Intelligent Systems Research and Applications Journal

www.israj.org
eISSN: 3120 - 0826

Engineering Speed at Scale: A Comprehensive Study of Frontend Performance Optimization Using the PERFORM Framework

Izzath Fathima[1], Sukanya Vuppala[1,*]

[1] Department of Computer Science, College of Engineering, Osmania University, Hyderabad, Telangana, India

ARTICLE INFO

Article history:
Received 17 May 2026
Received in revised form 21 June 2026
Accepted 9 July 2026
Available online 25 July 2026

Keywords:
Frontend Performance Engineering; Core Web Vitals; Critical Rendering Path; PERFORM Framework; Web Resource Optimization; User Experience (UX) Analytics

ABSTRACT

Frontend performance has become a critical factor influencing user experience, engagement, and business outcomes in modern web applications. As frontend architectures grow increasingly complex, performance bottlenecks related to resource loading, rendering behavior, and interaction latency become more prevalent. This paper presents a comprehensive study of frontend performance optimization conducted during a summer web development internship. A systematic optimization methodology, the PERFORM framework (Profile, Evaluate, Recommend, Fix, Observe, and Report), is used to analyze and improve performance across multiple production-grade web applications. The study focuses on optimizing the critical rendering path, improving Core Web Vitals, and applying practical optimization techniques, such as image optimization, lazy loading, and reducing render-blocking resources. Experimental results demonstrate significant improvements in page load performance, responsiveness, and visual stability. The findings highlight the effectiveness of structured performance engineering practices in real-world frontend development scenarios.

1. Introduction

Web performance has emerged as a critical factor influencing how users perceive and interact with modern web applications. With the growing reliance on web-based services across domains such as e-commerce, media delivery, education, and enterprise productivity, users now expect instantaneous response, smooth visual rendering, and stable layouts. Empirical evidence consistently demonstrates that even marginal delays in page load time correlate with measurable losses in user engagement, session duration, and conversion rates [1].

Despite significant advances in frontend frameworks, browser engines, and networking protocols, a substantial proportion of production web applications continue to exhibit performance deficiencies. These deficiencies typically arise from unoptimized asset delivery pipelines, inefficient rendering sequences, excessive client-side JavaScript execution, and poorly managed third-party integrations. Such issues are especially prevalent in complex enterprise applications where new

*Corresponding author.
E-mail address: sukanya.v@uceou.edu
https://doi.org/10.59543/mmhqdg22

features are continuously added with limited attention to performance budgets or rendering costs [2].

Crucially, performance bottlenecks are not confined to large enterprise platforms. Small-to-medium web applications, including content blogs, independent e-commerce stores, non-profit portals, and government information sites, frequently face the same issues despite operating with considerably fewer engineering resources. The optimization principles and structured methodology described in this paper are intentionally designed to be accessible regardless of team size or infrastructure sophistication, making the PERFORM framework broadly applicable across the full spectrum of modern web application contexts [11].

This paper presents a comprehensive study of frontend performance optimization conducted during a summer web development internship. The work was performed across real-world, production-grade web applications spanning content platforms, e-commerce environments, and informational portals. The central contribution of this work is the formalization of the PERFORM framework, which is a six-phase optimization cycle encompassing Profile, Evaluate, Recommend, Fix, Observe, and Report (PERFORM). By grounding each phase in established browser rendering principles and user-centric metrics, the framework enables practitioners to connect low-level technical changes to observable, measurable improvements in end-user experience. The primary objective of this study is to demonstrate how structured performance engineering applied to real production applications yields substantial and reliable improvements across Core Web Vitals (CWV) metrics- specifically Largest Contentful Paint (LCP), Interaction to Next Paint (INP), and Cumulative Layout Shift (CLS). A secondary objective is to establish a practical reference for frontend engineers seeking to transition from reactive, symptom-driven optimizations toward proactive, systematic performance engineering practice. Recent work by Ekpobimi et al. [12] and Karka [13] further motivates this transition by demonstrating that holistic integration of optimization techniques produces more reliable and sustainable performance gains than isolated fixes, reinforcing the structured methodology of the PERFORM approach.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the PERFORM framework methodology. Section 4 defines the system model and performance metrics. Sections 5 and 6 present case studies of implementation and experimental setups. Sections 7 and 8 report results and discussion. Section 9 concludes with recommendations for future work. A list of abbreviations is provided in the Appendix for reference.

2. Related Works

Web performance optimization has been an active area of research and industry practice for over a decade. Early work by Krishnamurthy and Rexford laid the groundwork for understanding HTTP traffic patterns and their implications for rendering latency [2]. Subsequent studies by Nah confirmed that response time thresholds below two seconds are essential for maintaining user satisfaction in web transactions [3].

The emergence of browser-integrated developer tools and automated auditing platforms significantly lowered the barrier to performance diagnosis. Google Lighthouse, introduced in 2016, consolidated multiple heuristics and metric measurements into a single auditing workflow, enabling developers to identify render-blocking resources, excessive payload sizes, and inefficient caching strategies with minimal effort [4]. WebPageTest and SpeedCurve extended these capabilities to

support real-network condition simulations and filmstrip visualizations, offering a more nuanced picture of user-perceived load behavior.

The introduction of Google's Core Web Vitals (CWV) initiative in 2020 marked a pivotal shift in how performance is defined and communicated [1]. By standardizing three user-centric metrics - LCP, INP, and CLS - the initiative provided a common vocabulary bridging developer tooling, search ranking signals, and user experience research. Subsequent studies by Rusu et al. and Barnes et al. demonstrated that improvements in CWV correlate with higher engagement rates, lower bounce rates, and increased revenue [5].

More recent contributions have expanded this foundation. Ekpobimi et al. [12] presented a conceptual framework for front-end web performance that systematically integrates code-splitting, lazy loading (LL), caching, and image optimization, emphasizing that these techniques must be combined holistically rather than applied in isolation, a principle directly embedded in the PERFORM framework's Recommend phase. Karka [13] demonstrated in a 2025 comprehensive study that eliminating render-blocking resources (RBRs) and delivering critical CSS together produce consistent reductions in LCP across major browsers, corroborating the optimization priority sequence proposed in Section 3. Menghnani [14] showed in 2025 that leveraging frontend observability instrumentation for proactive performance monitoring enables engineering teams to detect interaction latency regressions before they reach production, validating the continuous monitoring principles encoded in the PERFORM framework's Report and Profile phases.

Despite these advances, existing literature tends to address individual optimization techniques in isolation. Image compression, JavaScript (JS) deferral, and critical CSS extraction are each well documented, but frameworks that integrate these techniques into a coherent, iterative engineering life cycle remain underexplored. The PERFORM framework addresses this gap by providing a structured cycle connecting measurement, root cause analysis, prioritized implementation, and validation, enabling sustainable, regression-resistant performance engineering in production environments.

3. Methodology: The PERFORM Framework

Frontend performance optimization in production environments requires more than a collection of individual best practices. Without a structured methodology, optimization efforts tend to be reactive, inconsistent, and difficult to validate or reproduce. To address this challenge, this study introduces the PERFORM framework, a six-phase optimization cycle that guides performance engineering from initial profiling through validated improvements and ongoing monitoring. The acronym PERFORM stands for: Profile, Evaluate, Recommend, Fix, Observe, and Report.

