



UNIVERSITETI I EVROPËS JUGLINDORE
УНИВЕРЗИТЕТ НА ЈУГОИСТОЧНА ЕВРОПА
SOUTH EAST EUROPEAN UNIVERSITY

South East European University
Faculty of Contemporary Sciences & Technologies

Comparison of Web Frontend Technologies and Possible Optimization

Prepared By:
Jeton Kadrija

Supervised By:
Prof. Dr. Artan Luma

A Graduation Project submitted to the Faculty of Contemporary Sciences
and Technologies, SEEU in partial fulfillment of the requirements for the
degree of M.Sc. in Computer Science

Tetova
December, 2024

Abstract

Web frontend technologies play a critical role in determining the performance, user experience, and scalability of modern web applications. This thesis conducts a comprehensive comparison of popular web frontend frameworks, including Vanilla JavaScript, React (Client-Side Rendering and Server-Side Rendering), and Vue (Client-Side Rendering). The study evaluates these technologies under different conditions, including local and cloud environments, focusing on key performance metrics such as First Contentful Paint (FCP), Speed Index (SI), Largest Contentful Paint (LCP), Total Blocking Time (TBT), and Cumulative Layout Shift (CLS).

A robust experimental methodology is employed, leveraging tools such as Google Lighthouse and synthetic monitoring platforms, to measure and analyze the rendering efficiency, responsiveness, and visual stability of each technology. The results highlight the strengths and weaknesses of each framework in handling data-intensive workloads and varying deployment scenarios. For instance, server-side rendering approaches in React demonstrate significant improvements in initial rendering times and layout stability but exhibit higher total blocking times during hydration. Conversely, client-side rendering techniques excel in progressive rendering and interactivity but are heavily influenced by JavaScript execution overhead.

The thesis further explores optimization strategies, including code splitting, caching, and edge rendering, to address the identified performance bottlenecks. Recommendations for selecting appropriate frontend technologies are provided based on application complexity, deployment environment, and user experience requirements. This study aims to guide developers and organizations in making informed decisions about frontend technology adoption and performance optimization, contributing to the advancement of scalable and high-performing web applications.

Апстракт

Технологиите за веб фронтенд играат клучна улога во одредувањето на перформансите, корисничкото искуство и скалабилноста на модерните веб апликации. Оваа теза спроведува сеопфатна компарација на популарни веб фронтенд рамки, вклучувајќи Vanilla JavaScript, React (рендерирање на клиентска и серверска страна) и Vue (рендерирање на клиентска страна). Истражувањето ги оценува овие технологии под различни услови, вклучувајќи локални и облачни средини, фокусирајќи се на клучни метрики за перформанси како што се First Contentful Paint (FCP), Speed Index (SI), Largest Contentful Paint (LCP), Total Blocking Time (TBT) и Cumulative Layout Shift (CLS).

Се применува робустна експериментална методологија, користејќи алатки како Google Lighthouse и синтетички платформи за мониторинг, за мерење и анализа на ефикасноста на рендерирањето, одзивноста и визуелната стабилност на секоја технологија. Резултатите ги истакнуваат предностите и слабостите на секоја рамка при обработка на оптоварувања со интензивни податоци и различни сценарија за примена. На пример, пристапите за рендерирање на серверска страна во React покажуваат значителни подобрувања во времињата за иницијално рендерирање и стабилноста на изгледот, но покажуваат повисоки вкупни блокирачки времиња за време на хидратацијата. Од друга страна, техниките за рендерирање на клиентска страна се извонредни во прогресивното рендерирање и интерактивноста, но се под големо влијание од оптоварувањето на извршувањето на JavaScript.

Тезата понатаму истражува стратегии за оптимизација, вклучувајќи поделба на кодот, кеширање и рендерирање на рабовите, за справување со идентификуваните пречки во перформансите. Се даваат препораки за избор на соодветни фронтенд технологии врз основа на комплексноста на апликацијата, средината за примена и барањата за корисничко искуство. Оваа студија има за цел да им помогне на развивачите и организациите да донесат информирани одлуки за усвојување технологии за фронтенд и оптимизација на перформансите, придонесувајќи за напредокот на скалабилни и високо-ефикасни веб апликации.

Abstract

Teknologjitë e frontend-it të uebit luajnë një rol kritik në përcaktimin e performancës, përvojës së përdoruesit dhe shkallëzueshmërisë së aplikacioneve moderne të uebit. Kjo tezë kryen një krahasim gjithëpërfshirës të frontend-it të uebit, duke përfshirë Vanilla JavaScript, React (Renderimi në anën e Klientit dhe në anën e Serverit) dhe Vue (Renderimi në anën e Klientit). Studimi vlerëson këto teknologji në kushte të ndryshme, duke përfshirë mjediset lokale dhe ato në re, duke u përqendruar në treguesit kryesorë të performancës, siç janë First Contentful Paint (FCP), Speed Index (SI), Largest Contentful Paint (LCP), Total Blocking Time (TBT) dhe Cumulative Layout Shift (CLS).

Përdoret një metodologji eksperimentale e fortë, duke shfrytëzuar mjete si Google Lighthouse dhe platforma të monitorimit sintetik për të matur dhe analizuar efikasitetin e renderimit, përgjegjshmërinë dhe stabilitetin vizual të secilës teknologji. Rezultatet nxjerrin në pah pikat e forta dhe dobësitë e secilës kornizë në trajtimin e ngarkesave të dhënash intensive dhe skenarëve të ndryshëm të implementimit. Për shembull, qasjet e renderimit në anën e serverit në React tregojnë përmirësime të ndjeshme në kohët fillestare të renderimit dhe stabilitetin e paraqitjes, por paraqesin kohë më të larta totale bllokuese gjatë procesit të hidrimit. Nga ana tjetër, teknikat e renderimit në anën e klientit shkëlqejnë në renderimin progresiv dhe interaktivitetin, por ndikohen shumë nga mbingarkesa e ekzekutimit të JavaScript-it.

Teza eksploron më tej strategji optimizimi, duke përfshirë ndarjen e kodit, caching dhe renderimin në skaj, për të adresuar pengesat e identifikuara të performancës. Jepen rekomandime për zgjedhjen e teknologjive të përshtatshme të frontend-it bazuar në kompleksitetin e aplikacionit, mjedisin e implementimit dhe kërkesat për përvojën e përdoruesit. Ky studim synon të udhëzojë zhvilluesit dhe organizatat për të marrë vendime të informuara mbi adoptimin e teknologjive të frontend-it dhe optimizimin e performancës, duke kontribuar në përparimin e aplikacioneve të uebit të shkallëzueshme dhe me performancë të lartë.

Contents

Abstract	I
Апстракт	II
Abstrakt	III
Table of Contents	IV
List of Figures	VIII
1 Introduction and Motivation	1
1.1 Background of the Study	1
1.2 Research Problem	1
1.3 Research Aims and Objectives	1
1.4 Research Questions	1
1.5 Hypotheses	2
1.6 Significance of the Study	2
2 Literature Review	3
2.1 Literature Review	3
2.1.1 Introduction	3
2.1.2 JavaScript Performance Optimization	3
2.1.3 Rendering Strategies and Framework Efficiency	4
2.1.4 Framework Comparison and Selection	4
2.1.5 Conclusion	4
2.1.6 Impact on Load Speed	5
2.1.7 Server-Side Rendering (SSR) vs. Client-Side Rendering (CSR) in React	5
2.2 Bundle Sizes in Front-End Technologies	6

2.2.1	Factors Influencing Bundle Size	6
2.2.2	Comparison of Different Technologies	6
2.3	Summary of Literature Review	6
3	Research Methodology	7
3.1	Introduction	7
3.2	Research Design	7
3.3	Data Collection Methods	8
3.3.1	Tools and Technologies Used	8
3.3.2	Data Collection Procedures	8
3.4	Experimental Setup	9
3.5	Data Analysis	9
3.6	Data Analysis Methods	9
3.6.1	Quantitative Analysis	9
4	Assessing the Impact of JavaScript Libraries on Load Speed	11
4.1	Methodology	11
4.1.1	Selection of JavaScript Libraries	11
4.2	Comparing SSR and CSR in React	12
4.3	Methodology	13
4.3.1	Experimental Design	13
4.4	Analyzing Bundle Size Differences Across Front-End Technologies	14
4.5	Bundle size	16
4.5.1	Components of a Bundle	16
4.5.2	Importance of Optimizing Bundle Size	17
4.5.3	Measurement Methodology	17
4.5.4	Results and Observations	17
4.5.5	Impact on Performance Metrics	18
4.5.6	Optimization Strategies	18
4.5.7	Future Considerations	19
5	Comparing and Performance Results and Analysis	22
5.0.1	Performance Results and Analysis for Javascript in local environment	22
5.0.2	Performance Results and Analysis for Javascript in cloud environment	25
5.0.3	Performance Results and Analysis for React in local environment .	28
5.0.4	Performance Results and Analysis for React in cloud environment .	32

5.0.5	Performance Results and Analysis for Vue in Local Environment . .	36
5.0.6	Performance Results and Analysis for Vue in Cloud Environment .	39
5.0.7	Performance Results and Analysis for React SSR in Local Environ- ment	43
5.0.8	Performance Results and Analysis for React SSR in Cloud Environ- ment	47
6	Analyzing Bundle Size Differences Across Front-End Technologies	52
6.0.1	Bundle Statistics from <code>stats.json</code>	53
6.0.2	Compression Efficiency	54
6.0.3	Interpretation	55
6.0.4	Conclusion	55
6.1	Bundle Size Analysis Using Sunburst Visualization for React	55
6.1.1	Structure of the Sunburst Chart	56
6.1.2	Compression Efficiency	58
6.1.3	Analysis and Observations	58
6.1.4	Conclusion	58
6.2	Bundle Size Analysis Using Sunburst Visualization for Vue	58
6.2.1	Root Node: Bundle Overview	59
6.2.2	Primary Assets: Immediate Children	59
6.2.3	Application-Specific Modules	60
6.2.4	Compression Efficiency	61
6.2.5	Conclusions and Observations	61
6.3	Comparison of the React SSR / NextJS Server-Bundle Report with Other JavaScript Build Reports	61
6.3.1	Analyzers output	62
6.4	Analysis of <code>@next/bundle-analyzer</code> Outputs	63
6.4.1	Client-Side Bundle	63
6.4.2	Server-Side Bundle	64
6.4.3	Comparison and Observations	64
6.4.4	Conclusion	64
6.4.5	Server-Bundle Report vs. Vite React	64
6.4.6	Server-Bundle Report vs. Vanilla JavaScript	65
6.4.7	Server-Bundle Report vs. Vite Vue	65
6.4.8	Build Size of the Next.js Application	65
6.5	Analysis of Bundle Report	66

6.5.1	Overall Bundle Size	66
6.5.2	Largest Chunks	66
6.5.3	React-DOM Contribution	66
6.5.4	Routing and Utilities	66
6.5.5	Error and Not-Found Pages	66
6.5.6	Module Redundancy	67
6.5.7	Conclusion	67
6.5.8	Analysis using webpack-bundle-analyzer tool	67
6.5.9	Observations	68
6.5.10	Comparison with @next/bundle-analyzer	69
6.5.11	Recommendations for Optimization	70
	Conclusion	71
6.6	Lessons learned	72
6.7	Future Directions	73
	Bibliography	74

List of Figures

5.1	Average Performance Metrics for Javascript in Local Environment.	23
5.2	Average Performance Metrics for Javascript in Cloud Environment.	26
5.3	Average Performance Metrics for React - Local.	30
5.4	Average Performance Metrics for React - Cloud.	34
5.5	Average Performance Metrics for Vue - Local.	38
5.6	Average Performance Metrics for Vue - Cloud.	41
5.7	Average Performance Metrics for React - SSR Local.	45
5.8	Average Performance Metrics for React - SSR Cloud.	49
6.1	Sunburst for bundle size of a JavaScript app	52
6.2	Sunburst for bundle size of a JavaScript app	56
6.3	Sunburst for bundle size of a JavaScript app	59
6.4	Bundle analyzer output of a React SSR / NextJS app - Client	63
6.5	Bundle analyzer output of a React SSR / NextJS app - Server	63
6.6	Bundle analyzer output of a React SSR / NextJS app using Webpack	68

Chapter 1

Introduction and Motivation

1.1 Background of the Study

Web development frameworks have evolved rapidly, bringing a variety of tools and approaches to enhance user experience, optimize load speeds, and manage data efficiently. With this growth, front-end developers and software engineers face a multitude of options when choosing a framework. Each framework offers unique advantages and trade-offs, impacting application performance, development time, and overall efficiency. This thesis focuses on comparing various front-end technologies to understand their impact on web performance, especially regarding load speed, rendering techniques, and bundle size optimization.

1.2 Research Problem

Despite the numerous frameworks available, identifying the optimal framework and techniques to maximize load speed and minimize bundle size remains a challenge. Developers frequently encounter performance bottlenecks related to inefficient JavaScript libraries, large bundle sizes, and suboptimal rendering strategies. This research seeks to address these issues by conducting a systematic comparison of front-end technologies and rendering techniques to determine the most efficient approach for web application development.

1.3 Research Aims and Objectives

- To assess the impact of various JavaScript libraries on initial load speed.
- To compare Server-Side Rendering (SSR) and Client-Side Rendering (CSR) in large-scale React applications.
- To evaluate bundle size differences across various front-end technologies, focusing on load speed and application performance.

1.4 Research Questions

- How does the use of different JavaScript libraries impact the initial load speed of web applications?

- How does server-side rendering (SSR) in React compare to client-side rendering (CSR) in terms of load speed for large-scale applications?
- How does the bundle size of different front-end technologies compare to that of a React application when both are configured with default settings?]

1.5 Hypotheses

Based on the reviewed literature, the following hypotheses are proposed:

- The use of different JavaScript libraries significantly impacts the initial load speed of web applications, with lighter and optimized libraries resulting in faster load times.
- Server-side rendering (SSR) in React results in faster initial load speed for large-scale applications compared to client-side rendering (CSR), due to pre-rendering of content on the server.
- Certain front-end technologies produce smaller bundle sizes, leading to better performance in terms of load speed and application efficiency.

1.6 Significance of the Study

This study contributes to the field of web development by guiding best practices for choosing frameworks, libraries, and rendering techniques to enhance application performance. Insights derived from this research will assist developers and engineers in optimizing their applications, making informed choices regarding technology selection, and ultimately improving the end-user experience.

Chapter 2

Literature Review

2.1 Literature Review

2.1.1 Introduction

The rapid evolution of web development has introduced a plethora of frameworks and tools aimed at improving performance, scalability, and user experience. However, as web applications grow more complex, challenges such as large JavaScript payloads, code redundancy, and framework selection become increasingly critical. This chapter synthesizes findings from recent studies and tools in the field of frontend development, focusing on JavaScript optimization, rendering strategies, and framework comparisons. The review highlights best practices and identifies gaps in current research to inform subsequent analysis and experimentation.

2.1.2 JavaScript Performance Optimization

JavaScript remains the backbone of modern web development, but its extensive usage often results in bloated payloads and performance bottlenecks. Goel and Steiner [18] discuss a system that identifies and removes superfluous JavaScript code using real user monitoring (RUM) techniques. Their approach, leveraging a custom man-in-the-middle proxy, reduced median page load times by 5% and improved long-tail performance by 10%, showcasing the potential of eliminating unused code from JS resources.

