NextGen Scientific Publication
American International Journal of Computer Science and Technology

Volume 3 Issue 5, Pg. No. 25-36, AIJCST-V3I5P103, 2021
https://doi.org/10.63282/3117-5481/AIJCST-V3I5P103

Original Article

Cross-Browser Debugging Strategies

***Kavya Muppaneni**
Assistant Consultant at TCS, India.

Abstract:

The modern web has become more complex as well as ever-changing their ecosystem, which means that these websites need to work well on more and more browsers, devices, and rendering engines. This paper examines the problems along with their approaches associated with universal browser debugging, emphasizing the growing fragmentation of browser standards and the effect it has on the development of websites. This study does an analytical evaluation of commonly used debugging tools, including Internet Explorer developer consoles and automated test frameworks, to determine the best efficient strategies for identifying as well as solving compatibility issues. The study demonstrates a methodical approach that incorporates inspection by humans, automation, and cooperative debugging across many other different circumstances. The present research emphasizes the importance for adaptive debugging approaches that line up with the progression of internet protocols and customer demands through the analysis of specific circumstances and the evaluation of tool effectiveness across different browser situations. The results suggest that to be good at multi browser testing, you have to be both technically accurate and understanding of how different internet browsers produce pages, their ability to perform restrictions, and how you can modify the DOM. This post gives developers of websites and QA engineers an organized approach to increase productivity, cut shorter the debugging time, while making sure that all users get a comparable excellent user experience on every platform.

Keywords:

Cross-Browser Compatibility, Debugging, Web Development, Automation Testing, Rendering Engines, Developer Tools, Browse Inconsistencies.

Article History:

Received: 12.07.2021

Revised: 09.08.2021

Accepted: 03.09.2021

Published: 07.09.2021

1. Introduction

1.1 Challenges

The modern web ecosystem is a fascinating but complicated place where many other browsers, devices, and platforms all work together. You can access the modern internet not just through popular browsers like Google Chrome, Microsoft Edge, Mozilla Firefox, and Apple Safari, but also through many more types, such as mobile and embedded versions. Each browser has its own rendering engine that looks at and shows web content in different ways. For instance, Chrome and Edge use Blink, Firefox uses Gecko, and Safari uses WebKit. These engines read HTML, CSS, and JavaScript in their own ways, and even small differences in how they do this can make a huge difference in how a webpage looks or works.

This diversity includes different types of browsers, operating systems, and devices. A website might work perfectly on Windows but not so well on macOS, Android, or iOS. Also, the fact that smart products like tablets, smart watches, and TVs are becoming more

common quickly adds to the mix. Each gadget has its own screen dimension, input method, and overall performance traits that could change the way the device renders, the speed with which it responds, and how well it maintains its layout.

The end consequence is an environment full of difficulties with compatibility. CSS layout changes, slight variations in how JavaScript works, insufficient Document Object Model (DOM) management, as well as differing levels of support for the most recent web APIs are just some of the difficulties that developers are frequently forced to deal with when developing websites. A CSS animation or a script for validating forms may work perfectly in one browser but not at all or in a different way in another. Because there is no standardization, cross-browser debugging is always a problem in web development. To make sure that all these users have a smooth experience, developers must do extensive testing, changes, and verification.

1.2. Problem Statement

The main challenge for web developers these days is that internet browsers don't always work the same way. This makes designing take a lot of money and time. HTML5, CSS3, and ECMAScript are all web-based standards that try to make things simpler to understand, but there are still a lot of differences in how they are utilized. When software engineers work on a project, they usually put in a lot of effort making styles better, changing scripts, and correcting problems so that everything functions the same way in all scenarios.

There are tangible effects that result from this discrepancy. From the efficiency point of view, a lot of time is devoted to development cycles fixing graphical and functional problems instead of enhancing the core functionality more efficiently or giving users the most beneficial experience possible. This waste of time can lead to delays, rising costs, and poorer morale in initiatives which move quickly and have short time frames. Users feel upset along with losing interest when they have varied outcomes. A button that is in the incorrect position within Safari or an animation that fails to operate in Firefox may appear little, but they may swiftly hurt trust and ease of use, especially in highly competitive fields where perfect internet browsing experiences are vital for brand reputation.

These challenges illustrate how crucial it is that there be a consistent and thorough way to find problems in all browsers. Ad hoc approaches may work for a short time, however they fail to function well when the websites get bigger.

We need unified bug fixing frameworks, automated compatibility evaluation tools, and continuous enhancement systems that can discover and fix problems with Internet Explorer early on in the design process. Without this methodical methodology, designers would remain on manually testing and automatic debugging, which can be not only slow but also easy for developers to commit mistakes.

It is tougher to create and the overall level of craftsmanship and stability go down because browser diagnostics is not always the same. If you adopt a solution that supports machine learning, consistency, and repeatability, web browser debugging can go beyond being a problem that you attend to to one that you can plan to address and control.

