OPTIMIZING WEB DELIVERY: THE IMPACT OF RENDERING METHODS
ON USER EXPERIENCE ACROSS NETWORK CONDITIONS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Cagan Sevencan

June 2024
COMMITTEE MEMBERSHIP

TITLE: Optimizing Web Delivery: The Impact of Rendering Methods on User Experience Across Network Conditions

AUTHOR: Cagan Sevencan

DATE SUBMITTED: June 2024

COMMITTEE CHAIR: Franz Kurfess, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Javier Gonzalez-Sanchez, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Rodrigo Canaan, Ph.D.
Professor of Computer Science
ABSTRACT

Optimizing Web Delivery: The Impact of Rendering Methods on User Experience Across Network Conditions

Cagan Sevencan

In an era where web applications are pivotal for global information dissemination and user engagement, ensuring the performance and accessibility of static web content is paramount. This need is particularly significant given the diverse accessibility requirements worldwide, influenced by varying network generations and the real-time rendering of static elements such as text, images, and videos on devices ranging from laptops to cell phones.

This thesis embarks on a comparative study of client-side rendering (CSR) and server-side rendering (SSR), two fundamental techniques determining how various platforms present static content. Through detailed examination, the study aims to shed light on the nuances of web performance optimization, particularly how CSR and SSR affect content loading times, interactivity, and overall user experience. This investigation covers popular web browsers like Google Chrome, Brave Browser, and Microsoft Edge while considering network generations from 2G Good to 4G/LTE Regular. The analysis will utilize performance metrics such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), Cumulative Layout Shift (CLS), and Finish Time to provide a comprehensive evaluation. These metrics are essential for assessing the performance impact of CSR and SSR under different network conditions, offering a tangible measurement of user experience. By exploring the interplay between rendering techniques, device capabilities, and network environments, the research seeks to demystify the complexities of web performance optimization, ultimately offering strategies to enhance web application performance across different digital ecosystems.
ACKNOWLEDGMENTS

Thanks to:

- Dr. Kurfess for his guidance throughout this project.
- My parents for their support. Their sacrifices are what allowed me to pursue a higher education and gave me the motivation to try my best.
- Manpreet Singh for his mentorship and guidance.
- My girlfriend for her patience, understanding and unwavering support.
- My sister for her encouragement and faith in my abilities.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
</tbody>
</table>

## CHAPTER

1 INTRODUCTION .......................... 1

1.1 Overview ............................ 1

1.2 Chapter Outline ...................... 3

2 BACKGROUND ........................... 4

2.1 Metrics and Tools .................... 4

2.1.1 Browser-Centric Metrics ............. 4

2.1.2 User-Centric Metrics ............... 4

2.2 Core Web Vitals ..................... 5

2.2.1 First Contentful Paint (FCP) ........ 6

2.2.2 Largest Contentful Paint (LCP) ...... 8

2.2.3 Cumulative Layout Shift (CLS) ...... 9

2.2.4 First Input Delay (FID) ............ 11

2.3 Other Performance Metrics ........... 12

2.3.1 Time to First Byte (TTFB) .......... 12

2.3.2 Time To Interactive (TTI) .......... 13

2.3.3 Total Blocking Time (TBT) .......... 14

2.3.4 Interaction to Next Page (INP) ...... 16

2.4 Why Care About Performance? .......... 17

2.5 State of the Mobile Web .............. 18
# RELATED WORK

3.1 Client-Side Rendering vs Server-Side Rendering  . 22

3.2 Web Vitals and Performance Measurement  . 24

# RENDERING APPROACHES TO PERFORMANCE METRICS

4.1 Web Rendering  . 27

4.2 Hydration  . 29

4.3 Rendering Approaches  . 30

4.3.1 Client-Side Rendering (CSR)  . 30

4.3.2 Server-Side Rendering (SSR)  . 31

4.3.3 Alternative Rendering Approaches  . 33

# IMPLEMENTATION

5.1 Experiment Setup  . 36

5.1.1 Browser & Device Specifications  . 38

5.2 Overview of Next.js  . 38

5.3 Network Generations  . 39

5.4 Lighthouse & DevTools  . 41

5.5 Experiment Methodology  . 42

5.5.1 Data Collection  . 45

# RESULTS AND DISCUSSION

6.1 Finish Time and Load Time  . 47

6.1.1 Client-Side Rendering  . 50

6.1.2 Server-Side Rendering  . 51

6.1.3 Summary  . 51

6.2 Largest Contentful Paint (LCP) & First Contentful Paint (FCP)  . 53

6.2.1 LCP Analysis  . 55
<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>39</td>
</tr>
<tr>
<td>Network Throttling Profiles</td>
<td>39</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Core Web Vitals metrics and thresholds [1]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>FCP Timeline [2]</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Good FCP values are 1.8 seconds or less. Poor values are greater than 3.0 seconds [2]</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>LCP timeline from cnn.com [3]</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>LCP Metric Range [3]</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>CLS example from NYTimes [4]</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>CLS Metric Range - individual layout shift scores [4]</td>
<td>10</td>
</tr>
<tr>
<td>2.8</td>
<td>FID Facebook Login</td>
<td>11</td>
</tr>
<tr>
<td>2.9</td>
<td>FID Metric Range [5]</td>
<td>12</td>
</tr>
<tr>
<td>2.10</td>
<td>TTFB measures the elapsed time between startTime and responseStart [6]</td>
<td>13</td>
</tr>
<tr>
<td>2.12</td>
<td>TTI Visualization [6]</td>
<td>14</td>
</tr>
<tr>
<td>2.13</td>
<td>TBT Visualization - Long Tasks Blocking Main Thread</td>
<td>15</td>
</tr>
<tr>
<td>2.14</td>
<td>TBT Visualization - Excess Duration Beyond 50ms</td>
<td>15</td>
</tr>
<tr>
<td>2.15</td>
<td>The life of an interaction [7]</td>
<td>16</td>
</tr>
<tr>
<td>2.16</td>
<td>INP Values [7]</td>
<td>16</td>
</tr>
<tr>
<td>2.17</td>
<td>Asia Pacific Report 2023 [8]</td>
<td>19</td>
</tr>
<tr>
<td>2.18</td>
<td>Latin America Report 2023 [8]</td>
<td>19</td>
</tr>
<tr>
<td>2.19</td>
<td>Parse times for a 1MB bundle of JavaScript across desktop &amp; mobile devices of differing classes. [9]</td>
<td>20</td>
</tr>
</tbody>
</table>
4.1 Rendering Patterns ............................................... 28
4.2 Client-Side Rendering [10] ................................. 30
4.3 Server-Side Rendering [10] ................................. 32
4.4 Pre-rendering (Next.js) [11] ......................... 34
4.5 CSR with Prerendering [12] ......................... 35

5.1 Google Lighthouse Example ................................. 36
5.2 Chrome Dev Tools Metrics ............................... 37
5.3 Throttling Options in Chrome DevTools ............... 40
5.4 Chrome internal settings for DNS and socket pools .... 41
5.5 Next.js App Example Page ............................... 44
5.6 Disabling JavaScript in Chrome DevTools ............... 45

6.1 Finish and Load Times for Google Chrome ............... 48
6.2 Finish and Load Times for Brave Browser ............... 49
6.3 Finish and Load Times for Microsoft Edge .............. 50
6.4 Finish vs Load Time in Chrome Browser ............... 52
6.5 LCP for Google Chrome, Microsoft Edge, and Brave Browser 54
6.6 FCP for Google Chrome, Microsoft Edge, and Brave Browser 57
6.7 CLS for Google Chrome, Microsoft Edge, and Brave Browser 60
6.8 Finish Time Comparison Across Browsers ............... 63
6.9 Finish Time Comparison Across Browsers ............... 65
6.10 LCP Comparison Across Browsers ..................... 67
6.11 FCP Comparison Across Browsers ..................... 69
6.12 CLS Comparison Across Browsers ..................... 71
Chapter 1

INTRODUCTION

1.1 Overview

In the evolving landscape of the internet, web applications serve as the cornerstone for disseminating information, enabling interactions, and crafting captivating user experiences globally. Central to these applications are static elements such as text, images, and videos, which not only facilitate content delivery but also significantly influence the user’s navigational experience. Studies have shown that increased page load times may lead to customer abandonment, underscoring the importance of efficient content rendering [13]. This thesis delves into the nuanced domain of rendering these static components, a critical aspect determining the efficiency and responsiveness of web applications across various devices and network conditions.