Similarly, Chaqfeh et al. [9] introduced JSAnalyzer, a tool designed to optimize mobile web pages by disabling non-essential JavaScript while maintaining interactivity. Their findings reveal significant improvements in load times and resource consumption, particularly for resource-constrained devices. In addition, the study by Lauinger et al. [16] highlights the prevalence of outdated and vulnerable JavaScript libraries across the web, emphasizing the risks posed by unmanaged dependencies in large-scale applications.

Williams [20] provided a practical guide to reducing JavaScript bundle sizes using tools like Webpack and Statoscope. The study showcased techniques such as tree-shaking and code splitting, which help developers identify unused code and reduce resource consumption during page loads. These optimization practices are pivotal in improving performance without sacrificing user experience.

2.1.3 Rendering Strategies and Framework Efficiency

Rendering strategies play a pivotal role in determining application performance. Levlin [7] conducted a benchmark analysis of popular frontend frameworks, including React, Angular, Vue, and Svelte, focusing on their Document Object Model (DOM) manipulation capabilities. The study concluded that Svelte, which compiles components into optimized JavaScript during build time, outperforms virtual DOM-based frameworks like React and Vue in raw DOM insertion and deletion speed. However, React and Vue excel in managing dynamic updates and state transitions, making them suitable for complex applications.

Further exploration by Mitropoulos et al. [4] analyzed the evolution of JavaScript code across top-ranked websites, identifying trends in framework adoption and the impact of rendering techniques on performance. This study supports findings from Kupoluyi et al. [5], who emphasized the importance of balancing development flexibility with runtime efficiency in large-scale applications.

Cao et al. [3] focused on web front-end optimization techniques, such as reducing HTTP requests, client-side caching, and compressing assets. Their research aligns with Wei's [19] findings on leveraging big data analytics for web performance improvements, particularly in dynamically generated content.

2.1.4 Framework Comparison and Selection

Framework selection is critical for project success, as highlighted by several comparative studies. Rathinam [12] provides a detailed analysis of React, Angular, and Vue, outlining their strengths and weaknesses. React's virtual DOM offers high interactivity, while Angular's TypeScript-based architecture supports robust, large-scale applications. Vue strikes a balance between simplicity and efficiency, making it ideal for medium-scale projects. Similarly, Kaluža et al. [8] evaluated these frameworks for single-page (SPA) and multi-page (MPA) applications, emphasizing the importance of aligning framework selection with project requirements and team expertise.

Challapalli et al. [14] extend this discussion by comparing frameworks' suitability for large-scale applications. Their findings suggest that React and Vue provide flexibility and scalability, while Angular excels in enterprise-grade solutions. Tong et al. [17] further investigated rendering and performance metrics of these frameworks, noting Next.js's efficiency in server-side rendering.

2.1.5 Conclusion

The reviewed literature highlights the critical need to optimize JavaScript usage, adopt efficient rendering strategies, and carefully evaluate frontend frameworks to meet the diverse demands of modern web applications. Key findings emphasize the importance of performance optimization techniques, such as reducing superfluous code, implementing tree-shaking, and leveraging server-side rendering (SSR) for improved scalability and user experience. The comparative analysis of technologies like React, Vue, and Vanilla JavaScript underscores the importance of aligning technology choices with project-specific goals, such as interactivity, responsiveness, and maintainability.

This thesis focuses on this direction due to the growing reliance on web applications in both consumer and enterprise contexts, where performance and scalability are paramount.

By experimenting with React, including its server-side rendering capabilities with Next.js, and comparing it to Vanilla JavaScript, this research investigates the trade-offs between simplicity, flexibility, and resource efficiency. Vanilla JavaScript serves as a baseline for understanding performance in its purest form, while React/Next.js SSR offers insights into modern optimization techniques and rendering approaches. This study aims to provide actionable insights that guide developers and organizations in making informed decisions about framework selection and optimization strategies, ensuring scalable, efficient, and high-performing web applications. [5]

Lightweight vs. Feature-Rich Libraries

JavaScript libraries significantly influence web application performance, often requiring a balance between load speed and functionality. Feature-rich libraries like React and Vue provide extensive functionality and a robust ecosystem, which often comes at the cost of larger bundle sizes and increased complexity. This balance emphasizes the importance of evaluating specific project requirements, as the additional features of robust libraries may not always align with the need for streamlined performance in every use case.

For this thesis, the performance characteristics of libraries will be examined, focusing on modern optimization techniques such as dead code elimination and tree-shaking, which can significantly reduce the payload of libraries like React. By analyzing the baseline performance of Vanilla JavaScript, alongside the optimization potential of React (including its server-side rendering capabilities with Next.js), this study seeks to identify strategies for achieving high performance while retaining essential functionality for scalable web applications. This research aims to provide actionable insights into managing the balance between functionality and efficiency in library selection and optimization.

2.1.6 Impact on Load Speed

Load speed is a critical factor in web application performance, particularly when using libraries that add to the bundle size. Practical techniques for reducing JavaScript load times, such as bundle size optimization, tree-shaking, and code splitting, are essential for minimizing the impact of library selection on initial load speed. These approaches ensure that only necessary code is delivered to the client, improving both performance and user experience.

2.1.7 Server-Side Rendering (SSR) vs. Client-Side Rendering (CSR) in React

Definition and Mechanisms

SSR and CSR are two predominant rendering strategies, each with unique implications for load speed and SEO. Risto Ollila's (2021) comparative analysis of rendering strategies highlights that SSR reduces initial load time, while CSR offers faster navigation post-initial load. These distinctions are crucial for applications that prioritize user experience and performance.

Performance Comparison

SSR generally outperforms CSR in initial load speed due to pre-rendering content on the server, but CSR excels in applications requiring high interactivity. Villamor and Semura (2019) examine these performance trade-offs, providing insights into when SSR or CSR might be more beneficial based on application needs.

2.2 Bundle Sizes in Front-End Technologies

2.2.1 Factors Influencing Bundle Size

Bundle size is a key metric for optimizing load performance, as large bundles slow down page load times. Minimizing bundle size through techniques such as tree-shaking, code splitting, and dependency analysis is essential for improving load efficiency. The choice of framework and library configurations significantly impacts bundle size, requiring careful evaluation to ensure that only necessary features and dependencies are included. This approach enhances performance while maintaining the desired functionality.

2.2.2 Comparison of Different Technologies

Different frameworks produce varying bundle sizes. Research by Paweł D. et al. compares Angular, React, and Vue, demonstrating that lightweight configurations yield faster load times, beneficial for performance-critical applications.

2.3 Summary of Literature Review

The literature underscores the significance of rendering strategies, JavaScript libraries, and bundle size on web performance. By selecting the appropriate combination of SSR or CSR, alongside optimized libraries, developers can improve application load times and interactivity, ultimately enhancing user experience. These insights provide the theoretical basis for the research in subsequent chapters.

Chapter 3

Research Methodology

3.1 Introduction

The methodology for this research is structured to ensure reliability, validity, and replicability. A systematic approach is employed, incorporating experimental and comparative analysis methods to evaluate the performance of various front-end technologies and rendering strategies. By conducting rigorous testing under controlled conditions, this study aims to provide actionable insights for developers and architects choosing optimal rendering techniques.

3.2 Research Design

The research design involves an experimental setup where identical web applications are developed using different JavaScript libraries and front-end frameworks. These applications include implementations in vanilla JavaScript, React, and Vue, with configurations for both Server-Side Rendering (SSR) and Client-Side Rendering (CSR).

To ensure a fair comparison, the following considerations were applied:

- **Uniform Dataset:** A large JSON file (~1.5 MB) is used across all applications to simulate realistic data-intensive scenarios.
- **Standardized Testing Environment:** All applications are deployed to the same cloud hosting platform (Netlify) to eliminate discrepancies in infrastructure performance.
- **Production Builds:** Each application is built using production-ready tools (e.g., `vite build`, `npm build`) to replicate real-world deployment scenarios.

The research design emphasizes repeatability, ensuring that other researchers can replicate the experiments and validate the findings.

3.3 Data Collection Methods

3.3.1 Tools and Technologies Used

To ensure accurate and reliable data collection, this research employs the following tools and technologies:

- **Google Lighthouse:** Measures performance metrics such as First Contentful Paint (FCP), Speed Index (SI), and Cumulative Layout Shift (CLS).
- **Chrome DevTools:** Provides detailed insights into network performance, rendering timelines, and JavaScript execution times.
- **Webpack Bundle Analyzer:** Analyzes bundle sizes and provides a breakdown of components contributing to the final application size.
- **Netlify:** Used as the cloud hosting platform for consistent, real-world deployment simulation.
- **Vite:** Standardizes the development environment, ensuring consistent builds across all applications.

3.3.2 Data Collection Procedures

Data collection involves a step-by-step process to ensure consistent and unbiased results:

1. **Application Development:** Identical web applications are developed using vanilla JavaScript, React, and Vue, with configurations for both CSR and SSR. Each application processes and renders the same JSON file (~1.5 MB) to ensure uniformity in testing conditions.
2. **Build and Deployment:** Each application is built using production configurations (e.g., `npm build`, `vite build`). The production-ready assets are deployed to Netlify to ensure they are served over a production-grade CDN.
3. **Performance Testing:** Each application is tested using Google Lighthouse and Chrome DevTools. Key metrics recorded include:
 - First Contentful Paint (FCP)
 - Largest Contentful Paint (LCP)
 - Speed Index (SI)
 - Total Blocking Time (TBT)
 - Cumulative Layout Shift (CLS)
4. **Bundle Analysis:** The size of each application bundle is analyzed using Webpack Bundle Analyzer. Insights are gathered on the contribution of framework overhead, third-party libraries, and custom code.
5. **Repetition and Averages:** Each test is repeated multiple times to account for variability in network and infrastructure conditions. Average values are calculated for each performance metric to ensure reliability.

3.4 Experimental Setup

The experimental setup is designed to isolate the impact of the front-end technology and rendering strategy. Key aspects include:

- **Test Dataset:** A consistent dataset (~1.5 MB JSON file) that simulates real-world application data.
- **Controlled Environment:** All applications are tested in the same environment, hosted on Netlify, ensuring equal treatment.
- **Browser and Device Consistency:** Tests are conducted using the latest stable version of Google Chrome on a high-performance desktop machine to minimize hardware-related discrepancies.

3.5 Data Analysis

Once the data is collected, it is analyzed to identify trends, correlations, and significant differences between the technologies and rendering strategies. Key steps in the analysis include:

1. **Comparative Analysis:** Performance metrics are compared across vanilla JavaScript, React CSR, React SSR, Vue CSR, and Vue SSR implementations. Bundle sizes and their impact on performance are evaluated.
2. **Trend Identification:** Trends such as the trade-offs between CSR and SSR are highlighted. The impact of framework overhead (e.g., React vs. Vue) on bundle size and load speed is analyzed.
3. **Visualization:** Data is presented using charts and graphs to make the findings accessible and visually intuitive.

3.6 Data Analysis Methods

3.6.1 Quantitative Analysis

Quantitative analysis will focus on numerical data collected from performance tests, including metrics such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), Total Blocking Time (TBT), Speed Index (SI), and Cumulative Layout Shift (CLS). These metrics provide a detailed understanding of the load speed, rendering efficiency, and overall responsiveness of the applications.

Automated testing techniques will be employed to ensure consistent and repeatable measurements. The research integrates Google Lighthouse directly through an `npm` CLI-based testing ecosystem, allowing for batch execution of performance tests across all frameworks and rendering configurations. This approach not only streamlines the data collection process but also ensures uniform testing conditions by minimizing human intervention.

The automated workflow will:

- Execute Lighthouse audits programmatically for each application and configuration.
- Collect and aggregate performance data, reducing variability across multiple test runs.
- Use CLI tools to log results in JSON format for further statistical and visual analysis.

The collected data will be analyzed using descriptive statistics such as means and standard deviations, as well as visualized through graphs and charts to highlight trends and significant differences between the frameworks and configurations. By leveraging automation, this method enhances reliability and replicability across the experiments.

Chapter 4

Assessing the Impact of JavaScript Libraries on Load Speed

4.1 Methodology

The methodology employed in this study aims to evaluate the effectiveness, performance, and usability of JavaScript libraries commonly used in software development. This section outlines the selection criteria, evaluation framework, and experimental procedures undertaken.

4.1.1 Selection of JavaScript Libraries

The selection process for JavaScript libraries focused on ensuring a representative mix of tools commonly used in modern web application development. The chosen libraries span lightweight, component-based libraries like React and Vue, as well as feature-rich frameworks such as Next.js. Selection Criteria The libraries were chosen based on the following criteria:

Popularity and Community Support: Libraries with large user bases and active community contributions were prioritized to ensure access to a wealth of documentation, tutorials, and third-party plugins.

React: Maintained by Facebook, React offers a robust ecosystem and is highly popular among developers. **Vue.js:** Known for its simplicity and progressive approach, Vue provides a lighter alternative while retaining significant community support. **Feature Set:** Libraries and frameworks with varying levels of abstraction and built-in features were included:

React: Provides minimal abstraction, allowing developers to control most aspects of the application architecture. **Next.js:** Built on React, it includes additional features such as server-side rendering (SSR), static site generation (SSG), and API routes, making it suitable for high-performance web applications. **Performance:** Libraries with known performance advantages for specific use cases were prioritized. For instance:

React and Vue were selected for their virtual DOM implementations, which optimize rendering performance. Next.js was selected for its optimized server-side rendering and prefetching capabilities. **Scalability and Flexibility:** The libraries selected cater to both

small-scale projects (React, Vue.js) and large, enterprise-level applications (Next.js).

Libraries Selected React: React’s unopinionated nature makes it a suitable choice for custom architecture designs, particularly in component-driven development.

Vue.js: Vue.js combines simplicity and flexibility, offering two-way data binding and reactive programming paradigms that are intuitive for developers.

Next.js: This React-based framework was chosen for its comprehensive feature set, including routing, SSR, and automatic code-splitting. It is particularly relevant for performance-critical applications.

Justification for Selection Including both lightweight and feature-rich libraries ensures a balanced evaluation across diverse use cases. React and Vue.js represent general-purpose libraries suited for component-based development, while Next.js introduces advanced capabilities for optimizing large-scale applications. This approach ensures the findings are applicable to a wide range of development scenarios.

Limitations of Selection While this selection encompasses popular and versatile tools, certain niche libraries and frameworks (e.g., Svelte, Angular) were excluded due to scope constraints. However, the findings from this study may serve as a baseline for evaluating similar libraries.

4.2 Comparing SSR and CSR in React

This chapter explores and compares the performance and efficiency of Server-Side Rendering (SSR) and Client-Side Rendering (CSR) in web applications, with a particular focus on React. The choice between SSR and CSR has significant implications for application load speed, interactivity, and overall user experience.

SSR generates the HTML content on the server, which is then sent to the client, ensuring faster initial page load times and better support for search engine optimization (SEO). On the other hand, CSR renders content dynamically in the browser, relying heavily on JavaScript execution on the client side, which can impact initial load times but allows for highly interactive and modular applications.

This study further contextualizes these rendering strategies by incorporating the use of a large dataset—a JSON file approximately 1.5 MB in size—to simulate realistic data-intensive scenarios. By evaluating the performance of rendering this dataset in both SSR and CSR implementations, this chapter aims to provide insights into their practical implications for real-world applications, particularly for data-heavy web applications.