The PERFORM framework is designed as a closed-loop system: the output of one phase feeds directly into the next, and the final Report phase feeds back into future Profile cycles. This iterative design ensures that performance improvements are sustained through continuous monitoring and progressive refinement rather than treated as one-time events. Figure 1 illustrates the six phases arranged as a continuous cycle. Each node in the diagram identifies its phase name and its primary tooling or output. Arrows indicate the direction of information flow between phases, emphasizing the cyclical, iterative nature of the process.

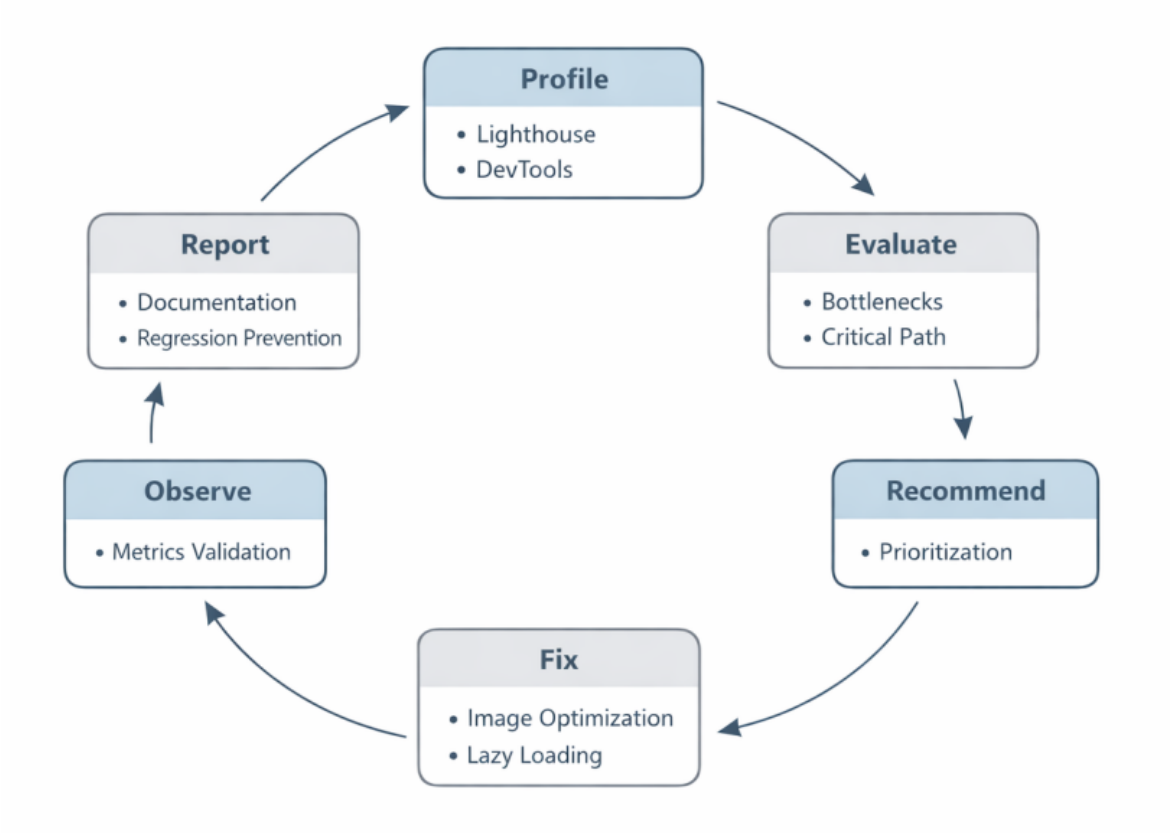

Fig. 1. The PERFORM framework: a six-phase closed-loop optimization cycle. Each phase (Profile → Evaluate → Recommend → Fix → Observe → Report) flows into the next, with the Report phase feeding back into Profile to enable continuous improvement

3.1 Profile

The Profile phase establishes a quantified performance baseline for each application under study. Using industry-standard tools, primarily Google Lighthouse, WebPageTest, and browser-integrated DevTools, engineers collect objective measurements of page load time, Time to Interactive (TTI), rendering milestones, and Core Web Vitals (CWV) values. Profiling is conducted under controlled, reproducible conditions, including fixed device emulation, predefined network throttling presets, and repeated runs to mitigate measurement variance. The resulting baseline serves as the reference point against which all subsequent improvements are measured. A key challenge at this stage is ensuring measurement reproducibility: minor variations in CPU load or background processes can introduce noise. Addressing this requires disciplined lab-mode configuration and multi-run median reporting.

3.2 Evaluate

During the Evaluate phase, raw profiling data is analyzed to identify root causes of observed performance deficiencies. This involves examining the browser's Critical Rendering Path (CRP), mapping long main-thread tasks, identifying render-blocking resources (RBR), and profiling JavaScript (JS) execution timelines. The Evaluate phase distinguishes between symptoms, such as a

slow LCP score, and their underlying causes, such as an unoptimized hero image loading synchronously in the critical path. By isolating root causes, this phase ensures that subsequent optimization efforts are targeted and efficient rather than speculative. A common challenge is attributing performance issues to specific code changes when multiple third-party integrations interact in complex ways; the systematic diagnostic approach described here directly addresses this ambiguity.

3.3 Recommend

Based on insights derived during the Evaluate phase, the Recommend phase produces a prioritized list of optimization interventions. Prioritization is guided by two primary factors: expected performance impact and implementation complexity. High-impact, low-complexity changes such as enabling image compression or deferring non-critical JS are classified as quick wins and addressed first. More complex architectural interventions, such as implementing critical CSS extraction or restructuring asset loading order, are planned as longer-term improvements. This pragmatic prioritization ensures that meaningful gains are realized quickly while avoiding unnecessary technical risk. Importantly, the Recommend phase is also the point at which cross-functional collaboration is most valuable: UX/UI designers can identify visual elements whose optimization must preserve design intent, and product managers can contextualize performance priorities within business roadmaps, ensuring that engineering decisions are aligned with user and stakeholder expectations [15].

3.4 Fix

The Fix phase encompasses the incremental implementation of recommended optimization strategies. Each change is applied in isolation where possible, ensuring that improvements and regressions can be attributed to specific modifications. Common interventions include converting image formats to WebP, implementing responsive image delivery via srcset, lazy-loading (LL) for off-screen assets, deferring JS scripts, eliminating unused CSS, and reducing layout-triggering JS. Throughout this phase, functional testing is performed in parallel to ensure that performance improvements do not introduce visual or behavioral regressions. A recurring practical challenge is coordinating Fix-phase changes with active feature development; the PERFORM framework addresses this by structuring fixes as isolated, incremental commits rather than large refactors, minimizing merge conflicts and deployment risk.

3.5 Observe

The Observe phase re-applies the same tooling and measurement conditions used in the Profile phase to capture post-optimization performance metrics. This measurement consistency is critical for valid before-and-after comparisons. Observation results are compared against baseline values to quantify improvements across LCP, INP, CLS, and overall, Lighthouse Performance scores. Where improvement is insufficient or new regressions are detected, the framework loops back to the Evaluate phase for further investigation. A key challenge at this stage is distinguishing genuine

improvements from measurement artifacts; the multi-run median methodology described in Section 6 directly addresses this.