At the core of our exploration are two predominant rendering techniques: Client-Side Rendering (CSR) and Server-Side Rendering (SSR). CSR, characterized by its dynamic content generation on the client-side, leverages the capabilities of the user’s device to render web content. In contrast, SSR takes a more traditional approach by pre-rendering content on the server, thus delivering a fully prepared page to the client’s browser. This dichotomy presents a complex landscape for developers and users alike, influencing the development process and end-user experience in terms of loading times, interactivity, and accessibility.

The primary inquiry of this research revolves around understanding how these rendering methods impact the performance and display of web content across various web browsers such as Google Chrome, Brave Browser, Microsoft Edge, and network gener-
ations (2G, 3G, 4G/LTE). A significant facet of this investigation is the examination of network transitions, such as the upgrade from a 2G Good to a 3G Slow network, and their implications on the rendering techniques’ efficacy. In order to conduct a thorough analysis, this thesis will include specific metrics such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), Cumulative Layout Shift (CLS), and Finish Time. These metrics are not functional requirements, but rather important factors in evaluating the performance implications of both client-side rendering (CSR) and server-side rendering (SSR) under varying network conditions.

Numerous contributions have been made in the fields of web development and optimization, such as the paper ”Evolution of Client-Side Rendering over Server-Side Rendering” by Pushkar Kishore and M. MahendraB. and [14], and ”On the Comparison of Software Quality Attributes for Client-side and Server-side Rendering” by Beke [15]. These papers provide insightful comparisons of different rendering techniques. Building upon the foundation established by these scholars, this study extends existing knowledge by simulating real-world network scenarios, factoring in varying network conditions and their impact on different rendering techniques. To ensure a comprehensive evaluation, this study also considers the use of various web browsers. The aim is to offer insights into optimal web development and optimization strategies against the backdrop of increasingly diverse and mobile internet usage.

As we navigate through this thesis, the reader will be equipped with a thorough understanding of the implications of rendering strategies in the digital age, informed by empirical data and contextualized within the broader web development and optimization discourse.
1.2 Chapter Outline

The remainder of this thesis is organized as follows:

- **Chapter 2: Background** delves into the fundamental concepts, including performance metrics, tools, and Core Web Vitals, essential for understanding web performance. It clarifies the role of mobile web optimization and underscores the importance of performance in enhancing user experience.

- **Chapter 3: Related Work** thoroughly examines studies that contrast CSR with SSR while shedding light on Core Web Vitals and the methodologies used to gauge web performance. It pinpoints the unexplored areas this thesis aims to address, setting a solid foundation for the investigation ahead.

- **Chapter 4: Methodology** outlines the methods employed in this study, focusing on the mechanics of CSR and SSR and their role in web rendering.

- **Chapter 5: Implementation** presents the experimental setup, detailing the selection of web browsers, network conditions, and the chosen metrics. It explains the methodological approach to investigating the performance implications of CSR and SSR.

- **Chapter 6: Results and Discussion** examines the findings from the empirical study, focusing on analyzing core web vital metrics within the contexts of CSR and SSR and across various network generations. Alongside our findings, we incorporated a comparative study of how different browsers stack up against each other.

- **Chapter 7: Conclusion** summarizes the study’s key findings and implications, offers recommendations for web developers, and suggests future research directions.
This chapter outlines various web metrics, rendering patterns, user experience principles, and design patterns for optimizing front-end web performance, providing a better understanding of the techniques used in the following chapters.

2.1 Metrics and Tools

Understanding web performance begins by identifying areas for improvement. Metrics play a crucial role in evaluating critical aspects of user experience, such as loading speed, interactivity, and visual stability. These metrics can broadly be divided into two categories: browser-centric and user-centric.

2.1.1 Browser-Centric Metrics

A metric that is important to the browser and the user is not affected directly by the metric. It consists of metrics like Time to First Byte (TTFB), First Paint, and Page Load.

2.1.2 User-Centric Metrics

User-centric metrics are crucial as they shape the web experience. These metrics cannot be defined or expressed purely through code because they relate to user-perceived performance. Key metrics include First Interactive (also known as Time To Interactive), Speed Index, and Largest Contentful Paint (LCP). Users also have
the freedom to create their own metrics. For instance, Twitter once used a metric called ‘time to first tweet’ [16].

2.2 Core Web Vitals

Before we start talking about different rendering patterns and performance optimizations, it is crucial to understand the Web Vitals [1]. Google Chrome provides valuable measurements used in its search index to assess website performance and user experience. A subset of these measurements, known as Core Web Vitals, consists of various metrics that may change over time. Led by the Google Chrome team, this initiative aims to pinpoint the crucial metrics that web developers must prioritize to improve web performance. These metrics are assessed at the 75th percentile for both mobile and web platforms, meaning the performance is measured against a benchmark where the majority (75%) of page views meet or exceed the standard.

Core Web Vitals encompass three main metrics: Largest Contentful Paint (LCP), First Input Delay (FID), and Cumulative Layout Shift (CLS). Each has defined thresholds that delineate performance as “good”, “needs improvement”, or “poor”. A page or site’s overall performance is determined by focusing on the 75th percentile of all page views. Essentially, a site is considered to have ”good” performance if at least 75 percent of page views meet the ”good” threshold; on the contrary, if at least 25 percent of page views meet the ”poor” threshold, then the site is considered to be having ”poor” performance. For instance, achieving a 75th percentile LCP of 2 seconds is considered ”good,” while a 75th percentile of 5 seconds is regarded as ”poor” [1]. By reflecting the majority of users’ experience, this method provides a realistic and comprehensive assessment of a site’s speed and responsiveness.
In addition to these core metrics, Web Vitals encompasses other significant indicators, such as First Contentful Paint and Total Blocking Time, which offer further insights into web performance measures. Given their significance in providing a rounded view of a site's speed and responsiveness, these additional metrics will be the focus of subsequent discussions.

### 2.2.1 First Contentful Paint (FCP)

First Contentful Paint (FCP) measures the interval from the initiation of a page load—triggered by a user clicking a link or entering a URL—to the appearance of the first significant piece of content on the screen. In the context of this metric, 'content' typically encompasses text, images, SVG elements, or non-white canvas elements. In the illustrated example shown in Figure 2.2, FCP occurs when the first text and image elements become visible to the user in the second frame of the page load timeline.

**Figure 2.1: Core Web Vitals metrics and thresholds [1]**
This instance marks the initial point at which the user perceives visual feedback from the page, signaling that the content is actively loading. Essentially, this metric captures the moment the user first perceives any visual response from the page, indicating that the page is loading. This immediate feedback assures users that the website is operational and the requested content is forthcoming, thus alleviating any uncertainty during the waiting period.

**Figure 2.2: FCP Timeline [2]**

**Figure 2.3: Good FCP values are 1.8 seconds or less. Poor values are greater than 3.0 seconds [2]**
2.2.2 Largest Contentful Paint (LCP)

During the loading process, various segments of the webpage become visible at different times, allowing us to assess the page’s visual development progress. We gauge this progress by identifying the moment when the largest section of the webpage becomes visible, known as the Largest Contentful Paint (LCP) [3]. This metric varies with screen size due to differences in viewport dimensions. Therefore, responsive designs that adjust to screen width can lead to different LCP values across devices such as mobiles, desktops, or tablets.

LCP captures the render time of the largest image or text block within the viewport from when the user first accesses the page. This measurement focuses solely on the pixel area of content elements. With each rendering cycle, the browser evaluates the dimensions of newly visible elements, calculating their total area. The point at which the content with the largest area renders defines the LCP. Figure 2.4 demonstrates how the browser continuously calculates different LCP candidates throughout the page load. For instance, in the third image, an element initially highlighted as the potential LCP eventually gets superseded by the actual image that becomes the actual LCP.
definitive LCP. Essentially, LCP serves as an indicator of when users perceive the webpage as being nearly fully loaded.

![LCP Metric Range](image)

**Figure 2.5: LCP Metric Range [3]**

### 2.2.3 Cumulative Layout Shift (CLS)

Cumulative Layout Shift (CLS) is a key Core Web Vital metric for assessing visual stability. It is a user-centric measure that quantifies the frequency of unexpected layout shifts to enhance user experience by promoting a stable and delightful page environment [4]. CLS fundamentally represents the product of two factors: the distance fraction and the impact fraction. The distance fraction accounts for the percentage of the largest viewport measurement altered, while the impact fraction measures the portion of the viewport affected by the layout shift.

Consider a scenario where a user navigates a webpage with the intention of interacting with content, such as clicking a button. However, due to asynchronous asset loading or dynamic DOM element insertion, the content shifts unexpectedly, potentially leading to interaction with an unintended item, such as an ad. These involuntary content movements, known as layout shifts, can recur multiple times, leading to user frustration. CLS captures the totality of these shifts throughout the page’s lifespan, offering a comprehensive view of its stability.
Figure 2.6: CLS example from NYTimes [4]

Figure 2.6 illustrates an example of such a shift, including a top banner ad that displaces the entire page downward. Similarly, if a primary image loads slowly, it can disrupt the layout as it appears, contributing to the overall CLS score. Notably, the measurement of CLS is not confined to the page’s loading phase but extends over its entire lifetime, accounting for any content added dynamically, such as through infinite scrolling. This process, which introduces new content and displaces existing elements, further impacts layout stability.