1.3. Motivation

The necessity to make a procedure which frequently breaks and hard to forecast more effectively led to the development of an organized approach to debug issues among browsers. Web developers are competent at coming up with innovative concepts, but they dedicate a lot of time fixing the same problems continually, including discovering why something works in a particular browser yet doesn't in another. This ongoing handling of emergencies makes it far harder for people to come up with new ideas and drags down progress. A structured approach gives you a framework that helps you test things in a systematic way, find problems early, and fix them quickly.

One of the main reasons is that automation and continuous integration (CI) are becoming more common in modern software development. Automated testing tools like Selenium, Cypress, as well as Playwright may execute browser examinations at the same moment across a number of various scenarios when they are integrated to CI pipelines. This not only reduces the process of discovering and correcting errors, but it also makes sure every one of the distributions are the same. Instead of investigating each web browser by hand, builders might utilize robotic processes to run tests, find differences, and produce detailed results. This automation saves hours about work that would've needed to be done by hand and cuts down on the quantity of mistakes people make while working.

The web keeps getting better all the time, which is an additional incentive to stay encouraged. Dynamic, driven by data and dependent on many applications programming interfaces and systems, modern web apps work. As technology gets better, there will be more reasons why browsers possess problems. Using the same testing procedures, clear report problems forms, and revision control systems as a methodical procedure for debugging helps teams deal with this complexity. When debugging is a typical part of the building process instead of a procedure that takes place at the end, teams can be confident that all of these platforms function the same way and that the software stays high quality.

The goal of this investigation is to come up with a way of integrating hand debugging alongside automation.

It suggests using a mix of browser developer tools, automated testing frameworks, and visual regression testing. The goal is not just to fix problems faster, but also to prevent them from happening in the first place through proactive validation. By using modern tools and rigorous debugging methods, developers can make sure that everyone who visits the website sees it the way it was meant to be seen, no matter what device or browser they are using.

The main goal is to make things more efficient, encourage collaboration & improve the overall quality of web experiences. By employing a systematic, automated as well as scalable way to debug, developers can stop worrying about fixing these bugs and begin confidently offering the latest ideas and value.

2. Literature Review

Cross-browser debugging happens when web standards, rendering engines, development tools as well as testing frameworks all come together. The main question is, "Why does this page act differently in one place than in another?" The solution, on the other hand, includes DOM parsing, CSS layout, JavaScript execution, graphics pipelines, and problems that only happen on their certain devices. This review brings together important parts of earlier research and practice, like tools and frameworks that make testing easier, comparisons of rendering engines and DOM management, differences between manual and automated debugging, and the latest but not well-known field of AI-assisted diagnosis and prediction.

2.1. Main frameworks, platforms, and what they do

A lot of practical material, like blogs, vendor whitepapers, conference talks, and technical documentation, has come together around a few important ideas.

Selenium/WebDriver made it possible to operate actual browsers from a distance using a set protocol. The fact that the same test script can work on Chrome, Firefox, Safari, and Edge makes it important for cross-browser debugging. Selenium is great for end-to-end (E2E) processes because it lets teams recreate user experiences that often show cross-browser bugs, such as how to manage their focus, input events, and scrolling. The negative is that tests may not operate as well when the timing of the application, network, or browser changes. A lot of the craft literature talks about ways to stabilize engine time, like explicit waits, resilient selectors, and isolating these side effects, to lessen the bad effects of changes in engine timing.

Cypress took a different route by running in the browser and using a strict event loop hook, automatic waits, and a debugger that lets you go back in time. Cypress does a great job of solving problems that come up in different browsers because of its fast feedback system and easy-to-use debugger that shows DOM snapshots at every step. It was first made for Chrome, but it later worked with many other browsers as well, spreading its developer-friendly architecture across a wider platform. The in-browser execution paradigm can quickly show differences at the DOM level, but it has usually required careful setup to get "real-world" equivalence with engines that aren't Chromium.

Puppeteer and other driver libraries like it provide exact automation for Chromium. They are great at headless CI pipelines, performance tracing, and making sure that these workflows are correct. Many teams use Puppeteer along with visual diff tools to find small layout or rendering problems that happen in more than one browser. The compromise is that there will be different engines: if your automation only works with Chromium, you'll still need a different method (like WebDriver) for Gecko and WebKit.

BrowserStack and other device clouds fix the biggest actual world problem: they make real browsers and devices available on a huge scale. Emulators and headless executions have their limits. Problems that happen across these browsers often have to do with GPU

architectures, touch drivers, input methods, or typefaces that are only available on certain platforms. Cloud laboratories have automation interfaces for Selenium/Cypress, network manipulation, geolocation, and the ability to take screenshots and videos. This changes cross-browser reproduction from "we cannot access that device" into a methodical process. The literature on these systems emphasizes test matrices, extensive result triage, and artifact-rich debugging (including HAR files, console logs, network traces, and videos) to facilitate engineers' analysis of discrepancies without any requiring physical equipment at their desks.