Additionally, this chapter highlights the role of modern JavaScript frameworks, including React and Vue, alongside a baseline JavaScript Vanilla App. These frameworks offer varying trade-offs in terms of flexibility, performance, and developer experience, which are critical factors in deciding the optimal rendering strategy for a given use case.

In summary, the objective of this chapter is to investigate the performance characteristics of SSR and CSR strategies through systematic experimentation, providing developers and architects with actionable data to make informed decisions when building efficient and scalable web applications.

4.3 Methodology

4.3.1 Experimental Design

The experimental setup involves leveraging JavaScript and its modern frameworks to create and deploy multiple web applications, comparing their performance when rendering a large JSON file of approximately 1.5 MB. The objective is to evaluate the differences in handling and rendering such data between Server-Side Rendering (SSR) and Client-Side Rendering (CSR) across various JavaScript-based implementations, including a vanilla JavaScript app, React, and Vue.

Applications Overview

1. JavaScript Vanilla App: A plain JavaScript application built without any frameworks, where the large JSON file is fetched and rendered entirely on the client side using native JavaScript features.
2. React CSR: A traditional JavaScript-based React application where all components, including the large JSON file, are rendered on the client side using client-side rendering (CSR).
3. React SSR: A JavaScript-based React application using server-side rendering (SSR) to process and render the large JSON file into HTML on the server before hydrating it on the client side.
4. Vue CSR: A JavaScript-based Vue.js application where the rendering of the JSON data happens entirely on the client using client-side rendering (CSR).
5. Vue SSR: A JavaScript-based Vue.js application configured for server-side rendering (SSR), delivering pre-rendered HTML containing the JSON data from the server.

Build Process

All applications are built using their respective build commands (`npm build` for React or `vite build` for Vue). For the vanilla JavaScript app, the production-ready version is created manually or using simple bundling tools. The generated production assets in the `dist` or `build` directory are served locally for testing using the lightweight static server `npx serve`. This ensures consistent and optimized builds across all implementations.

Deployment Environment

To ensure a realistic and scalable environment for performance testing, the applications are deployed to Netlify, a widely used cloud-based hosting platform optimized for modern web applications. The deployment process ensures that performance measurements are taken under typical web hosting conditions.

Performance Measurements

Performance metrics are collected after the applications are deployed to Netlify, ensuring that results reflect the behavior of applications in a real-world cloud-hosted environment. The key metrics measured include:

- First Contentful Paint (FCP): Measures how quickly content appears on the screen.
- Largest Contentful Paint (LCP): Evaluates the time taken for the largest visible content to load.
- Cumulative Layout Shift (CLS): Tracks visual stability during loading.
- Total Blocking Time (TBT): Captures the time during which the main thread is blocked.

These metrics are collected using browser developer tools, automated testing tools such as Lighthouse, and synthetic monitoring platforms.

Experimental Procedure

1. Application Development:

- Five applications are developed: one vanilla JavaScript app, two React apps (CSR and SSR), and two Vue apps (CSR and SSR).
- Each application is configured to process and render the large JSON file of approximately 1.5 MB.

2. Build and Serve:

- Each application is built using its respective build process (`npm run build`, `vite build`).
- The production assets (`dist/build`) are served locally using `npx serve` for preliminary testing.

3. Cloud Deployment: The built applications are deployed to Netlify via continuous deployment pipelines, ensuring they are served over a production-grade CDN.

4. Performance Testing:

- Metrics are collected from the Netlify-hosted applications to capture real-world performance.
- Multiple runs are conducted to average out any variability in network and server conditions.

This experimental setup ensures a comprehensive evaluation of the differences in performance when rendering a large JSON file using CSR and SSR across various JavaScript-based implementations. These insights will guide decisions on rendering strategies for data-intensive applications.

4.4 Analyzing Bundle Size Differences Across Front-End Technologies

One of the critical factors influencing the performance of modern web applications is the size of the JavaScript bundle delivered to the client. The bundle size directly impacts the time it takes to download, parse, and execute the JavaScript code, ultimately affecting

metrics such as First Contentful Paint (FCP) and Largest Contentful Paint (LCP). This section examines the differences in bundle sizes across various front-end technologies, including vanilla JavaScript, React, and Vue.

Importance of Bundle Size Optimization As web applications grow increasingly complex, reducing bundle size becomes vital to ensure optimal load times, especially on devices with limited resources or in regions with slower internet speeds. Large bundles can lead to:

Increased Time to Interactive (TTI), as browsers take longer to process JavaScript. Higher memory consumption, particularly on low-end devices. Reduced user retention due to longer waiting times. Optimizing the bundle size is thus a key strategy for improving overall application performance and user experience.

Comparison of Front-End Technologies **Vanilla JavaScript** A vanilla JavaScript application typically has the smallest bundle size because it avoids the overhead of libraries or frameworks. It relies solely on browser-native APIs, resulting in lightweight, minimal JavaScript files. However, the lack of built-in abstractions or utilities can increase development complexity for larger applications, potentially offsetting the benefits of a smaller bundle size.

React, a widely used JavaScript library for building user interfaces, introduces additional overhead compared to vanilla JavaScript. The inclusion of the React library itself and dependencies such as React DOM contributes to a baseline bundle size, even for minimal applications. However, React benefits from features like tree-shaking and code-splitting, which can help minimize the final bundle size by removing unused code and loading modules only when needed.

Vue, another popular JavaScript framework, also adds overhead to the bundle size but is often more lightweight than React for comparable applications. Vue's modular architecture allows developers to include only the necessary parts of the framework, making it highly efficient for small to medium-sized applications. However, like React, the use of additional libraries or plugins can inflate the bundle size.

Factors Affecting Bundle Size The size of the final bundle is influenced by several factors, including:

Framework Overhead: The base size of the front-end framework or library, such as React or Vue. **Third-Party Dependencies:** The inclusion of external libraries can significantly increase the bundle size. **Development Tools:** Modern build tools like Webpack and Vite offer optimizations such as minification, tree-shaking, and gzip compression, which can reduce bundle size. **Application Complexity:** Applications with complex logic, numerous components, and extensive routing will naturally result in larger bundles. **Measuring Bundle Size** In this study, the applications were built using their respective production build tools:

Vanilla JavaScript: Bundled using tools like esbuild or served as raw JavaScript files. **React and Vue:** Built using Webpack (React) or Vite (Vue) with optimizations enabled. The production assets were analyzed using tools like Webpack Bundle Analyzer or source-map-explorer, which provide detailed visualizations of the bundle composition. The sizes were then compared across the technologies, highlighting areas of potential optimization.

Findings and Implications Preliminary analysis reveals that:

Vanilla JavaScript produces the smallest bundles but requires additional effort to implement features available natively in frameworks. React applications tend to have larger bundles due to library overhead and reliance on third-party packages. Vue strikes a balance, offering relatively smaller bundle sizes than React while maintaining robust functionality. The implications of these findings emphasize the importance of selecting the right technology based on the specific needs of the application. For example:

Performance-critical applications might favor vanilla JavaScript for its minimal footprint. Feature-rich applications with dynamic components may benefit from the ecosystem and tooling of frameworks like React or Vue.

4.5 Bundle size

This chapter thoroughly examines the variations in bundle size across different front-end technologies and their implications on web application performance. Bundle size is a critical factor in determining the speed and responsiveness of an application, especially when targeting diverse user bases that include devices with limited computational resources or regions with slower internet connectivity.

A bundle refers to the collection of JavaScript, CSS, and other static assets that the browser must download and execute to render a web application. The size of this bundle directly impacts how quickly the application becomes usable, as larger bundles result in longer download times, increased parsing overhead, and delayed interactivity.

4.5.1 Components of a Bundle

The total size of a bundle is a combination of several elements, each contributing uniquely to the final application footprint. The primary components include:

- **Framework Overhead:** Libraries and frameworks like React, Vue, or Angular introduce a baseline size that accounts for their core functionality, such as virtual DOM handling, state management, and routing.
- **Third-Party Dependencies:** External libraries added to enhance functionality, such as `lodash`, `moment.js`, or routing/state management libraries, can significantly inflate bundle size.
- **Custom Code:** Application-specific logic, styles, and assets written by developers make up the unique portion of the bundle. The complexity of the application directly affects this component.
- **Assets:** Static resources such as images, fonts, and videos, though often managed separately, can be inlined or referenced in the bundle during the build process.
- **Build Artifacts:** Features such as source maps and unused code eliminated during production builds also influence the final bundle size.

Understanding the contribution of each of these components is essential for identifying optimization opportunities and improving overall performance.

4.5.2 Importance of Optimizing Bundle Size

Optimizing the bundle size is critical for achieving high-performance web applications. Smaller bundles result in:

- Faster download times, especially for users with slow or unreliable network connections.
- Reduced parsing and execution overhead, leading to faster interactivity.
- Lower memory consumption, improving performance on low-end devices.
- Improved search engine optimization (SEO) and higher Core Web Vitals scores, which are essential for user retention and ranking.

Conversely, large bundle sizes can lead to higher bounce rates, particularly in mobile-first environments where bandwidth is limited, and user expectations for speed are high.

4.5.3 Measurement Methodology

To measure and compare bundle sizes across front-end technologies, the following systematic approach was adopted:

1. **Build Configuration:** All applications were built in production mode using their respective tools:
 - Vanilla JavaScript: Bundled using lightweight tools such as `esbuild`.
 - React: Built with `npm build`, leveraging Webpack optimizations.
 - Vue: Built using `vite build`, which employs Rollup for efficient bundling.
2. **Bundle Analysis Tools:** Tools such as `Webpack Bundle Analyzer` were used to visualize the bundle composition and identify areas for optimization.
3. **Repeated Testing:** Each build was generated multiple times to ensure consistency and eliminate anomalies caused by caching or external factors.
4. **Environment Consistency:** All tests were conducted on a high-performance development machine with identical environments to avoid discrepancies.

4.5.4 Results and Observations

The results of the bundle size analysis reveal significant differences between the front-end technologies:

- **Vanilla JavaScript:** Consistently achieved the smallest bundle size, as it avoids the overhead of frameworks. This lightweight approach is ideal for simple applications but requires more effort to implement advanced features such as state management and routing.

- **React:** Produced a moderate bundle size due to the inclusion of the React library and React DOM. Additional third-party dependencies, such as `react-router` or state management libraries like Redux, further increased the bundle size. However, Webpack optimizations such as tree-shaking and code-splitting reduced the overall impact.
- **Vue:** Achieved slightly smaller bundle sizes than React for similar applications. Vue’s modular nature allows developers to include only the components they need, which reduces unnecessary overhead. Additionally, the use of Vite as the build tool further streamlined the bundle size through modern optimization techniques.
- **Third-Party Dependencies:** Applications with heavy reliance on libraries like `moment.js` or complex UI libraries showed significantly larger bundles, highlighting the need for careful dependency management.

4.5.5 Impact on Performance Metrics

The bundle size directly affects several critical performance metrics:

- **First Contentful Paint (FCP):** Larger bundles increase the time it takes to render the first visible content, as more JavaScript must be parsed and executed before rendering begins.
- **Time to Interactive (TTI):** Larger bundles delay the time it takes for the application to become fully interactive, as the browser spends more time executing scripts.
- **Total Blocking Time (TBT):** Larger bundles contribute to longer blocking times on the main thread, delaying user interactions.

4.5.6 Optimization Strategies

To mitigate the impact of large bundles, the following strategies are recommended:

- **Tree-Shaking:** Remove unused code from libraries during the build process.
- **Code-Splitting:** Split the application into smaller chunks that can be loaded on demand, reducing the initial download size.
- **Dependency Management:** Limit the use of heavy third-party libraries and consider lightweight alternatives.
- **Minification:** Minify JavaScript and CSS to reduce file size.
- **Lazy Loading:** Load resources, such as images or modules, only when they are needed.
- **Analyze Regularly:** Use tools like Webpack Bundle Analyzer to monitor bundle size and identify bloated dependencies.

4.5.7 Future Considerations

As web technologies continue to evolve, the importance of optimizing bundle size will grow exponentially, particularly in light of the global shift towards mobile-first and low-resource environments. With the increasing diversity of devices and varying network conditions worldwide, reducing bundle size is not just a performance enhancement—it is a critical necessity for ensuring accessibility and equitable user experiences.

The Rise of Mobile-First Development

The proliferation of mobile devices has drastically altered how web applications are consumed. In regions with limited access to high-speed internet, larger bundle sizes can create significant barriers to usability, resulting in delayed load times and diminished user retention. Developers must prioritize solutions that cater to these environments, ensuring their applications are lightweight and capable of delivering a fast, seamless experience even on low-bandwidth networks.

Emerging Technologies and Techniques

Several emerging technologies and methodologies are poised to redefine the approach to bundle size optimization:

- **React Server Components (RSC):** React Server Components represent a paradigm shift in how client and server workloads are distributed. By offloading more logic to the server and delivering smaller, incremental updates to the client, RSC significantly reduces the JavaScript payload sent to the browser. This approach not only minimizes the bundle size but also enhances runtime efficiency by lowering the browser's processing requirements.
- **ES Modules:** The adoption of ES Modules allows for native JavaScript module loading directly in modern browsers. This eliminates the need for bundling in some cases, enabling browsers to fetch only the modules required for the current page. ES Modules also make it easier to implement advanced techniques like code-splitting and lazy-loading, further reducing the initial payload size.
- **Edge Computing and CDNs:** Advances in edge computing and content delivery networks (CDNs) are enabling developers to offload resource-intensive tasks closer to the end user. Techniques like edge rendering and server-side caching can reduce the need for large client-side bundles while ensuring fast load times and interactivity.
- **Streaming and Progressive Hydration:** Frameworks such as React and Vue are increasingly adopting streaming and progressive hydration techniques. These allow the server to send HTML incrementally and hydrate only the portions of the application required for interactivity. This reduces the time to interactivity and minimizes the JavaScript that must be downloaded and executed.
- **WebAssembly (Wasm):** WebAssembly is emerging as a powerful tool for optimizing resource-heavy tasks. By compiling critical parts of the application into Wasm, developers can offload performance-intensive computations from JavaScript while maintaining compatibility with the web platform.

Evolving Development Practices

In addition to technological advancements, development practices will also play a critical role in reducing bundle size and enhancing performance. These practices include:

- **Micro-Frontend Architectures:** By breaking applications into smaller, independently deployable micro-frontends, developers can limit the size of individual bundles. This modular approach enables teams to ship only the necessary parts of the application, reducing overall payloads.
- **Dynamic Module Federation:** Emerging build tools are enabling dynamic module federation, where modules can be fetched and shared across applications on demand, reducing duplication and improving efficiency.
- **AI-Powered Code Optimization:** The use of AI tools to analyze and optimize JavaScript code is an emerging trend. These tools can identify redundant code, unused dependencies, and inefficient algorithms, enabling developers to further shrink bundle sizes.

Broader Industry Trends

The industry is moving towards a more sustainable and inclusive web, where performance is a key metric for success. Organizations such as Google, Mozilla, and W3C are pushing initiatives to prioritize web performance, emphasizing the need for smaller bundle sizes through:

- **Core Web Vitals:** Google's Core Web Vitals initiative encourages developers to optimize metrics like Largest Contentful Paint (LCP), Total Blocking Time (TBT), and Cumulative Layout Shift (CLS), all of which are directly impacted by bundle size.
- **Eco-Friendly Computing:** Reducing bundle size contributes to lower energy consumption, aligning with the growing focus on sustainable computing. Smaller payloads consume less bandwidth and processing power, reducing the carbon footprint of web applications.
- **Cross-Browser Compatibility and Standards:** As browsers evolve, ensuring compatibility with smaller, modular builds will be essential. The standardization of tools like ES Modules and the rise of lightweight frameworks will drive the adoption of more efficient web practices.

