International Journal of Emerging Research in Computer Science
Volume 2 | Issue 3 | March 2026 | ISSN: 3139-101X (Online)
Available online at: https://ijercs.com/
DOI: https://doi.org/10.59828/ijercs.v2i3.23

Minification of CSS, JavaScript, and HTML for Faster Websites

Budgude Siddharth Dattatray[1] ✉, **Dr. Sonali Doifode Gholve** [2] ✉

[1]Department of Computer Science, Sarhad College of Arts, Commerce & Science, Katraj, Pune, India
[2]Assistant Professor, Department of Computer Science, Sarhad College of Arts, Commerce & Science, Katraj, Pune, India

Received: 06 March 2026 | **Accepted:** 19 March 2026 | **Published:** 27 March 2026

ABSTRACT

Over the last decade, web pages have grown dramatically in size and complexity. Serving a modern single-page application can easily mean pushing 2–5 MB of JavaScript, CSS, and HTML to a browser before the user sees anything meaningful on the screen. Minification is one of the oldest and still most effective tools we have for fighting this problem. Put simply, minification strips out everything in source code that a browser does not need at runtime whitespace, comments, long variable names without changing how the code actually behaves. This paper walks through how that process works at a technical level, covers the history of the major tools, and presents test results from a simulated e-commerce page that show just how much difference minification makes in practice, particularly on slow mobile connections. We also look at how minification pairs with modern compression formats like Brotli, and argue that treating the two as independent choices is a mistake they work best when used together.

Keywords: *Web Performance Optimization, Minification, Critical Rendering Path, Tree Shaking, Code Splitting, Content Delivery Network, Largest Contentful Paint.*

1. INTRODUCTION: WHY FILE SIZE STILL MATTERS

Ask any web developer what slows down a web page, and they will likely mention large images, slow servers, or too many HTTP requests. File size of the code itself is easy to overlook but it is a very real factor, especially on mobile networks. A file that is three times larger takes roughly three times longer to download and three times longer to parse. For a user sitting on a slow 3G connection in a rural area or on a bus, those extra seconds can mean the difference between staying on a page and leaving it.

The core tension in front-end development is that readable code and lean code are two different things. Good developers write clean, well-commented code with meaningful variable names like calculateUserTotalScore. That is how it should be. No one wants to maintain a codebase written in cryptic single-letter identifiers. But once that code is ready to ship, those long names and generous whitespace padding become dead weight. A JavaScript engine does not need comments to run code, and it has no use for indentation. So the question becomes: why send those extra characters at all?

Minification is the practical answer to that question. It is a build-time transformation that takes developer-friendly source code and produces a functionally identical but much smaller output file. This paper examines how that transformation actually happens under the hood, traces how minification tools have evolved over twenty-plus years, and shows with real numbers what the performance difference looks like.

2. METHODOLOGY: HOW MINIFICATION ACTUALLY WORKS

It is tempting to think of minification as just a fancy find-and-replace for spaces and tabs. In practice, the better tools do quite a lot more than that. The pipeline has several distinct stages, each with its own logic.

2.1 Tokenization: Reading the Code

Before a minifier can change anything, it has to understand what it is looking at. The first step is tokenization the minifier reads the source file character by character and breaks it into tokens, which are the smallest meaningful pieces of the language: keywords like var and function, operators like + and =, string literals, numbers, and identifiers.

As part of this process, the minifier separates out the tokens that can simply be thrown away comments and whitespace. A comment like // TODO: fix this later carries zero information for the browser. Same with the four spaces of indentation at the start of every line in a well-formatted file. The minifier discards all of it. That said, it has to be careful: you cannot blindly remove every space. Remove the space between return and true, for example, and you get return true, which is an identifier, not a return statement followed by a Boolean. The minifier keeps spaces only where they are syntactically necessary.

2.2 Abstract Syntax Tree Refactoring

The most powerful minifiers Terser and esbuild being the two main ones today do not stop at the token level. They parse the code into an Abstract Syntax Tree, or AST, which is essentially a structured representation of what the code means rather than just what it looks like. Working at the AST level opens up some genuinely impressive optimizations.

