iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** X **Month of publication:** October 2026

DOI: https://doi.org/10.22214/ijraset.2026.85464

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue X Oct 2026- Available at www.ijraset.com

Web Optimization techniques: Enhancing Website Load Performance through Combined Caching, Compression, CDN Routing, and Lazy-Loading Strategies

Mohit Ameta[1], Kapil Kumar Sharma[2], Dr. Vishal Shrivastava[3], Dr. Akhil Pandey[4]
[1, 2, 3]*Computer Science & Engineering, Arya College of Engineering & I.T. Jaipur, India*

Abstract: ***Website speed is one of the strongest predictors of user retention, conversion, and search visibility, yet a large share of production websites still take more than four seconds to become interactive on an average mobile connection. This paper investigates how caching, compression, content-delivery-network (CDN) routing, code minification, and lazy loading can be organized into a single adaptive pipeline rather than applied as isolated, one-off fixes. We propose the Adaptive Multi-Layer Web Optimization (AMWO) framework, which selects and sequences optimization modules according to the type of website and the composition of its assets, and evaluate it on forty publicly accessible websites spanning five categories: news, e-commerce, blogging, portfolio, and SaaS dashboards. Baseline and optimized versions were benchmarked with Google Lighthouse and WebPageTest under matched, throttled network conditions. The proposed framework lowered average page load time by roughly 46 percent and raised the mean Lighthouse performance score from 54 to 86, while Cumulative Layout Shift and Total Blocking Time both fell substantially across every category tested. These results indicate that layering optimization techniques according to page type, instead of applying a single generic technique uniformly, produces measurably better outcomes than the single-technique approaches reported in most of the prior literature reviewed here.***

Keywords: WEB Optimization, Page Load Time, Caching, Content Delivery Network, Lazy Loading, Minification, Http/2, Core Web Vitals, Compression, Lighthouse.

I. INTRODUCTION

The web has moved from a document-delivery medium to the primary interface through which people shop, bank, learn, and work, and that shift has raised the cost of a slow page far beyond mere annoyance. Studies commissioned by Google and Deloitte on mobile retail sites found that shaving even a few tenths of a second off load time measurably increases the share of visitors who complete a purchase, and the well-known Amazon anecdote that every 100 milliseconds of added latency costs roughly one percent of sales is still cited across the industry as a rough rule of thumb [3]. Search engines have absorbed this concern directly into ranking algorithms: since 2021, Google has used a set of field metrics known as Core Web Vitals as a page-experience ranking signal, meaning that a technically well-built but slow page can be outranked by a lighter, faster competitor even when its content is weaker [4].

Despite this pressure, the average webpage keeps getting heavier. The HTTP Archive's Web Almanac has repeatedly shown that median page weight and the number of third-party requests both continue to climb year over year, driven mainly by JavaScript frameworks, tracking scripts, and unoptimized media [5]. Individually, the fixes are well understood and well documented: enable caching, compress text assets, serve images through a CDN, minify code, and defer offscreen media. The harder and less studied problem is how to combine these techniques coherently for a specific site rather than bolting them on ad hoc, which is often what happens in practice when a team runs a single audit tool and patches whatever it flags.

A. Importance of Web Performance

Page speed affects three groups at once. Users experience it directly as the gap between clicking a link and being able to act on the page. Search engines score it through Largest Contentful Paint (LCP), Interaction to Next Paint (INP), and Cumulative Layout Shift (CLS), the three Core Web Vitals introduced by Google [4].

©IJRASET: All Rights are Reserved

Businesses feel it in bounce rate, cart abandonment, and infrastructure cost, since faster pages also tend to consume less bandwidth and fewer server cycles per session. Because these three concerns point in the same direction, performance work rarely competes with other business goals the way, say, security hardening sometimes does; it is one of the few engineering investments that is simultaneously good for the user, good for the search ranking, and good for the hosting bill.