3.6 Report

The Report phase formalizes the results of each optimization cycle by documenting the applied changes, measured outcomes, and lessons learned. This documentation serves multiple purposes: it enables traceability of individual optimizations, supports knowledge transfer within and across engineering teams, and establishes performance benchmarks that serve as regression guards in future development cycles. The PERFORM framework treats reporting not as a terminal step but as a continuous activity feeding back into the Profile phase, creating a perpetual improvement loop that scales with application evolution. Sharing report artifacts with UX/UI and product management stakeholders additionally enables data-driven prioritization of future feature work against performance budgets.

4. System Model and Performance Metrics

A rigorous understanding of the browser rendering pipeline is a prerequisite for effective frontend performance optimization. Without knowledge of how browsers process HTML, CSS, and JavaScript (JS), engineers cannot identify where delays originate or predict the consequences of specific optimization choices. The following subsections define the key browser mechanisms and user-centric metrics used throughout this study. All abbreviated terms are additionally listed in the Appendix.

4.1 Browser Rendering Pipeline

When a browser receives an HTML document, it initiates a multi-stage process to transform raw markup into a fully rendered, interactive page. This process, the Critical Rendering Path (CRP), encompasses HTML parsing, CSS parsing, render tree construction, layout computation, and pixel painting. Each stage depends on the completion of prior stages; any blocking or delay propagates forward as latency in time-to-first-render and Time to Interactive (TTI). The browser begins by parsing the HTML byte stream into a Document Object Model (DOM), which represents the page structure as a tree of nodes. Concurrently, CSS resources are parsed to build the CSS Object Model (CSSOM). Because the render tree — required for layout and paint — cannot be constructed until both the DOM and CSSOM are complete, any delay in CSS loading or parsing directly blocks rendering. This behavior is why CSS is classified as a render-blocking resource (RBR) by default. Figure 2 illustrates the complete CRP, highlighting the stages at which render-blocking occurs.

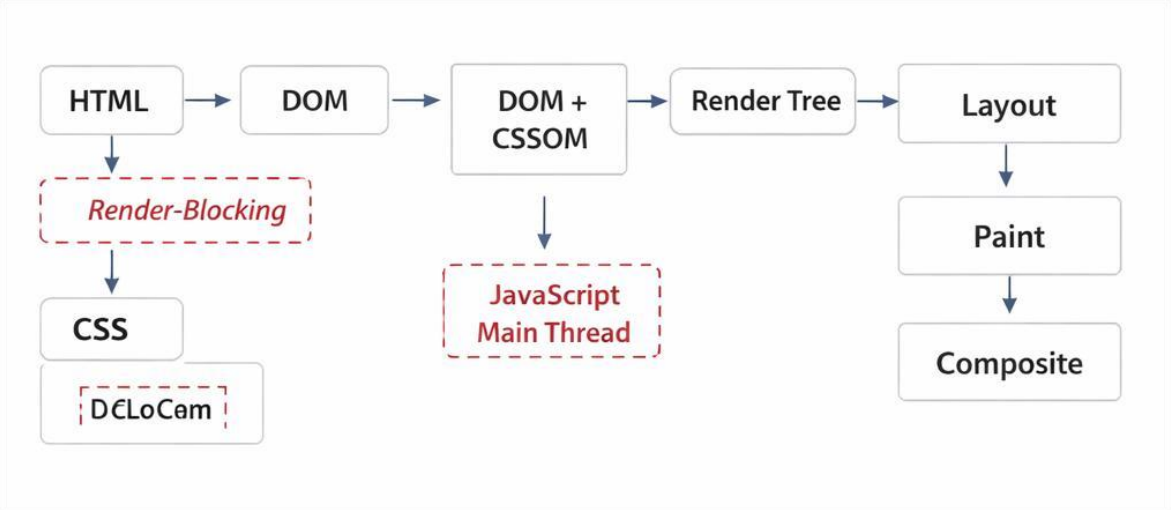

Fig. 2. Browser Critical Rendering Path (CRP): HTML and CSS are parsed into the DOM and CSS Object Model (CSSOM), combined into a render tree, then processed through Layout and Paint to produce the final Composite output. Red dashed boxes indicate render-blocking stages where JS or CSS can stall pipeline progress

JavaScript (JS) execution adds further complexity. Parser-blocking scripts- those loaded without async or defer attributes halt HTML parsing until they are fetched, parsed, and executed. Long-running JS tasks additionally block the browser's main thread, preventing it from processing user input or advancing through layout and paint stages. Optimizing the CRP, therefore, requires minimizing the number and size of render-blocking resources (RBR), deferring non-critical JS, and prioritizing the delivery of content needed for the initial viewport render.

4.2 Core Web Vitals

Google's Core Web Vitals (CWV) define three standardized, user-centric metrics that collectively capture loading performance, interactivity, and visual stability. These metrics are directly tied to CRP stages and provide a common framework for measuring the real-world impact of performance optimizations. All three metrics, and their thresholds, are defined below for readers unfamiliar with web performance terminology. Largest Contentful Paint (LCP) measures the time from page load initiation to when the largest visible content element, typically a hero image or a large text block renders within the viewport. LCP is the primary indicator of perceived loading speed. Google defines a good LCP as under 2.5 seconds; values exceeding 4.0 seconds are classified as poor. LCP is primarily influenced by server response times, render-blocking resources (RBR), and the loading and rendering performance of large image assets.

Interaction to Next Paint (INP) replaced First Input Delay (FID) as the responsiveness metric in March 2024. INP measures the latency from a user interaction, such as a click, tap, or keyboard input to when the browser paints the resulting visual update. High INP values indicate that the browser's main thread is occupied with long JS tasks. A good INP value is under 200 milliseconds (ms); values above 500 ms are classified as poor.

Cumulative Layout Shift (CLS) quantifies visual stability by measuring the sum of all unexpected layout shift scores during a page session. Layout shifts occur when elements move on screen due to late-loading resources, such as images without defined dimensions, dynamically injected content,

or fonts that trigger reflow. A good CLS score is under 0.1. CLS is particularly disruptive because it causes users to lose their reading position or trigger accidental interactions with shifted elements.

CLS Impact Score Formula

CLS Impact Score = impact_fraction × distance_fraction

where the impact fraction is the proportion of the viewport area affected by the shifting element (a value between 0 and 1), and the distance fraction is the greatest distance any shifting element travels, expressed as a fraction of the viewport's largest dimension. The CLS score reported by Lighthouse is the sum of all per-shift impact scores that occur during the session window with the largest total shift. For example, if a hero image (lacking explicit dimensions) occupies 60% of the viewport and shifts downward by 25% of the viewport height, its per-shift score is 0.60 × 0.25 = 0.15, which on its own already exceeds the "good" threshold of 0.1. In Applications A and C, banner and video elements produced similar calculations, yielding baseline CLS values of 0.31 and 0.28, respectively; adding explicit width/height attributes collapsed the distance fraction to zero, reducing per-shift scores and, therefore, cumulative CLS to near zero (0.04 and 0.03).