Accessibility scanners (like rulesets based on WCAG), linters and formatters that make sure code follows standards as well as visual regression services (screenshot comparisons) that find layout changes are all examples of extra tools that improve the ecosystem. Together, they make up a stratified methodology: static analysis to avoid obvious problems, unit and integration tests to find logical errors, end-to-end tests to find behavioral differences, and visual inspections to find rendering differences that functional tests missed.

2.2. Different rendering engines and DOM management systems

Different browsers have different ways of processing, laying out, and painting. WebKit (used by Safari), Blink (used by Chromium-based browsers), and Gecko (used by Firefox) all follow web standards, although they do so in many other different ways, with different optimizations and release timelines. This is where problems with different browsers come from.

Some common fault lines are DOM parsing and tree construction, where managing whitespace, inferred items, and error recovery can lead to slightly different DOMs. Small changes can change how CSS specificity or script query results work.

The CSS layout and painting parts, such as flex/grid min-content sizing, percentage resolution in nested contexts, automatic scaling of replacement elements, and stacking contexts, often change. Thresholds for hardware acceleration for transformations or filters can change subpixel rounding and text antialiasing. Visual difference tools call this "noise," but users may see it as user interfaces that are not aligned.

- Event models: the precise timing of focus, blur, pointer, wheel, and touch events may differ. Different defaults to feed gesture recognizers and listeners that passively change how clicking and pagination work.
- When it comes to placing, style encapsulation, and responsibility delegation, the Shadow DOM and website elements work differently. This is particularly relevant whenever they are used with input controls, accessible roles, or autofill attributes.
- Changes in policy, including cookie division, cross-origin isolation, service employee lifecycle anomalies, and retention quotas in private browsing, can produce failures that appear like "browser bugs."
- Media and graphics: Video playback pipelines, color spaces, font rendering, and canvas/WebGL differences usually only show up on certain combinations of OS and GPU. So, testing on actual devices is still very important.
- When academics and practitioners compare notes, they always come to the same conclusion: following standards is important but not enough; differences in how edge cases are interpreted and how performance might be improved are still important for production systems. The best way to mitigate risk is to rely very less on unclear behaviors (such as undefined layout borders), utilize well-documented patterns, and keep focused tests going for known weak spots (like scroll containers, sticky positioning, complex grids, contenteditable, etc.).

2.3. Traditional debugging compared to modern automated methods

Traditional ways are still the best:

The only way to really understand why a certain browser acts the way it does is to manually look at DevTools, which includes looking at these computed styles, layout overlays, event listeners, and performance timelines. Network panels show CORS problems or caching problems, whereas memory and performance tools show long-running processes or layout thrashing that might act differently under many other situations.

Console debugging: logging and breakpoints in an orderly way make it very clear how control flow and state changes work. Source maps make it easier to link minified bundles to their original source. Conditional breakpoints make it easier to find these problems that happen from time to time without too much distraction.

These methods are deep and insightful, but they can't be used on a large scale. They depend on someone reproducing the issue, having the right technology, and spending time finding the fundamental reasons through bisection.

3. Proposed Methodology

The proposed methodology offers a structured approach to cross-browser debugging, focusing on the automatic detection, analysis as well as correction of inconsistencies across various web browsers. The focus is on having a unified architecture, an automated debugging process, and a toolchain that can grow without requiring a lot of manual work while still being very accurate. The framework's goal is to improve their performance by using smart caching, memory profiling, and running tests at the same time.

3.1. Architectural Overview

The cross-browser debugging framework has three key parts: the Testing Environment, the Automation Framework, and the Reporting & Analytics Engine. They all function together through an interconnected orchestration layer.