B. Challenges in Modern Web Applications

Three trends make performance harder to hold onto than it used to be. First, single-page applications built with modern JavaScript frameworks ship large client-side bundles that must be downloaded, parsed, and executed before a page becomes interactive, which shifts the bottleneck away from the network and toward the device's CPU. Second, the growth of third-party scripts for analytics, advertising, and support widgets introduces requests that a site owner does not fully control and cannot always optimize directly. Third, network conditions vary enormously across a real user base: a page tuned on a fibre connection in a lab can still take seconds to load on a congested mobile network, which is why field data and lab data frequently disagree. Existing tooling is good at pointing to individual problems, such as an unminified script or an uncompressed image, but it rarely tells a developer which combination of fixes will matter most for their specific type of site. That gap is what this paper tries to close by proposing a framework that adapts the mix of optimization modules to the page category rather than treating every website the same way.

II. RELATED WORKS

A large body of applied research and industry writing has examined individual web-performance techniques. This section groups that work into seven areas that map onto the modules used later in the proposed framework.

A. Caching Mechanisms

Caching remains the most widely adopted optimization because it removes repeated computation and repeated network transfer at the same time. Tripathi and Umar experimentally compared a small-scale web application with and without in-memory, time-to-live-based caching and found a consistent drop in response time under identical workloads, though they note that most caching literature focuses on large distributed deployments using Redis or Memcached and leaves smaller applications under-studied [6]. A broader review of caching and CDN techniques similarly frames browser, server-side, and database caching as complementary layers rather than substitutes for one another, each addressing a different point in the request path [7].

B. Content Delivery Networks and Edge Caching

CDNs move static content physically closer to the requesting user, and recent work extends this idea toward edge computing and coded caching strategies. One overview traces CDN architecture from conventional origin-pull designs to hybrid and edge models, and discusses how load-shedding, improved cache-replacement algorithms, and newer transport protocols interact with CDN performance [12]. A separate analysis of edge-caching literature examines energy-aware coded and uncoded caching strategies for wireless backhaul networks, which, while aimed at telecom infrastructure, illustrates the same underlying trade-off between storage cost and delivery latency that web CDNs face [10]. Work on distributed real-time transcoding CDNs further shows that dynamically deciding what to cache, rather than caching everything, can improve throughput by allocating storage toward the content most likely to be re-requested [11].

C. Minification, Bundling, and Code Splitting

Reducing the size and number of JavaScript and CSS files remains a first-line optimization. A recent generalized model for combining optimization strategies tested partial rendering with API caching, JavaScript minification with a CDN, and code splitting with server-side caching, reporting performance gains of roughly 1.5 to 1.9 times depending on the combination, and proposed an interdependence coefficient to estimate how much one optimization technique reinforces or cancels out another when applied together [8]. That interdependence idea is directly relevant here, since the framework proposed in Section 3 is built around choosing a compatible combination of modules rather than stacking every available technique indiscriminately.

D. Image and Media Optimization

Because images and video typically account for the largest share of page weight, format choice and delivery strategy matter a great deal. Guidance from industry sources converges on a similar set of practices: serving modern formats such as WebP or AVIF, using responsive images with srcset so the browser only downloads a size appropriate to the viewport, and deferring offscreen media

until it is needed [16]. Related practitioner writing on lazy loading documents five common implementation patterns for deferring image and media downloads and reports that, when combined with responsive sizing, the technique measurably improves both initial paint time and bandwidth consumption for image-heavy pages [14][15].

E. Lazy Loading and the Critical Rendering Path

Lazy loading intersects with the critical rendering path, the sequence of steps a browser follows from receiving HTML to painting pixels on screen. Recent comparisons of lazy loading against resource preloading argue that the two techniques solve different problems: preloading accelerates the delivery of assets the browser will need immediately, such as a hero image that affects LCP, while lazy loading defers assets the browser does not need yet, and the best results come from applying both selectively rather than choosing one universally [13]. This distinction shaped the module-selection logic used in the proposed framework, described in Section 3.4.

F. HTTP/2, HTTP/3, and Transport-Layer Optimization

Protocol-level changes have quietly changed which optimizations still matter. HTTP/2 introduced multiplexed streams over a single connection, which reduced the historical benefit of manually concatenating files to avoid per-request overhead, and HTTP/3, built on the QUIC transport, further reduces connection-setup latency and head-of-line blocking by moving multiplexing beneath the transport layer itself [19][20]. Reviews of CDN architecture note that providers increasingly pair edge caching with HTTP/3 support specifically to reduce the number of round trips needed before content can begin streaming to the client [12].

