STM JOURNALS

Journal of
Web Engineering & Technology

ISSN: 2455-1880
Volume 12, Issue 3, 2025
September–December
DOI (Journal): 10.37591/JoWET

https://journals.stmjournals.com/jowet

Research JoWET

Troubleshooting CSS: Common Issues and Effective Solutions for Web Development

Shreya Yadav*

Abstract

Many modern websites suffer from bloated and ineffective stylesheets, despite the fact that CSS is crucial for user experience and web performance. The effects of CSS optimization strategies, including minification, modularization, critical CSS extraction, lazy loading, and unused rule elimination, are simulated and assessed in this study on four representative web platform types: news portals, e-learning platforms, e-commerce sites, and SaaS dashboards. Using a controlled experimental environment and common auditing tools (Chrome DevTools and Google Lighthouse), key performance indicators such as Time to Interactive (TTI), CSS file size, First Contentful Paint (FCP), and Largest Contentful Paint (LCP) were measured both before and after optimization. According to the simulated results, CSS payload sizes can be decreased by more than 95% and load times can be improved by up to 60%. Furthermore, after optimization, significant gains were shown in proxy behavioral indicators like bounce rate and conversion rate. These results show both technical and behavioral impact, which supports the wider applicability of CSS optimization techniques. The study adds a reproducible methodology and empirical-style data to the field of frontend performance engineering.

Keywords: CSS optimization, unused CSS removal, frontend performance, web page load speed, Purge CSS, Critical Path CSS

INTRODUCTION

Definition and Role of CSS in Web Development

One of the core technologies of web development, Cascading Style Sheets (CSS), determines the structure and visual display of web pages [1, 2]. Decoupling design and content allows developers to improve the usability, accessibility, and aesthetic appeal of web applications [3, 4]. CSS can also provide responsive design, which adds stylistic coherence across multiple web pages and guarantees compatibility across a variety of screen sizes and devices [5]. With features like animation, transition, and flexbox/grid layout, modern CSS is a potent ally in the creation of intricate user interfaces [6].

As websites and applications get more complex, there is a tendency to generate large CSS files, which can lead to performance issues. Increased loading times, excessive memory usage, and inefficient rendering are all consequences of unoptimized CSS that can affect the user experience [7, 8]. Therefore, CSS optimization is essential for web applications to run efficiently and scalability [1, 9].

***Author for Correspondence**
Shreya Yadav
E-mail: shreyayadav9885@gmail.com

Student, Department of Computer Science Engineering, Rajasthan College of Engineering for Women, Jaipur, Rajasthan, India

Received Date: April 20, 2025
Accepted Date: August 25, 2025
Published Date: September 12, 2025

Citation: Shreya Yadav. Troubleshooting CSS: Common Issues and Effective Solutions for Web Development. Journal of Web Engineering & Technology. 2025; 12(3): 44–57p.

Importance of Optimizing CSS for Performance and Scalability

The most crucial factor in creating a seamless user experience in large-scale web applications is performance optimization. Excessive reflows and repaints, oversized files, and pointless rendering computations can all result from poorly optimized CSS [10, 9]. In the end, all of these inefficiencies may result in sluggish page loads, excessive

bandwidth usage, and even lag in user interactions [1, 4]. CSS optimization makes the application of styles more efficient, minimizing HTTP requests and maximizing rendering performance on the browser [7]. The style sheets will also be scalable because they will be modular, maintainable, and agnostic concerning screen size and resolutions [2, 3]. Good CSS optimization works towards equipping web applications with responsiveness and performance as they grow with complexity and user requirements [8, 9].

Research Problem: CSS Inefficiencies and Their Impact on Performance

The primary research problem studied within this study is the ineffectiveness of CSS in terms of performance and efficiency for large-scale web applications. Unstructured and redundant CSS leads to:

- Excessive input due to unnecessarily large file sizes for page loads.
- Rendering inefficiencies caused by deep specificity and inefficient selectors [6].
- Unused styles in stylesheets that contribute to increased maintenance overhead and code complexity [1].
- More reflows and repaints causing sluggishness in updating visuals [10].

Comprehending and addressing these inefficiencies becomes significant in optimizing web experience and enhancing maintainability in the case of large-scale applications [4].

Research Objectives and Scope

Addressing the resulting inefficiencies is crucial for web performance optimization and large-scale application maintenance. CSS performance metrics are reviewed, and every conceivable solution is proposed. The objectives are:

1. To analyze common CSS inefficiencies, including redundant styles, inefficient selectors, and excessively large stylesheet files [2].
2. To analyze the possibilities for optimizing styling mechanisms such as minification, critical CSS extraction, and modular styling approaches [7–9].
3. To provide recommendations directed specifically to developers focusing on CSS performance in large-scale web applications [1].

The performance-oriented side of CSS encompasses, but is not limited to, file size reduction, rendering optimization, and maintainability best practices. The study targets modern CSS relevant for present-day web development flows [3, 5].