Figure 2.7: CLS Metric Range - individual layout shift scores [4]
2.2.4 First Input Delay (FID)

First Input Delay (FID) measures the duration from a user’s first interaction with the page—clicking a link, tapping a button, or engaging with a custom, JavaScript-powered control—to the moment the browser can start processing event handlers in response [5]. FID evaluates a webpage’s readiness from a user’s perspective. If the browser is busy with tasks such as downloading or processing JavaScript when a user initiates an interaction, it might extend the response time. FID quantifies this delay, offering insight into the user’s initial impression of the site’s interactivity and responsiveness.

![Figure 2.8: FID Facebook Login](image)

Figure 2.8 exemplifies this scenario: after the Largest Contentful Paint (LCP) loads, a user attempts to click on the "Log In" button. Meanwhile, the browser, preoccupied with other tasks, postpones the user’s actions until it completes them. Optimizing other Core Web Vitals scores, such as LCP and FCP, may involve deferring significant processing until after the content loads. While generally beneficial, excessive deferment can lead to substantial JavaScript execution pending post-content load. Consequently, when a user interacts with the seemingly ready page, the browser may still be busy executing deferred tasks, causing a delay before the first click handler
activates. FID only comes into play during user interactions; without any interaction, this metric would remain unrecorded.

![FID Metric Range](image)

**Figure 2.9: FID Metric Range [5]**

2.3 Other Performance Metrics

When examining Core Web Vitals, recognizing additional metrics becomes essential. These metrics cover various performance aspects relevant to user experiences, extending beyond the scope of Core Web Vitals to provide more detailed insights. Websites can leverage these detailed metrics for benchmarking against competitors, thereby enhancing their understanding of performance in a broader context.

2.3.1 Time to First Byte (TTFB)

Time to First Byte (TTFB) measures the time from the moment one makes a resource request until the first byte of the response arrives [6]. It signifies when the client receives the first byte from the server, and the parser begins to process the HTML. As shown in Figure 2.10, TTFB includes Redirects, HTTP Caches, DNS domain lookups, and other service connections and workers until the reception of the first byte of the response.
While TTFB does not classify as a Core Web Vital and exerts a minimal direct effect on metrics, it offers valuable insights for enhancing First Contentful Paint (FCP) and Largest Contentful Paint (LCP) scores [17].

![TTFB Diagram](image)

**Figure 2.11: TTFB Metrics [6]**

### 2.3.2 Time To Interactive (TTI)

Time To Interactive (TTI) marks the point following the completion of a lengthy task when the main thread has remained free from any such tasks for a minimum of five seconds. According to MDN Web Docs [18], the system classifies any task occupying the main thread for over 50 milliseconds as a lengthy task. Should a user attempt interaction during these extensive tasks, the browser’s response time will extend,
potentially hampering performance as these tasks monopolize the main thread and delay user interaction.

Figure 2.12: TTI Visualization [6]

Figure 2.12 demonstrates the calculation of long tasks: after initiating First Contentful Paint (FCP), we identify a 'quiet window,' which is five seconds without any lengthy tasks, with the allowance of no more than two network GET requests. The conclusion of the final lengthy task preceding this interval helps identify TTI, with the span from FCP to this point representing the TTI duration. If no long tasks exist, FCP can be used as the TTI.

2.3.3 Total Blocking Time (TBT)

Total Blocking Time (TBT) emerges as a pivotal metric alongside Largest Contentful Paint (LCP) and First Contentful Paint (FCP). It measures the duration during which
long tasks obstruct the main thread, impacting a page’s usability. TBT reflects the page’s responsiveness prior to achieving full interactivity [19].

![Figure 2.13: TBT Visualization - Long Tasks Blocking Main Thread](image1)

![Figure 2.14: TBT Visualization - Excess Duration Beyond 50ms](image2)

Figures 2.13 and 2.14 showcase the identification of tasks exceeding 50 milliseconds from the onset of First Contentful Paint to the end of Time To Interactive. The first figure shows tasks that prolong over 50 milliseconds, blocking the main thread. The methodology involves subtracting 50 milliseconds from each lengthy task to determine its blocking time, subsequently aggregating these durations. In this specific timeline, three tasks contribute to a combined TBT of 149 milliseconds (40 + 5 + 104), representing the excess duration beyond the 50 millisecond mark. TBT serves as a reliable measure of a website’s interactivity level before reaching TTI, making it a reliable metric due to its resilience against outliers compared to TTI. Google regards a TBT of less than 200 milliseconds as a “good” metric for optimal user experience [19].
2.3.4 Interaction to Next Page (INP)

As of March 2024, Interaction to Next Paint (INP) will join the Core Web Vital metrics and is poised to replace First Input Delay (FID) [7]. INP measures the average response time from user interaction to the visual update on the webpage. This metric comes into play when a user exits the webpage, capturing the site’s or a particular page’s responsiveness to user inputs until the changes are visible.

![Figure 2.15: The life of an interaction [7]](image)

Figure 2.15 illustrates that INP accounts for processing events like pointer, mouseup, and click, as well as rendering, painting, and compositing. Ultimately, the system identifies the highest (worst) INP score recorded throughout the page’s lifespan upon the user’s departure. Since JavaScript plays a crucial role in driving interactivity, the extent of INP largely depends on the frequency and nature of user interactions, including clicks, taps, or keystrokes.

![Figure 2.16: INP Values [7]](image)

**Figure 2.16: INP Values [7]**
2.4 Why Care About Performance?

Measurement, optimization, and monitoring performance are essential to achieving optimal results for any modern web experience. High-performing websites are more likely to engage and retain users, so optimizing for user-centric happiness metrics is key to understanding and improving website performance. Fortunately, tools such as Lighthouse [20] (accessible through web.dev) are instrumental in identifying these metrics and guiding performance enhancements. Implementing and adhering to performance budgets ensures sustained fast loading times and user satisfaction post-launch.

The significance of page performance is underscored by research linking Core Web Vitals to improved user engagement and business outcomes [1]. For instance, meeting Core Web Vitals thresholds has been associated with a 24% decrease in user abandonment during page loads [21]. Moreover, incremental improvements in metrics such as Largest Contentful Paint (LCP) and Cumulative Layout Shift (CLS) have demonstrated tangible benefits. A 100ms reduction in LCP led to a 1.3% increase in conversion rates for Farfetch [22], while a 0.2 reduction in CLS resulted in a 15% rise in page views per session and 13% longer session durations for Yahoo! JAPAN [23]. Similarly, Netzwelt’s focus on Core Web Vitals yielded an 18% rise in advertising revenue and a 27% increase in page views [24]. RedBus experienced a significant uplift in domain rankings globally by reducing its CLS from 1.65 to 0 [25].

Since the integration of Core Web Vitals by Google in 2021, these metrics have become a pivotal ranking factor, significantly influencing a site’s visibility on the world’s premier search engine. While the primacy of content relevance and quality remains unchanged, Google now favors websites offering superior page experiences, notably those excelling in Core Web Vitals. This adjustment means that in scenar-
ios where competitors produce content of comparable excellence, superior Core Web Vitals scores could be a decisive advantage in achieving higher search page rankings. Research by Sistrix reveals that websites meeting all Core Web Vitals criteria gain an additional 3.7% visibility on Google’s Search Engine Results Pages compared to those without [26]. This tangible SEO signal from Google underscores the importance of prioritizing the enhancement of Core Web Vitals for any site reliant on SEO.