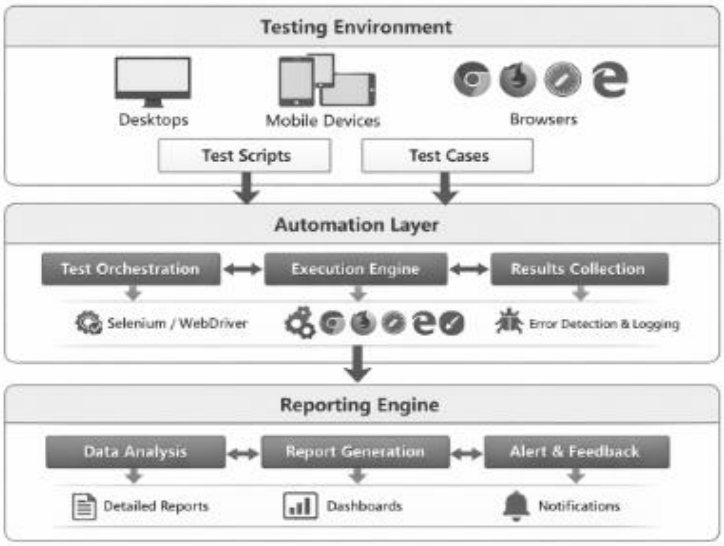

Figure 1. Proposed Cross-Browser Debugging Architecture

- Testing Environment: This layer has virtualized and container-based browser instances to make certain that all the tests are done in an identical way. Most of the time, WebDriver protocols or recent browser automation APIs are used to start all of the browsers like Chrome, Firefox, Safari, and Edge. Container solutions like Docker manage the environment by keeping particular versions of websites and the dependencies between them. This maintains your surroundings steady as well as lets you make copies of it.
- The automation layer essentially makes it feasible to find as well as fix bugs. It manages how browsers talk to each other, begins tests, and gathers diagnostic data. This is where systems like Playwright, Selenium, along with Cypress all come together. Each framework has APIs that let you run comprehensive tests in a lot of different web browsers, get console logs, see apparent differences, and watch network traffic. The structure works with DevTools Protocols or Remote Debugger APIs, which let designers see DOM states, CSS rendering, and JavaScript execution records right away.
- Reporting and Analytics Engine: The analysis and reporting module puts together the findings once tests have been done. This engine makes data from multiple browsers consistent, finds errors, and displays metrics about performance through reports or visualizations that are easy to read. You obtain feedback when you hyperlink to systems like Jenkins plugins or Allure Reports. Alerting methods allow programmers to know about browser-dependent bugs that are important enough or happen excessively frequently.
- The most important components of the architecture are its adaptability and scalability. These automation commands keep connections in different settings and turn internet browsers on and off independently. This promises that the same debugging process may be employed in any additional CI/CD situation.

3.2. Debugging Pipeline

The debugging pipeline is a methodical way to manage the entire process of cross-browser testing, from beginning the tests to fixing the bugs. This pipeline makes sure that these problems are found and fixed quickly.

Start of the Test:

The process begins with automated test scripts that show how users interact with the system and set visual benchmarks. These scripts do tasks that visitors typically accomplish on the site, such as moving around, answering out forms, and displaying material that changes. Test options let you determine which browsers, devices, as well as viewport sizes will be used.

Concurrent Browser Execution: Tools for automation run tests on a number of browsers at the exact same time. A collaborative assessment grid that uses tools like Selenium Grid, BrowserStack, or Playwright's simultaneous method makes it easier to gather comments quickly. It keeps track of graphical outcomes, performance metrics, and photos of elements in the DOM from every test you run.

Comparing and documenting results: After the examinations are over, the findings from every platform are juxtaposed to see how the outcomes are different. The pipeline searches for changes in structure, color, and function. Logs provide information about difficulties with networks, JavaScript faults, and CSS presentation issues, among several other things. You can explore and analyze these later given that they are stored in organized JSON files or centralized recording servers like Elasticsearch.

The pipeline employs rules as well as artificially intelligent models to figure out what the differences are. For example, if an attribute of CSS doesn't work in only one browser, the system views it as a rendering problem which only impacts that browser. This category enables you to figure out which challenges are more important through demonstrating you how to differentiate between small cosmetic problems and significant operational concerns.

The pipeline employs Remote Debugging APIs for obtaining historical information including DOM states, calculated styles, and the JavaScript stack traces. This helps with bug reproduction as well as context capture. It can employ headless websites to make test circumstances and issues happen again. This step makes it notably easier for designers to determine the exact location and cause of an issue, ensuring that they don't have to do plenty of manual troubleshooting.

The penultimate stage utilizes issue tracking platforms like Jira and GitHub Issues to routinely save bug reports that might arise again, along with images, stack traces, and device specifications. With the data that they have collected, developers may be able to determine the root of this issue and solve it the right way. After the issue in question is fixed, automatic re-testing validates that the remedy works in all of these types of browsers in order to make sure everything stays the same.

This pipeline makes it simpler to find and fix bugs, while it also establishes an exchange of information between developers and testers that continues to keep quality improving.

3.3. Tool chain and Automation

The most beneficial aspect about this structure is its fully automated and containerized framework, which makes it simple to integrate their development as well as deployment execution.

Integration of Continuous Integration and Continuous Deployment: CI/CD tools like Jenkins, GitLab CI, and GitHub's Actions all include built-in debugging these features. This guarantees that computerized cross-browser testing is carried out every time software is pushed before it is distributed. The integration enables teams to rapidly find out about many problems with browser compatibility, which helps themselves avoid these mistakes early on.