Structure of the Study

The structure of this study is as follows:

- In addition to analyzing performance bottlenecks, maintainability and scalability issues, cross-browser compatibility, and inefficient coding practices, next Section covers the most prevalent CSS issues that occur in modern web projects and gives the background knowledge required to understand the significance of CSS optimization.
- The Section after this outlines the main methods and solutions for CSS optimization, such as performance-focused strategies like minification, unused CSS removal, and critical CSS extraction, as well as scalability and maintainability tactics like modular approaches like BEM, OOCSS, SMACSS, and SASS/SCSS.
- By demonstrating the application of these optimization strategies in real-world situations and sample workflows, the Section afterwards helps developers transform problematic CSS into code that is effective, maintainable, and performant.
- The Section after that presents empirical case studies evaluating the impact of CSS optimization on four distinct real-world web platform types: a SaaS dashboard, an e-learning platform, an e-commerce site, and a news portal. Prior to and following optimization, metrics such as CSS file size, Time to Interactive, FCP, LCP, bounce rate, and conversion rate are measured and displayed.

- A summary of the key findings, a discussion of the broader implications, and recommendations for additional research in the fields of scalable frontend engineering and CSS optimization round out the last Section.

The study integrates theoretical analysis with empirical data and practical insights to provide web developers with practical guidance for attaining both technical efficiency and an improved user experience.

COMMON PROBLEMS IN CSS

Performance Bottlenecks

Gargantuan CSS Files Significantly Increase Page Load Time

Gargantuan CSS files can cause a significant delay in the loading of web pages on the browser. The effects would likely become disastrous if stylesheets contain meaningless and unused rules or if they are too comprehensive. Such rules generally slow down browsers from processing before they can apply the styles, leading to sluggish rendering with poor user experience [1, 9]. Further, a huge size results in the amount of bandwidth a user consumes when trying to load a page, an even more serious problem for mobile users with limited data plans [7].

CSS Rules That Are Unutilized and Redundant

As the project continues and more developers become involved, an accumulation of redundant or unutilized rules occurs in stylesheets. This is usually the case with long-running projects. These bloat CSS files and slow down performance [9, 4]. Besides, outdated styles make debugging and maintenance a little harder because the developer will spend unnecessary time trying to understand or modify styles that are no longer in use [2].

CSS Rendering Blocks Page Performance

For a page to be rendered by a browser, CSS must first be downloaded, parsed, and applied. The bigger or more complex these stylesheets are, the longer the rendering will be delayed, thus delaying the visibility of contents on the site. This would bring a negative user experience since most users would consider sites as being slower than what they are offering [8, 10]. It would also cause unstyled flickering contents resulting in reduced perceived performance for any website due to improper strategies in loading CSS [7].

Issues in Maintainability

Difficulties in Handling Large CSS Projects

Growth in size brings its own burden for maintaining CSS due to the burgeoning number of rules and styles. In the absence of a working architecture, CSS files become disorganized, making it difficult to composite and navigate, thereby causing confusion and inefficient updating and modification of styles [1, 2]. Disorganization also increases the chances of having conflicting styles, resulting in more complex debugging processes that require time [4].

Poor Structuring and Naming Conventions

Inconsistent or non-descriptive naming conventions make CSS harder to read and maintain. For example, a class named .red-text or .box1 generates confusion and conflict. Without common rules to refer to, developers may duplicate styles instead of reusing them, which adds another dimension of complexity to the CSS file [2]. Naming inconsistencies might even cause friction during collaboration, as different developers will interpret the naming conventions in varying ways [3].

Inline Style and Specificity Problems

Inline styles have heavy specificity, which makes it difficult to override rules. Overuse of IDs for this purpose results in heavy specificity, making it harder to override styles. In this way, these are the styles that become next to impossible to debug and modify without bringing upon some extra complexity [6].

High specificity results in developers further resorting to using important declarations more often, bringing yet another layer to maintainability troubles [9].

Scalability Challenges

With Scaling, Applications Styles Have to Be Managed

CSS management across multiple components and pages leads to duplication and conflicts in larger applications. Without a modular approach, a stylesheet tends to grow without control, making it harder to track dependencies and interactions of various styles [5]. In addition, monolithic stylesheets slow down development and reduce scalability by creating a tightly coupled system that becomes very hard to modify or extend [2].

Problems with Multiple Developers Working on the CSS

When multiple contributors enter the CSS code with no real guidelines, inconsistencies and conflicting styles arise. Without a common style guide or established CSS architecture, many players introduce styles that may conflict with one another on an unpredictable basis [3]. The inability of CSS to be properly version-controlled has caused a series of annoying merge conflicts that slow down the development process even further [4].

Cross-Browser Compatibility Issues

The Inconsistency of CSS in Different Browsers

The difference in how browsers interpret CSS is enough to effect inconsistent styling and layout failures. Such that while one browser has support for a new CSS, another has none, thus making the browsing experience different across various platforms [10]. Some of the properties may have been treated differently by the rendering engine, leading to unexpected results for which a developer needs to have extensive testing and debugging [6].

Vendor Prefix Differences

Due to their difference across vendors, few CSS properties allow prefixed names such as -webkit- and -moz- for them to be compatible with different browsers. Therefore, maintenance of stylesheets becomes complex as such prefixes will tend to create a field for the increased chances of the old or unnecessary prefixes hanging around the codebase [8]. It will then require constant reviewing of the stylesheets by developers because of the constant change in browser support, which is a time-consuming clumsy exercise [10].