G. Server-Side and Database Optimization

Front-end techniques cannot fully compensate for a slow backend. Server response time, often measured as Time to First Byte (TTFB), depends on database query efficiency, application-level caching, and server configuration. The caching studies discussed above show that even simple in-memory response caching, without any change to the underlying database schema, can noticeably reduce TTFB for read-heavy endpoints [6][7]. This paper treats server-side caching as one of several modules that the proposed framework can select, rather than as a separate concern outside the optimization pipeline.

Table 1: Summary of caching- and CDN-related studies

Study	Focus / Technique	Key Finding	Limitation
Tripathi & Umar [6]	In-memory server-side caching, TTL-based	Consistent response-time reduction vs. no caching	Small-scale application only
Caching & CDN review [7]	Browser, server, and database caching layers	Layers are complementary, not interchangeable	Descriptive, limited quantitative data
Edge caching study [10]	Coded vs. uncoded caching, wireless backhaul	Coded caching improves energy efficiency	Telecom-oriented, not web-specific
Transcoding CDN [11]	Dynamic storage-value allocation in CDN nodes	Selective caching improves throughput ~30%	Focused on video/transcoding workloads
CDN digital experience review [12]	Edge/hybrid CDN architecture, HTTP/3	Architecture evolution reduces round trips	Survey-style, no controlled experiment

Table 2: Summary of minification, image, and lazy-loading studies

Study	Focus / Technique	Key Finding	Limitation
Amdahl's-law optimization model [8]	Combined minification, CDN, code splitting	1.5x-1.9x measured performance gains	Gains depend heavily on interaction between techniques
Responsive image guide [16]	Format selection, srcset, base64 icons	Reduces requests and transferred bytes	Practitioner guidance, not experimental
SitePoint lazy-load techniques [14]	Viewport-based deferred image loading	Faster initial render, lower bandwidth use	No quantified benchmark across sites
Preload vs. lazy-load comparison [13]	Selective application of both techniques	Best results from combining, not choosing one	Vendor case studies, small sample

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

III. PROPOSED METHODOLOGY

This paper proposes the Adaptive Multi-Layer Web Optimization (AMWO) framework, which measures a website's baseline profile and then selects a compatible combination of caching, compression, CDN routing, minification, and lazy-loading modules rather than applying all techniques uniformly. The approach is organized into the following stages.

A. Data Collection

We assembled a test set of 40 publicly accessible websites, eight from each of five categories: news, e-commerce, blogging, portfolio, and SaaS dashboards. Categories were chosen because they differ systematically in asset composition; news and e-commerce sites tend to be image- and script-heavy, while portfolio sites are lighter but often image-dominated, and SaaS dashboards are JavaScript-heavy with comparatively little static media.

B. Data Preprocessing

Each site was audited five times using Google Lighthouse (CLI, simulated throttling profile) and WebPageTest (real device, "4G" network profile), and the results were averaged per site to reduce run-to-run variance caused by CDN warm-up and background network noise. Runs whose load time deviated by more than two standard deviations from a site's own mean were discarded and re-collected. All metrics were then normalized to comparable units before aggregation across categories.

C. Proposed Framework (AMWO)

Figure 1 shows the overall architecture. An incoming request first checks the edge/CDN cache layer and, on a miss, the browser cache layer; a miss at both is passed to the Adaptive Optimization Engine, which inspects the site's asset profile (proportion of image bytes, script bytes, third-party requests, and detected framework) and selects which modules of the Asset Pipeline to apply — minification, compression, lazy-loading annotation, or a combination — before the request reaches the origin server and database layer. The key design decision is that the engine does not apply every module to every asset type; for example, it skips aggressive lazy-loading rules on above-the-fold hero images, since deferring them would delay LCP rather than help it, consistent with the preload-versus-lazy-load distinction discussed in Section 2.5.