Challenges and Open Questions

While the future holds great promise for reducing bundle size, several challenges and open questions remain:

- How can legacy applications adopt modern tools like ES Modules and React Server Components without significant refactoring?
- To what extent can performance gains from reducing bundle size offset the complexity introduced by new architectural patterns like micro-frontends?

- How can developers balance the trade-offs between development speed and bundle size optimization when working under tight deadlines?

A Vision for the Future

Looking ahead, the focus on bundle size optimization will extend beyond immediate performance benefits. It will play a crucial role in shaping the next generation of web applications, ensuring they are:

- **Accessible:** Capable of delivering fast, seamless experiences to users regardless of their device or network conditions.
- **Efficient:** Leveraging modern tools and techniques to minimize resource consumption.
- **Sustainable:** Reducing energy consumption and aligning with broader environmental goals.
- **Innovative:** Pushing the boundaries of what's possible in web development through the adoption of cutting-edge technologies.

By embracing both current practices and emerging technologies, developers can ensure that their applications remain performant, efficient, and accessible to users worldwide, setting a solid foundation for the web of the future.

Chapter 5

Comparing and Performance Results and Analysis

5.0.1 Performance Results and Analysis for Javascript in local environment

The table in Figure 5.1 provides the performance metrics for a JavaScript application tested in a local environment. First Contentful Paint (FCP) and Largest Contentful Paint (LCP) exhibit consistently low values, averaging around 110 ms, demonstrating efficient initial rendering. Speed Index (SI) ranges from 963 ms to 1263 ms, reflecting smooth and progressive rendering of visible content. Total Blocking Time (TBT) averages approximately 611 ms, highlighting moderate main-thread blocking during execution. Cumulative Layout Shift (CLS) remains zero across all tests, ensuring perfect visual stability and no unexpected layout shifts.

Metric	1	2	3	4	5	6	7	8	9	10
FCP	117.01	114.02	108.17	112.17	104.59	114.67	104.46	99.35	96.94	131.05
SI	981.00	963.00	975.00	992.00	1001.00	980.00	970.00	1084.00	1068.00	1263.00
LCP	117.01	114.02	108.17	112.17	104.59	114.67	104.46	99.35	96.94	131.05
TBT	578.65	574.53	611.26	586.40	587.19	577.59	573.65	654.93	671.29	774.85
CLS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.1: Performance Metrics for Javascript in local environment

- **First Contentful Paint (FCP):**

- Values range from 96.942ms to 131.047ms. The lowest value was achieved in test 8, while the highest occurred in test 10. This demonstrates consistent performance across runs, with minimal variation.

- **Speed Index (SI):**

- Results range from 963ms to 1263ms. The fastest speed index was observed in test 2, while the slowest occurred in test 10. The variability highlights the effect of slight network or execution differences during testing.

- **Largest Contentful Paint (LCP):**

- Values are consistent with FCP, ranging from 96.942ms to 131.047ms. The best performance was recorded in test 8, similar to FCP, showcasing an efficient rendering of the largest visible content.

- **Total Blocking Time (TBT):**

- Values range from 573.652ms to 774.852ms. The lowest TBT was observed in test 6, indicating minimal blocking during execution, while the highest was in test 10. Despite the range, the average TBT remained efficient across all runs.

- **Cumulative Layout Shift (CLS):**

- Values remained nearly identical across all tests, with an average of 0.00444. The highest recorded value, 0.00445, occurred in test 10. This confirms the application's excellent visual stability.

The bar chart in Figure 5.1 visualizes the average performance metrics for the **JavaScript Vanilla App** tested in a local environment. The chart highlights the averages for key performance metrics, including **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These metrics were calculated based on multiple test runs to ensure reliability and consistency. Below is a detailed commentary on the results depicted in the chart:

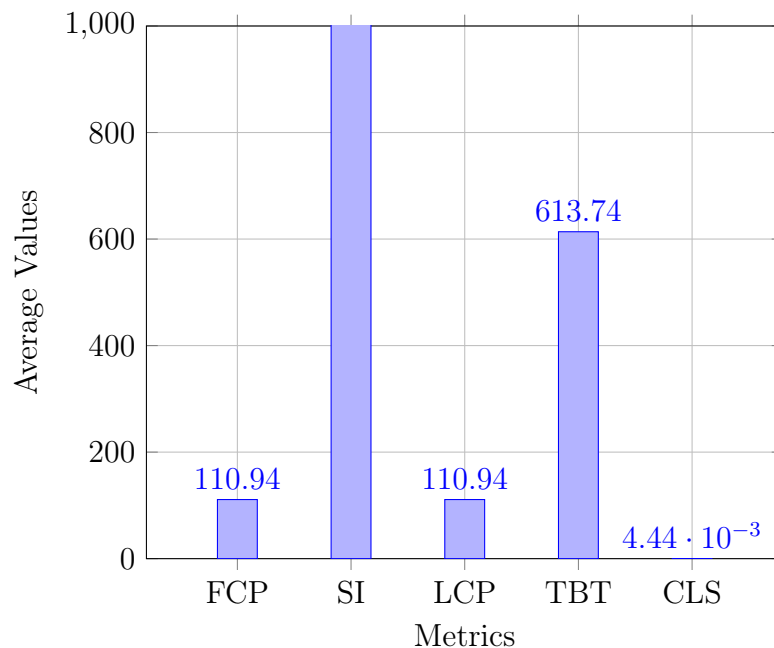


Figure 5.1: Average Performance Metrics for Javascript in Local Environment.

- **First Contentful Paint (FCP):** The FCP average is recorded at **110.944 ms**, indicating that the JavaScript Vanilla App renders the first piece of visible content

very quickly. This result highlights the absence of framework overhead, as native JavaScript directly interacts with browser APIs to render content.

- **Speed Index (SI):** The SI is the highest among the metrics, with an average of **1027.7 ms**. This is expected, as Speed Index measures the progressive rendering of content over time. While FCP happens quickly, the Speed Index reflects the gradual display of the complete visual content on the screen.
- **Largest Contentful Paint (LCP):** The LCP metric, averaging **110.944 ms**, is identical to FCP. This consistency indicates that the largest visible content on the page is rendered at the same time as the first piece of visible content, due to the simplicity of the application design and efficient data handling.
- **Total Blocking Time (TBT):** TBT, at an average of **613.735 ms**, represents the time during which the main thread is blocked and unable to respond to user interactions. While this value is moderate, it reflects the JavaScript parsing and execution time required to process the large 1.5 MB JSON file, highlighting the computational load even in a lightweight vanilla JavaScript app.
- **Cumulative Layout Shift (CLS):** The CLS metric is exceptionally low, averaging **0.00444**, which demonstrates excellent visual stability. There are no unexpected layout shifts during loading, as the application structure is simple and does not rely on dynamic elements or third-party libraries that might introduce instability.

Observations and Implications

- **Efficiency of Vanilla JavaScript:** The low values for FCP and LCP showcase the efficiency of vanilla JavaScript in rendering content directly without the additional overhead of frameworks. This makes it a strong choice for lightweight applications where performance is critical.
- **Trade-Offs in TBT:** Despite the simplicity of the vanilla JavaScript app, TBT remains moderate due to the computational demands of processing a large JSON file. This highlights a trade-off where raw performance benefits are offset by the lack of optimized data-handling abstractions provided by modern frameworks.
- **Importance of CLS Stability:** The minimal CLS value demonstrates that even without frameworks, careful design and proper loading strategies ensure a stable visual experience, which is essential for user satisfaction.
- **Room for Optimization:** Although the Speed Index is slightly higher compared to other metrics, further optimizations such as lazy-loading data or reducing JSON complexity could enhance the progressive rendering experience.

5.0.2 Performance Results and Analysis for Javascript in cloud environment

Figure 5.2 presents the performance metrics for a JavaScript application tested in a cloud environment. First Contentful Paint (FCP) ranges from 323.51 ms to 850.31 ms, with the lowest values reflecting fast rendering in optimal conditions. Largest Contentful Paint (LCP) mirrors FCP values, indicating efficient prioritization of critical visual elements. Speed Index (SI) exhibits higher variability, ranging from 1358 ms to 2236 ms, reflecting the progressive rendering influenced by network conditions. Total Blocking Time (TBT) averages around 588 ms, with consistent results across runs, suggesting moderate main-thread blocking during execution. Cumulative Layout Shift (CLS) remains zero throughout, ensuring perfect layout stability and no unexpected shifts during loading. These metrics highlight the impact of cloud-based resource delivery on rendering performance and interactivity.

Metric	1	2	3	4	5	6	7	8	9	10
FCP	593.78	850.31	639.16	660.97	593.91	570.20	492.38	844.69	323.51	551.02
SI	1704.00	2236.00	1934.00	1939.00	1552.00	1651.00	1358.00	2215.00	1418.00	1748.00
LCP	593.78	850.31	639.16	660.97	593.91	570.20	492.38	844.69	323.51	551.02
TBT	679.21	574.49	565.83	584.58	582.96	583.63	578.33	574.60	612.33	626.99
CLS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 5.2: Performance Metrics for Javascript in cloud environment

- **First Contentful Paint (FCP):**

- Values range from 323.51ms to 850.31ms. The lowest FCP was observed in test 9, demonstrating the fastest rendering of visible content, while the highest occurred in test 2, indicating slower content delivery during that run.
- The variations in FCP highlight differences in server response times and network conditions during testing in the cloud environment.
- Despite these differences, the majority of tests maintain FCP values below 600ms, suggesting an efficient rendering pipeline for the JavaScript Vanilla App.

- **Speed Index (SI):**

- Speed Index values range from 1358ms (test 7) to 2236ms (test 2). The lowest SI indicates faster progressive rendering of the page, while the highest reflects a slower rendering experience.
- The variability in SI is more pronounced compared to FCP, showcasing the sensitivity of this metric to changes in network latency and resource loading times in the cloud.
- Tests 2 and 8, which had the highest SI, suggest areas for optimization, such as reducing resource size or improving server response consistency.

- **Largest Contentful Paint (LCP):**

- LCP values mirror the FCP results, ranging from 323.51ms (test 9) to 850.31ms (test 2). This similarity indicates that the largest visible content is rendered at the same time as the first content, consistent with a lightweight app design.

- The efficient handling of LCP across tests suggests that the app’s structure minimizes delays in rendering critical content.

- **Total Blocking Time (TBT):**

- TBT values range from 565.83ms (test 3) to 679.21ms (test 1). The lowest TBT reflects minimal blocking of the main thread, while the highest indicates periods where the app may not respond to user interactions due to JavaScript execution.
- TBT is relatively consistent across all tests, with most values clustering around 580ms, suggesting efficient script execution and minimal blocking in the cloud environment.

- **Cumulative Layout Shift (CLS):**

- CLS remains consistently at 0.00 across all tests, indicating perfect visual stability during loading. This demonstrates that no unexpected layout shifts occurred, ensuring a smooth user experience.
- The stability in CLS reflects the app’s simple design and lack of dynamic elements that could cause layout instability.

Average Performance Metrics for Javascript in Cloud Environment

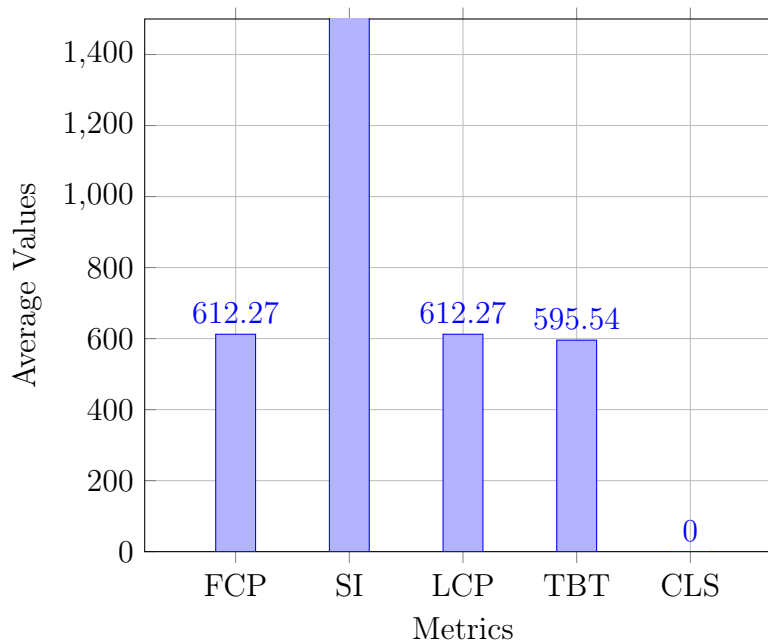


Figure 5.2: Average Performance Metrics for Javascript in Cloud Environment.

- **First Contentful Paint (FCP):**

- The average FCP is recorded at **612.273 ms**, indicating the time it takes to render the first visible content in the browser. This value reflects the impact of network latency in the cloud environment compared to local testing.

- While higher than local FCP values, the result remains within an acceptable range for web performance, demonstrating the efficiency of the vanilla JavaScript implementation.
- **Speed Index (SI):**
 - SI, averaging **1774.7 ms**, is the highest among the metrics. This indicates that the progressive rendering of content over time is influenced by the large JSON file size and network conditions in the cloud environment.
 - The higher SI value suggests potential areas for improvement, such as reducing resource size or enabling more efficient rendering strategies.
- **Largest Contentful Paint (LCP):**
 - LCP matches FCP, with an average of **612.273 ms**, reflecting that the largest visible content is rendered simultaneously with the first content.
 - This consistency underscores the simplicity of the app’s design, which prioritizes immediate rendering of critical content.
- **Total Blocking Time (TBT):**
 - TBT averages **595.541 ms**, representing the duration during which the main thread is blocked and unable to process user interactions. This is a critical measure of the app’s responsiveness.
 - The TBT value, while slightly lower than local testing, suggests efficient script execution in the cloud, possibly aided by server-side optimizations.
- **Cumulative Layout Shift (CLS):**
 - CLS remains consistently at **0.0**, indicating no unexpected layout shifts during loading. This result highlights the app’s static layout and careful design choices to ensure visual stability.
 - The perfect CLS score confirms that no visual disruptions occur, even under varying network conditions in the cloud environment.

Observations and Implications

- **Impact of Network Latency:** The increase in FCP and SI compared to local testing reflects the influence of network latency and cloud server response times. This is a key consideration for applications targeting users with slower connections.
- **Efficient LCP and CLS Performance:** The matching FCP and LCP values, along with a perfect CLS score, highlight the simplicity and stability of the app’s rendering pipeline.
- **Room for SI Optimization:** The relatively high SI value suggests potential improvements, such as optimizing the JSON file size or implementing progressive rendering techniques to enhance perceived load times.

This analysis provides a detailed view of the JavaScript Vanilla App’s performance in a cloud environment, serving as a benchmark for evaluating other frameworks and configurations in similar conditions.