2.5 State of the Mobile Web

The mobile web has become a dominant force in the digital landscape now that mobile devices are the primary source for more than half of the world’s online traffic [27]. This shift has led to changes in web development practices, emphasizing mobile-first design and performance optimization. A 2023 study by GSMA Intelligence [8] presented data from 2022 and projected expectations for 2030, highlighting the varying degrees of network connectivity across regions. The report shows that only 60% of worldwide connections are 4G, and a mere 12% are 5G, with a substantial portion on slower networks. In regions like Asia Pacific and Latin America, older generation networks such as 2G and 3G are still prominent, accounting for 15% and 9% (2G) and 11% and 25% (3G), respectively, which are significantly higher percentages than in North America or Europe (Figures 2.17 & 2.18). These insights underscore the importance of optimizing web performance to meet diverse global audiences’ needs, especially those dealing with slow network speeds.
Latency—the time delay between sending and receiving a data packet—becomes a critical factor in user experience, particularly in 2G and 3G environments, where it can severely degrade web performance [28]. In 2G and 3G environments, HTTP requests—essential for web interaction and involve sending and receiving data—suffer from exceptionally high latency, further deteriorating the web experience in these network conditions. Despite improvements with 4G and 5G, these networks still fall short compared to conventional Digital Subscriber Line (DSL) internet speeds in terms of latency. Additionally, many users worldwide have slower phones with slower CPUs and GPUs, leading to longer Parse/Compile times than on desktops [9].
The focus on mobile web performance is often underestimated. Much of the testing is focused on desktop setups that fail to account for the specific hurdles presented by cellular networks. Moreover, when mobile performance is evaluated, tests are commonly run on desktop browsers, notably Chrome, rather than on actual mobile devices. This approach fails to address issues related to latency and bandwidth and does not accurately replicate the true mobile user experience. This oversight is critical given Google’s mobile-first indexing, which prioritizes mobile versions of websites for ranking, highlighting the imperative to optimize for mobile. To deliver an optimal user experience across all platforms, developers must address not just the aesthetic aspects but also the technical performance challenges inherent to mobile web browsing.
Chapter 3

RELATED WORK

While the academic community has yet to extensively explore rendering paradigms for client-side and server-side rendering, several publications from leading companies like Airbnb [29] and Netflix [30], as well as the personal accounts of developers or developer teams [31], offer insights into the advantages and disadvantages of each approach. These contributions, often found in blog posts, provide valuable perspectives on these rendering techniques’ practical applications and challenges. Though academic literature on this topic is scarce, some scholarly papers, such as "Evolution of Client-Side Rendering over Server-Side Rendering" by Pushkar Kishore and M. MahendraB. [14] and "On the Comparison of Software Quality Attributes for Client-side and Server-side Rendering" by Beke [15], offer comprehensive analyses and comparisons of rendering methods, contributing valuable perspectives to a predominantly industry-led discussion. Moreover, [32] offers additional comparative insights into rendering methods concerning metrics such as First Contentful Paint (FCP), Speed Index, and Time to Interactive (TTI), further enriching the understanding of web performance optimization.

Several studies extend the conversation beyond rendering techniques alone. For instance, [33] investigates how load times critically affect e-commerce conversion rates. Additionally, [34] and [35] explore the wider impacts on web performance, emphasizing the crucial role of Web Vitals in assessing user experience, a key aspect I will address in my paper.
3.1 Client-Side Rendering vs Server-Side Rendering

Choosing between client-side and server-side rendering paradigms significantly impacts various aspects of web development, including page load times, development and maintenance costs, search engine optimization, and browser compatibility. These choices can influence user engagement, with longer load times potentially leading to customer abandonment.

Several papers discuss the implications and shortcomings of preferring one rendering method over another. Beke’s research [15] provides an in-depth analysis of the quality attributes of both rendering methods through a pilot study of self-written web applications and a case study on existing open-source projects. The paper also provides a decision tree to guide developers in choosing between these rendering paradigms.

This study encompasses performance benchmarks, lines of code measurements, and availability monitoring, focusing on projects developed in Go, Laravel, and WordPress, employing Vue as the front-end framework for both rendering approaches. The case studies further extend to applications utilizing major front-end frameworks such as Vue, Angular, and React.

Beke concluded that client-side rendering offers advantages in server bandwidth, server throughput, subsequent page load times, usability, and scalability. In contrast, server-side rendering shows benefits for initial page loads, development efforts, search engine optimization, and browser compatibility. The paper concludes with revenue calculations based on load times and conversion rates to aid developers in selecting the most suitable rendering paradigm for their projects, thereby optimizing performance.
Two additional papers delve into the nuances of client-side and server-side rendering. Kishore et al. [14] traces the transition from server-side to client-side rendering in web applications, highlighting robustness, availability, and responsiveness improvements. It also highlights the adoption of frameworks like reactive and functional programming and flux architecture and their impact on application scalability. This study explores the history of server-side rendering and its prevalent usage and highlights its significant shortcomings. The paper also declares that client-side rendering addresses many of these limitations, evidenced by its widespread acceptance among the developer community. Overall, it concludes by noting that many disadvantages of server-side rendering, such as full page reloads on route changes and memory leaks, have been fully resolved with client-side routing.

Another paper by Iskandar [32] discusses the shift towards client-side rendering for more efficient software development. It compares technical aspects and emphasizes the importance of proper planning and architecture selection. Though not in-depth, it focuses on choosing the correct architecture in software development, using PHP for server-side rendering, and preferring JavaScript for client-side rendering in single-page applications. The study’s main findings are that client-side rendering is quicker and more efficient in response to the increasing demand for web applications. Besides offering superior Search Engine Optimization (SEO), server-side rendering performs better in specific technical metrics such as First Contentful Paint (FCP), Speed Index, and Time to Interactive (TTI). Conversely, client-side rendering enhances accessibility, marking a pivotal consideration in the ongoing discourse on rendering paradigms.
3.2 Web Vitals and Performance Measurement

One notable paper I encountered during my research was from Sroka et al. [34], which examined geoportal performance in rural areas with limited internet access. The study explored how users in such areas could enhance geoportal speeds and whether performance metrics could be tailored to specific locations, connection speeds, and devices. Sroka et al. adopted a methodology that tested geoportals in rural mountainous regions, focusing on user experience and portal quality, with performance measurements regarding Core Web Vitals, similar to those used in our study.

The research highlighted the ad-hoc measurement approach, evaluating immediate performance, in contrast to continuous monitoring, which involves more complex considerations due to increased server loads and potential disruptions over time. The study acknowledged limitations, concentrating on user optimization rather than broader application development strategies. It discussed the minimal impact of actions like clearing history and memory on geoportal speed and the inherent constraints of ad-hoc measurements. Additionally, the study delved into comparing Web Vitals scores between desktop and mobile devices, incorporating radio link and broadband lab data from WebPageTest \(^1\) to provide a comprehensive analysis.

The findings revealed that most performance index scores fell below 50, indicating suboptimal performance, with fully loaded times surpassing 10 seconds for all geoportals and even reaching 20 seconds in certain instances. The paper concluded with a call for further research into optimization techniques aligning with Core Web Vitals standards and suggested conducting user studies to investigate the effects of cache clearing, script blocking, and other user-side optimizations on geoportal performance.

\(^1\)https://pagespeed.web.dev/
Such studies would compare geoportal performance before and after implementing user-guided recommendations to enhance browsing speed.

A study conducted by Basalla et al. [36] examines the impact of latency on the performance of e-commerce websites, focusing on factors like user navigation speed and device usage that shape the browsing experience. The research underscores the consequences of latency for fast navigators and mobile device users and its implications for building lasting customer relationships.

The research centers on the following inquiries:

- Does latency disproportionately affect users with faster navigation speeds and those using mobile devices?
- What is the long-term impact of latency on customers’ likelihood of revisiting a site?
- How does latency affect users who are already familiar with a website?

The study explores various hypotheses, including:

- Fast navigators and mobile users are more significantly affected by latency.
- Latency plays a role in determining whether customers return to the site in the future.
- Users who are well acquainted with a website are less affected by latency.
- Analyzing customers based on their navigation speed could enhance real-time behavioral insights.
- Improving web page responsiveness could boost revenue through better customer satisfaction.
The paper also proposes areas for a study, like exploring how latency impacts user actions, such as searching and buying habits, and investigating how latency combines with other factors affecting website performance. My research goal is to expand on these findings by investigating how network conditions, latencies, and upload speeds impact web performance to gain an understanding of these dynamics.
Chapter 4

RENDERING APPROACHES TO PERFORMANCE METRICS

This chapter will provide an overview of the rendering patterns utilized in the case study, as well as the tools and frameworks involved in performance testing. It is crucial to comprehend each rendering method individually, particularly before discussing the benefits of client and server-side rendering.

4.1 Web Rendering

Since the introduction of the World Wide Web, the earliest iterations of websites predominantly featured static HTML files hosted on a web server. These files contained both the content and design elements of the website. However, introducing server-side programming languages like PHP\(^1\) and ASP\(^2\) revolutionized web development by enabling dynamic websites that could render HTML content on the fly. This shift meant that navigating between different sections of a website required full page reloads, a process inherently reliant on server-side rendering. During this time, JavaScript had a limited role and was mainly used to enable or disable HTML elements [12].