One of these is constant folding. If your code says const secondsInDay = 60 * 60 * 24, the minifier can see that this is a purely arithmetic expression with no side effects and no runtime dependencies. It will replace the whole expression with the literal value 86400. The browser never needs to compute that multiplication.

A more impactful technique is identifier munging. The minifier builds a complete map of every variable and function name in the codebase, along with the scope each one belongs to. It then replaces long names with the shortest possible alternatives a, b, c, and so on. Because it tracks scope, it knows it can safely reuse a in a different function without any collision. For a large codebase, this alone can cut JavaScript size by thirty to forty percent.

2.3 CSS-Specific Optimizations

CSS minification has its own set of tricks that go beyond removing whitespace between braces. One common optimization is value shortening: 0.5 becomes .5, 1000ms becomes 1s, and margin: 10px 10px 10px 10px becomes margin: 10px. These feel like small wins, but they add up across thousands of lines of stylesheet.

More advanced CSS minifiers will also look for duplicate rule sets. If. button-primary and. button-secondary share five identical property declarations, the tool will merge them into a single rule with a grouped selector. Similarly, color values get normalized #000000 shortens to #000, and some named colors can be replaced with shorter hex equivalents.

2.4 HTML Structural Pruning

HTML minification is the most conservative of the three. The HTML5 specification explicitly lists several tags as optional the browser will infer their presence. The opening and closing tags for html, head, body, tbody, and li are all technically optional in valid HTML5. Minifiers that follow the spec closely will remove them. Attribute quotes are also fair game: type=text is perfectly valid HTML; the quotes around text are not required when the value has no spaces.

Boolean attributes get simplified too: disabled="disabled" becomes just disabled, and checked="checked" becomes checked. None of this changes how a browser renders the page it is purely cosmetic from the parser's point of view.

2.5 Tree Shaking: Only Ship What You Use

Tree shaking deserves a section of its own because it is where modern JavaScript bundling tools have made the biggest impact. The idea is straightforward: if your application imports a utility library that exports thirty functions, but you only ever use three of them, why send the other twenty-seven to the user? Tree shaking performs a static analysis of your import and export statements and removes any exported code that nothing in the application actually imports.

This requires ES Modules the import/export syntax introduced in ES6 because those imports are static and can be analyzed at build time. CommonJS require() calls cannot be reliably tree-shaken because they can be dynamic. This is one of the main reasons the JavaScript ecosystem has pushed so strongly toward ES Modules over the past several years.

2.6 Why Minification and Compression Are Both Necessary

A common misconception is that if you are already gzipping or using Brotli compression on your server, minification is redundant. This is not correct. Minification and compression operate on different kinds of redundancy, and they complement each other rather than overlap.

Compression algorithms like Brotli work by finding repeated byte sequences and encoding them more efficiently. They are very good at this but they are working with whatever input you give them. Source code that has not been minified contains a lot of low-value repetition: the same four-space indent pattern repeated hundreds of times, comments that follow predictable formats, whitespace between every selector in a CSS file. The compression algorithm has to spend dictionary space on all of that.

When you minify first, you remove that noise before Brotli even starts. The result is that the compression has more dictionary budget to spend on the parts of the code that actually vary and the final compressed size ends up meaningfully smaller than Brotli-only would produce. Section 4.5 has the numbers.

3. A BRIEF HISTORY OF MINIFICATION TOOLS

3.1 The Early Days: 1995 – 2005

JavaScript was introduced in 1995, but for the first several years it was used so sparingly that file size was not a serious concern. Scripts were short, mostly doing simple things like validating a form field before submission. Developers who cared about performance would manually keep their scripts compact, but there was no tooling for it.

That changed in 2001 when Douglas Crockford released JSMin, the first automated JavaScript minifier. It was simple by today's standards it removed comments and unnecessary whitespace but it was genuinely useful and introduced the idea that minification should be a step in the build process rather than something you think about while writing code.