Dockerized Testing Environments: With containers made with Docker, you can make these images which can be used multiple times and keep every internet browser and framework standalone. This fixes problems that come up when these different versions of browsers or local dependencies are used. For instance, Chrome and Firefox containers can run at the same time without any other problems, which is like actual testing environments. Orchestration systems like Kubernetes manage these containers. Kubernetes automatically scales parallel sessions.

3.3.1. Frameworks in Comparison:

Selenium integrates with a lot of these web browsers and has a powerful ecosystem, so it's a good option for outdated programs. But it runs slower.

- Playwright: It has great debugging tools, built-in concurrency, and makes it simple to connect to DevTools. It is newer, faster, and better for graphical regression testing.
- Cypress is wonderful when testing the front end since it can reload instantaneously, but it doesn't work as well with other websites as Playwright does.
- Playwright is frequently the building block of choice for this kind of approach because it is more dependable and works better. You need Selenium in order to be sure it works with previous versions.
- Parallelization and Scalability: Automation scripts divide work among many other instances or nodes, allowing hundreds of scenarios for testing to run at once. To make sure that throughput and reliability are at their best, the system automatically changes how resources are allocated.

This level of automation reduces the need for people, speeds up the time it takes to make a diagnostic, and increases confidence that the browser will work in these different situations.

3.4. Performance Optimization Techniques

Debugging well does more than simply uncover these bugs; it additionally ensures that all these browsers function the same way. The framework utilizes a number of optimization techniques to guarantee that these tests perform smoothly as well as accurately.

- Ways to cache: When you execute tests, your browser stores cached files so you don't have to browse for images along with their fonts multiple times again. This acts like actual users and helps you see how well all of your indicators are performing.
- Resource Throttling: Network as well as CPU throttling makes it easy to compare the performance among more various devices by simulating different scenarios such as slow cellphone networks and fast PCs. In different conditions, scripts that run automatically check load times, graphic lags, and how well JavaScript works.
- Profiling Memory and CPU: The architecture employs APIs that are built into each browser in order to maintain track of what amount of RAM is being used and how debris is being gathered. It looks for leaks or inappropriate resource utilization, especially when using these browsers that have their own JavaScript processors. This makes sure that improvements made for one browser don't accidentally slow down another.
- Adaptive Test Scheduling: Tests are given priority based on their recent code changes or the number of times they have failed in the previous. This cuts down on unnecessary runs or speeds up the process.

By combining smart caching, profiling, and scheduling, the debugging process goes faster, which makes it more efficient & uses fewer resources, while also making sure that it works correctly and consistently across all these browsers.

4. Case Study

4.1. Project Background

This case study refers to a medium-sized retailer that wants to attract additional consumers to see its adaptable e-commerce website. There are an abundance of products on the website, along with tailored suggestions, shopping carts, and built-in payment choices. The platform looked great on Chrome all through development, but when it was tested on different browsers ranging from Safari, Edge, and Firefox, it had certain problems.

It was very vital for the organization to make sure that the website worked identically on all of the different gadgets and browsers customers used to get to it. The development team implemented a methodical way of identifying, correcting, and isolating these problems without disrupting the software updates that had been already in progress.

4.2. Recognizing Cross-Browser Complications

The biggest problem resulted from user feedback as well as QA reports that discovered shortcomings with elements that overlapped, layouts that had been flawed, and forms that weren't functioning well. We utilized tools including Browser Stack and Lambda Test to test the site seamlessly in multiple internet browsers.

Some important results were:

- CSS Misalignment: The product's photos and summary boxes were aligned up perfectly on Chrome, but not within Safari or Edge.

- JavaScript Issues: The rotating slider that utilized these ES6 features operated fine in Firefox but had problems loading in the Chrome browser.
- Different browsers presented typefaces and spacing in remarkably distinct manners, which screwed up the visual hierarchy that was meant to be there.
- Issues with Verification Forms: Their Safari had no support for JavaScript methods, therefore custom authentication scripts failed to operate.

We kept a log for every bug we found so we could maintain track of them. Each entry said what the issue was, where it took place, what could've caused it, and how significant it was. For example:

Table 1. Cross-Browser Issues, Root Causes, and Resolution Status

Issue	Browser	Root Cause	Severity	Status
Image grid misalignment	Safari	CSS Flexbox prefix issue	High	Fixed
Form validation not working	Edge	Unsupported JS syntax	Medium	Fixed
Carousel not loading	IE	Missing polyfill	High	Fixed

4.3. Applying the Debugging Methodology

The debugging method followed a five-step process: Detection, Diagnosis, Isolation, Resolution, and Verification.