In 2006, the web development scene changed significantly with the introduction of Ajax. This innovation brought about Single-Page Applications (SPAs), which allow for dynamic requests without full page reloads, thereby improving user interaction. Subsequently, JavaScript became more crucial for implementing features and handling data presentation on the client side. This shift led to the creation of client-side

\(^1\)https://www.php.net/

\(^2\)https://dotnet.microsoft.com/en-us/apps/aspnet
frameworks, like jQuery\textsuperscript{3}, Ember\textsuperscript{4}, Angular\textsuperscript{5}, and notably React\textsuperscript{6}, which is widely popular today.

While these frameworks offer numerous benefits, they also pose a set of limitations. This has led developers to delve into rendering strategies that blend the features of both client-side and server-side rendering. These emerging trends aim to improve performance metrics by combining aspects of Client-Side Rendering (CSR) and Server-Side Rendering (SSR). Investigating these mixed methods demonstrates a continuous effort to enhance effectiveness and user satisfaction in web development.

\textsuperscript{3}https://jquery.com/
\textsuperscript{4}https://emberjs.com/
\textsuperscript{5}https://angular.io/
\textsuperscript{6}https://react.dev/
4.2 Hydration

To understand how different rendering methods work, grasping the concept of hydration is imperative. React employs numerous techniques to facilitate rendering, one of which involves using a root component to represent the application:

ReactDOM.render(<App />, document.getElementById('root'));

The ReactDOM.render function locates the specific Document Object Model (DOM) node to anchor the application. In Server-Side rendering (SSR), this root node usually contains rendered content, unlike in Client-Side Rendering (CSR), where ReactDOM.render fills an initially empty node with the client-side application. In cases where SSR is used with React, the framework does not just replace the existing content but attaches the application's content to it, initiating the hydration process. During hydration, React checks the DOM nodes and matches them with their respective JavaScript components.

React, alongside frameworks like Vue.js, implements a concept called virtual DOM (VDOM), which stores the copy of a DOM in memory as a JavaScript object. Using a VDOM helps frameworks identify if a user interaction necessitates a repaint, thereby preventing excessive repaints. It ensures that only modified parts of the site are repainted and consolidates repaints so that a single user interaction does not trigger multiple repaints across the entire page. By not attempting to repaint the whole page for each change, VDOM effectively minimizes unnecessary DOM manipulations, enhancing performance and efficiency.
4.3 Rendering Approaches

4.3.1 Client-Side Rendering (CSR)

Client-side rendering (CSR) is a method where the server sends HTML, CSS, and JavaScript files to the client rather than delivering a fully rendered page. With CSR, the server provides a basic HTML structure with links to necessary JavaScript files. The tasks of applying logic, fetching data, creating templates, and managing routing – all essential for displaying content on the webpage – are handled by JavaScript code that executes in the client’s browser [37]. This approach allows for dynamic content assembly and presentation directly on the user’s device. Since the initial data sent from the server is minuscule, CSR has an advantage when it comes to some of the Core Web Vitals, such as FCP and TTFB [10]. On the contrary, there is an increased score in TTI (negative) due to all the JavaScript that has to be downloaded in order for content to be rendered and become interactive for the user.

Figure 4.2: Client-Side Rendering [10]
CSR has another disadvantage regarding LCP scores since the client side has to download all the JavaScript files, render the UI, fetch all the data from the client side, and render the largest element on the screen. Moreover, search engine optimization (SEO) for CSR still needs improvement because large payloads and a waterfall of network requests may delay rendering meaningful content for crawlers to index [37].

The popularity of CSR has grown alongside the global trend of using frameworks such as React\(^7\), Angular\(^8\), and Vue.js\(^9\). These frameworks handle all the data fetching and routing and provide developers with well-documented libraries.

### 4.3.2 Server-Side Rendering (SSR)

Server-side rendering (SSR) involves generating and showing web content on the server before transmitting it to the user’s web browser. The server handles the requested page in this process and sends a complete HTML document to the client. SSR follows a more conventional approach by pre-rendering content on the server side, delivering a fully prepared webpage to be displayed on the client’s browser.

The server assumes various tasks like running JavaScript, fetching data from databases, interacting with APIs, and customizing content. This is achievable because servers have faster internet connections and robust processors and can promptly generate HTML for immediate delivery to visitors accessing the page [10].

\(^7\)https://react.dev/
\(^8\)https://angular.io/
\(^9\)https://vuejs.org/
In contrast to CSR, SSR provides a notable advantage in search engine optimization (SEO) because it serves pre-rendered static content from the server. This reduces the reliance on browsers and search engines to process JavaScript files, resulting in faster metrics like FP and FCP. Sending less JavaScript to the client also speeds up TTI, as browsers spend less time processing JavaScript. This boost in performance is mainly beneficial for mobile devices due to reduced JavaScript workload.

However, SSR can cause a delay in TTFB, particularly when the server needs to compile large amounts of data or when multiple users are causing excess load on the server. This delay creates a gap between the user’s request and the server’s response, potentially negatively impacting the user experience (UX) by showing a blank screen while waiting for the server. Once the browser gets hold of the HTML content, it becomes visible right away, usually aligning FCP scores closely with TTFB.

**Figure 4.3: Server-Side Rendering [10]**
The popularity of SSR has grown alongside the global trend of using frameworks such as Next.js\textsuperscript{10}, Nuxt.js\textsuperscript{11}, and Gatsby\textsuperscript{12}.

4.3.3 Alternative Rendering Approaches

Up to this point, two primary rendering methods have been the focus of our discussion. However, the rise of frameworks like React and Next.js has sparked interest in exploring new rendering techniques. In particular, Next.js, along with other frameworks such as Gatsby\textsuperscript{13} and VuePress\textsuperscript{14}, offers the capability to pre-render pages on the server per request, also known as Static Generation (SSG). This approach involves meticulously preparing HTML for each page in advance, moving away from the traditional reliance on client-side JavaScript. The use of SSG enhances performance and augments Search Engine Optimization (SEO) capabilities by reducing the need for JavaScript \textsuperscript{12}.

\textsuperscript{10}https://nextjs.org/
\textsuperscript{11}https://nuxtjs.org/
\textsuperscript{12}https://www.gatsbyjs.com/
\textsuperscript{13}https://www.gatsbyjs.com/
\textsuperscript{14}https://vuepress.vuejs.org/
Next.js provides flexibility by allowing users to choose between different pre-rendering forms for each generated page. This means that a hybrid app can be created, utilizing either static generation or server-side rendering (SSR) based on the page’s complexity. SSG is especially beneficial for content that does not change often, like blog posts, e-commerce listings, and marketing materials. Minimizing dependency on JavaScript for dynamic content rendering accelerates the display process.
As shown in 4.5, server processing time is significantly reduced with pre-generated HTML responses, resulting in faster TTFB. The minimal use of JavaScript for static pages means they become interactive almost immediately after the client receives the response, contributing to positive performance metrics like FCP and TTI. While this approach offers advantages, it is essential to acknowledge the challenges of numerous HTML files to cover all user paths, such as different product views in an e-commerce setting. Furthermore, the static nature of SSG makes it challenging for websites that require customized content delivery, such as tailored recommendations on e-commerce sites. Additionally, since SSG relies on pre-generated HTML files, any changes made to the content would necessitate redeploying the entire site as it is unsuitable for dynamic content requiring larger JavaScript file sizes. Therefore, it is crucial to consider the rendering approach by balancing performance benefits with potential drawbacks associated with static content generation.

Figure 4.5: CSR with Prerendering [12]
5.1 Experiment Setup

This chapter performs a case study on various quality attributes in order to achieve a quantifiable comparison between client-side rendering (CSR) and server-side rendering (SSR). To better understand the experiment setup, it is paramount to comprehend the different metrics used in this study to evaluate the performance of CSR and SSR and how we obtain them.

Figure 5.1: Google Lighthouse Example
Google Chrome DevTools and Lighthouse, the performance tools used in this study, are open-source tools provided by Google that help developers assess and improve the quality and performance of web pages. These tools audit web pages and provide insights and recommendations for areas such as performance, accessibility, best practices, and SEO. Figure 5.1 presents an example illustrating the usage of Google Lighthouse and the metrics it provides. In our study, we extrapolate the First Contentful Paint (FCP), Largest Contentful Paint (LCP), and Total Blocking Time (TBT) for analysis and evaluation. One more metric is used in our analysis: finish time, gathered from the Chrome Dev Tools after clicking on "Inspect Element" by right-clicking on the web page you wish to analyze.

Figure 5.2 displays various metrics, including FCP, LCP, finish time (Total), DOMContentLoaded (DCL) time, load time, and other Core Web Vital values. Among these metrics, our primary focus is the finish time, which signifies the completion of the browser’s initial preparation work to display the web page. Since we are solely...
dealing with static components, the finish time can also be considered the response
time.