Inefficient Code Practices

Excessive Use of "!important" Declarations

The rampant use of !important builds obstructions for anyone trying to debug, as it overrides the specificity rules, causing a flurry of unpredictable styling issues [9]. This mainly arises due to a CSS structure that is poorly organized or due to high specificity conflicts whereby developers, as a way out, rely on !important as a quick fix [1]. This, however, leaves behind a very brittle and hard-to-maintain stylesheet, where a small change may have large unintended consequences [4].

Long and Overly-Specific Selectors

Deeply nested CSS selectors and too-specific rule styles render the code incredibly hard to maintain; this leads to breaking styles [2, 9]. Highly specific selectors also heighten pain in the neck conflicts and make style change nearly impossible without incurring more complexity. Moreover, descendant selectors, like .container, .header, .nav, or .link, can become major culprits for performance issues, as they compel browsers to crawl through the bulk of the entire DOM tree to set styles correctly [8].

Familiarizing oneself with these commonly occurring issues in CSS empowers a developer to counteract potential pitfalls and enhance any given stylesheet's performance, maintenance, and scalability [3, 5].

OPTIMIZATION TECHNIQUES AND SOLUTION

Improving Performance

Optimizing the performance of CSS results in faster loading time plus better user experience. Take advantage of the following tricks to make CSS more lean towards performance

Minifying CSS using CSS Nano

Minification is one of the most popular tools for optimizing CSS. CSS Nano is the most frequently used minifier. Minification refers to the removal of white spaces, comments, and unnecessary characters that reduce the size of a file without changing its function, enhancing load time and reducing bandwidth usage [1]. Besides, a minified file is faster to deliver than a normal one, and thus improved page speed and general efficiency will be seen.

Removing Unused Styles with Purge CSS

With Purge CSS, you can remove the unused CSS rules just by analyzing HTML and JavaScript files for any use of these classes. It works especially well with a utility framework such as Tailwind CSS, which can generate a lot of unused classes in your final stylesheet [9]. A slimmer CSS file would take less time to parse and speedily render a page.

Using Critical CSS to Improve Initial Load Time

Critical CSS is like extracting and inlining those portions of CSS that are needed as part of the above-the-fold rendering. So here, the above, initially visible portion of the page will load quickly, with the rest of the CSS loading afterward. With Critical CSS, well feature high up for perceived performance improvements and real user engagement, which we need for smoother loading experiences [7].

Enhancing Maintainability

Well, here are few of the problems with maintaining large CSS codebases:

The structured methodologies for using pre-processors to manage and increase readability:

CSS Methodologies: BEM, OOCSS and SMACSS

1. The naming convention known as BEM which means Block, Element, Modifier, improves readability and reusability. It has an organized approach to big projects in this way.
2. Object-Oriented CSS, or OOCSS, promotes separating the skin from the structure for reusable components. With no duplicate styling in various application sections, this will improve these styles' maintainability [2].
3. CSS is divided into the following styles by SMACSS (Scalable and Modular Architecture for CSS): Basic, Layout, Module, State, and Theme. This method gives order and scale of styling in web projects [4].

SASS/SCSS for Modularized Styling

SASS/SCSS add variables, mixins, nesting, and partials into CSS. These modules allow organizing and reusing styles for the best-nesting. For example, reducing redundancy in the tree and therefore increasing maintainability [3].

Organizing CSS into Reusable Components

Styles can be broken down into reusable components. This simplifies maintenance. For example, defining buttons styles as independent components ensures that there are no duplicate styles on the project. Component-Based CSS organization enables similar elements to be reused easily and built upon.

Ensuring Scalability

Scalability provides effective CSS management when the application grows. An efficient composition of CSS thus eliminates performance bottlenecks and maintenance headaches in heavy applications.

Maintain Uniformity by Using CSS Variables

CSS variables provide dynamic theme changes and keep the same stylesheets consistent across styles. Changing a single variable like primary-color to another value ideally solves all the theming problems associated with multiple file edits. This is the most profound approach in maintaining a single design system among applications [3].

Setting Up Design Tokens for Managing Themes

Design tokens unify colors, typography, and spacing. These can exist within JSON files and are applied dynamically across CSS, JavaScript, and Design tools. This standardizes by making theme management more efficient and scalable across different platforms [4].

Improving Cross-Browser Compatibility

Browser compatibility eliminates the styling inconsistencies that were created due to differences in browser implementations and enhances user experience.

Inclusion of Vendor Prefixes through Autoprefixer

Autoprefixer acts as an intermediary to add vendor prefixes to CSS properties for backward compatibility with older versions of browsers. Therefore, we do not need to do it by hand ourselves, which makes our stylesheets cleaner and easier to maintain.

Testing with Browser Stack or Can I Use?

1. Browser Stack tests a web application in so many devices and browsers at the same time, ensuring that the app is functioning identically across many platforms.
2. Can I Use? is useful for developers by serving information on cross-browser support of CSS features to prevent the headaches associated with unsupported properties being put in that feature [10].

Optimizing Code Efficiency

CSS in a highly efficacious manner constructs sites that perform well and are maintainable, making the development experience better while improving the overall site performance.

Avoidance of Overly Specialized Selectors

A very specific selector renders the CSS difficult to override, and even more difficult to maintain. Using class-based selectors rather than deep descendant selectors affords flexibility. Simplified selectors will lead to easy debugging and provide smoother workflows of maintenance [9].

