micro-Frontends: A Multivocal Literature Review", *Information and Software Technology*, vol. 136, art. 106571, August 2021. Available: <https://doi.org/10.1016/j.infsof.2021.106571>
- Stefan Huber et al., "A comparative study on the energy consumption of Progressive Web Apps", *Information Systems*, vol. 108, art. 102017, September 2022. Available: <https://doi.org/10.1016/j.is.2022.102017>
- Tolga Buyuktanir and Mehmet S. Aktas, "On the Use of Embedding Techniques for Modeling User Navigational Behavior in Intelligent Prefetching Strategies", *Concurrency and Computation: Practice and Experience*, vol. 37, no. 3, art. e8356, February 2025. Available: <https://doi.org/10.1002/cpe.8356>
- Yasmeen Hammad et al., "An empirical study on the performance overhead of code instrumentation in containerised microservices", *Journal of Systems and Software*, vol. 230, art. 112573, December 2025. Available: <https://doi.org/10.1016/j.jss.2025.112573>
- Yuhong Xie et al., "Short video preloading via domain knowledge assisted deep reinforcement learning", *Digital Communications and Networks*, vol. 10, no. 6, pp. 1826–1836, December 2024. Available: <https://doi.org/10.1016/j.dcan.2024.01.006>