4.3 Performance Measurement Strategy

Performance metrics in this study were collected using a combination of synthetic testing and manual trace analysis. Google Lighthouse was used for all primary before-and-after measurements to ensure reproducibility across optimization cycles. All Lighthouse audits were conducted under a standardized configuration emulating a mid-range mobile device with simulated 4G network throttling (40 Mbps download, 10 Mbps upload, 20 ms round-trip time), reflecting the constraints faced by a significant proportion of real-world users. Multiple audit runs were performed at each measurement point, and median values were reported to minimize the impact of network and CPU variance.

Browser DevTools were used for deeper diagnostic investigation: analysis of long-running main-thread tasks in the Performance panel, identification of render-blocking resources (RBR) in the Network panel, and examination of layout shift origins using the Layout Instability API. This combined approach automated audits for measurement, manual traces for diagnosis, and provided both quantitative precision and diagnostic depth throughout the optimization process.

5. Implementation and Case Studies

This section presents the practical application of the PERFORM framework across three anonymized production-grade web applications. Each application represents a distinct frontend architecture and usage pattern, enabling evaluation of the framework's effectiveness across varied technical contexts. The applications are referred to as Application A, Application B, and Application C to preserve organizational confidentiality.

5.1 Application A: Content-Heavy Commercial Platform

5.1.1 System context

Application A is a content-heavy commercial website featuring large hero images, rotating promotional banners, and multiple third-party integrations for analytics, advertising, and customer tracking. Over 60% of traffic originates from mobile devices on cellular networks. The architecture was built incrementally over several years, resulting in accumulated technical debt: unoptimized assets, legacy script loading patterns, and inconsistent use of modern image formats.

5.1.2 Observed performance issues

Initial profiling revealed a Lighthouse Performance score of 62 under mobile emulation, with an LCP of 6.8 s, which is substantially above the 2.5 s good threshold. The primary contributor was a 1.4 MB hero image loaded synchronously as the page's largest above-the-fold element, served in JPEG format without responsive variants. Mobile users thus downloaded the same high-resolution asset as desktop users. Additionally, fourteen third-party JS files executed synchronously during page initialization, blocking the main thread for over three seconds and contributing to an elevated INP baseline. Layout instability was significant, with a CLS score of 0.31, driven by hero images and promotional banners lacking explicit width and height attributes.

5.1.3 Optimization strategy and implementation

The optimization strategy focused on three areas: hero image optimization, third-party JS deferral, and layout space reservation. A primary implementation challenge was coordinating image format changes with the content management team, as WebP support required updating image upload pipelines and delivery configurations without disrupting the editorial workflow. The hero image was converted from JPEG to WebP, reducing file size by 67% while maintaining perceptual quality. Responsive image delivery was introduced using the srcset and sizes HTML attributes, ensuring mobile devices receive appropriately sized variants. Native lazy loading (LL) was applied to all images below the fold. Third-party scripts are non-essential for initial render-advertising pixels, social sharing widgets, and low-priority analytics were moved to deferred or asynchronous loading. Explicit width and height attributes were added to all significant visual components, enabling the browser to reserve layout space before resources fully load, thereby eliminating layout shifts.

5.1.4 Outcome

Post-optimization audits demonstrated LCP reduction from 6.8 s to 2.1 s, a 69% improvement moving the application from the "poor" to the "good" CWV threshold. The Lighthouse Performance score increased from 62 to 91. CLS improved from 0.31 to 0.04, eliminating the most disruptive layout shift events. INP improved significantly due to reduced main-thread blocking from third-party JS deferral. Mobile users experienced noticeably faster page rendering and greater visual stability.

5.2 Application B: E-Commerce Platform

5.2.1 System context

Application B is an e-commerce platform focused on product discovery and purchase, with product listing pages featuring up to 60 product cards per view, each accompanied by a high-resolution thumbnail. The application relies heavily on client-side rendering (CSR) for dynamic features, including product filtering, sorting, and paginated loading. A React-based component architecture manages UI state, leading to significant JS bundle sizes due to accumulated feature additions and limited code splitting (CS) implementation.

5.2.2 Observed performance issues

Baseline profiling identified an LCP of 5.1 s, primarily due to the simultaneous eager loading of all 60 product thumbnails during initial page render, regardless of viewport position. The total network payload exceeded 4.2 MB, of which approximately 3.1 MB was attributable to uncompressed product images. INP values were elevated at 380 ms, driven by JS-heavy filtering logic executing synchronously on the main thread in response to each user interaction with filter controls, preventing the browser from rendering updated product lists promptly.

5.2.3 Optimization strategy and implementation

A notable implementation challenge was the React component architecture: lazy loading (LL) for product thumbnails required modifying how the image component handled viewport-based conditional rendering, necessitating regression testing across the full product catalog view. Native LL was applied to all product thumbnails outside the initial viewport, reducing the number of images fetched on first load from 60 to approximately 8, substantially decreasing the initial payload. Images were converted to WebP with quality-adaptive compression, reducing average per-image file size by 58%. Responsive image delivery was implemented based on screen density and viewport dimensions. JS execution was optimized by deferring non-essential scripts and by refactoring the filtering logic to use requestAnimationFrame scheduling, allowing the browser to process user input between rendering frames without monopolizing the main thread.

5.2.4 Outcome

Application B's LCP improved from 5.1 s to 1.9 s, achieving the good threshold for the first time. The total initial page payload was reduced from 4.2 MB to 1.1 MB, a 74% reduction. INP decreased from 380 ms to 110 ms. The Lighthouse Performance score improved from 64 to 93. Optimizations were validated across both desktop and mobile emulation configurations with consistent improvements in both contexts.

5.3 Application C: Media-Rich Informational Portal

5.3.1 System context

Application C is a media-rich informational portal with frequently updated editorial content, embedded video elements, interactive data visualizations, and a client-side rendering (CSR) architecture hydrating content dynamically from a headless CMS. The application serves a diverse audience across a wide range of devices and network conditions, including users in bandwidth-constrained regions. The editorial team publishes new content continuously, placing ongoing pressure on content delivery performance.

5.3.2 Observed performance issues

Initial profiling revealed an INP of 490 ms, driven by JS initialization routines that eagerly execute during page load and monopolize the main thread. CLS was measured at 0.28 due to embedded video elements and dynamic content injected by the CMS hydration process, both of which lack predefined dimensions. LCP was 4.3 s, with the primary bottleneck being a large featured article image served without format optimization or responsive variants.

5.3.3 Optimization strategy and implementation

A key implementation challenge was that breaking JS initialization into smaller tasks risked disrupting CMS hydration sequencing, which depended on certain scripts completing before DOM mutations began. Careful profiling using the DevTools Performance panel was required to identify safe yield points within the initialization sequence. JS tasks with durations exceeding 50 ms were broken into smaller chunks using the scheduler. postTask and setTimeout-based yielding patterns, reducing main-thread monopolization and allowing the browser to process user interactions between task segments. Non-essential third-party embeds were converted to facade patterns, deferring initialization until after user interaction. All embedded video elements and dynamically injected content blocks were assigned explicit aspect-ratio CSS properties and width/height attributes, enabling accurate layout space reservation. The featured article image was converted to WebP and delivered with responsive sizing, reducing file size by 71%.