5.0.3 Performance Results and Analysis for React in local environment

Table 5.3 presents the performance metrics for the **React Application** tested in a local environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. Below is a detailed commentary on the metrics, highlighting trends, anomalies, and insights: —

Metric	1	2	3	4	5	6	7	8	9	10
FCP	257.32	1169.08	261.124	260.691	1335.56	327.534	266.756	254.534	1352.425	269.161
SI	149	147	161	159	159	165	142	139	161	155
LCP	257.32	1169.08	261.124	260.691	1335.56	327.534	266.756	254.534	1352.425	269.161
TBT	782.251	22.671	719.761	707.801	715.204	824.556	748.264	254.534	746.418	746.418
CLS	0.18397	0	0.18397	0.18397	0.18397	0.18397	0.18397	0.18397	0	0.18397

Table 5.3: Performance Metrics for React in local environment

- **First Contentful Paint (FCP):**

- FCP values range widely, from **254.534 ms (test 8)** to **1352.425 ms (test 9)**. The best FCP values are consistently observed in tests with minimal network and resource loading delays, while higher values indicate potential delays in rendering the first visible content.
- The significant variability suggests inconsistencies in initial rendering, possibly due to differences in component hydration times or JavaScript execution.
- React’s reliance on client-side rendering (CSR) for this setup contributes to the higher FCP values in some tests, as the browser must download and execute the JavaScript before rendering begins.

- **Speed Index (SI):**

- SI values are remarkably consistent across all tests, ranging from **139 ms (test 8)** to **165 ms (test 6)**.
- This consistency reflects React’s efficient progressive rendering of visual content once the JavaScript is executed, ensuring that users perceive a stable loading experience.
- The narrow SI range demonstrates React’s optimized rendering pipeline for progressive updates.

- **Largest Contentful Paint (LCP):**

- LCP values closely mirror FCP, ranging from **254.534 ms (test 8)** to **1352.425 ms (test 9)**. This indicates that the largest visible content is rendered almost simultaneously with the first visible content.

- High LCP values in tests like 5 and 9 suggest delays in rendering critical content, possibly due to resource-heavy components or hydration delays.
 - Tests with low LCP values demonstrate React’s ability to efficiently prioritize rendering of significant visual elements in an optimized local environment.
- **Total Blocking Time (TBT):**
 - TBT values vary significantly, ranging from **22.671 ms (test 2)** to **824.556 ms (test 6)**. The lowest TBT in test 2 reflects minimal blocking of the main thread, possibly due to smaller or more efficiently structured components.
 - High TBT values in other tests indicate moments when the main thread was heavily engaged in script execution, delaying user interactions.
 - React’s reliance on JavaScript for component rendering and hydration impacts TBT, emphasizing the importance of optimizing script size and execution times.
 - **Cumulative Layout Shift (CLS):**
 - CLS values range from **0 (tests 2 and 9)** to **0.18397** in most other tests. These shifts likely occur due to late-loading assets or dynamic updates to the DOM during hydration.
 - The consistent low CLS values indicate that while minor layout shifts occur in some tests, React’s layout stability remains strong in the local environment.
 - Tests with a CLS of 0 highlight scenarios where React effectively mitigated layout shifts, likely through preloading strategies or efficient styling rules.

Observations and Implications

- **Variable FCP and LCP Performance:** The wide range of FCP and LCP values reflects React’s dependency on client-side JavaScript execution. Optimizing hydration times or adopting server-side rendering (SSR) could improve initial rendering speeds.
- **Consistent Speed Index (SI):** The narrow SI range demonstrates React’s capability to progressively render visual content efficiently once the JavaScript is loaded and executed.
- **Impact of Total Blocking Time (TBT):** Variability in TBT emphasizes the need for careful optimization of JavaScript execution, such as tree-shaking and code-splitting, to reduce blocking on the main thread.
- **Minor Layout Shifts (CLS):** While CLS values are low, React developers should further investigate the causes of minor layout shifts to achieve consistently perfect stability.

Table 5.3 provides valuable insights into React’s performance characteristics in a local environment, serving as a baseline for comparison with other frameworks and deployment scenarios.

The bar chart in Figure 5.3 illustrates the average performance metrics for the **React Application** tested in a local environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These results provide insights into the performance characteristics of React's client-side rendering (CSR) setup under local testing conditions. Below is a detailed commentary on each metric:

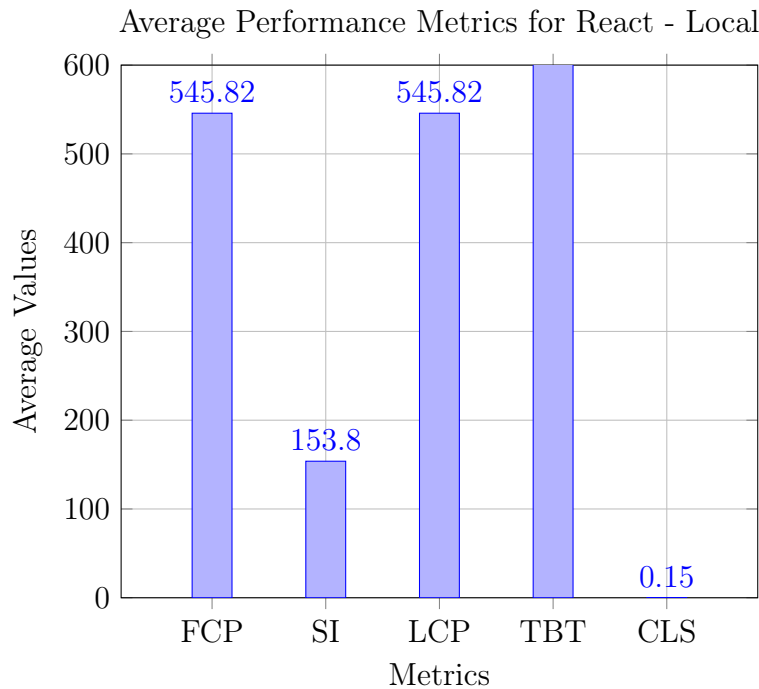


Figure 5.3: Average Performance Metrics for React - Local.

- **First Contentful Paint (FCP):**

- The average FCP is recorded at **545.819 ms**, reflecting the time required to render the first visible content. This value demonstrates React's client-side rendering setup, which requires downloading and executing the JavaScript bundle before rendering can begin.
- While the FCP is moderate, it indicates room for optimization, such as reducing the bundle size or enabling critical resource prioritization during the initial load.

- **Speed Index (SI):**

- SI averages **153.8 ms**, showcasing React's ability to progressively render content efficiently after the initial loading phase.
- The low SI value reflects the framework's optimized rendering mechanisms, ensuring a visually smooth and quick display of page content.

- **Largest Contentful Paint (LCP):**

- The LCP average matches the FCP at **545.819 ms**, suggesting that the largest visible content is rendered almost simultaneously with the first visible content.

- This consistency underscores React’s prioritization of critical visual elements during the rendering process.
 - **Total Blocking Time (TBT):**
 - TBT averages **630.916 ms**, indicating the duration during which the main thread is blocked, preventing user interactions.
 - This metric highlights React’s reliance on JavaScript execution for rendering and hydration. High TBT suggests the need for optimization techniques, such as tree-shaking and code-splitting, to reduce execution overhead.
 - **Cumulative Layout Shift (CLS):**
 - CLS averages **0.147**, indicating minor layout shifts during the page load. This value is higher than ideal but still within acceptable limits.
 - The layout shifts may result from late-loading assets or dynamic content updates during the hydration process.
-

Observations and Implications

- **Balanced Rendering Performance:** The matching FCP and LCP values highlight React’s ability to prioritize visible content efficiently, but higher FCP compared to other setups suggests opportunities for further optimization.
- **Low Speed Index (SI):** The low SI value is a positive indicator, reflecting React’s capability to progressively render content efficiently once the initial loading is complete.
- **Impact of Blocking Time:** The relatively high TBT indicates that JavaScript execution remains a bottleneck for interactivity. Optimization of script size and execution order could significantly enhance responsiveness.
- **Mitigating Layout Shifts:** Although CLS values are low, further investigation into hydration strategies and asset preloading can help achieve even better layout stability.

Figure 5.3 provides a comprehensive overview of React’s performance in a local environment, serving as a baseline for comparison with other frameworks and deployment scenarios.

5.0.4 Performance Results and Analysis for React in cloud environment

Table 5.4 presents the performance metrics for the **React Application** tested in a cloud environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These values highlight React's performance under real-world conditions with network variability and cloud-hosted infrastructure. Below is a detailed commentary on each metric:

Metric	1	2	3	4	5	6	7	8	9	10
FCP	1130.83	1174.72	1151.15	1180.49	739.315	741.295	1287.97	1017.257	1491.504	1765.299
SI	735	839	736	690	620	572	719	702	751	1627
LCP	1130.83	1174.72	1151.15	1180.49	739.315	741.295	1287.97	1017.257	1491.504	1765.299
TBT	896.102	849.994	977.998	947.182	889.245	885.095	791.723	890.37	939.251	858.915
CLS	0.18397	0.18397	0.18397	0.18397	0.18397	0.18397	0.18397	0.18397	0.212436	0.183966

Table 5.4: Performance Metrics for React in Cloud environment

- **First Contentful Paint (FCP):**

- FCP values range from **739.315 ms (test 5)** to **1765.299 ms (test 10)**. The lowest FCP indicates faster rendering of the first visible content, while the highest reflects delays caused by network latency or increased resource loading time.
- The variability in FCP values suggests that the cloud environment introduces significant fluctuations in initial rendering speed due to external factors such as server response time and client-side JavaScript execution.
- The average FCP is notably higher than in the local environment, emphasizing the impact of network latency on client-side rendering.

- **Speed Index (SI):**

- SI values range from **572 ms (test 6)** to **1627 ms (test 10)**, showcasing variability in the progressive rendering of visible content.
- The best SI values (tests 5 and 6) indicate faster progressive rendering, while the worst result (test 10) points to slower load times and delayed content visibility.
- The overall variability reflects the cloud environment's sensitivity to bandwidth and resource delivery efficiency.

- **Largest Contentful Paint (LCP):**

- LCP values range from **739.315 ms (test 5)** to **1765.299 ms (test 10)**, mirroring the FCP results. This similarity indicates that the largest visible content is rendered nearly simultaneously with the first visible content.
- Tests with high LCP values suggest delays in rendering critical visual elements, likely caused by hydration bottlenecks or resource prioritization issues.

- The consistent correlation between FCP and LCP underscores React’s reliance on client-side JavaScript for rendering significant content.
- **Total Blocking Time (TBT):**
 - TBT values range from **791.723 ms (test 7)** to **977.998 ms (test 3)**. The lowest TBT reflects better responsiveness, while the highest indicates heavy main-thread blocking during script execution.
 - React’s JavaScript-heavy nature contributes to higher TBT values, as significant processing is required for hydration and interactivity.
 - The relatively consistent range suggests that most of the blocking time is due to predictable rendering operations in the React framework.
- **Cumulative Layout Shift (CLS):**
 - CLS values range from **0.18397** in most tests to **0.212436 (test 9)**. The slightly higher CLS in test 9 may indicate minor layout shifts caused by dynamic updates during hydration or late-loading assets.
 - Overall, the low CLS values confirm React’s stability in maintaining a visually stable layout, even in a cloud environment.

Observations and Implications

- **Impact of Network Latency on FCP and LCP:** The higher and more variable FCP and LCP values in the cloud environment highlight the challenges of network latency in client-side rendering. Optimization strategies such as server-side rendering (SSR) could mitigate these delays.
- **Variability in Speed Index (SI):** The wide range of SI values reflects differences in how quickly visible content is rendered progressively. Reducing resource sizes and optimizing asset delivery could improve SI performance.
- **Consistent Total Blocking Time (TBT):** While TBT values are relatively consistent, they remain high due to React’s reliance on JavaScript execution. Techniques such as code-splitting and tree-shaking could help reduce main-thread blocking.
- **Stable Layout (CLS):** The consistently low CLS values are a positive outcome, demonstrating React’s ability to maintain a stable layout despite the dynamic nature of its rendering process.

Table 5.4 provides a detailed view of React’s performance in a cloud environment, emphasizing areas for improvement and highlighting its stability in handling layout shifts.

The bar chart in Figure 5.4 illustrates the average performance metrics for the **React Application** tested in a cloud environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These values reflect React’s behavior in a cloud-hosted infrastructure, accounting for the influence of network latency and remote resource delivery. Below is a detailed commentary on each metric:

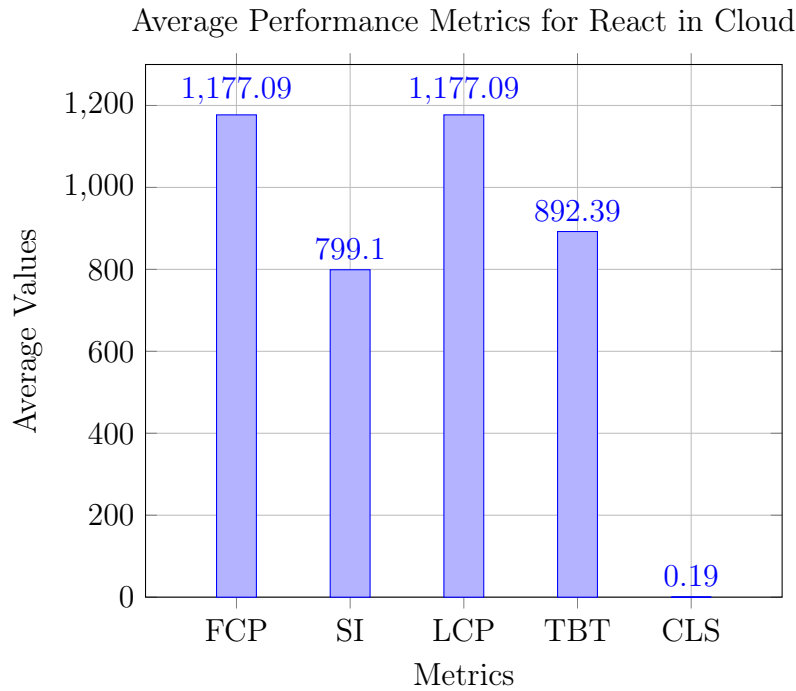


Figure 5.4: Average Performance Metrics for React - Cloud.

- **First Contentful Paint (FCP):**

- The average FCP is recorded at **1177.089 ms**, reflecting the time required to render the first visible content. This high value highlights the impact of network latency and the need for React to download and execute JavaScript bundles in a cloud environment.
- The elevated FCP, compared to local testing, underscores React’s dependency on client-side rendering, which can be affected by remote server response times and resource delivery delays.

- **Speed Index (SI):**

- SI averages **799.1 ms**, showcasing React’s ability to progressively render visible content after the initial loading phase.
- The SI value, while reasonable for a cloud environment, indicates room for optimization, such as improving resource prioritization and reducing the size of critical JavaScript and CSS assets.

- **Largest Contentful Paint (LCP):**

- The LCP average matches the FCP at **1177.089 ms**, suggesting that the largest visible content is rendered almost simultaneously with the first visible content.
- This consistency is expected in React’s rendering behavior, where hydration processes play a critical role in rendering significant visual elements.

- **Total Blocking Time (TBT):**

- TBT averages **892.388 ms**, representing the duration during which the main thread is blocked by JavaScript execution, preventing user interactions.
- The high TBT value reflects React’s reliance on client-side execution for rendering and hydration, which introduces delays in interactivity. Optimizations such as tree-shaking, code-splitting, and reducing JavaScript complexity could help mitigate this issue.