3.2 The Age of the Big Compressors: 2006 – 2015

Google's V8 engine, released in 2008, was a turning point. JavaScript was now fast enough to build real applications, and developers started building real applications with it. Codebases grew. Yahoo's YUI Compressor, and later Google Closure Compiler, raised the bar considerably they used proper language parsing and could rename variables, inline small functions, and apply algebraic simplifications. Closure Compiler's advanced mode was

particularly aggressive, capable of whole-program optimization that could produce dramatically smaller output, though it required code written in a specific style to do so safely.

3.3 The Build Pipeline Era: 2016 – Present

The arrival of frameworks like React, Angular, and Vue, combined with ES6's module system, changed how front-end code is structured and delivered. You no longer concatenate a few script files you have an entire build pipeline that transpiles JSX, compiles TypeScript, resolves hundreds of module imports, and bundles everything together before minifying the result.

Webpack became the dominant bundler for several years, with Terser as its default minifier. More recently, esbuild written in Go and designed from the ground up for parallelism has redefined what fast minification looks like. On large codebases, esbuild is roughly 100 times faster than JavaScript-based alternatives. For teams running builds in CI/CD pipelines dozens of times a day, that is not a minor convenience it translates to real money in compute costs and real time saved for developers waiting on builds.

4. PRACTICAL APPLICATION AND RESULTS

Choosing a minification tool in a real project involves trade-offs. The table below summarizes the four tools we evaluated for this paper:

Tool	Language Focus	Key Algorithm	Best Use Case
Terser	JavaScript	AST-based Munging	Production builds via Webpack
esbuild	JS / CSS	Parallel Go routines	Fast development builds
CSSO	CSS	Structural optimization	Heavy CSS frameworks
HTML-Minifier-Terser	HTML	Regex + DOM parsing	Content-heavy SEO sites

4.1 esbuild vs. Terser: Speed vs. Output Size

In our testing, Terser consistently produced slightly smaller output than esbuild typically two to four percent smaller after gzip. For a 500 KB JavaScript bundle, that difference is roughly 10–20 KB, which is noticeable but not dramatic. What is dramatic is the speed difference. On a project with around 400,000 lines of TypeScript, Terser took 68 seconds to complete a production build. esbuild finished the same task in under one second.

For most teams, the practical choice depends on pipeline requirements. If you are building once a day for a production release, Terser's output quality justifies the wait. If you have dozens of developers triggering CI builds constantly, esbuild's speed matters more than shaving an extra few kilobytes off the bundle.

4.2 The Math Behind Minification and Compression

There is an information-theoretic reason why minification helps even when you are already using Brotli. Compression algorithms are bounded by the Shannon entropy of the input the more predictable and repetitive the data, the more it compresses, but there is still wasted work encoding the predictable parts. Minification reduces the total size of the input (L) before the compressor runs. Since compressed size is roughly proportional to L times the entropy H of the content:

$$C \approx L \times H$$

Reducing L before computing H means the compressor is working on a smaller, denser input and the final C ends up smaller than Brotli alone would achieve on the original file. This is not just theoretical; Section 4.5 shows it empirically.

4.3 Test Setup: Simulated E-Commerce Landing Page

To get concrete numbers, we built a simulated e-commerce landing page that is representative of what you would see on a mid-tier retail site: a product grid with filtering, a cart component, a header with navigation, and some promotional banners. The unoptimized total came to 512 KB across HTML, CSS, and JavaScript combined.

We ran three versions of the page: unoptimized, minified only (Terser for JS, CSSO for CSS, HTML-Minifier-Terser for HTML), and minified plus Brotli level 6 compression served over HTTP/2.

4.4 Network and Device Conditions

- Network: Simulated slow 3G 1.6 Mbps downstream, 300 ms round-trip latency
- Device: Mid-tier Android handset (Snapdragon 665, 4 GB RAM)
- Browser: Chrome 124, Dev Tools throttling enabled

4.5 Results

Metric	Unoptimized	Minified Only	Minified + Brotli
Total Transfer Size	512 KB	310 KB	88 KB
Time to Interactive (TTI)	8.4 s	5.2 s	2.1 s
CPU Parse Time	1.2 s	0.7 s	0.6 s

The TTI improvement from 8.4 seconds down to 2.1 seconds is the most striking result. Research consistently shows that a significant share of mobile users abandons a page if it takes more than three seconds to become usable so the difference between 8.4 and 2.1 is not just a benchmark number, it is the difference between a page that works and one that most users never see.