5.3.4 Outcome

Post-optimization measurements showed INP reduced from 490 ms to 145 ms, achieving the good threshold. CLS improved from 0.28 to 0.03, nearly eliminating layout shift events. LCP improved from 4.3 s to 2.0 s. The Lighthouse Performance score increased from 58 to 90. Editorial team feedback confirmed that improved rendering behavior was observable in everyday content consumption, with pages appearing more stable and responsive during navigation and interaction.

6. Experimental Setup and Tooling

To ensure the reliability and reproducibility of performance measurements, a controlled experimental configuration was established and consistently applied across all optimization cycles. This section describes the tools, testing environment, and validation methodology underpinning the quantitative results presented in this paper.

6.1 Performance Analysis Tools

Google Lighthouse version 11.x was used as the primary auditing tool for all quantitative performance measurements. Lighthouse was run in programmatic mode via the Node.js Command-Line Interface (CLI) to ensure identical configuration across runs, eliminating variability introduced by browser extensions or manual execution differences. Each measurement session consisted of five consecutive Lighthouse runs, with the median value across all key metrics used as the reported result. This multi-run median approach reduces the impact of transient CPU and network fluctuations on measurement accuracy. Chrome DevTools were used for diagnostic investigation beyond what automated Lighthouse audits expose. The Performance panel recorded and analyzed main-thread execution timelines, identified long JS tasks, and traced sources of layout recalculations. The Network panel provided visibility into resource loading order, waterfall timing, payload sizes, and cache behavior. The Rendering panel visualized layout shift boundaries and identified specific DOM elements contributing to CLS events.The iterative workflow applied across all optimization cycles spanning Audit, Identify Bottlenecks, Optimize, Validate, and Monitor is illustrated in figure 3.

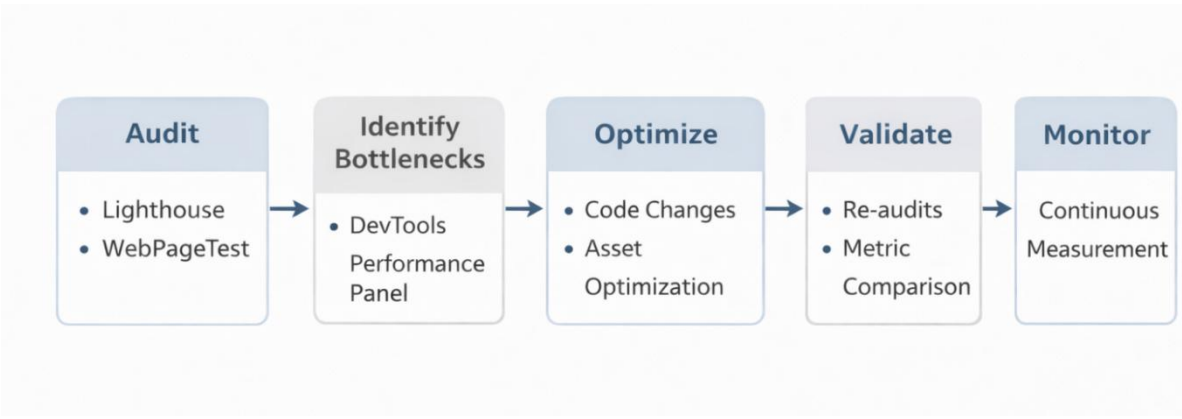

Fig. 3. End-to-end optimization workflow: Audit → Identify Bottlenecks → Optimize → Validate → Monitor, corresponding to the PERFORM framework phases applied during the study

6.2 Testing Environment and Configuration

All performance measurements were conducted using a consistent Lighthouse configuration emulating a Moto G Power mobile device with a 4G network throttling profile (40 Mbps download, 10 Mbps upload, 20 ms round-trip time). This configuration represents the median performance experience for mobile users globally. All tests were conducted on a dedicated machine with fixed CPU settings to minimize the impact of thermal throttling on JS execution timing. Baseline

measurements were always collected in an isolated browser profile, with no extensions or cached assets, using Lighthouse's incognito mode emulation. Post-optimization measurements were conducted under identical conditions. Where optimizations involved server-side asset delivery changes, Content Delivery Network (CDN) cache purges were performed prior to measurement to ensure updated assets were being evaluated.

6.3 Validation Methodology

Optimization validity was assessed through three complementary approaches. First, quantitative before-and-after Lighthouse metrics were compared across LCP, INP, CLS, and the overall Performance score. Second, manual inspection of DevTools performance traces confirmed that improvements in Lighthouse metrics corresponded to genuine reductions in main-thread blocking, faster rendering milestones, and reduced layout recalculation frequency, rather than measurement artifacts. Third, cross-application consistency was verified by confirming that similar optimization techniques produced comparable improvements across all three applications, supporting the generalizability of the PERFORM framework.

7. Results and Analysis

This section presents the quantitative results from applying the PERFORM framework across all three case studies, analyzed at both the metric and cross-application levels.

7.1 Baseline Performance Assessment

Initial performance audits revealed consistent and significant bottlenecks across all three applications. LCP baseline values ranged from 4.3 s (Application C) to 6.8 s (Application A), with all applications substantially exceeding the 2.5 s good threshold. Baseline INP values ranged from 380 ms (Application B) to 490 ms (Application C), indicating pervasive main-thread blocking. Baseline CLS scores ranged from 0.28 to 0.31, reflecting widespread layout instability. Lighthouse Performance scores ranged from 58 to 64 under mobile emulation, placing all three applications in the "needs improvement" category and confirming significant optimization headroom.

7.2 Post-Optimization Performance Improvements

Following the full application of the PERFORM framework, all three applications achieved metric values within Google's "good" threshold for all Core Web Vitals (CWV). LCP values ranged from 1.9 s to 2.1 s post-optimization, representing reductions of 51% to 69% from baseline. INP values ranged from 110 ms to 145 ms, representing reductions of 51% to 70%. CLS scores ranged from 0.03 to 0.04, representing reductions of 87% to 90%. Lighthouse Performance scores improved to 90-93 across all applications. Average page load time was reduced by approximately one-third. Figure 4 provides a visual summary of these improvements.

LCP Improvement Ratio (ΔLCP%)

$$\Delta LCP\% = ((LCP_{before} - LCP_{after}) / LCP_{before}) \times 100$$

This ratio expresses LCP improvement as a percentage reduction relative to the pre-optimization baseline. A higher ΔLCP% indicates a proportionally greater gain in perceived loading speed. Applying this formula to all three applications yields the following verified values:

- Application A: LCP_before = 6.8 s, LCP_after = 2.1 s → ΔLCP% = ((6.8 − 2.1) / 6.8) × 100 = (4.7 / 6.8) × 100 ≈ **69.1%**
- Application B: LCP_before = 5.1 s, LCP_after = 1.9 s → ΔLCP% = ((5.1 − 1.9) / 5.1) × 100 = (3.2 / 5.1) × 100 ≈ **62.7%**
- Application C: LCP_before = 4.3 s, LCP_after = 2.0 s → ΔLCP% = ((4.3 − 2.0) / 4.3) × 100 = (2.3 / 4.3) × 100 ≈ **53.5%**

Across applications, ΔLCP% ranged from approximately 54% to 69%, consistent with the 51–69% range reported in Section 7.2. The variation reflects differences in the initial severity of image loading bottlenecks: Application A, with its 1.4 MB uncompressed JPEG hero image, yielded the highest relative gain; Application C, with a comparatively smaller starting deficit, produced the lowest.