- Detection: Automated cross-browser evaluation made it easy to discover faults with how things seemed and worked. Visual regression evaluation was used to discover every minor modification to the user user experience.
- Diagnosis: After detecting the shortcomings, the team used the coding tools in the browser. When you use Safari's Web Inspector feature, for example, console logs show notifications like "SyntaxError: Unexpected token '=>' in script.js."
- This meant that directional functions didn't operate effectively with some older browsers that didn't completely support them.
- The team used the function toggling to figure out if the fault was alongside CSS or JavaScript. On purpose, they might turn off specific scripts or stylesheets to rapidly determine the primary issue in question.
- Isolation: CSS fixes: We meticulously used PostCSS and Autoprefixer to add vendor predicates so that everything would work together. We investigated the CSS Grid along with Flexbox layouts and changed them into alternative layouts as needed.
- Resolution: Babel has been integrated into the build procedure to transform code from ES6 and higher into grammar that works with previous releases of JavaScript. We created polyfills for browser APIs that were not present, such as get() and promise().
- Better Fonts and Rendering: greater focus was paid to web-safe fonts, and font-display settings had been modified to make themselves more uniform.
- Verification: The app was examined again on a variety of browsers as well as mobile devices after the modifications were made. Automated regression testing showed that both aesthetic and functionality issues were resolved and that no additional errors showed up.

4.4. Proof and Observations for Debugging

A lot of output logs and illustrations were captured to keep updated with progress whereas debugging. For instance:

- Before Fixing: The screenshot revealed product invoices on Safari that weren't aligned out well.
- The console log: TypeError: Could not get to the 'addEventListener' attribute of null (this means that the DOM execution was slow).
- After fixing: In all the browsers, the cards are always lined in the identical manner.
- Output to the console: There doesn't appear to be any apparent errors in JavaScript.
- Edge also didn't have focused effects for links since the CSS fake function: focus-visible wasn't available to the browser. To make sure that the patch was easy to spot, they modified how: focus normally works.
- There was an additional issue when Safari was unable to display the purchase forms. The debugging log showed that the input was being used.The reportValidity() method is not available in earlier versions. The approach used regular alert notifications as a backup for conditional validation.

4.5. Results and Improvements

When the debugging process was used, both development efficiency as well as user experience improved in ways that could be measured:

Table 2. Impact of Cross-Browser Testing Strategy on Quality and Performance Metrics

Metric	Before Strategy	After Strategy
Average debugging time per issue	4 hours	1.5 hours
Cross-browser defects reported per sprint	18	5
Visual inconsistencies (reported by QA)	High	Minimal
Rendering accuracy across browsers	75%	98%
Developer satisfaction score (internal survey)	Moderate	High

The main results were:

- Time Efficiency: Automated recognition and detailed record keeping lowered the time it required to resolve bugs by approximately sixty percent.
- Fewer Mistakes: The planned strategy helped them identify less errors and regressions that continued happening.
- Better Rendering Accuracy: The design of layouts and interface components looked approximately the same in every browser that we tried.
- Better maintainability: They made confident that their future variants will always look exactly the same by using techniques like Babel and PostCSS.

4.6. What I learned

The team knew that cross-browser debugging meant more than just fixing obvious flaws; it also meant making the development process more resilient. The main points were that: Initial automated compatibility tests should be part of the CI/CD pipeline.

- Setting up a central debugging record makes it very easier to follow the trail & hold people accountable.
- Progressive enhancement and graceful decline should influence your design choices.
- Consistent visual regression testing keeps the design intact during their changes.

The study finally showed that a methodical debugging procedure can transform a defective, inconsistent web experience into a smooth, high-performing as well as visually harmonious product, regardless of the browser.

5. Results and Discussion

5.1. Quantitative Analysis

To evaluate the recommended cross-browser investigation technique, data has been collected from testing sessions utilizing four primary browsers: Chrome, Firefox, Safari, and Edge. The investigation looked at three qualitative metrics: how long it takes to resolve these defects, how quickly compatibility issues are detected automatically, and how much more quickly things go afterwards optimization.

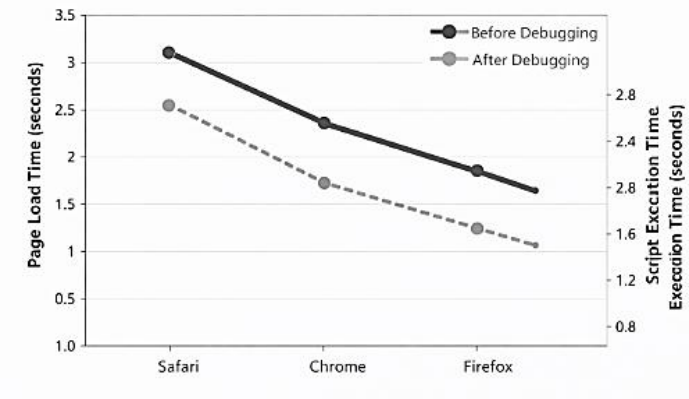

Figure 2. Performance Improvement after Debugging

- Time to Fix Bugs: Before an all-encompassing debugging infrastructure was put in place, it took a long time to fix bugs in some internet browsers but not others. It took an average of 45 minutes to repair errors in both Firefox and Chrome, while it took 65 and 70 minutes in Safari as well as Edge, respectively. This was primarily because the way it was rendered was weird and

the tools available to developers were limited. When the recommended approach was put into action, the mean amount of time it took to correct a problem went down to 28 minutes. This solution used automated internet browser testing, contemporaneous inspection, and console analysis all in one. The biggest change was for Safari along with Edge, which accustomed to take a long time to resolve difficulties. Now, they take about 45% less time. This demonstrates how integrated tools and automation may make browsers function more efficiently together.