5.1.1 Browser & Device Specifications

Experiments were conducted on an Apple Macbook Pro 16-inch 2021 model with 32 GB of memory and an M1 Max chip running macOS Sonoma 14.4.1. Although we weren’t able to test with actual mobile devices, we focused on simulating different network conditions, a decision made to account for a factor that primarily affects mobile users. To ensure the validity and reliability of the experiment results, we used three different web browsers: Google Chrome (Version 123.0.6312.87—arm64), Brave (Version 1.64.113 Chromium: 123.0.6312.86—arm64), and Microsoft Edge (Version 123.0.2420.65—arm64).

5.2 Overview of Next.js

Next.js\(^1\), developed by Vercel, enhances React’s capabilities by incorporating server-side rendering (SSR), a key focus of our rendering methods comparison. In addition, it supports a variety of rendering methods such as static site generation (SSG) and incremental static regeneration (ISR); Next.js also streamlines modern web application development with features like client-side routing and code splitting.

In this experiment, we used Next.js (Version 13.4.3) in combination with React (Version 18.2.0) to provide a meaningful comparison and minimize the complexity of employing disparate frameworks for server-side and client-side applications. Traditionally, this would necessitate different setups—for SSR, one might use PHP/Laravel,

\(^1\)https://nextjs.org/
and for CSR, React, or Vue.js. Adopting Next.js mitigates the need for such variances, thus eliminating performance discrepancies attributable to specific languages or framework implementations.

### 5.3 Network Generations

This study incorporates a range of network throttling profiles, detailed in Table 5.1, to accurately reflect user experiences on various network generations. These profiles, including 2G Good, 3G Slow, 3G Good, and 4G/LTE Regular, provide a comprehensive overview of each network’s download speed, upload speed, and latency capabilities.

<table>
<thead>
<tr>
<th>Profile</th>
<th>Download (kb/s)</th>
<th>Upload (kb/s)</th>
<th>Latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2G Good</td>
<td>450</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>3G Slow</td>
<td>780</td>
<td>330</td>
<td>200</td>
</tr>
<tr>
<td>3G Good</td>
<td>1500</td>
<td>750</td>
<td>40</td>
</tr>
<tr>
<td>4G/LTE</td>
<td>4000</td>
<td>300</td>
<td>20</td>
</tr>
</tbody>
</table>

We utilized the throttling options available in Chrome’s DevTools to simulate these network conditions, as depicted in Figure 5.3. While Chrome offers default profiles like Slow 3G and Fast 3G, we also incorporated custom profiles based on specifications from [38]. The process involves:

- Navigating to the Network tab in Chrome DevTools
- Accessing the throttling dropdown menu
- Selecting 'Add...' under the Custom section to input the desired network parameters
It is important to note that Chrome DevTools employs request-level throttling, which applies a delay only after receiving the server’s response [39]. This method does not account for individual network events such as DNS lookups or the process of establishing TCP/SSL connections, which can significantly impact the real-world performance of web applications.
5.4 Lighthouse & DevTools

Lighthouse simulates network conditions by initially loading web pages without throttling and then retrospectively estimating performance metrics for slower connections [39]. This contrasts with DevTools’s applied throttling approach, where pages inherently load slower to mimic real-world conditions. Although Lighthouse offers valuable insights, its evaluations are primarily based on unthrottled network connections. After the initial data collection, a simulation with higher latency is applied, and the performance metrics are adjusted to reflect these conditions [40]. This study employs both DevTools and the Web Vitals Chrome extension to provide a more realistic assessment of first-time page loads.

Figure 5.4: Chrome internal settings for DNS and socket pools

To accurately simulate a user’s initial website visit, we disable the cache in the network tab, as seen in Figure 5.3; by doing so, we can obtain unbiased performance metrics and ensure that the website’s performance is not reliant on cached content, thus providing us with a realistic performance assessment [41].

It is also essential to clear Chrome’s DNS and connection caches. This process involves navigating to chrome://net-internals/#dns and selecting ”Clear host cache,” as illustrated in Figure 5.4. Additionally, refreshing the socket pools via chrome://net-
internals/#sockets and clicking "Flush socket pools" ensures that each page reload accurately represents the DNS lookup and TCP connection establishment times, closely mirroring a real user's experience [42].

To enhance the accuracy of our performance assessment, we integrate an additional no-caching strategy using Charles Proxy [43], a web monitoring tool that examines HTTP and HTTPS traffic. This application’s SSL Proxying feature allows us to closely examine SSL requests and responses. We employ the No Caching tool within Charles to ensure that no resources are stored in the browser cache, thereby guaranteeing that each request is directed to the server, presenting the most current version of the website. The No Caching feature works by adjusting the HTTP headers related to response caching. It eliminates headers like If Modified Since and If None Match, from requests and adds Pragma: no cache and Cache Control: no cache. For outgoing responses, it eliminates Expires, Last-Modified, and ETag headers, and includes Expires: 0 and Cache-Control: no-cache [44]. This detailed manipulation of HTTP headers guarantees an initial user experience by preventing cached content from affecting performance measurements.