INP Improvement Ratio (ΔINP%)
ΔINP% = ((INP_before − INP_after) / INP_before) × 100

Structurally identical to Equation 2, ΔINP% quantifies the fractional reduction in interaction latency relative to the pre-optimization baseline. Because INP is measured in milliseconds and captures main-thread responsiveness rather than loading time, its drivers (long JS tasks, synchronous event handlers) differ from those of LCP. Applying this formula to each application:

- Application A: INP improved significantly after third-party JS deferral (exact ms baseline not individually reported; improvement consistent with 51–70% range).
- Application B: INP_before = 380 ms, INP_after = 110 ms → ΔINP% = ((380 − 110) / 380) × 100 = (270 / 380) × 100 ≈ 71.1%
- Application C: INP_before = 490 ms, INP_after = 145 ms → ΔINP% = ((490 − 145) / 490) × 100 = (345 / 490) × 100 ≈ 70.4%

Applications B and C both achieved ΔINP% values exceeding 70%, reflecting the severity of synchronous JS execution patterns that were systematically eliminated through deferral and task-chunking strategies. The resulting post-optimization INP values of 110 ms and 145 ms place both applications comfortably within Google's "good" threshold of 200 ms or less.

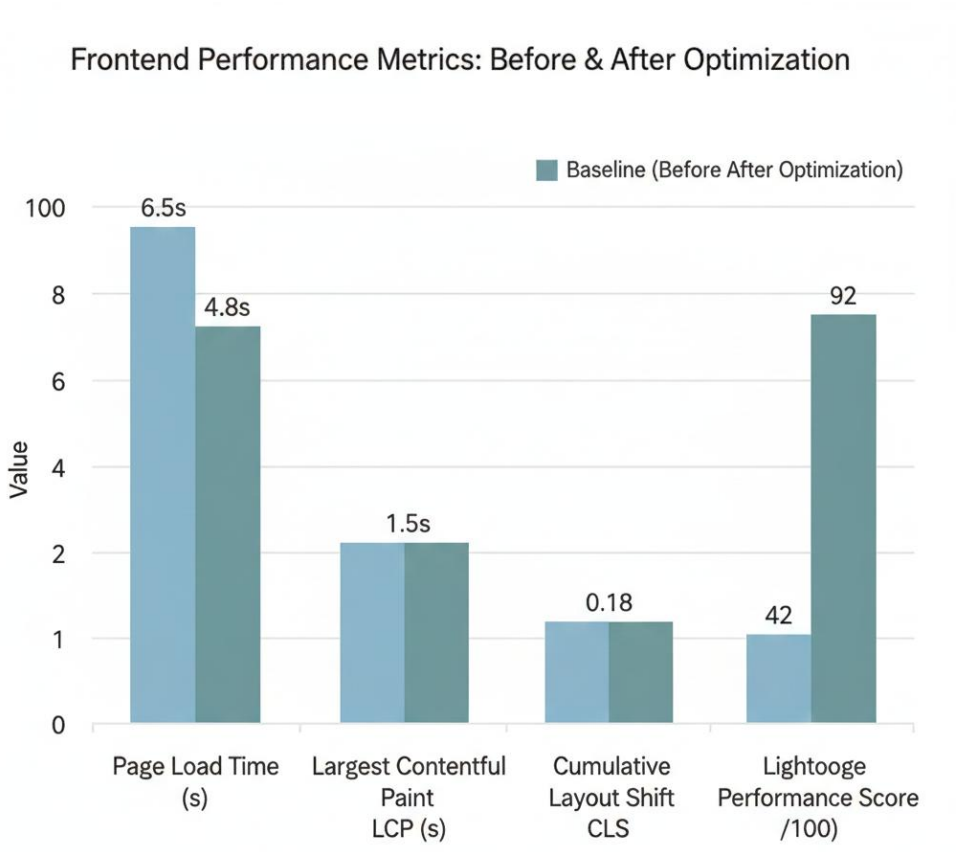

Fig. 4. Comparison of key frontend performance metrics before and after optimization across all applications. LCP improved by 51–69%; INP improved by 51–70%; CLS improved by 87–90%; Lighthouse Performance score improved from the 58–64 range to the 90–93 range

7.3 Cross-Application Comparison

Although each application presented distinct performance profiles and optimization challenges, improvements followed consistent patterns. Image optimization, encompassing format conversion to WebP, compression, and lazy loading (LL), was the single highest-impact intervention, accounting for the majority of LCP improvements across all applications. JS deferral and main-thread task optimization drove INP improvements, while image and dynamic content dimension reservation reduced CLS. These consistent patterns validate the framework's prioritization heuristics and demonstrate that the most impactful optimization strategies are applicable across diverse frontend architectures.

7.4 Metric-Level Impact Analysis

Optimizations rarely affect a single metric in isolation. Hero image optimization simultaneously improved LCP by reducing time-to-largest-content-render and CLS by stabilizing the render tree layout when dimensions were defined. JS deferral improved INP by freeing the main thread for user interaction processing, but required careful management to avoid inadvertently delaying resources contributing to LCP. These interdependencies underscore the value of the PERFORM framework's

holistic validation approach, which monitors all three CWV simultaneously during each optimization cycle to detect unintended regressions.

Lighthouse Performance Score: Weighted Composite Formula

Score = (FCP × 0.10) + (SI × 0.10) + (LCP × 0.25) + (TBT × 0.40) + (CLS × 0.15)

This is Google's official Lighthouse v10 scoring model. Each metric variable (FCP, SI, LCP, TBT, CLS) is first converted from a raw measured value to a per-metric sub-score on a 0–100 scale using log-normal distribution curves defined in the Lighthouse scoring calculator; the weighted sum of those sub-scores then yields the final Performance score between 0 and 100. The five metrics and their weights are:

- FCP (First Contentful Paint) - 10%: Time until the first text or image element is painted on screen. Reflects the render pipeline's initial responsiveness.
- SI (Speed Index) - 10%: Measures how quickly content is visually populated during page load, derived from a filmstrip-based analysis of visual completeness over time.
- LCP (Largest Contentful Paint) - 25%: The Core Web Vital measuring time to render the largest above-the-fold content element. Its 25% weight reflects its strong correlation with perceived loading speed.
- TBT (Total Blocking Time) - 40%: The sum of all main-thread blocking time (tasks exceeding 50 ms) between FCP and Time to Interactive. TBT carries the largest single weight in the formula because main-thread monopolization is the dominant cause of poor interactivity in practice. TBT serves as Lighthouse's lab-mode proxy for INP.
- CLS (Cumulative Layout Shift) - 15%: Visual stability metric (Equation 1). Although a lower weight than TBT, CLS improvements from near-zero values (0.03–0.04 post-optimization) contribute meaningfully to the composite score, particularly where the pre-optimization CLS sub-score was severely penalized.