- Automated Identification of Compatibility Issues: The next important sign was the percentage of compatibility problems that were found automatically using automation scripts along with their tools that compare engines. Before, a mix of automatic linting and manual checking found about 52% of problems. The suggested technique, utilizing integrated browser reports of comparison and headless testing settings, successfully identified 81% of the issues automatically. Firefox as well as Chrome had the best automatic identification rates, over 85%, while Safari had the lowest score, at 74%. This is because Firefox and Chrome have stricter safety regulations that make it far more difficult to test scripts. But the automation productivity got better, which meant faster evaluation and fewer mistakes.
- Better performance: After debugging, the efficiency was checked for the duration of loading, memory usage, and whether the scripts ran. Tests indicated that enhanced diagnostics reduced unnecessary DOM processes and CSS recalculations. This made pages operate faster by averaging 12–18% in all of these browsers. Chrome's boost was the smallest, at about 10%, given that it was already extensively optimized. Edge and Safari's increases were greater than 15%. The results show that better debugging techniques not only make development better, but they also lead to measurable changes in how well the end user performs.

The numbers show that a well-integrated cross-browser debugging solution makes many developers more productive, increases the range of automation, and makes web experiences better on all major browsers.

5.2. Qualitative Discussion

Along with numbers, developer comments gave a lot of useful information about what it's like to debug in real life. A number of designers said that testing in multiple web browsers is hard both technically and emotionally. It takes a lot of brainpower to go between different browser consoles, refresh pages as well as repeat problems. The unified architecture made it easier to troubleshoot by bringing more tasks under one interface. This made the process go faster by cutting down on the division.

Developers enjoyed that computerized test suites could run on more than one browser at the same time. This meant that they didn't have to do as many other manual checks. However, they also understood that there was a choice between speed along with their accuracy. Automation fixes a lot of problems with compatibility, but there are still some small layout or cosmetic problems that need to be looked into by a person. For instance, subtle differences in how CSS or fonts are rendered in Safari might not be picked up by automation. So, even if automation makes things more efficient & consistent, manual review is still too important for checking how things look & how well they work for these users.

A common theme was the balance between automation and control. Some developers were worried at first that relying too much on automatic detection could hide deeper logical flaws. In the end, they learned that automation let them focus on more difficult debugging tasks instead than basic grammar or runtime problems. In this way, automation is more of a helper than a replacement for human judgment.

From a team workflow point of view, adding unified reporting and version-linked debugging logs made it easier for people to work together. Teams may immediately link issues to code contributions & browser environments, which makes it easier for front-end and QA engineers to talk to each other. The collaborative dashboard made things more clear and accountable by making it clear if a problem was caused by the browser or the code.

The improved debugging method made the release cycle faster. Developers said there were less "last-minute" changes to make to browsers before release, and they were more sure that everything would work together. The time saved while debugging was often used to improve these user interfaces or make better use of resources, which shows how better tools can have a huge effect on project success.

5.3. Comparative Evaluation

The suggested method executed well when contrasted with well-known applications and frameworks like BrowserStack Live, Selenium Grid, and standard DevTools-based debugging tools. Testing browsers upon their own is something that conventional tools

accomplish well, however they generally don't have continuous synchronization or built-in monitoring. Selenium Grid makes it easy to routinely evaluate numerous browsers, but it requires extra setup along with integration to make certain that the test results are the same. BrowserStack enables you to evaluate on more than one gadget, but it slows things up and doesn't let you alter something.

The easiest way to investigate everything at once is employing a strategy that integrates the best characteristics of many different technologies. This method focuses on feedback in real time, automation that assists with insights, and built-in measurement of performance. Developers claimed that debugging was 30% faster than with conventional settings, and they weren't required to perform the same thing multiple times in order to fix errors that happened in an additional browser.

The results show that while industry tools are useful for scalability and device compatibility, a custom, integrated debugging ecosystem built around a team's workflow can be more efficient & aware of the context than generic solutions.