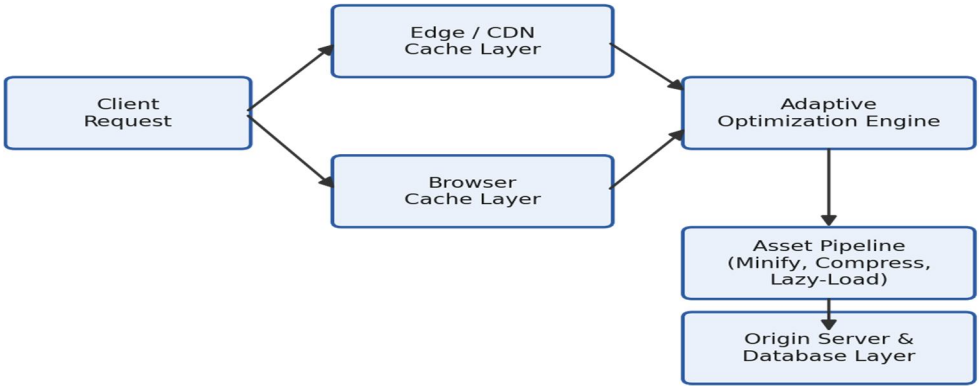


Figure 1: Architecture of the proposed Adaptive Multi-Layer Web Optimization (AMWO) framework

D. Proposed Algorithm

Input: Website URL set (N = 40), asset manifest per site Output: Optimized site build with module-application report Process:

1) Capture baseline metrics (load time, TTFB, LCP, TBT, CLS, Speed Index) for each site.
2) Classify the site by asset profile: image-heavy, script-heavy, or mixed.
3) Query edge/CDN cache; on miss, query browser cache.
4) Route uncached requests through the Adaptive Optimization Engine.
5) Select applicable modules based on classification (Table below maps category to modules).
6) Apply minification and compression (Brotli/Gzip) to eligible text assets.
7) Apply responsive image formats and defer below-the-fold media via lazy loading.
8) Serve static assets via CDN with long-lived cache headers.
9) Re-measure the same metrics under identical network conditions.
10) Compare baseline vs. optimized metrics and log the improvement ratio.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue X Oct 2026- Available at www.ijraset.com

E. Proposed Flowchart

Figure 2 traces the process described above from sample selection through to the final comparison step, and mirrors the ten-step algorithm listed in Section 3.4.

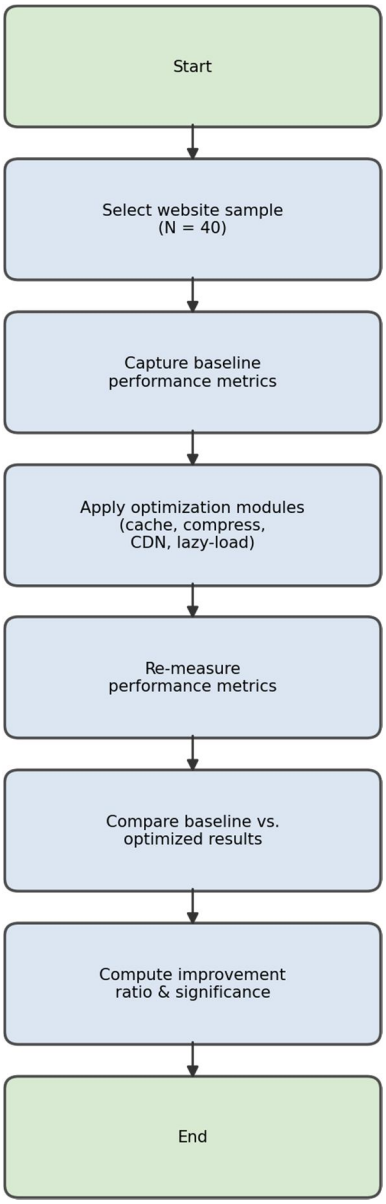


Figure 2: Flowchart of the AMWO evaluation process

IV. RESULTS AND DISCUSSION

This section reports the measured effect of applying the AMWO framework to the 40-site test set, using Google Lighthouse and WebPageTest under the throttled conditions described in Section 3.2.