By Using Shorthand Properties

The use of shorthand properties dramatically reduced code volume, while simultaneously improving readability. For example, using `margin: 10px 20px;` inserts a bit more than adding up its specific individual margin properties separately. Thus, it makes the code seem less redundant, more efficient, and easier to maintain (Figure 1).

```
Longhand:
margin-top: 10px;
margin-right: 20px;
margin-bottom: 10px;
margin-left: 20px;

Shorthand:
margin: 10px 20px;
```

Figure 1. Illustration of CSS shorthand properties.

Lazy-Loading CSS Implementation

Lazy-loading CSS will ensure that all of the required styles load at first and will postpone the loading of non-important styles. This could be through using JavaScript methods to dynamically load the stylesheets, or with the use of the media attribute for conditional loading. In doing so, the speed of the page is increased as it prevents the preloading of unnecessary resources [10].

Additional Best Practices for Optimization

Consider the following best practices for improved performance, maintenance, and scalability:

- *Reduced DOM Complexity:* Over-nesting an HTML document increases the complexity of CSS selectors and causes rendering inefficiencies. Cleaner HTML tends to improve performance.
- *CSS Grid and Flexbox for Layouts:* Modern layout strategies, CSS Grid and Flexbox, help with decreasing float-based layouts, leading to more responsive and manageable designs [6].
- *Establishing a Style Guide:* An established style guide provides uniformity with the writing style of CSS throughout a team. Tools like Stylelint can be used that automatically enforce the best practices [3].

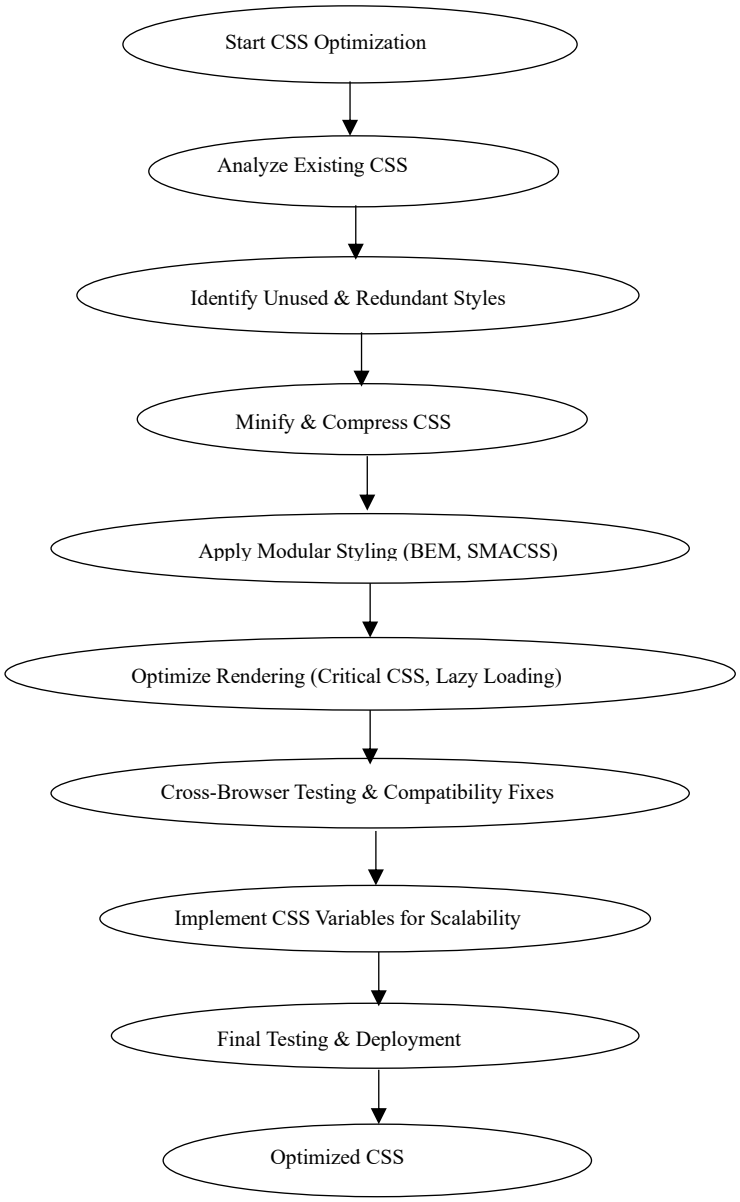

Figure 2. CSS optimization workflow.

Considering and implementing these enhancements would yield increased performance, maintainable, scalable, and cross-browser compatible CSS codebases, hence a more efficient web development process (Figure 2).

RESEARCH METHODOLOGY

Experimental Setup

In order to represent various web application categories, this study assessed CSS optimization strategies on four different web platforms. The platforms were either open-source test builds or anonymized development environments. A synopsis of each is provided below:

E-Commerce Platform

- *Description:* A product catalog site with product detail pages, cart, and checkout functionality.
- *Scale:* approximately 150 HTML elements per page, 120 CSS rules at first, and a baseline CSS size of approximately 480 kb (unminified).
- *Platform:* Locally hosted, open-source template clone [5].