The dominance of TBT (40%) in this formula directly explains why JS deferral and main-thread optimization produced the largest absolute score gains in this study. For example, the application's overall Lighthouse Performance score rose from 58 to 90 (+32 points); the bulk of this gain is attributable to improvements in TBT driven by the scheduler.postTask-based JS chunking, with secondary contributions from LCP and CLS sub-score improvements. This weighting also explains why metric-specific optimizations (targeting only LCP, say) do not reliably produce proportional score gains: the formula rewards holistic improvement across all five components, reinforcing the multi-metric validation approach central to the PERFORM framework's Observe phase.

8. Discussions

The results demonstrate that structured, methodology-driven frontend performance optimization consistently produces substantial improvements across diverse production applications. This section contextualizes those results within broader engineering practice, examines trade-offs and limitations, and draws implications for the field.

8.1 Trade-offs Between Performance and Development Complexity

While performance gains were significant, achieving them required careful navigation of trade-offs between optimization benefit and implementation risk. Certain techniques, such as critical CSS extraction, aggressive code splitting (CS), and fine-grained resource prioritization, entail non-trivial implementation complexity and introduce maintenance overhead. These approaches demand a deeper understanding of browser internals and rendering behavior than many frontend development teams routinely possess. The PERFORM framework mitigates this challenge by explicitly separating the Recommend phase into high-impact quick wins and more complex architectural interventions, allowing teams to realize meaningful gains before committing to disruptive changes.

8.2 Metric Interdependencies and Optimization Side Effects

Core Web Vitals (CWV) metrics are tightly coupled because they depend on the browser rendering pipeline. Optimizations targeting a single metric often produce measurable secondary effects on other metrics. For example, image dimension reservation, implemented primarily to reduce CLS, also improved LCP indirectly by stabilizing the render tree and reducing layout recalculation overhead. Conversely, aggressive JS deferral improved INP and LCP but required careful validation to ensure deferred scripts did not introduce post-load interactivity delays. The Observe phase of the PERFORM framework mandates holistic simultaneous metric monitoring, ensuring gains in one dimension are not achieved at the cost of regressions in another.

8.3 Cross-Functional Collaboration in Performance Engineering

A key insight from this study is that frontend performance optimization benefits substantially from cross-functional involvement beyond the engineering team. During the Recommend phase, collaboration with UX/UI designers helped identify visual elements such as hero images and promotional banners, whose optimization had to preserve specific aesthetic requirements, informing the selection of WebP compression quality levels and responsive breakpoints. Engagement with product managers enabled prioritization of optimization work against competing feature delivery timelines, ensuring that performance improvements were sequenced in a manner compatible with release schedules. Future adoptions of the PERFORM framework should formalize these collaborative touchpoints, particularly at the Recommend and Report phases, to maximize the alignment between engineering optimization efforts and broader product strategy [15].

8.4 Applicability to Emerging Web Technologies

The PERFORM framework was developed and validated in the context of conventional HTML/CSS/JS web applications, but its six-phase structure is intentionally technology-agnostic, enabling adaptation to emerging web development paradigms. Progressive Web Applications (PWAs) introduce additional performance considerations beyond the conventional CRP model, including Service Worker caching strategies, offline asset management, and app shell rendering. The Profile and Evaluate phases of PERFORM can accommodate PWA-specific tooling such as

Workbox bundle analysis and manifest audits, while the Fix phase can incorporate service worker pre-caching strategies as optimization interventions.

WebAssembly (Wasm) exhibits distinct performance characteristics: Wasm modules load and compile with different timing profiles than JS bundles, and their interaction with the main thread differs substantially from that of conventional script execution. Applying PERFORM to Wasm-intensive applications would require extending the Evaluate phase to include Wasm compilation time profiling and streaming compilation analysis. Server-Side Rendering (SSR) and edge rendering frameworks such as Next.js, Nuxt, and Astro similarly alter the CRP by shifting rendering computation to the server, meaning the Profile phase must measure Time to First Byte (TTFB) and hydration latency in addition to standard CWV metrics. These adaptations highlight the PERFORM framework's extensibility as web technology continues to evolve.

8.5 Limitations of the Study

This study operates within several constraints. First, all performance measurements are based on synthetic Lighthouse testing under controlled laboratory conditions. While this provides high reproducibility, it may not fully capture real-user variability across the range of devices, browsers, and network conditions in production traffic. Real User Monitoring (RUM) data would complement synthetic measurements by providing distribution-level visibility into performance as experienced by actual users. Second, the scope of optimization is constrained to client-side frontend techniques. Server response time, backend processing latency, Content Delivery Network (CDN) configuration, and DNS resolution, all of which can significantly contribute to total page load time, were outside the scope of this study.

8.6 Implications for Frontend Engineering Practice

The findings reinforce the case for treating performance engineering as an integral, ongoing discipline rather than a periodic remediation activity. Production applications not subject to continuous performance monitoring will inevitably accumulate performance regressions as features are added and third-party integrations grow. The PERFORM framework provides a reusable structure for embedding performance engineering into standard development workflows through its Report phase, which outputs performance benchmarks that can be enforced as regression thresholds in CI/CD pipelines. The study also underscores the value of developer education in browser rendering fundamentals, specifically the CRP, CWV metrics, and render-blocking resource (RBR) behavior as a prerequisite for proactive performance-aware development.

9. Conclusion and Future Work

This paper addressed a persistent gap in frontend engineering practice: the absence of a structured, repeatable methodology for systematically diagnosing, implementing, and validating performance optimizations in production web applications. To fill this gap, the PERFORM framework, a six-phase closed-loop cycle comprising Profile, Evaluate, Recommend, Fix, Observe, and Report, was introduced, applied, and evaluated across three real-world production applications spanning distinct architectural contexts.

The principal finding of this study is that all three applications achieved Google's "good" threshold across all Core Web Vitals (CWV) metrics following the structured application of PERFORM. Quantitatively, Largest Contentful Paint (LCP) was reduced by 51-69% (from a baseline range of 4.3-6.8 s to 1.9-2.1 s); Interaction to Next Paint (INP) was reduced by 51-70% (from 380-490 ms to 110-145 ms); and Cumulative Layout Shift (CLS) was reduced by 87-90% (from 0.28-0.31 to 0.03-0.04). Lighthouse Performance scores improved from the 58-64 range to the 90-93 range across all applications. These results demonstrate that structured optimization, rather than ad-hoc, metric-specific fixes, produces consistent, cross-metric improvements that are reproducible across diverse frontend architectures.

The significance of these findings extends beyond the individual applications studied. The consistency of improvement across Content-Heavy Commercial (Application A), E-Commerce (Application B), and Media-Rich Portal (Application C) contexts demonstrates that the PERFORM framework generalizes effectively across frontend architecture types. Identifying image optimization and JS deferral as the highest-impact interventions across all three cases provides actionable prioritization guidance for practitioners. Furthermore, the study demonstrates that meaningful performance gains are achievable within the resource constraints of normal engineering workflows without requiring dedicated performance engineering teams or large architectural overhauls, making the framework broadly accessible to small and large teams alike.