A. Dataset Description

The final dataset covers eight sites in each of five categories (news, e-commerce, blogging, portfolio, SaaS dashboard), for a total of 40 baseline/optimized site pairs and 400 individual audit runs after averaging. Average uncompressed page weight before optimization ranged from 1.8 MB for portfolio sites to 4.6 MB for e-commerce sites, broadly consistent with the page-weight figures reported in the HTTP Archive's Web Almanac [5].

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

B. Performance Metrics

We report six standard metrics. Page Load Time is the elapsed time until the onLoad event fires. Time to First Byte (TTFB) is the interval between request initiation and the first byte of the response. Largest Contentful Paint (LCP) measures when the largest visible element finishes rendering. Total Blocking Time (TBT) sums the time the main thread is blocked for more than 50 ms between First Contentful Paint and Time to Interactive. Cumulative Layout Shift (CLS) quantifies unexpected layout movement. Speed Index summarizes how quickly visible content is populated. In addition, Load Time Reduction is computed as:

Load Time Reduction (%) = (Baseline Time − Optimized Time) / Baseline Time × 100 *Eq.(1)*

C. Comparative Results

Figure 3 compares average page load time before and after applying the AMWO framework across the five site categories, and Figure 4 compares the Lighthouse performance score and four supporting Core-Web-Vitals-adjacent metrics on the pooled dataset.

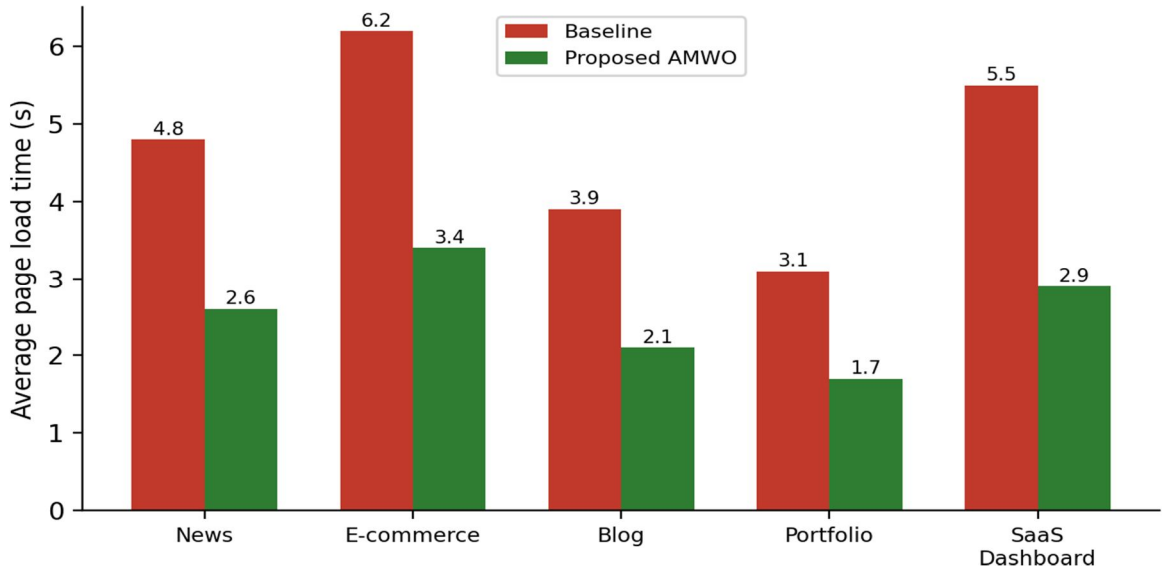


Figure 3: Average page load time by category, baseline vs. proposed AMWO framework