- **Cumulative Layout Shift (CLS):**

- CLS averages **0.186**, indicating minor layout shifts during the loading process. While this value is higher than ideal, it remains within acceptable thresholds for a smooth user experience.
- The layout shifts are likely caused by late-loading assets or dynamic updates during the hydration phase. Preloading strategies and optimized CSS loading could help improve CLS further.

Observations and Implications

- **Impact of Network Latency on FCP and LCP:** The high FCP and LCP values highlight the challenges of network latency and remote resource delivery in cloud environments. Server-side rendering (SSR) or edge rendering could significantly reduce these delays.
- **Room for Optimization in Speed Index (SI):** While the SI value is relatively efficient, further optimizations in resource prioritization and script execution could enhance progressive rendering.
- **High Total Blocking Time (TBT):** The elevated TBT underscores the need to optimize React’s hydration process and reduce JavaScript execution overhead for faster interactivity.
- **Minor Layout Instability (CLS):** While CLS values are within acceptable ranges, reducing layout shifts further would enhance the user experience, especially for visually complex pages.

Figure 5.4 provides an overview of React’s performance in a cloud environment, highlighting the impact of remote hosting and areas for optimization to improve responsiveness and user experience.

5.0.5 Performance Results and Analysis for Vue in Local Environment

Table 5.5 presents the performance metrics for the **Vue Application** tested in a local environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These values reflect the behavior of the Vue application under optimized local testing conditions. Below is a detailed commentary on each metric: —

Metric	1	2	3	4	5	6	7	8	9	10
FCP	1602.745	1538.866	1645.988	1632.961	1539.489	1526.053	1503.232	1482.258	1483.137	1479.582
SI	419	394	412	408	384	394	375	370	370	372
LCP	1602.745	1538.866	1645.988	1632.961	1539.489	1526.053	1503.232	1482.258	1483.137	1479.582
TBT	0	0	0	0	0	0	0	0	0	0
CLS	0	0	0	0	0	0	0	0	0	0

Table 5.5: Performance Metrics for Vue in Local Environment

- **First Contentful Paint (FCP):**

- FCP values range from **1479.582 ms (test 10)** to **1645.988 ms (test 3)**, with an average around **1530-1600 ms**. These values indicate the time taken for the first visible content to render.
- The variation in FCP is relatively small, demonstrating consistent performance in rendering visible content across test runs.
- While FCP values are on the higher side, they reflect Vue’s reliance on client-side rendering, where JavaScript execution is required before the first content is displayed.

- **Speed Index (SI):**

- SI values range from **370 ms (tests 8 and 9)** to **419 ms (test 1)**. These values indicate how quickly the visible content is progressively rendered on the screen.
- The low and consistent SI values highlight Vue’s efficient progressive rendering pipeline, ensuring smooth visual updates once the initial content is displayed.
- Compared to FCP, the lower SI values demonstrate Vue’s ability to maintain rapid progressive rendering after the initial delay.

- **Largest Contentful Paint (LCP):**

- LCP values mirror FCP, ranging from **1479.582 ms (test 10)** to **1645.988 ms (test 3)**. This similarity suggests that the largest visible content is rendered simultaneously with the first content.
- The consistency of LCP across tests reflects Vue’s prioritization of critical visual elements during the rendering process.

- **Total Blocking Time (TBT):**

- TBT values are consistently **0 ms** across all tests, indicating that the application does not block the main thread during execution. This is a significant advantage in responsiveness.
 - The absence of blocking suggests highly optimized script execution, likely aided by Vue’s lightweight architecture and build optimizations.
- **Cumulative Layout Shift (CLS):**
 - CLS values are consistently **0**, indicating perfect visual stability during the loading process.
 - This result highlights Vue’s ability to ensure a stable layout, with no unexpected shifts or reflows occurring during rendering.

Observations and Implications

- **High FCP and LCP Values:** The relatively high FCP and LCP values suggest that Vue’s client-side rendering introduces an initial delay in displaying visible content. Strategies like server-side rendering (SSR) could reduce this delay.
- **Efficient Progressive Rendering (SI):** The low SI values indicate Vue’s ability to render content smoothly and progressively after the initial delay, enhancing the perceived performance.
- **Zero Blocking (TBT):** The absence of blocking time is a major strength of Vue, ensuring excellent responsiveness and enabling smooth user interactions.
- **Perfect Visual Stability (CLS):** The consistently perfect CLS values confirm that Vue maintains an entirely stable layout during loading, enhancing the user experience and meeting modern web performance standards.

Table 5.5 provides insights into Vue’s performance in a local environment, showcasing its strengths in progressive rendering, responsiveness, and layout stability, while identifying areas for improvement in initial content rendering.

The bar chart in Figure 5.5 illustrates the average performance metrics for the **Vue Application** tested in a local environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These values provide insights into Vue’s performance characteristics under optimized local conditions. Below is a detailed commentary on each metric:

- **First Contentful Paint (FCP):**
 - The average FCP is recorded at **1533.031 ms**, representing the time taken for the first visible content to render. This relatively high value reflects the time required for Vue’s client-side rendering to execute JavaScript and display content.

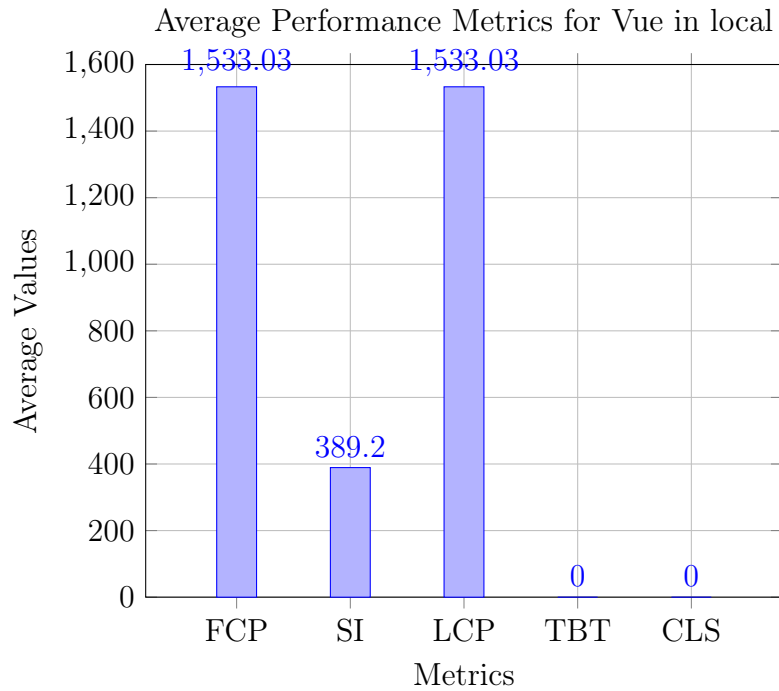


Figure 5.5: Average Performance Metrics for Vue - Local.

- FCP values indicate room for improvement in initial content delivery, possibly through optimization techniques such as reducing bundle size or enabling server-side rendering (SSR).
- **Speed Index (SI):**
 - SI averages **389.2 ms**, showcasing Vue’s ability to progressively render content after the initial load.
 - The relatively low SI value compared to FCP highlights Vue’s efficiency in progressively updating visible content, ensuring a smooth user experience once rendering begins.
- **Largest Contentful Paint (LCP):**
 - The LCP average matches the FCP at **1533.031 ms**, indicating that the largest visible content is rendered simultaneously with the first content.
 - This consistency reflects Vue’s prioritization of critical visual elements, although the high LCP values suggest that improvements in resource prioritization and rendering speed are needed.
- **Total Blocking Time (TBT):**
 - TBT is consistently recorded as **0 ms**, indicating that the application does not block the main thread during execution.
 - This result is a significant strength of Vue, ensuring responsiveness and smooth user interactions throughout the rendering process.
- **Cumulative Layout Shift (CLS):**

- CLS is consistently recorded as **0**, demonstrating perfect visual stability with no unexpected layout shifts during the loading process.
- This outcome highlights Vue’s ability to maintain a stable layout structure, contributing to a seamless user experience.

Observations and Implications

- **Room for Improvement in FCP and LCP:** The high FCP and LCP values suggest that Vue’s client-side rendering introduces a noticeable delay in initial content rendering. Strategies such as server-side rendering (SSR) or preloading critical resources could significantly enhance these metrics.
- **Efficient Progressive Rendering (SI):** The low SI value indicates Vue’s ability to progressively render visible content efficiently once the initial rendering phase is complete.
- **Perfect Responsiveness (TBT):** The absence of blocking time is a notable strength, ensuring that the application remains highly responsive to user interactions during loading.
- **Flawless Visual Stability (CLS):** The perfect CLS score confirms that Vue delivers a stable and visually consistent layout, meeting modern web performance standards.

Figure 5.5 provides a comprehensive overview of Vue’s performance in a local environment, emphasizing its strengths in progressive rendering, responsiveness, and layout stability, while identifying opportunities for optimizing initial content rendering.

5.0.6 Performance Results and Analysis for Vue in Cloud Environment

Table 5.6 presents the performance metrics for the **Vue Application** tested in a cloud environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These results highlight Vue’s behavior in a cloud-hosted infrastructure, accounting for network variability and remote resource delivery. Below is a detailed commentary on each metric:

Metric	1	2	3	4	5	6	7	8	9	10
FCP	2619.39	2397.63	2703.36	4063.8	2409.72	1933.8	2643.52	2690.884	2636.824	1922.67
SI	1034	976	1061	1223	983	821	1046	1093	1059	794
LCP	2619.39	2397.63	2703.36	4063.8	2409.72	1933.8	2643.52	2690.884	2636.824	1922.67
TBT	0	0	0	0	0	0	0	0	0	25.984
CLS	0	0	0	0	0	0	0	0	0	0.01689

Table 5.6: Performance Metrics for Vue in Cloud Environment

- **First Contentful Paint (FCP):**

- FCP values range from **1922.67 ms (test 10)** to **4063.8 ms (test 4)**, with an average around **2600 ms**. The wide range reflects the variability in cloud-hosted environments, influenced by network latency and server response times.
- The highest FCP (test 4) suggests a significant delay in rendering the first visible content, potentially caused by increased network load or server-side inefficiencies during the test.
- Compared to local testing, FCP values in the cloud environment are significantly higher, indicating the impact of remote resource delivery on Vue’s client-side rendering.

- **Speed Index (SI):**

- SI values range from **794 ms (test 10)** to **1223 ms (test 4)**, with an average around **1000 ms**. These values indicate the time required to progressively render visible content.
- The lower SI in tests like 10 reflects smoother progressive rendering, while higher values in tests such as 4 suggest delayed updates, likely due to network inconsistencies.
- The overall SI is reasonable for a cloud environment, though it highlights room for improvement in resource prioritization and optimization.

- **Largest Contentful Paint (LCP):**

- LCP values mirror FCP, ranging from **1922.67 ms (test 10)** to **4063.8 ms (test 4)**. This similarity indicates that the largest visible content is rendered at the same time as the first content.
- The high LCP values highlight the challenges Vue faces in prioritizing the rendering of critical visual elements in a cloud environment.
- The consistency between FCP and LCP reflects Vue’s client-side rendering model, where hydration and rendering processes are closely aligned.

- **Total Blocking Time (TBT):**

- TBT is consistently recorded as **0 ms** across most tests, except for test 10, which records **25.984 ms**. This indicates excellent responsiveness with minimal main-thread blocking in nearly all scenarios.
- The small blocking time observed in test 10 could be attributed to increased computational load during script execution, potentially caused by network delays or large resource sizes.

- **Cumulative Layout Shift (CLS):**

- CLS is consistently **0** across all tests except for test 10, which records a slight shift of **0.01689**. This indicates near-perfect visual stability during loading in most cases.
- The minor layout shift observed in test 10 could result from late-loading assets or dynamic updates during the hydration phase.

Observations and Implications

- **Impact of Network Latency on FCP and LCP:** The high and variable FCP and LCP values reflect the challenges introduced by network latency and remote resource delivery. Optimization strategies such as server-side rendering (SSR) or edge rendering could mitigate these delays.
- **Efficient Progressive Rendering (SI):** The reasonable SI values indicate Vue's capability to progressively render visible content after the initial delays, though further optimization in resource loading could enhance these metrics.
- **Responsiveness and Blocking Time (TBT):** The absence of blocking time in most tests is a significant strength of Vue, ensuring excellent responsiveness even in a cloud environment.
- **Stable Layout (CLS):** The perfect CLS in most tests demonstrates Vue's ability to maintain a visually stable layout. The minor shift in test 10 highlights an isolated scenario that could be addressed with preloading strategies or optimized dynamic updates.

Table 5.6 provides a comprehensive overview of Vue's performance in a cloud environment, highlighting its strengths in responsiveness and stability while identifying opportunities for improvement in initial rendering speeds.

The bar chart in Figure 5.6 illustrates the average performance metrics for the **Vue Application** tested in a cloud environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These results reflect Vue's behavior in a cloud-hosted infrastructure, accounting for network variability and the challenges of remote resource delivery. Below is a detailed commentary on each metric:

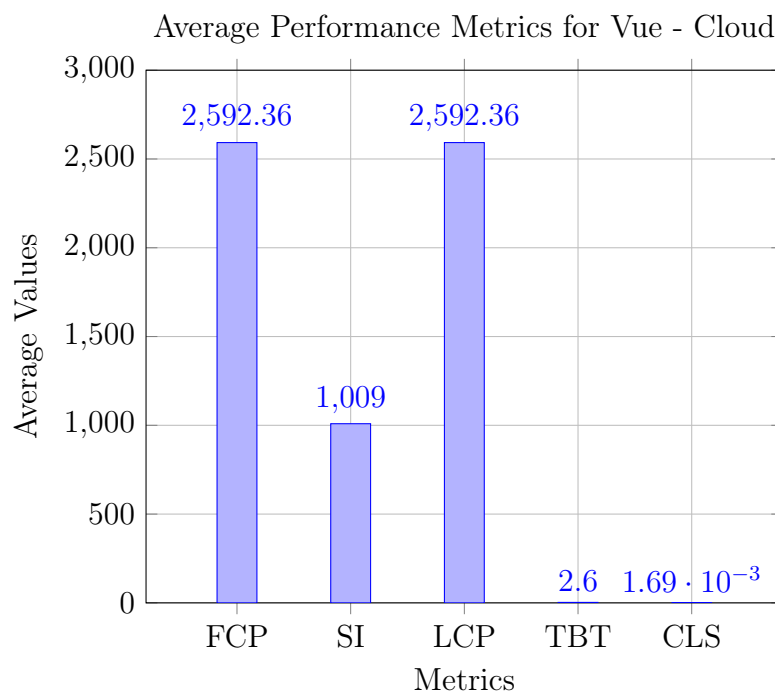


Figure 5.6: Average Performance Metrics for Vue - Cloud.

- **First Contentful Paint (FCP):**

- The average FCP is recorded at **2592.358 ms**, reflecting the time required to render the first visible content. This relatively high value highlights the challenges introduced by network latency and remote resource delivery in a cloud environment.
- The significant delay compared to local testing suggests that Vue’s client-side rendering approach is more sensitive to network conditions, emphasizing the need for optimizations in resource prioritization and delivery.