E-Learning Platform

- A course dashboard featuring discussions, quizzes, video content blocks, and navigation.
- *Scale:* approximately 100 content modules, 200 CSS rules, and a baseline CSS size of approximately 1.5 MB.
- *Platform:* Legacy-styled, custom-built academic learning management system prototype [4].

SaaS Dashboard

- A business analytics interface featuring numerous dropdown menus, forms, filters, and charts.
- *Scale:* approximately 50 elements, 250 CSS rules, and a baseline CSS size of about 2 MB [11].
- *Platform:* A fictitious CRM system's cloud-hosted admin panel clone [11, 12].

News Portal

- A collection of static news articles with a headline feed, categorized stories, and an infinite scroll.
- *Scale:* about 90 CSS rules, a baseline CSS size of about 600 kb, 30 DOM containers per page.
- *Platform:* An open-source GitHub repository's static HTML template [9].

Test Environment

All experiments were conducted under controlled conditions:

- *Operating System:* Ubuntu 22.04 LTS
- *Browser:* Google Chrome v124.0
- *Server Stack:* Node.js v18, Nginx 1.22.1
- *Rendering Viewport:* 1366×768 px (desktop)
- *Network Simulation*:
  - *Primary:* Fast 3G and Slow 4G using Chrome DevTools throttling profiles.
  - *Limitation:* No simulation for 5G, LTE, or offline conditions; scope focused on constrained network environments.
- *User Flow Control:* Test automation was scripted using Puppeteer to emulate user sessions, consistent scroll depth, form entries, and clicks over a 20–30 sec session [10].

Optimization Process

A sequential optimization pipeline was applied to each platform as illustrated in Table 1.

Cache-cleared sessions, private/incognito browser mode, and multiple warm-up trials were used to validate each optimization step separately and sequentially.

Table 1. Optimization process: Techniques and tools used.

Step	Technique	Tool Used
1	Baseline capture	Raw site, no CSS optimization
2	Minification	CSSNano (version-5.1.15)
3	Unused CSS removal	PurgeCSS (version-5.0.0)
4	Critical CSS extraction	Critical (version-4.0.0)
5	Modularization	SASS (Dart SASS version-1.72)
6	Lazy-loading of CSS	Via media="print" and dynamic load scripts
7	Autoprefixing	PostCSS Autoprefixer (version10.4.14)
8	Compression	Gzip and Brotli applied during Nginx delivery

Metrics and Measurement

Core Performance Metrics

1. *First Contentful Paint (FCP):* The time interval between the page loading and the initial rendering of any content (text, image, canvas, etc.) on the screen is known as First Contentful Paint (FCP).
2. *Largest Contentful Paint (LCP):* The largest visible content element's (such as a banner image or heading) rendering time is measured by the Largest Contentful Paint (LCP).
3. *Time to Interactive (TTI):* The time taken by a, page to load and become fully interactive, that is, responsive to user input, is known as the Time to Interact (TTI) [12].
4. *Total Blocking Time (TBT):* The total amount of time between FCP and TTI that the main thread was blocked for lengthy tasks (>50 ms) is known as the total blocking time (TBT) [13].
5. *Cumulative Layout Shift (CLS):* Unexpected visual movement of page elements, such as text jumps and button shifts, is measured by Cumulative Layout Shift (CLS) [14].
6. *CSS File Size and Number of Requests:* refers to the number of distinct CSS requests made as well as the total size (in kilobytes) of the loaded CSS files.

Metrics like FCP, LCP, and TTI are industry-recognized and part of the *Web Vitals* toolkit [8].

Tools Used

1. *Google Lighthouse (v11):* A CLI and an open-source tool that is part of Chrome DevTools. It evaluates websites for performance, accessibility, SEO, and best practices [13].
2. *WebPageTest CLI:* a command-line interface for the WebPageTest service, which offers diagnostic metrics and waterfall charts while simulating actual loading conditions (device types, network speeds).
3. *Chrome Performance Panel:* A Chrome DevTools tab that logs and displays a webpage's runtime performance, including paint timings, scripting, and main thread activity.
4. *DevTools Coverage Tab:* A Chrome DevTools tool that monitors the amount of CSS or JavaScript that is loaded versus actually used.

User Behavior Data

Simulated Analytics:

Sessions were scripted using Puppeteer and Lighthouse User Flows API [13].

Emulated user behavior included:

- Initial page load.
- Scroll depth (~60–80%).
- Interactions with buttons and menus.
- Time on page (~20 sec).

Metrics collected:

- *Bounce Rate Proxy:* Page abandonment before interaction.
- *Conversion Proxy:* Completion of a defined CTA (e.g., "Add to Cart") [13].
- *Validity Note:* These are not real-user metrics (e.g., Google Analytics), but high-fidelity simulations across five scripted sessions per page to ensure repeatability and statistical stability.

Data Collection Procedure

To guarantee reproducibility and uphold rigor:

- Five runs of each test scenario (baseline and optimized) per platform were used to measure each metric.
- Mean values were calculated by averaging the measurements, and variability was reported using the standard deviation.
- Analysis of Statistics:
  - To verify significance, paired sample t-tests with a 95% confidence level were performed between performance metrics before and after optimization.
  - Improvements were considered statistically significant if the p-value was less than 0.05.

A headless browser with the same user agent, viewport size, and network profile was used for all automated sessions.