5.5 Experiment Methodology

The GitHub repository at [45] contains the source code for the Next.js web application utilized in this study. The experiment methodology is divided into two main sections: the first focuses on client-side rendering (CSR) specifically, while the second section evaluates server-side rendering (SSR). Using Next.js dynamic function to conditionally toggle between SSR and CSR by setting the ssr option based on a flag (ENABLE_SSR) allows us to test both rendering methods. Additionally, prefetch is
set to false to ensure Next.js does not prefetch these components since we are trying to simulate a more realistic loading scenario for first-time visitors.

```javascript
const ENABLE_SSR = true;

// Dynamic imports with conditional SSR/CSR
const Component = dynamic(() => import('.component'), {
  ssr: ENABLE_SSR,
  prefetch: false
});

const Image = dynamic(() => import('.image'), {
  ssr: ENABLE_SSR,
  prefetch: false
});

Next, we decided to run all of the web performance tests in Incognito Mode for each browser to eliminate the possible use of browser extensions and ensure that we start from a clean slate: without any browsing history, cookies, or profiles.
Figure 5.5: Next.js App Example Page

Figure 5.5 shows the main page of the Next.js app showcasing solely CSR components, encompassing 20 text segments and 20 images. Aside from the primary content blocks, we incorporated a toggle button designed to fetch new images dynamically. This specific element was strategically placed to serve as an effective tool for assessing Interaction to Next Paint (INP) scores.

To launch the Next.js application, we execute the following command: `npm run dev` in Incognito mode for each browser with the custom network throttling profiles that are configured in Chrome DevTools. For both CSR and SSR, we perform three tests for each profile and browser combination to ensure the reliability and consistency of the results.
Furthermore, to validate the effectiveness of client-side rendering (CSR) in our Next.js application, JavaScript is strategically disabled using Chrome DevTools, as illustrated in Figure 5.6. When CSR mode is activated, it fails to display JavaScript-dependent components like images and text blocks, only revealing the page title and the button alone. This approach helps differentiate between the rendering methods, clearly demonstrating how Next.js handles content delivery under each rendering paradigm.

5.5.1 Data Collection

Throughout the trials, we gathered a range of performance indicators such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), Cumulative Layout Shift
(CLS), Response (Finish) time, and Load time. These measurements offer perspectives into the loading speed, efficiency, smoothness, and overall user satisfaction with web pages, especially when comparing the two different rendering techniques: client-side and server-side. By analyzing these metrics, we can compare the performance of CSR and SSR and evaluate their suitability for different use cases.
Chapter 6

RESULTS AND DISCUSSION

In this section, we present the results of our comprehensive experiment evaluating the performance of client-side rendering (CSR) and server-side rendering (SSR). This investigation extended to three widely used web browsers—Google Chrome, Brave, and Microsoft Edge—under four distinct network conditions: 2G Good, 3G Slow, 3G Good, and 4G/LTE Regular.

6.1 Finish Time and Load Time

Our analysis observed notable differences in the Load Times, correlating with the DomContentLoaded events, and Finish Times, often regarded as the overall response time when all content has fully loaded on the browser. These differences mainly arise from how content is delivered and rendered in CSR and SSR approaches. Figures 6.1, 6.2 and 6.3 illustrate the respective Finish and Load times recorded for each browser under the study’s specified network conditions, providing insights into the efficiency of each rendering method.
Figure 6.1: Finish and Load Times for Google Chrome
Figure 6.2: Finish and Load Times for Brave Browser
6.1.1 Client-Side Rendering

In CSR, the initial HTML document is often smaller and quicker to load, as it generally contains less embedded content. Due to the minimal content, the browser rapidly parses the HTML and reaches the DOMContentLoaded event relatively swiftly. However, the bulk of the content in CSR is delivered and rendered through JavaScript.
after the initial load. Therefore, despite the DOMContentLoaded event occurring quickly, the full content might not be immediately visible to the user. The content only appears after JavaScript has completed fetching, processing, and rendering it.

6.1.2 Server-Side Rendering

SSR involves the server sending a fully-rendered HTML page with all its content from the server. This approach requires the server to do more work upfront to prepare and send the complete page. As a result, the initial server response is typically slower, as the server must finalize the HTML content before transmission. Once the browser receives the HTML, the content is immediately visible to the user, given that the HTML already encompasses the rendered content.

6.1.3 Summary

In terms of Load Times, CSR generally performs better, with an average Load Time of 11.608 seconds compared to SSR’s score of 12.233 seconds across all tested browsers and network conditions. This efficiency in CSR is partly attributed to quicker DOMContentLoaded times, a direct result of the reduced initial data payload. SSR, although slower to start due to more substantial initial content, consistently surpasses CSR in Finish Time. With an average Finish Time of 12.451 seconds, SSR showcases its effectiveness in delivering a complete and ready-to-interact page faster than CSR, which records an average Finish Time of 13.743 seconds.
The transition from 2G to 4G/LTE showcases a remarkable improvement in Finish Time, with CSR reducing from 26.96 seconds to 3.89 seconds and SSR reducing from 24.46 seconds to 2.87 seconds. Similarly, Load Time for CSR decreases from 23.04 seconds to 2.73 seconds and for SSR from 24.06 seconds to 2.82 seconds. While SSR provides a clear advantage in low-bandwidth conditions, offering faster content visibility, CSR’s dynamic and interactive nature becomes more viable and effective as network conditions improve.
6.2 Largest Contentful Paint (LCP) & First Contentful Paint (FCP)

The variations in the LCP and FCP measurements when using Server-Side Rendering (SSR) versus Client-Side Rendering (CSR) can be explained by the inherent distinctions in how these rendering approaches handle content distribution and rendering.
Figure 6.5: LCP for Google Chrome, Microsoft Edge, and Brave Browser
6.2.1 LCP Analysis

6.2.1.1 Network Generation Improvements

- From 2G to 4G/LTE, significant reductions in LCP times are observed across all browsers for both CSR and SSR. For example, Chrome decreases from 26.963s to 3.891s for CSR and from 24.261s to 2.854s for SSR, indicating improvements of approximately 86% and 88%, respectively.

- The most considerable single-generation improvement was observed moving from 3G Slow to 3G Good, where, for example, Edge’s LCP for CSR improved by 53.93% from 19.293s to 8.887s.

6.2.1.2 CSR vs. SSR

- SSR consistently exhibits lower LCP times across all network conditions and browsers, signifying faster rendering of the largest content elements. For instance, in the 3G Slow category, Edge shows a 41% decrease in LCP for SSR compared to CSR (14.384s vs. 19.293s).

- The average improvement of SSR over CSR ranges from 8% to 10% across different networks and browsers, highlighting SSR’s efficiency in delivering the main content faster.

6.2.1.3 Summary

SSR’s advantage in LCP comes from its ability to send fully rendered content directly from the server. This approach allows the browser to quickly paint large content elements without waiting for client-side JavaScript execution. This proves especially
beneficial for users on slower devices or connections, where JavaScript execution can significantly delay content rendering.
Figure 6.6: FCP for Google Chrome, Microsoft Edge, and Brave Browser
6.2.2 FCP Analysis

6.2.2.1 Network Generation Improvements

- Similar to LCP, FCP scores improve markedly with network upgrades. For instance, Chrome’s FCP under CSR improved by 88.12% from 23.249s to 2.771s from 2G Good to 4G/LTE. SSR in Chrome also shows an 88% improvement, from 24.283s to 2.872s.

- The transition from 3G Slow to 3G Good showed significant FCP improvements, with Brave’s FCP for CSR improving by 48.97% from 13.826s to 7.063s.

6.2.2.2 CSR vs. SSR

- In contrast to LCP, CSR tends to have slightly better FCP scores than SSR in most cases, indicating quicker visibility of the first content piece. For example, Chrome’s FCP under CSR was consistently lower than SSR across all network conditions.

- In the 4G/LTE category, Edge’s CSR FCP is 2.811s compared to SSR’s 3.085s, demonstrating a faster initial paint by approximately 9%.

6.2.3 Summary

CSR leverages asynchronous loading of content through JavaScript. While the main content might take longer to become fully interactive (affecting metrics like LCP), the browser can quickly render initial UI elements or placeholders, contributing to a quicker FCP. This approach can make the application feel responsive, even if the larger content elements and full interactivity are still pending.
6.2.4 LCP vs. FCP

- Across browsers and network conditions, SSR’s overall average LCP is lower than CSR’s, indicating SSR’s efficiency in delivering the largest content element faster.

- CSR showcases a slight edge in FCP times, emphasizing its capability to present the first piece of content to users quickly. The average FCP times for CSR are marginally better than SSR’s, reflecting CSR’s potential to enhance perceived performance.

- The transition from 3G Slow to 3G Good shows a notable performance jump in both LCP and FCP times, emphasizing the impact of even slight improvements in network conditions on content rendering speeds.
6.3 Cumulative Layout Shift (CLS)

Figure 6.7: CLS for Google Chrome, Microsoft Edge, and Brave Browser
CLS measures the total of all individual layout shift scores for every unexpected layout shift that occurs during the page’s lifespan. A lower CLS means the page is more stable, with higher values indicating more frequent or significant shifts that can affect user experience.

6.3.1 General Analysis

6.3.1.1 SSR and CLS

A standout observation is SSR’s CLS score of 0 across all networks and browsers, indicating no visual instability or layout shifts, which is ideal for user experience.

In Server-Side Rendering (SSR), the web page’s content, including text, images, and media, is pre-rendered on the server, resulting in a fully formed HTML document sent to the client. This pre-rendering establishes the layout at the initial load, minimizing content shifts as the page completes loading. Although SSR pages may use some client-side JavaScript for enhanced interactivity or dynamic content loading, the fundamental structure and layout are less reliant on it. This reduced dependency on JavaScript minimizes layout shifts often caused by dynamic DOM manipulations or asynchronous content loading.

6.3.1.2 CSR and CLS

CSR’s CLS scores show variability across different network conditions and browsers. For instance, Chrome’s CLS under CSR slightly increases from 0.258 in 3G Slow and 3G Good to 0.285 in 4G/LTE, indicating a slight increase in layout shifts even with improved network conditions.