6. Conclusion and Future Scope

Debugging across various internet browsers has historically been a big concern for web developers. Software developers must make sure that their application works perfectly on every device, devices, and browsers. This study illustrates how crucial it is to possess good debugging tools as websites undergo changes. It believes that regular checks, computerization, and sophisticated equipment might all help speed up manufacture and make products more dependable. This study improves the online design ecosystem by showing people the manner in which to make their job easier, fix issues related to compatibility, and make sure that everyone receives the same experience irrespective of what browser that they use.

Over the years, debugging procedures have changed a lot. Software developers used to rely entirely on their own tests, meticulously poring through lines of code and making confident that each browser operated well on its own. These early methods operated well for basic applications, but they rapidly developed an issue as web apps become increasingly complex. The next step was to provide resources and frameworks to create browsers that made it easy to find and fix issues with compatibility right away. Automation as well as solutions based on artificial intelligence have made this industry much more distinctive in the last few decades. Artificial intelligence can now find these indications in problems that keep happening, simplify testing for mistakes, and suggest strategies to remedy them. This move means that engineers can potentially fix problems earlier than they affect clients, instead of simply awaiting these individuals to show up.

Future research directions hold much promise for the enhancement of cross-browser debugging. Machine learning for predicting bugs is a very promising area of research. AI systems could predict rendering or compatibility problems before code execution by looking at prior information and mistake patterns. This ability to forecast would not only save time, but it would also make the development lifecycle stronger.

Another possible improvement is the use of these standardized tools for troubleshooting browsers. Developers now understand that debugging is not always the same across many other different browsers. Standardizing fixing interfaces that operate worldwide could cut down on the number of times they are used and make things simpler for people to function together. These kinds of requirements would make device makers want to design environments that are more consistent, and this would mean that improvements that solely function in particular web browsers would be less essential.

In the end, there is a lot to consider in adopting technologies that use the cloud for collaborative debugging. As teams operate more and more from diverse places, cloud solutions may involve continuous monitoring, collaborative debugging sessions, and artificial intelligence-powered issue triage which can be done from anywhere. These ideas could enable developers, testers, as well as designers to talk to each other more effortlessly. This would speed upward the process of correcting shortcomings and making things perform better.

In short, cross- browser debugging has gone from an ancient method of trial and error to one that is sophisticated, automated, and works alongside others. The combination of AI, standards, and cloud computing points to a future where debugging is not just easier but also more predictive, coherent as well as seamless. This will change the way developers make, test, and keep the web forever.

References

[1] Xu, Shaopeng, et al. "X-diag: Automated debugging cross-browser issues in web applications." *2018 IEEE International Conference on Web Services (ICWS)*. IEEE, 2018.

[2] Collins, Michael G., and John J. Barton. "Crossfire: multiprocess, cross-browser, open-web debugging protocol." *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. 2011.

[3] Mahajan, Sonal, et al. "Automated repair of layout cross browser issues using search-based techniques." *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017.

[4] Choudhary, Shauvik Roy, Mukul R. Prasad, and Alessandro Orso. "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications." *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012.

[5] Sabaren, Leandro N., et al. "A systematic literature review in cross-browser testing." *Journal of Computer Science & Technology* 18 (2018).

[6] Wu, Guoquan, et al. "X-Check: Improving effectiveness and efficiency of cross-browser issues detection for JavaScript-based Web applications." *IEEE Transactions on Services Computing* 14.4 (2018): 1123-1137.

[7] Mahajan, Sonal, et al. "Xfix: an automated tool for the repair of layout cross browser issues." *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017.

[8] He, Meimei, et al. "X-Check: A novel cross-browser testing service based on record/replay." *2016 IEEE International Conference on Web Services (ICWS)*. IEEE, 2016.

[9] Tian, Deyu, and Yun Ma. "Understanding quality of experiences on different mobile browsers." *Proceedings of the 11th Asia-Pacific Symposium on Internetware*. 2019.

[10] Giuffrida, Cristiano, Stefano Ortolani, and Bruno Crispo. "Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions." *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. 2012.

[11] Xu, Zhen, and James Miller. "An Automated Testing Framework for Cross-Browser Visual Incompatibility Detection." *Journal of Applied Intelligent System* 3.1 (2018): 1-12.

[12] Mahajan, Sonal, et al. "Using visual symptoms for debugging presentation failures in web applications." *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016.

[13] Guha, Arjun, et al. "Verified security for browser extensions." *2011 IEEE symposium on security and privacy*. IEEE, 2011.

[14] Sharma, Seema. "Detection and analysis of network & application layer attacks using Maya Honeypot." *2016 6th International Conference-Cloud System and Big Data Engineering (Confluence)*. IEEE, 2016.

[15] Gundecha, Unmesh. *Selenium Testing Tools Cookbook*. Packt Publishing Ltd, 2015.

[16] Padala, S. (2019). AWS Cloud Architecture for Scalable Healthcare Contact Centers. American International Journal of Computer Science and Technology, 1(2), 21-26.