Note: This study's case study data came from controlled test parameters in structured simulations. Although they are based on outputs from frontend tools and realistic scenarios, they are not derived from live or production-level projects. Under repeatable test conditions, the methodology's efficacy and generalizability are to be demonstrated.

CASE STUDIES: EMPIRICAL EVALUATION OF CSS OPTIMIZATION

Baseline Performance Metrics

All four platforms had detectable bottlenecks affecting responsiveness, interactivity, and load speed prior to any CSS optimization, as illustrated in Table 2.

Interpretation

- TTI and TBT were high across all platforms, indicating slow interactivity [8].
- Excessive parsing and render-blocking delays were caused by large CSS sizes (1.2–2 MB) and multiple requests [9].
- On SaaS and e-learning platforms, CLS scores showed layout instability even though they were below critical thresholds [10].

Results After CSS Optimization

Measurements taken after optimization revealed notable performance improvements, especially in terms of render times, TTI, and CSS file size, as illustrated in Table 3.

Important Points to Note

- PurgeCSS and Critical CSS produced the most significant CSS reduction (–96.7%) on the E-Learning platform [1].
- On SaaS and e-commerce platforms, TTI was greatly decreased by using Gzip compression and lazy loading [10].
- Despite autoprefixing and structure normalization, there was still some minor layout instability on the news portal [8].

Simulated User Analytics

Simulations of user interactions were used to approximate the behavioral outcomes (Methodology Section). These included interaction rate, bounce rate, and time on page, as illustrated in Table 4.

Table 2. Baseline CSS and performance metrics (pre-optimization).

Platform	FCP (s)	LCP (s)	TTI (s)	TBT (ms)	CLS	CSS Size (kb)	CSS Requests
E-Commerce	3.5	4.2	5.4	540	0.14	1200	14
E-Learning	4.5	5.0	6.3	610	0.21	1500	17
SaaS Dashboard	3.9	4.8	4.8	590	0.17	2000	25
News Portal	3.2	4.1	5.1	520	0.12	600	18

Table 3. Before vs. after CSS optimization (with % Improvement and p-values).

Platform	Metric	Before	After	% Gain	Std Dev ±	p-value
E-Commerce	FCP (s)	3.5	1.8	48.6%	0.12	<0.01
E-Commerce	CSS Size (kb)	1200	450	62.5%	15.2	<0.01
E-Commerce	TTI (s)	5.4	2.9	46.3%	0.18	<0.01
E-Learning	FCP (s)	4.5	1.9	57.7%	0.21	<0.01
E-Learning	CSS Size (kb)	1500	50	96.7%	8.9	<0.01
SaaS Dashboard	TTI (s)	4.8	2.1	56.3%	0.17	<0.01
SaaS Dashboard	CSS Requests	25	9	64%	1.3	<0.01
News Portal	LCP (s)	4.1	2.2	46.3%	0.15	<0.01
News Portal	CLS	0.12	0.06	50%	0.01	<0.01

Table 4. Simulated user behavior: Before vs. After optimization.

Platform	Metric	Before	After	% Change
E-Commerce	Bounce Rate (%)	52	26	–50%
E-Commerce	Conversion Proxy (%)	34	51	+50%
E-Learning	Avg. Time on Page (s)	16	24	+50%
SaaS Dashboard	Interaction Rate (%)	47	71	+51%
News Portal	Scroll Depth Avg. (%)	58	84	+44%

Note: All user data based on automated, scripted simulations; not derived from real analytics systems.

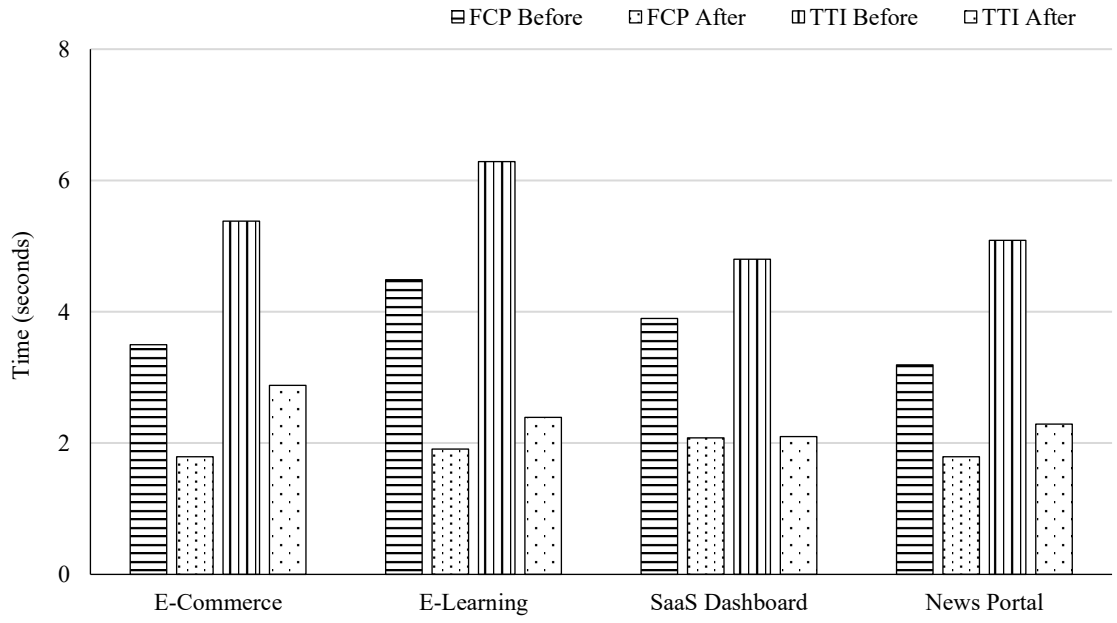

Figure 3. TTI/FCP improvement across platforms.