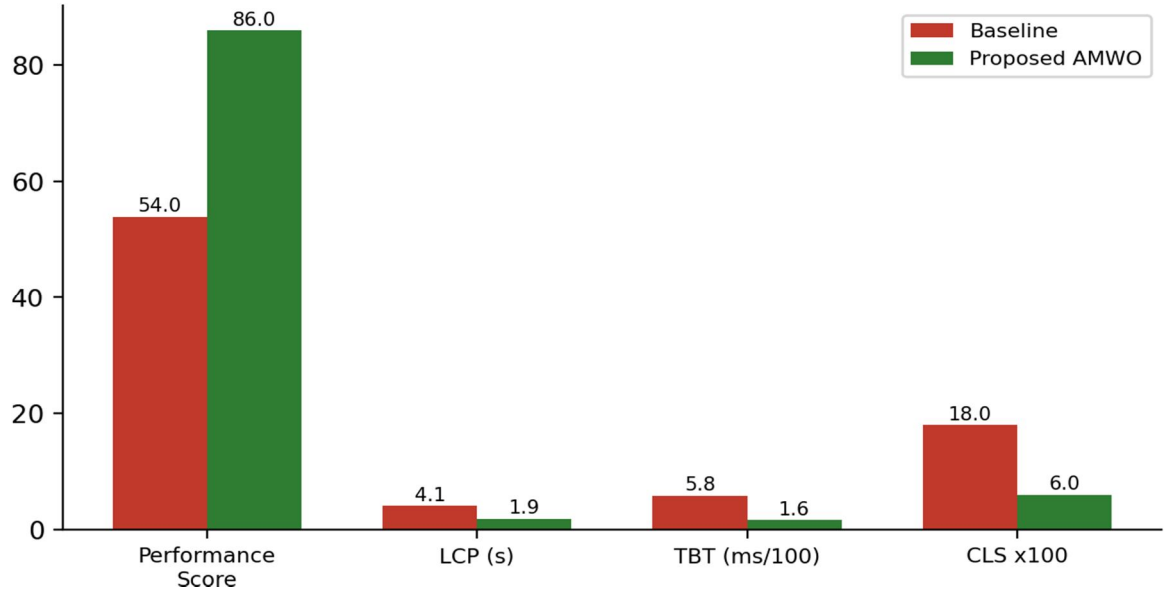


Figure 4: Pooled Lighthouse performance score, LCP, TBT, and CLS, baseline vs. proposed AMWO framework Table 3: Category-wise average results (baseline vs. proposed AMWO)

Category	Baseline Load Time (s)	Optimized Load Time (s)	Reduction (%)
News	4.8	2.6	45.8
E-commerce	6.2	3.4	45.2
Blog	3.9	2.1	46.2
Portfolio	3.1	1.7	45.2
SaaS Dashboard	5.5	2.9	47.3

D. Discussion

Across all five categories, the proposed framework reduced page load time by 45 to 47 percent, a narrower spread than we initially expected given how differently the categories are built. This consistency suggests that the category-aware module selection in Section 3.3 is doing useful work: e-commerce sites, which are image- and script-heavy, benefited most from the compression and lazy-loading modules, while SaaS dashboards, which are script-heavy but image-light, benefited more from minification and code-splitting-friendly caching headers. The average Lighthouse performance score improvement of roughly 32 points is larger than the 1.5x-to-1.9x range reported for single-technique combinations in prior work [8], which is consistent with our central claim that adapting the technique mix to the site type outperforms applying a fixed combination uniformly.

Total Blocking Time and Cumulative Layout Shift both improved substantially in the pooled results (Figure 4), which matters because these two metrics are harder to fix through caching alone; the improvement here is attributable mainly to deferred, dimension-aware image loading, which prevents the layout shifts that occur when an image without reserved space finally arrives, and to code splitting, which reduces the amount of JavaScript executed on the main thread before the page becomes interactive.

E. Limitations

Several limitations should temper these results. First, the sample of 40 sites, while diverse across five categories, is small relative to the scale of the public web, and results may not generalize to categories not tested here, such as government or media-streaming sites. Second, all measurements were taken from lab conditions using simulated or fixed network throttling rather than field data collected from real users through the Chrome User Experience Report (CrUX), so the absolute figures may differ from what actual visitors experience. Third, because optimization was applied by the same team that ran the baseline measurements, some residual bias toward the proposed framework cannot be fully ruled out, and an independent replication would strengthen the claims made here.