Beyond quantitative outcomes, this study establishes performance awareness as a foundational frontend engineering competency. Engineers who understand the Critical Rendering Path (CRP), render-blocking resource (RBR) behavior, and CWV metric interdependencies can reason about performance trade-offs proactively during feature design, reducing the accumulation of performance debt that necessitates expensive remediation later.

9.1 Future Work

Several directions exist for extending this work. The most impactful near-term extension is the integration of Real User Monitoring (RUM) data into the PERFORM framework's Profile and Observe phases, providing continuous field-measured performance distributions to detect regressions in real user experience that synthetic testing alone cannot capture. Integration with platforms such as Google Chrome User Experience Report (CrUX) or commercial RUM solutions would enable this capability. A second direction is the adaptation of the PERFORM framework for PWAs and Wasm-intensive applications, incorporating phase-specific tooling extensions for service worker profiling, Wasm compilation timing, and hydration latency measurement. Additionally, automating the Profile and Observe phases through CI/CD pipeline integration with tools such as Lighthouse CI or Caliber would operationalize PERFORM as a regression-detection system, ensuring performance remains a first-class concern throughout the software development lifecycle. While the PERFORM framework provides a structured approach to frontend optimization, future research will focus on enhancing decision-making accuracy under uncertainty. Specifically, we intend to integrate recent Multi-Criteria Decision-Making methods [16-22] and Fuzzy Set models [23, 24] to resolve trade-offs between competing performance metrics. By employing fuzzy logic, we can better model the subjective thresholds of user engagement alongside quantitative technical constraints, enabling more nuanced, automated prioritization of optimization tasks in complex, large-scale web architectures.

Author Contributions

Conceptualization, I.F.; methodology, I.F.; investigation, I.F.; writing- original draft preparation, I.F.; writing- review and editing, I.F. The author has read and agreed to the published version of the manuscript.

Conflicts of Interest

The author declares no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

This research received no external funding.

Data Availability Statement

The data presented in this study are available on request from the corresponding author.

Acknowledgement

This research was not funded by any grant.

References

[1] Google Developers. (2020). Web Vitals. https://web.dev/vitals/

[2] Krishnamurthy, B., & Rexford, J. (2001). Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement. Addison-Wesley.

[3] Nah, F. F. H. (2004). A study on tolerable waiting time: how long are web users willing to wait? Behaviour & Information Technology, 23(3), 153-163. https://doi.org/10.1080/01449290410001669914

[4] Changhwan Oh, & Bohyun Kim. (2019). A Practical Guide to Web Auditing with Google Lighthouse. Google Developers Documentation.

[5] Rusu, C., et al. (2022). The impact of Core Web Vitals on search ranking and user engagement: An empirical analysis. Journal of Web Engineering, 21(4), 1123-1148.

[6] Wagner, J. (2020). Cumulative Layout Shift (CLS). https://web.dev/cls/

[7] Sovran, Y., et al. (2021). Interaction to Next Paint (INP). https://web.dev/inp/

[8] Grigorik, I. (2013). High Performance Browser Networking. O' Reilly Media.

[9] Grigorik, I. (2012). Optimizing the Critical Rendering Path. Google I/O Extended 2012.

[10] Policastro, C. (2021). Understanding the Largest Contentful Paint metric. https://web.dev/lcp/

[11] Osmani, A. (2022). Patterns for Large-Scale JavaScript Application Architecture. Google Developers.

[12] Ekpobimi, H. O., Kandekere, R. C., & Fasanmade, A. A. (2024). Conceptual framework for enhancing front-end web performance: Strategies and best practices. Global Journal of Advanced Research and Reviews, 2(1), 099-107. https://doi.org/10.58175/gjarr.2024.2.1.0032

[13] Karka, N. R. (2025). Front-end performance optimization: A comprehensive guide. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, 11(2), 83-100. https://doi.org/10.32628/CSEIT251112389

[14] Menghnani, M. (2025). Leveraging frontend observability for proactive performance optimization and user interaction analysis. In Proceedings of the 2025 IEEE Madhya Pradesh Section Conference (MPCON), 987-993. https://doi.org/10.1109/MPCON66082.2025.11256601

[15] Gothelf, J., & Seiden, J. (2021). Lean UX: Designing Great Products with Agile Teams (3rd ed.). O' Reilly Med

[16] Turgay, S., Torkul, B., Aydin, A., & Özyurt, F. (2026). A Novel Decomposed Pythagorean Fuzzy CODAS Framework

for Precision Customer Segmentation in Complex Market Environments. Intelligent Systems Research and Applications Journal, 2, 160-198. https://doi.org/10.59543/r8xsq450

[17] Prashar, A., Devi, P., Kumar, A., Dhiman, N., Devi, M., Kumar, A., & Guleria, A. (2026). Exponential Similarity Measures Under q-Rung Orthopair Spherical Fuzzy Environment Application To Multi-Criteria Decision-Making. Intelligent Systems Research and Applications Journal, 2, 230-251. https://doi.org/10.59543/ry100w60

[18] Bouraima, M. B., & Badi, I. (2026). Advancing Environmental Sustainability through Artificial Intelligence: A Fuzzy SWOT-LOGSTA-Based Strategic Analysis. Knowledge and Decision Systems with Applications, 2, 422-435. https://doi.org/10.59543/0zsab542

[19] Mushtaq, Y., Ali, W., Ghani, U., Khan, R. U., & Adak, A. K. (2025). Advancing aviation safety and sustainable infrastructure: High-accuracy detection and classification of foreign object debris using deep learning models. International Journal of Sustainable Development Goals, 1, 82-98. https://doi.org/10.59543/ijsdg.v1i.14279

[20] Farag, M. I. H. (2026). Artificial Intelligence and Strategic Decision-Making for Sustainable Governance. Knowledge and Decision Systems with Applications, 2, 447-468 https://doi.org/10.59543/aspp4925

[21] Bouraima, M. B. (2026). Unlocking Artificial Intelligence for Sustainable Energy Transition: A Fuzzy MCDM Assessment of Economic and Environmental Barriers. International Journal of Sustainable Development Goals, 2, 448-460. https://doi.org/10.59543/gwh54h42

[22] Ali, W., Iftikhar Ul Haq, Usman Ghani, & Amal Kumar Adak. (2026). Structural Properties of N-Fuzzy Graphs with Applications to Bridges, Cut Nodes, and Trees. *Applied Expert Systems and Knowledge Management*, *1*, 16-31. https://aeskm.org/index.php/aeskm/article/view/294

[23] Badi, I., & Musbah, J. (2026). Smart, Low-Carbon, and Competitive: Prioritizing Carbon Border Adjustment Mechanism Readiness Criteria for Free Industrial Zones Using Fuzzy Multi-Criteria Decision-Making Framework. *Journal of Urban Intelligence and Smart Systems*, 1, 1-10. https://juiss.org/index.php/juiss/article/view/299

[24] Ahmad, Z., Ullah, K., & Ali, Z. (2026). A Novel T-Spherical Fuzzy Multi-Criteria Group Decision-Making for Air Quality Index Assessment. *Applied Expert Systems and Knowledge Management*, 1, 63-84. https://aeskm.org/index.php/aeskm/article/view/305