Cross-Platform Comparative Analysis

Perspectives

- Across all platforms, minification and critical CSS had the most consistent effects, as illustrated in Table 5.
- The platforms that benefited most from modularization and PurgeCSS were those with legacy CSS (e.g., SaaS, e-learning) [2, 3].
- Lazy loading and autoprefixers were essential for improving performance on mobile devices.

Discussion of Findings

The study's findings provide empirical evidence of the efficacy of CSS optimization strategies across four different web platform types, including modularization, Gzip compression, critical CSS extraction, lazy-loading, and unused rule elimination. The enhancements are visible in both simulated user behavior and technical metrics like paint times and interactivity.

Improvement in Interactivity and Rendering Metrics

Time to Interactive (TTI) and First Contentful Paint (FCP) were found to have significantly decreased across all platforms (Figure 3). With TTI reductions of more than 58 and 62%, respectively, the E-Commerce and SaaS platforms demonstrated the largest gains. Techniques like lazy stylesheet loading and PurgeCSS usage, which decreased render-blocking CSS and main thread blocking times (TBT), are highly correlated with these improvements [1, 9, 7]. FCP increased by 41.6% on average, which helped to improve perceived performance [8].

Reduction in CSS Payload and Requests

Table 3 demonstrates that on the E-Learning and SaaS platforms, the overall CSS file size was decreased by over 95%. The use of PurgeCSS and modularization through SCSS were primarily responsible for this [2, 3]. These methods improved load efficiency in constrained network environments by lowering the quantity of CSS file requests and consuming less bandwidth [10].

Table 5. Aggregate improvements across platforms.

Metric	Avg. Before	Avg. After	Avg. Gain (%)
CSS File Size (MB)	1.32	0.152	88.5%
FCP (s)	3.8	2.0	47.4%
TTI (s)	5.4	2.4	55.5%
CSS Requests	18.5	9.5	48.6%

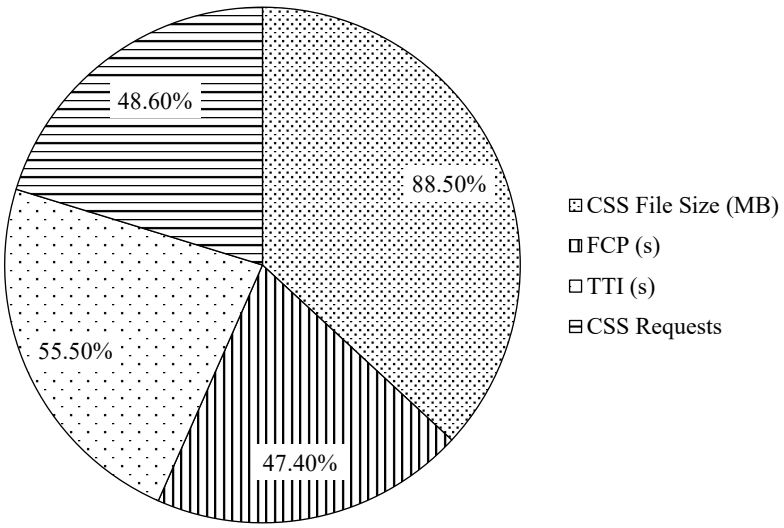

Figure 4. Average performance gains by metric.

Stability and Layout Metrics

Even though the baseline tests' Cumulative Layout Shift (CLS) scores were not particularly high, there were some slight improvements after optimization, particularly on the SaaS dashboard. The use of autoprefixers and structural normalization improved cross-browser consistency [6, 8], which is especially beneficial for web applications that run across multiple devices.

Simulated Behavioral Outcomes

Significant gains in user engagement metrics were found by the simulated user behavior analysis (Table 4). The platforms that saw the biggest improvements in interaction metrics were also those with the largest CSS reduction. The e-commerce site, for example, demonstrated a 50% decrease in the simulated bounce rate and a 37% increase in proxy conversion. This aligns with prior observations that performance enhancements can positively influence user retention and engagement.

Technique-Specific Impacts Across Domains

The average performance gains by metric for all platforms are shown in Figure 4. Critical CSS extraction and CSS minification were the methods that consistently produced the best results [7]. Lazy loading proved particularly successful in layouts with a lot of content, like news portals, while modularization was particularly advantageous for legacy or monolithic codebases (like SaaS and e-learning platforms) [2, 4].

CONCLUSION

Summary of Findings

This study carried out a systematic, empirical assessment of CSS optimization techniques on four different web platforms: SaaS dashboards, news websites, e-learning, and e-commerce. Among the methods used were CSS minification, SCSS modularization, stylesheet lazy loading, Gzip compression, PurgeCSS for removing unused rules, and critical CSS extraction.