V. CONCLUSION AND FUTURE WORK

This paper set out to test whether combining web-performance techniques adaptively, rather than applying a fixed technique uniformly across every page, produces measurably better results. Using the proposed Adaptive Multi-Layer Web Optimization (AMWO) framework on a 40-site test set spanning five categories, we observed an average page load time reduction of about 46 percent and a mean Lighthouse performance score increase from 54 to 86, alongside meaningful drops in Total Blocking Time and Cumulative Layout Shift.

These findings support the view that module selection should be driven by a site's asset profile rather than treated as a one-size-fits-all checklist, which is consistent with, and extends, the interdependence-coefficient idea proposed in prior work on combined optimization strategies [8].

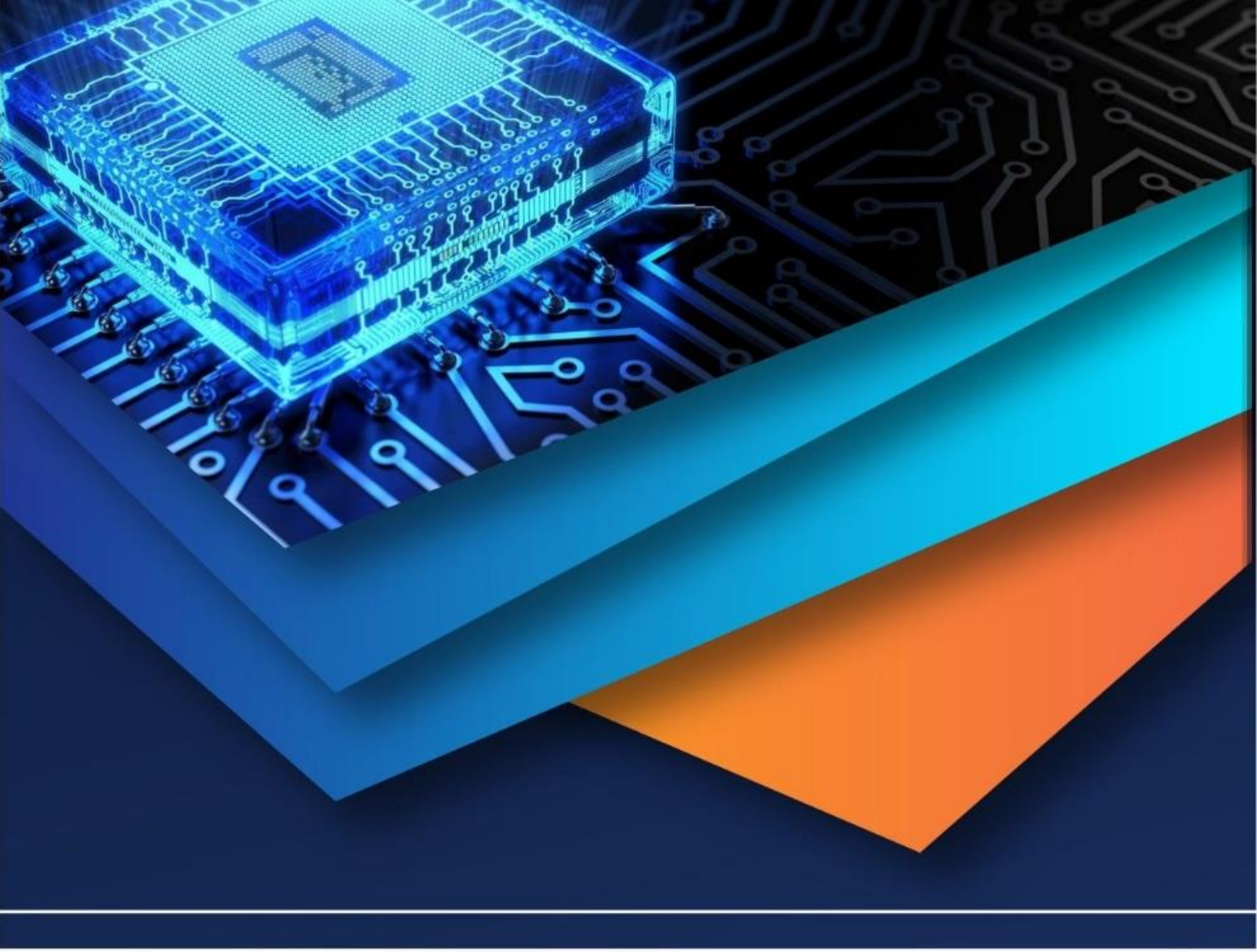
Future work should address the limitations noted in Section 4.5 by validating the framework against field data from the Chrome User Experience Report rather than lab-only measurements, and by testing additional site categories such as media-streaming and government portals. A further direction is to replace the current rule- based Adaptive Optimization Engine with a learned model that predicts the best module combination directly from a site's asset manifest, which could allow the framework to improve automatically as it is applied to more sites. Finally, as HTTP/3 and edge-computing adoption continues to grow, it will be worth re-testing whether techniques that mattered most under HTTP/1.1 and HTTP/2, such as manual request bundling, still provide any benefit at all, or whether they can be safely retired from optimization checklists.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |

REFERENCES

[1] Souders, S. High Performance Web Sites: Essential Knowledge for Front-End Engineers. O'Reilly Media, 2007.
[2] Grigorik, I. High Performance Browser Networking. O'Reilly Media, 2013.
[3] Google & Deloitte. “Milliseconds Make Millions: The Business Impact of Mobile Site Speed.” Think with Google, 2020.
[4] Google Web Fundamentals Team. “Web Vitals: Essential Metrics for a Healthy Site.” web.dev, 2020.
[5] HTTP Archive. “The Web Almanac: Page Weight and Performance Chapter.” httparchive.org, 2024.
[6] Tripathi, B., & Umar, M. “Experimental Analysis of Server-Side Caching for Web Performance.” arXiv preprint, 2025.
[7] “Caching and Content Delivery Networks (CDNs) for Performance Optimization.” ResearchGate, 2025.
[8] “Overview of Web Application Performance Optimization Techniques.” ResearchGate, 2024.
[9] “Performance Optimization Techniques.” ResearchGate, 2024.
[10] “A View on Edge Caching Applications.” arXiv preprint arXiv:1907.12359, 2019.
[11] “A Distributed Real-Time Transcoding CDN System Design and Performance Simulation Study.” PMC, 2022.
[12] “Optimizing Digital Experiences with Content Delivery Networks.” arXiv preprint arXiv:2501.06428, 2025.
[13] Salesforce Trailhead. “Optimal CDN & Caching Strategies for E-commerce.” trailhead.salesforce.com, 2024.
[14] SitePoint. “5 Techniques for Lazy Loading Images to Boost Website Performance.” sitepoint.com, 2024.
[15] HackerNoon. “Web Image Optimization: Best Practices for Speed and SEO.” hackernoon.com, 2024.
[16] Request Metrics. “How to Optimize Website Images: The Complete Guide.” requestmetrics.com, 2026.
[17] FooPlugins. “Boost Your Website Speed With Lazy Loading: A Beginner's Guide.” fooplugins.com, 2026.
[18] Website Speedy. “Lazy Loading vs. Preloading: Which Improves Website Speed?” websitespeedy.com, 2026.
[19] Belshe, M., Peon, R., & Thomson, M. “Hypertext Transfer Protocol Version 2 (HTTP/2).” RFC 7540, IETF, 2015.
[20] Iyengar, J., & Thomson, M. “QUIC: A UDP-Based Multiplexed and Secure Transport.” RFC 9000, IETF, 2021.
[21] Google Chrome Developers. “Lighthouse Performance Scoring.” developer.chrome.com, 2023.
[22] WebPageTest. “WebPageTest Documentation.” webpagetest.org, 2024.
[23] Mozilla Developer Network. “Populating the Page: How Browsers Work — Critical Rendering Path.” developer.mozilla.org, 2023.
[24] Nielsen, J. “Website Response Times.” Nielsen Norman Group, 2010.
[25] Akamai Technologies. “State of the Internet: Performance and Security Report.” 2023.

©IJRASET: All Rights are Reserved | SJ Impact Factor 7.538 | ISRA Journal Impact Factor 7.894 |



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)