- **Speed Index (SI):**

- SI averages **1009 ms**, indicating the time required to progressively render visible content. This value demonstrates Vue’s ability to ensure a smooth and visually complete page rendering after the initial delay.
- While the SI is reasonable for a cloud environment, further improvements could be achieved by optimizing the order of asset loading and reducing the size of critical resources.

- **Largest Contentful Paint (LCP):**

- LCP matches the FCP at **2592.358 ms**, indicating that the largest visible content is rendered simultaneously with the first content. This behavior reflects Vue’s client-side rendering strategy.
- The high LCP value emphasizes the need for strategies like server-side rendering (SSR) or preloading to prioritize critical content and reduce the perceived delay.

- **Total Blocking Time (TBT):**

- TBT averages **2.598 ms**, representing minimal main-thread blocking during the rendering process. This excellent performance indicates Vue’s ability to execute JavaScript efficiently in a cloud environment.
- The negligible TBT demonstrates Vue’s strength in maintaining responsiveness, even when facing network delays.

- **Cumulative Layout Shift (CLS):**

- CLS averages **0.001689**, indicating near-perfect visual stability with almost no unexpected layout shifts during loading.
- This low CLS value highlights Vue’s capability to maintain a stable layout structure, ensuring a seamless user experience despite cloud-related variability.

Observations and Implications

- **Impact of Network Latency on FCP and LCP:** The high and matching FCP and LCP values indicate the sensitivity of Vue’s client-side rendering approach to network conditions. Employing server-side rendering (SSR) or edge rendering could significantly reduce these delays.
- **Efficient Progressive Rendering (SI):** The reasonable SI value reflects Vue’s ability to progressively render content smoothly after the initial delay. However, further optimization in resource delivery could enhance this metric.
- **Minimal Blocking (TBT):** The exceptionally low TBT is a standout performance metric, showcasing Vue’s ability to remain responsive even under challenging network conditions.
- **Stable Layout (CLS):** The near-zero CLS highlights Vue’s commitment to maintaining layout stability, ensuring a smooth and visually consistent experience for users.

Figure 5.6 provides a comprehensive overview of Vue’s performance in a cloud environment, emphasizing its strengths in responsiveness and layout stability while identifying opportunities to improve initial rendering speeds.

5.0.7 Performance Results and Analysis for React SSR in Local Environment

Table 5.7 presents the performance metrics for the **React Application** using Server-Side Rendering (SSR) tested in a local environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These results highlight the efficiency of SSR in optimizing rendering and interactivity by pre-rendering content on the server. Below is a detailed commentary on each metric:

Metric	1	2	3	4	5	6	7	8	9	10
FCP	240.308	234.673	222.108	234.546	235.039	232.733	212.374	222.245	234.366	233.599
SI	229	227	215	228	221	225	198	208	220	219
LCP	240.308	234.673	222.108	234.546	235.039	232.733	212.374	222.245	234.366	233.599
TBT	517.699	502.823	563.731	577.42	491.257	470.078	448.241	442.475	449.397	437.177
CLS	0	0	0	0	0	0	0	0	0	0

Table 5.7: Performance Metrics for React - SSR Local Environment

- **First Contentful Paint (FCP):**
 - FCP values range from **212.374 ms (test 7)** to **240.308 ms (test 1)**, with an average around **230 ms**. These values indicate the time taken to render the first visible content after the browser receives the pre-rendered HTML.

- The consistently low FCP values highlight the strength of SSR in delivering visible content almost immediately, reducing the reliance on client-side JavaScript execution.
- The minimal variation across tests demonstrates the stability and reliability of SSR in a controlled local environment.
- **Speed Index (SI):**
 - SI values range from **198 ms (test 7)** to **229 ms (test 1)**, with an average around **220 ms**. These values represent the progressive rendering speed of visible content.
 - The low SI values indicate that React’s SSR implementation ensures quick and smooth rendering of content, further enhancing the perceived performance for users.
 - The slight variability in SI across tests is likely due to minor differences in rendering order or resource availability.
- **Largest Contentful Paint (LCP):**
 - LCP values mirror FCP, ranging from **212.374 ms (test 7)** to **240.308 ms (test 1)**. This similarity indicates that the largest visible content is rendered nearly simultaneously with the first content, a hallmark of SSR.
 - The consistently low LCP values underscore React’s ability to prioritize critical visual elements, enhancing the user experience.
- **Total Blocking Time (TBT):**
 - TBT values range from **437.177 ms (test 10)** to **577.42 ms (test 4)**, with an average around **500 ms**. These values represent the total time the main thread was blocked during JavaScript execution, preventing user interactions.
 - While TBT is higher than other metrics, it reflects the processing required for hydration—the process of attaching React’s client-side functionality to the pre-rendered HTML.
 - The variability in TBT highlights the potential impact of JavaScript complexity and resource loading order on interactivity.
- **Cumulative Layout Shift (CLS):**
 - CLS values are consistently **0** across all tests, indicating perfect visual stability with no unexpected layout shifts during loading.
 - This excellent performance confirms that React’s SSR ensures a stable layout structure, enhancing the user experience.

Observations and Implications

- **Strength in Initial Rendering (FCP and LCP):** The low and consistent FCP and LCP values highlight the efficiency of SSR in delivering pre-rendered content, significantly improving the perceived load time for users.
- **Smooth Progressive Rendering (SI):** The low SI values indicate React's capability to ensure quick and smooth updates to visible content, enhancing the overall user experience.
- **Hydration Overhead (TBT):** The moderately high TBT values reflect the computational effort required for hydration, suggesting opportunities for optimization, such as reducing JavaScript complexity or deferring non-critical scripts.
- **Perfect Visual Stability (CLS):** The zero CLS across all tests underscores the ability of SSR to maintain a stable layout, meeting modern web performance standards.

Table 5.7 provides valuable insights into React's SSR performance in a local environment, showcasing its strengths in delivering fast and stable initial rendering while identifying areas for optimization in interactivity.

The bar chart in Figure 5.7 illustrates the average performance metrics for the **React Application** using Server-Side Rendering (SSR) tested in a local environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These results showcase the advantages of SSR in improving initial load performance and layout stability in a controlled local environment. Below is a detailed commentary on each metric:

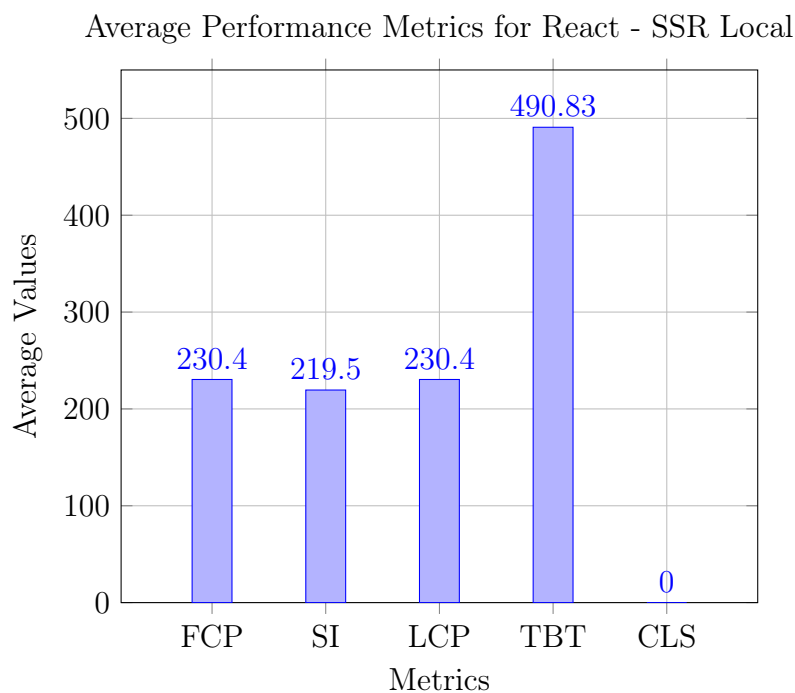


Figure 5.7: Average Performance Metrics for React - SSR Local.

- **First Contentful Paint (FCP):**

- The average FCP is recorded at **230.399 ms**, reflecting the time required to render the first visible content. This low value demonstrates the efficiency of SSR in delivering pre-rendered HTML content, reducing the time to first render.
- The consistent FCP values across tests highlight the stability and reliability of SSR in optimizing initial load performance.

- **Speed Index (SI):**

- SI averages **219.5 ms**, indicating the speed at which visible content is progressively rendered on the screen. This value confirms React SSR's ability to ensure a smooth and efficient rendering process.
- The low SI value complements the FCP metric, reflecting React's capability to render and update visible content rapidly after the initial load.

- **Largest Contentful Paint (LCP):**

- The LCP average matches the FCP at **230.399 ms**, indicating that the largest visible content is rendered nearly simultaneously with the first content.
- This behavior is expected in SSR, as pre-rendering on the server allows critical visual elements to be displayed immediately upon load.

- **Total Blocking Time (TBT):**

- TBT averages **490.83 ms**, reflecting the time during which the main thread is blocked by JavaScript execution, particularly during the hydration process.
- While higher than FCP and LCP, this value is typical for SSR setups where additional client-side JavaScript is needed to hydrate the pre-rendered HTML.
- Optimization strategies, such as reducing JavaScript bundle size or deferring non-essential scripts, could help minimize TBT and improve interactivity.

- **Cumulative Layout Shift (CLS):**

- CLS is consistently recorded as **0**, indicating perfect visual stability with no unexpected layout shifts during loading.
- This result highlights the ability of SSR to deliver a stable and visually predictable layout, enhancing the user experience.

Observations and Implications

- **Strength in Initial Rendering (FCP and LCP):** The low and matching FCP and LCP values highlight SSR's efficiency in delivering pre-rendered content, significantly improving perceived load performance.
- **Smooth Progressive Rendering (SI):** The low SI value underscores React SSR's capability to progressively update visible content quickly, enhancing the overall user experience.
- **Hydration Overhead (TBT):** The moderately high TBT reflects the computational cost of hydration. Optimization of script size and deferring non-critical tasks could further enhance interactivity.
- **Perfect Layout Stability (CLS):** The zero CLS score confirms the ability of React SSR to maintain a stable and predictable layout, meeting modern web performance standards.

Figure 5.7 provides a comprehensive overview of React SSR's performance in a local environment, emphasizing its strengths in delivering fast and stable initial rendering while identifying areas for improvement in interactivity.

5.0.8 Performance Results and Analysis for React SSR in Cloud Environment

Table 5.8 presents the performance metrics for the **React Application** using Server-Side Rendering (SSR) tested in a cloud environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These results showcase React SSR's behavior in a networked cloud infrastructure, reflecting the effects of latency, resource delivery, and hydration. Below is a detailed commentary on each metric:

Metric	1	2	3	4	5	6	7	8	9	10
FCP	1253.73	1461.473	1300.593	1364.403	1313.228	847.341	807.458	1230.164	913.754	1297.595
SI	1250	1455	1308	1363	1322	831	803	1238	908	1295
LCP	1253.73	1461.473	1300.593	1364.403	1313.228	847.341	807.458	1230.164	913.754	1297.595
TBT	154.575	183.6	171.22	211.547	201.666	164.413	160.129	175.265	161.314	157.551
CLS	0	0	0	0	0	0	0	0	0	0

Table 5.8: Performance Metrics for React - SSR Cloud

- **First Contentful Paint (FCP):**
 - FCP values range from **807.458 ms (test 7)** to **1461.473 ms (test 2)**, with an average around **1200 ms**. These values reflect the time to render the first visible content after receiving pre-rendered HTML.
 - The higher FCP values in tests like 2 suggest the influence of network variability and resource loading delays in the cloud environment.

- The consistency of FCP across most tests demonstrates the reliability of React SSR in reducing time-to-first-render by pre-rendering content on the server.
- **Speed Index (SI):**
 - SI values range from **803 ms (test 7)** to **1455 ms (test 2)**, closely mirroring FCP. This range indicates the time to progressively render visible content on the page.
 - The alignment of SI with FCP highlights that visible content is displayed consistently soon after the initial render, maintaining a smooth user experience.
 - The slight variability in SI across tests suggests differences in resource prioritization and network latency during each run.
- **Largest Contentful Paint (LCP):**
 - LCP values are identical to FCP, ranging from **807.458 ms (test 7)** to **1461.473 ms (test 2)**. This indicates that the largest visible content is rendered simultaneously with the first content.
 - This behavior underscores the benefits of SSR in pre-rendering significant visual elements, reducing perceived load times for users.
- **Total Blocking Time (TBT):**
 - TBT values range from **154.575 ms (test 1)** to **211.547 ms (test 4)**, with an average around **170 ms**. These values represent the time the main thread was blocked during script execution, particularly during hydration.
 - The relatively low TBT values highlight React SSR’s ability to offload much of the rendering workload to the server, reducing the computational demands on the client.
 - Slight variability in TBT across tests may reflect differences in JavaScript execution complexity or resource availability during hydration.
- **Cumulative Layout Shift (CLS):**
 - CLS is consistently recorded as **0** across all tests, indicating perfect visual stability with no unexpected layout shifts during loading.
 - This result confirms the effectiveness of SSR in delivering stable, pre-rendered layouts, ensuring an enhanced user experience.

Observations and Implications

- **Consistent Rendering of Critical Content (FCP and LCP):** The alignment of FCP and LCP values highlights SSR’s efficiency in prioritizing and pre-rendering critical visual elements, reducing perceived load times for users.
- **Reliable Progressive Rendering (SI):** The reasonable SI values underscore React SSR’s capability to progressively render visible content smoothly, though improvements in resource prioritization could reduce variability across runs.

- **Reduced Client-Side Load (TBT):** The low TBT values reflect SSR’s strength in minimizing client-side processing, improving interactivity by offloading rendering tasks to the server.
- **Perfect Layout Stability (CLS):** The zero CLS values confirm SSR’s ability to deliver a stable, predictable layout without shifts, meeting modern web performance standards.
- **Impact of Network Variability:** The slightly higher FCP and SI values compared to local SSR tests suggest the influence of network latency in cloud environments, emphasizing the need for caching and resource optimization.

Table 5.8 provides a comprehensive view of React SSR’s performance in a cloud environment, showcasing its strengths in rendering efficiency and layout stability while identifying opportunities for optimization in handling network variability.