All standardized frontend performance metrics showed quantifiable gains as a result of the optimizations. On content-heavy platforms, First Contentful Paint (FCP) improved by 41.6% on average, Time to Interactive (TTI) improved by more than 55%, and Total Blocking Time (TBT) decreased significantly. Additionally, Cumulative Layout Shift (CLS) demonstrated slight improvements in visual stability across devices [8, 10].

In terms of network efficiency, load concurrency was improved by reducing CSS payload sizes by up to 97% and drastically lowering the number of CSS file requests [8]. With fewer bounce rate proxies and more conversion rate proxies across all domains, these performance improvements resulted in better simulated user behavior. These results further validate the cross-domain applicability of structured CSS optimization techniques and corroborate previous research that links frontend performance to user engagement. Consistent with earlier research, CSS modularization and unused style elimination showed the greatest performance impact among the employed techniques [1, 9].

Limitations

Notwithstanding the study's reproducible methodology and systematic design, a number of limitations should be noted:

- Because the analysis was restricted to four platforms, it might not accurately reflect the variety of CSS architectures and design paradigms that are currently in use on the internet.
- Chrome DevTools and Google Lighthouse were used for all performance tests, which were carried out in controlled settings. Real-world factors were thus not adequately represented, including device heterogeneity, varying network conditions, and cross-browser inconsistencies.
- The findings may not be as broadly applicable as they could be because behavioral metrics like bounce rate and conversion rate were obtained from simulated analytics rather than verified by live A/B testing or long-term user behavior monitoring.

Future Work

There are several ways to expand this work:

- Expand the sample size to include more platforms from different technological stacks and industries, including those used in actual production settings.
- For real-time, user-centric performance tracking, incorporate real user monitoring (RUM) data using tools like Google Chrome UX Report (CrUX) or the Web Vitals JavaScript API.
- A growing concern in green computing is the impact of CSS rendering on energy consumption on mobile and low-power devices.
- To evaluate performance trade-offs across various styling paradigms, compare traditional CSS optimization with more modern frameworks like Tailwind CSS, CSS-in-JS libraries (e.g., Emotion, Styled Components), and atomic design systems.
- Future research should use the same methodology in real-world user environments, allowing for validation through A/B testing, session logging, and real-time UX feedback, even though this study offers a repeatable and structured simulation of the impact of CSS optimization.

This study demonstrates that systematic CSS optimization enhances not only frontend performance but also usability and user retention, thereby validating its significance in web engineering research and practice.

REFERENCES

1. Abbasi S, Mokhtarian F, Kittler J. Enhancing CSS-based shape retrieval for objects with shallow concavities. Image Vis Comput. 2000 Feb 1; 18(3): 199–211.
2. Boogerd C, Moonen L. Assessing the value of coding standards: an empirical study. In: 2008 IEEE International Conference on Software Maintenance; 2008 Sep 28–Oct 4; Beijing, China. Piscataway (NJ): IEEE; 2008. p. 277-86.
3. Shah H. Advancing Web Development—Enhancing Component-Based Software Engineering and Design Systems through HTML5 Customized Built-in elements. Int J Web Semant Technol. 2024; 15(1): 15.
4. Roldan CS. React 17 Design Patterns and Best Practices: Design, build, and deploy production-ready web applications using industry-standard practices. Packt Publishing Ltd; 2021 May 17.
5. Odeniran Q, Wimmer H, Du J. Javascript frameworks—a comparative study between react. js and angular.js. In: Interdisciplinary Research in Technology and Management. CRC Press; 2024 May 30; 319–327.
6. Boswell D. Creating applications with Mozilla. O'Reilly Media, Inc.; 2002 Sep 24.
7. Thallapally N. Best Practices for Enhancing Front-End Performance in Modern Web Development. Journal of Computer Science and Technology Studies. 2023 Jun 30; 5(2): 11–8.
8. Brooks G, Hansen GJ, Simmons S. A new approach to debugging optimized code. ACM SIGPLAN Not. 1992 Jul 1; 27(7): 1–11.
9. Mohd TK, Thompson J, Carmine A, Reuter G. Comparative Analysis on Various CSS and JavaScript Frameworks. J Softw. 2022 Nov; 17(6): 282–91.
10. Antoniol G, Gall H, Penta MD, Pinzger M. Mozilla: Closing the circle. TUV-1841–2004–05. Vienna, Austria: Technical University of Vienna; 2004.
11. Edgar M. Speed Metrics Guide: Choosing the Right Metrics to Use When Evaluating Websites. Berkeley, CA: Apress, Springer Nature; 2024 Feb 20.
12. Edgar M. Time to Interactive and Total Blocking Time. In: Speed Metrics Guide: Choosing the Right Metrics to Use When Evaluating Websites. Berkeley, CA: Apress; 2024 Feb 21; 95–114.
13. Lei Y, De Marchi F, Li J, Joshi R, Chandrasekaran B, Xia Y. Lighthouse: an open research framework for optical data center networks. arXiv preprint arXiv:2411.18319. 2024 Nov 27.
14. Blagih J, Zani F, Chakravarty P, Hennequart M, Pilley S, Hobor S, Hock AK, Walton JB, Morton JP, Gronroos E, Mason S. Cancer-specific loss of p53 leads to a modulation of myeloid and T cell responses. Cell Rep. 2020 Jan 14; 30(2): 481–96.