What the CPU parse time tells us is equally important. Even after Brotli decompresses the file on the client, the browser still has to parse and compile the JavaScript. Minification reduces that cost because there is simply less code to process dead code is gone, and identifier names are shorter. The compression alone does not help here; minification does.

4.6 Obfuscation as a Secondary Benefit

Worth noting briefly: minification makes code harder to read, which provides a degree of protection against casual reverse engineering. When validateCreditCard() becomes v(), the intent of the function is not obvious to someone inspecting the shipped code. Comments that might expose internal API paths or architecture decisions are gone. Conditional logic rewritten as ternary chains is harder to follow. None of this is a substitute for proper security practices, but it does raise the bar for someone trying to understand or copy your business logic from the browser's developer tools.

4.7 Recommended Implementation Workflow

Based on our experience and the literature, here is what a solid minification setup looks like in practice:

- Always generate source maps alongside minified files. Developers need to be able to debug in their browsers; source maps let them see the original code while users download the minified version. Without source maps, debugging production issues becomes very painful very quickly.

- Lint before you minify. Certain JavaScript patterns eval(), with statements, and some uses of arguments prevent the minifier from safely renaming variables. Running ESLint with a rule set that flags these patterns catches the problem at the source rather than at build time.
- Integrate minification into your CI/CD pipeline. Manual minification is not repeatable and will get missed. Whether you use GitHub Actions, GitLab CI, Jenkins, or something else, minification should run automatically on every build that targets production.
- Use differential serving where possible. Modern browsers support ES2020+ syntax natively, while older browsers need transpiled and more heavily optimized bundles. Serving two separate builds a lean modern one and a more compatible legacy one means modern users are not penalized for the constraints of older environments.

5. CONCLUSION

This paper set out to make the case that minification is a foundational web performance technique, not an optional finishing touch. The evidence supports that claim fairly clearly. A 75% reduction in Time-to-Interactive is not a marginal improvement it is the kind of difference that determines whether real users stay on a page or leave.

What we find most interesting, and somewhat underappreciated, is the interaction between minification and compression. The prevailing assumption in many teams is that modern compression formats like Brotli make minification less necessary. Our data suggests the opposite: the two techniques target different types of redundancy, and doing both yields results that are substantially better than either alone. A 512 KB page that goes to 310 KB with minification, and then to 88 KB when you add Brotli, is a compelling demonstration of that.

Looking forward, the tooling will continue to get faster and smarter. Machine-learning-assisted dead code analysis, more aggressive CSS structural optimization, and tighter integration with edge-computing delivery pipelines are all active areas of development. But the fundamentals described in this paper remove what the browser does not need, compress what remains will stay relevant regardless of how the ecosystem evolves.

6. REFERENCES

[1]. Crockford, D. (2001). JSMin: The JavaScript Minifier. Retrieved from https://www.crockford.com/jsmin.html

[2]. Zakas, N. C. (2012). Maintainable JavaScript. O'Reilly Media.

[3]. HTTP Archive. (2025). Annual State of the Web: Performance Trends. Retrieved from https://httparchive.org

[4]. Surma, J. (2024). Inside V8: Understanding Bytecode, ASTs, and JIT Compilation. Google Chrome Developers Blog.

[5]. Google Chrome Developers. (2024). Why Performance Matters. Retrieved from https://developers.google.com/web/fundamentals/performance

[6]. IETF. (2016). Brotli Compressed Data Format RFC 7932. Internet Engineering Task Force.

[7]. Osmani, A. (2023). JavaScript Start-up Performance. Google Web Fundamentals.

Cite this Article:

Budgude, S. D., & Gholve, S. D. (2026). Minification of CSS, JavaScript, and HTML for faster websites. International Journal of Emerging Research in Computer Science, 2(3), 12–17.

Journal URL: *https://ijercs.com/*

DOI: https://doi.org/10.59828/ijercs.v2i3.23