CSR relies heavily on JavaScript to fetch, render, and often re-render content after the initial page load. This dynamic loading and rendering can lead to content shifting as different elements are added to the DOM, resized, or moved, contributing to higher CLS values. Resources such as images, advertisements, and additional code might be loaded asynchronously, potentially altering the layout if their dimensions were not initially reserved, contributing to layout instability. In CSR, the client’s browser is responsible for constructing the page structure based on JavaScript execution, making the layout susceptible to shifts during content staging or as a response to user actions.

6.3.2 Summary and Insights

- The clear difference between CSR and SSR regarding CLS highlights SSR’s advantage in delivering a visually stable experience. The absence of layout shifts in SSR enhances user satisfaction by preventing unexpected content movement.

- Contrary to expectations, network speed has minimal impact on CSR’s CLS scores. This indicates that layout shifts in CSR are more closely tied to factors like JavaScript execution and DOM manipulation rather than network conditions.

- SSR’s zero CLS scores across all conditions and browsers highlight its robustness in maintaining visual stability, making it a preferable choice for pages where layout shifts could significantly impact user experience.

- While CSR offers dynamic and interactive experiences, it faces challenges in maintaining visual stability, as indicated by the non-zero CLS scores. Developers leveraging CSR must implement strategies to minimize layout shifts, such as specifying dimensions for images and media or employing placeholders.
In conclusion, while SSR provides a clear advantage in terms of visual stability, careful optimization can mitigate CSR’s challenges, ensuring a balance between dynamic interactivity and user-centric stability.

6.4 Browser Analysis

6.4.1 Finish Time

![Figure 6.8: Finish Time Comparison Across Browsers](image)

Figure 6.8: Finish Time Comparison Across Browsers
The charts above showcase the Finish Time for CSR and SSR across Google Chrome, Brave, and Microsoft Edge browsers under various network conditions. All three browsers show a trend of decreasing finish and load times from 2G to 4G/LTE.

**Improvement from 2G to 4G/LTE:**

- **Chrome** sees an 85.57% improvement in CSR and an 88.24% improvement in SSR from 2G Good to 4G/LTE.
- **Brave** exhibits an 83.66% improvement for CSR and an 87.85% improvement for SSR.
- **Edge** demonstrates an 86.20% improvement in CSR and an 87.47% improvement in SSR.

**Comparison Between Browsers:**

From 3G Slow to 3G Good, Chrome’s Finish Time improves by 48.13%, slightly more than Brave’s 45.23% and Edge’s 44.49%. This suggests that Chrome may have optimizations that slightly edge out in mid-range network conditions.
6.4.2 Load Time

![CSR Load Time Comparison Across Browsers](chart1)

![SSR Load Time Comparison Across Browsers](chart2)

**Figure 6.9: Finish Time Comparison Across Browsers**

As illustrated in Chart 6.9, all browsers show improved performance metrics from 2G to 4G/LTE. The differences in load times between the browsers are relatively minor, indicating that load time performance is fairly consistent across these browsers for CSR.
Comparison Between Browsers:

- **Edge**’s Load Times improve consistently with network upgrades, with a notable 48.83% improvement for SSR from 3G Slow to 3G Good (from 14.22s to 7.51s). Moreover, from 2G to 4G/LTE, Edge demonstrates an 88.11% improvement in CSR and an 87.72% improvement in SSR.

- For Load Times in **Brave**, the transition from 3G Slow to 3G Good shows a 48.53% improvement for CSR, with times reducing from 13.6 seconds to 7.01 seconds.

- **Chrome**’s Load Time for CSR improves by 88.16% and SSR by 88.26% from 2G to 4G/LTE.
6.5 Largest Contentful Paint (LCP)

The charts above illustrate the Largest Contentful Paint (LCP) times for CSR and SSR across Google Chrome, Brave, and Microsoft Edge browsers under various network conditions.

**Comparison Between Browsers:**

![CSR LCP Comparison Across Browsers](chart1)

![SSR LCP Comparison Across Browsers](chart2)

Figure 6.10: LCP Comparison Across Browsers
• **Edge’s LCP Anomaly in 3G Slow**: Edge exhibits a notably higher CSR LCP score of 19.293s in the 3G Slow condition, which is approximately 10.9% and 22.87% higher than Chrome’s 17.271s and Brave’s 15.72s, respectively. This deviation suggests a potential inefficiency in Edge’s content rendering or resource prioritization in slower network conditions.

• **Improvement from 2G Good to 4G/LTE**: Chrome demonstrates the most significant improvement in CSR LCP, reducing from 26.963s to 3.891s, a reduction of 85.57%. In contrast, Brave and Edge show improvements of 83.67% and 86.32%, respectively.
6.6 First Contentful Paint (FCP)

The charts above illustrate the First Contentful Paint (FCP) times for CSR and SSR across Google Chrome, Brave, and Microsoft Edge browsers under various network conditions.

Figure 6.11: FCP Comparison Across Browsers
Comparison Between Browsers:

- **Edge’s Initial SSR FCP**: Edge starts with the highest SSR FCP score of 24.809s, approximately 2.17% higher than Chrome’s 24.283s and 1.16% higher than Brave’s 24.525s in the 2G Good condition.

- **Improvement from 2G Good to 4G/LTE**:
  - Chrome shows an 88.11% improvement in CSR FCP, reducing from 23.249s to 2.771s.
  - Brave follows with an 88.12% improvement in CSR FCP, from 23.393s to 2.785s.
  - Edge demonstrates an 87.93% improvement, from 23.267s to 2.811s.
  - In SSR, Chrome leads with an 88.21% improvement, followed by Brave at 87.78% and Edge at 87.57%.

- **Narrowing Gap in 4G/LTE**: The gap between CSR and SSR FCP scores significantly narrows in 4G/LTE, with the smallest difference observed in Chrome (3.65%) and the largest in Edge (9.72%). This convergence indicates that under optimal network conditions, the distinction between CSR and SSR in initial content visibility becomes minimal.
6.7 Cumulative Layout Shift (CLS)

Figure 6.12: CLS Comparison Across Browsers

Given the consistent zero values for SSR across all browsers, this analysis focuses exclusively on CSR.

Browser Specific Results:

- **Chrome’s CLS in CSR** fluctuates minimally, starting and ending at 0.285 with a slight decrease to 0.258 in mid-range network conditions (3G Slow and 3G Good). This indicates a stable visual experience with minor layout shifts.

- **Brave’s CLS in CSR** shows remarkable consistency at 0.284 across the first three network conditions, with a slight improvement to 0.257 in 4G/LTE.

- **Edge’s CLS in CSR** demonstrates a unique trend, starting at a lower value of 0.233 in 2G Good, maintaining this stability into 3G Slow, then experiencing an increase to 0.26 in 3G Good, and peaking at 0.287 in 4G/LTE. This upward
trend in CLS scores with improved network conditions may indicate challenges in maintaining layout stability as content loads more rapidly.

• **Comparative Insights:**

  – **Edge** starts with the lowest CLS scores, indicating superior layout stability in slower network conditions. However, its increasing trend towards 4G/LTE raises concerns about its ability to maintain this stability as network performance improves.

  – **Chrome** and **Brave** exhibit greater consistency in their CLS scores, with Brave slightly improving 4G/LTE, potentially indicating more effective optimization for high-speed networks.
Chapter 7

CONCLUSION AND FUTURE WORK

7.1 Future Work

1. **Impact of 5G Network:** With the rollout of 5G networks, it would be interesting to delve into how the new generation of network technology affects client-side and server-side rendering. This investigation will reveal the improvements that 5G brings to performance and user experience, especially in its interaction with different rendering approaches.

2. **Comparative Analysis of Mobile Browsers:** Expand the analysis to include a more comprehensive array of mobile browsers, such as Safari on iOS devices and Firefox for Android. By considering the performance and compatibility of client-side and server-side rendering on these browsers, the study can provide a deeper understanding of the rendering techniques across diverse mobile platforms.

3. **Testing on Non-Chromium browsers:** Non-Chromium web browsers like Safari and Firefox have their rendering engines (WebKit for Safari and Gecko for Firefox), which may differ from Chromium (used by Google Chrome, Brave Browser, and Microsoft Edge). This variance in rendering engines can introduce compatibility challenges and optimization that influence the performance metrics, and studying these different browsers will give a comprehensive study from a wide variety of browsers.
4. **Long-Term Performance Analysis:** Conducting a long-term performance analysis will reveal the sustainability of client-side and server-side rendering over extended periods. This investigation should consider the roles of caching mechanisms, periodic browser updates, and the evolving nature of network technologies. Such an examination is crucial for assessing various rendering methods’ lasting effectiveness and dependability.

7.2 **Conclusion**

This thesis explores the performance and user experience implications of Client-Side Rendering (CSR) and Server-Side Rendering (SSR) across browsers such as Google Chrome, Brave, and Microsoft Edge. The study carefully assesses metrics such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), Cumulative Layout Shift (CLS), and Finish Time across different network speeds from 2G to 4G/LTE.

Our findings reveal a distinct advantage of SSR in Finish Time, indicating that SSR generally delivers a fully rendered page faster than CSR, which is particularly beneficial under constrained network conditions. In contrast, CSR stands out in Load Time due to its dynamic rendering approach, which often allows for quicker initial content interaction despite occasional delays in full-page rendering. While CSR occasionally faces challenges in displaying main content promptly, reflected in LCP metrics, it consistently delivers superior FCP times due to the rapid loading of minimal initial HTML documents. SSR, which delivers a fully rendered HTML page from the server to the client, excels in LCP performance, offering quicker main content visibility and more stable page layouts, as evidenced by lower CLS scores.

In conclusion, our study highlights the inherent trade-offs between CSR’s interactivity and SSR’s content availability and stability. The findings highlight the importance of
considering browser-specific behaviors and network conditions in web development to optimize user experience and performance. This thesis lays the groundwork for further exploration into how emerging network technologies may influence web rendering strategies.
BIBLIOGRAPHY


https://www.patterns.dev/react/static-rendering/.

[13] Fiona Nah. A study on tolerable waiting time: How long are web users willing
to wait? In Behaviour & Information Technology - Behaviour & IT,

server-side rendering. Recent Trends in Information Technology and its

Antwerp, Antwerp, Belgium, 2018.


[17] Optimize time to first byte | articles | web.dev. Accessed: 2024-03-06. URL:
https://web.dev/articles/optimize-ttfb.

[18] PerformanceLongTaskTiming - web APIs | MDN. Accessed: 2024-03-06. URL:

[19] Total blocking time (TBT) | articles. Accessed: 2024-03-07. URL:
https://web.dev/articles/tbt.


Acronyms

CSR  Client-Side Rendering.

FCP  First Contentful Paint.

FP   First Paint.

LCP  Largest Contentful Paint.

SSG  Static Site Generation or Static Generation.

SSR  Server-Side Rendering.

TTFB Time to First Byte.

TTI  Time to Interactive.