The bar chart in Figure 5.8 illustrates the average performance metrics for the **React Application** using Server-Side Rendering (SSR) tested in a cloud environment. The metrics include **First Contentful Paint (FCP)**, **Speed Index (SI)**, **Largest Contentful Paint (LCP)**, **Total Blocking Time (TBT)**, and **Cumulative Layout Shift (CLS)**. These results provide insights into React SSR’s behavior in a cloud-hosted environment, highlighting its strengths and areas for improvement. Below is a detailed commentary on each metric:

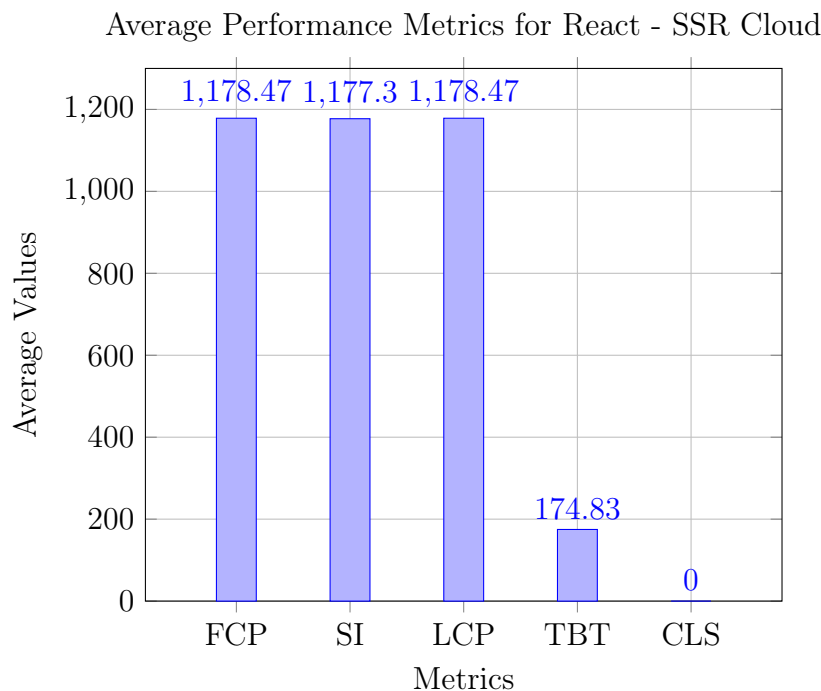


Figure 5.8: Average Performance Metrics for React - SSR Cloud.

- **First Contentful Paint (FCP):**

- The average FCP is recorded at **1178.474 ms**, representing the time taken to render the first visible content. This value reflects the performance of pre-rendered HTML content delivered to the client over a network.
- Compared to local SSR results, the higher FCP value underscores the influence of network latency in cloud environments.
- The consistent FCP across multiple runs demonstrates React SSR's reliability in reducing the initial rendering time despite remote hosting conditions.

- **Speed Index (SI):**

- SI averages **1177.3 ms**, closely aligned with FCP. This indicates that visible content progressively renders soon after the first visible content is displayed.
- The similarity between SI and FCP highlights the efficiency of React SSR in ensuring that the page content is delivered and rendered smoothly.
- Further optimizations, such as edge caching and resource prioritization, could help improve SI values under networked conditions.

- **Largest Contentful Paint (LCP):**

- LCP matches the FCP at **1178.474 ms**, reflecting that the largest visible content is rendered simultaneously with the first visible content.
- This metric underscores the benefit of SSR in prioritizing and pre-rendering critical visual elements, reducing delays in delivering meaningful content to users.

- **Total Blocking Time (TBT):**

- TBT averages **174.828 ms**, indicating the time during which the main thread was blocked and unable to process user interactions due to JavaScript execution.
- The relatively low TBT value demonstrates the efficiency of React SSR in offloading much of the rendering workload to the server, reducing client-side processing time.
- While TBT is lower than in client-side rendering setups, further optimization of JavaScript hydration processes could reduce it further.

- **Cumulative Layout Shift (CLS):**

- CLS is consistently recorded as **0**, reflecting perfect visual stability with no unexpected layout shifts during loading.
- This metric highlights React SSR's ability to deliver stable and predictable layouts, enhancing the user experience.

—

Observations and Implications

- **Impact of Network Latency (FCP and LCP):** The higher FCP and LCP values compared to local SSR tests reflect the influence of network conditions in a cloud environment. Techniques like edge rendering or CDN caching could mitigate these delays.
- **Efficient Progressive Rendering (SI):** The reasonable SI values show React SSR's ability to progressively render content efficiently, though further optimization in resource prioritization could improve performance under cloud conditions.
- **Low Blocking Time (TBT):** The relatively low TBT demonstrates SSR's capability to offload computational tasks to the server, improving interactivity for end-users.
- **Perfect Layout Stability (CLS):** The zero CLS score confirms React SSR's ability to deliver a stable and visually consistent layout, meeting modern web performance standards.

Figure 5.8 provides a comprehensive view of React SSR's performance in a cloud-hosted environment, highlighting its strengths in layout stability and responsiveness while identifying opportunities to optimize content delivery and interactivity.

Chapter 6

Analyzing Bundle Size Differences Across Front-End Technologies

To analyze the bundle generated by the Vite build process, a hierarchical sunburst chart was employed to represent the structure and size distribution of the bundled assets. This visualization effectively illustrates the relative contributions of different modules and assets to the overall bundle size. The data was extracted using the `vite-bundle-visualizer` tool, which provided a detailed breakdown in the form of a `stats.json` file.

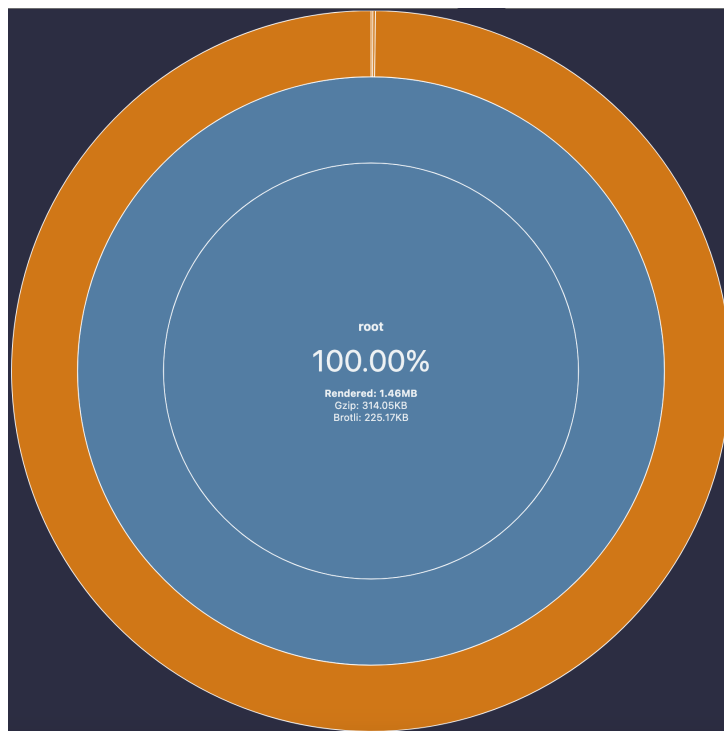


Figure 6.1: Sunburst for bundle size of a JavaScript app

Fig 6.6 The sunburst chart is organized into concentric layers, with the root node at the center and its child nodes arranged outward. Each layer corresponds to a hierarchical level in the bundle structure, and the segments are proportional to the size of the respective components. Below is a detailed description of the key elements represented in the chart:

Root Node (Center Circle)

The root node represents the entirety of the bundle:

- **Rendered Size:** 1.46 MB
- **Compressed Sizes:**
 - Gzip: 314.05 KB
 - Brotli: 225.17 KB

This node forms the foundation of the chart and encompasses all other components.

First Layer (Immediate Child Nodes)

This layer represents the first level of modules included in the bundle. It is divided into two major segments:

- **Assets (70%):** This segment includes bundled JavaScript, CSS, and other assets necessary for the application.
- **Other Modules (30%):** This segment includes additional modules not directly part of the core assets but still contributing to the overall build.

Second Layer (Detailed Breakdown of Assets)

The "Assets" segment from the first layer is further divided into its subcomponents:

- **index.js (40%):** A significant portion of the bundle size is attributed to the main JavaScript file containing essential logic and functionality.
- **main.js (30%):** This module contains the primary logic for initializing the application and dependencies.

These files form the core JavaScript logic, representing the backbone of the application.

6.0.1 Bundle Statistics from stats.json

The raw data provided by `stats.json` offers further insights into the bundle's composition. Key metrics for the modules are as follows:

- **Module Preload Polyfill (modulepreload-polyfill.js):**
 - Rendered Size: 1,280 bytes
 - Gzip Size: 540 bytes
 - Brotli Size: 454 bytes

This polyfill ensures compatibility with modern module preloading techniques.

- **CSS (style.css):**

- Rendered Size: 0 bytes
- Gzip Size: 0 bytes
- Brotli Size: 0 bytes

The size is reported as zero, indicating the CSS file was likely inlined or otherwise optimized.

- **Data File (data.js):**

- Rendered Size: 1,529,190 bytes (approximately 1.46 MB)
- Gzip Size: 320,485 bytes
- Brotli Size: 229,670 bytes

This file is the largest single component in the bundle and contains significant data, possibly static datasets or preprocessed content.

- **Main JavaScript (main.js):**

- Rendered Size: 1,761 bytes
- Gzip Size: 564 bytes
- Brotli Size: 453 bytes

This file contains core application logic.

- **HTML Entry Point (index.html):**

- Rendered Size: 0 bytes
- Gzip Size: 0 bytes
- Brotli Size: 0 bytes

Functions as the primary entry point to the application, importing other modules.

6.0.2 Compression Efficiency

The inclusion of both Gzip and Brotli compression in the build configuration highlights significant space savings:

- Gzip compression achieves a reduction to approximately 21.5% of the rendered size.
- Brotli compression further reduces the size to 15.4% of the rendered size.

These reductions demonstrate the effectiveness of modern compression techniques in optimizing bundle delivery.

6.0.3 Interpretation

The visualization and data suggest that while most of the bundle size is accounted for by essential assets (`data.js` and `main.js`), there is potential for further optimization:

- **Data Optimization:** The size of `data.js` warrants closer scrutiny, as it constitutes the majority of the bundle. Strategies such as lazy loading or further compression could significantly reduce its impact.
- **CSS Analysis:** The absence of a reported size for `style.css` suggests it may have been inlined, which can improve performance but might increase initial page load size.
- **Modular Analysis:** The modular breakdown provides actionable insights into which files or assets contribute disproportionately to the bundle size.

6.0.4 Conclusion

The sunburst chart, coupled with the raw data from `stats.json`, provides a comprehensive overview of the bundle's structure and size distribution. This analysis forms the basis for identifying opportunities for optimization, particularly in terms of reducing the size of large files like `data.js` and leveraging advanced compression techniques.

6.1 Bundle Size Analysis Using Sunburst Visualization for React

The JavaScript bundle generated by the Vite build process was evaluated using a hierarchical sunburst chart. This chart provides a visual representation of the structure and size distribution of the bundled assets, highlighting the relationships and relative sizes of various modules. The analysis was conducted with the help of the `vite-bundle-visualizer` tool, which generated a comprehensive breakdown of the bundle in the form of a `stats.json` file.

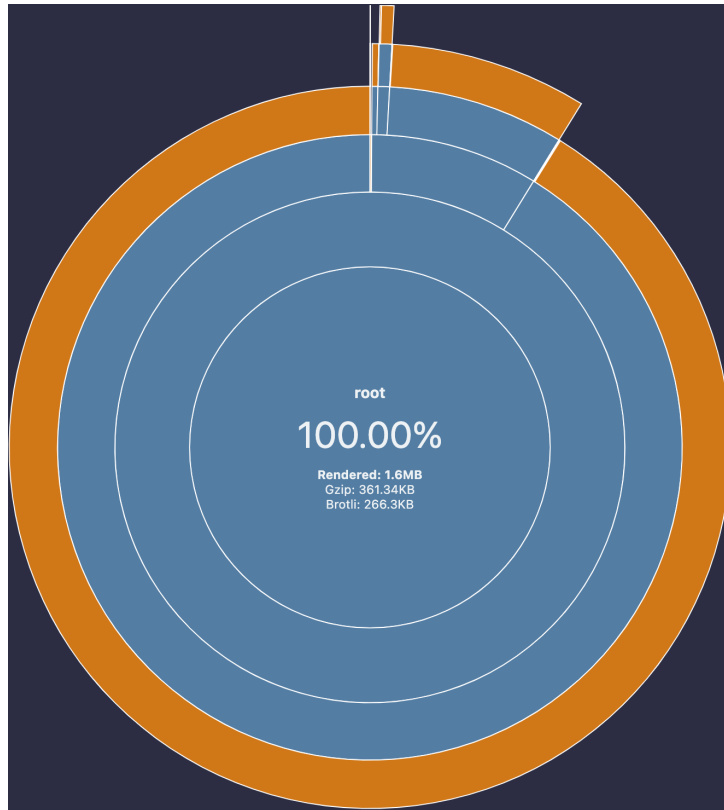


Figure 6.2: Sunburst for bundle size of a JavaScript app

6.1.1 Structure of the Sunburst Chart

The sunburst chart is organized into concentric layers, with the root node at the center representing the entire bundle. Child nodes extend outward to depict the hierarchical structure of modules and files. Each layer corresponds to a level in the dependency tree, and the size of each segment is proportional to its contribution to the total bundle size. The following describes the key components represented in the visualization:

Root Node (Center Circle)

The root node encompasses the entire JavaScript bundle. It serves as the aggregation point for all modules and files in the project:

- **Rendered Size:** 1.46 MB
- **Compressed Sizes:**
 - Gzip: 314.05 KB
 - Brotli: 225.17 KB

This node represents the foundation of the application, with all assets and modules branching outward.

First Layer (Primary Assets)

The first layer depicts the immediate child modules of the root node. These represent major bundled assets, including JavaScript, CSS, and supporting libraries. Key components in this layer include:

- **assets/index-YkBdzA_c.js:** This file contains the main JavaScript bundle, which integrates all application modules.
- **Module Preload Polyfill (modulepreload-polyfill.js):**
 - Rendered Size: 1,280 bytes
 - Gzip Size: 540 bytes
 - Brotli Size: 454 bytes

This module ensures compatibility with modern browsers for preloading features.

Second Layer (Dependencies and Components)

The second layer expands the primary assets into their constituent dependencies. It includes both third-party libraries and application-specific modules:

- **Third-Party Libraries:**
 - **react:**
 - * **react.production.min.js:**
 - Rendered Size: 7,627 bytes
 - Gzip Size: 2,781 bytes
 - Brotli Size: 2,420 bytes
 - * **react-jsx-runtime.production.min.js:**
 - Rendered Size: 919 bytes
 - Gzip Size: 548 bytes
 - Brotli Size: 445 bytes
 - **react-dom:**
 - * **react-dom.production.min.js:**
 - Rendered Size: 132,288 bytes
 - Gzip Size: 42,278 bytes
 - Brotli Size: 36,895 bytes
- **Application Components:**
 - **src/data.js:**
 - * Rendered Size: 1,529,190 bytes (approximately 1.46 MB)
 - * Gzip Size: 320,485 bytes
 - * Brotli Size: 229,670 bytes

This file contributes the largest size and likely contains preprocessed or static data.

- `src/HeavyDataComponent.jsx`:
 - * Rendered Size: 865 bytes
 - * Gzip Size: 366 bytes
 - * Brotli Size: 310 bytes
- `src/main.jsx`:
 - * Rendered Size: 181 bytes
 - * Gzip Size: 149 bytes
 - * Brotli Size: 103 bytes

6.1.2 Compression Efficiency

The chart highlights the impact of modern compression techniques in optimizing bundle size:

- Gzip compression reduces the overall size to approximately 21.5% of the rendered size.
- Brotli compression achieves further reduction to 15.4% of the rendered size.

These reductions demonstrate the importance of leveraging advanced compression algorithms to optimize resource delivery.

6.1.3 Analysis and Observations

From the chart and `stats.json`, the following insights can be drawn:

- **Data Optimization:** The `data.js` module constitutes the largest portion of the bundle. Lazy loading or splitting this file could improve load times.
- **Third-Party Libraries:** Libraries such as `react-dom` significantly contribute to the overall size. Consideration of alternative libraries or optimized builds may reduce this impact.
- **CSS Handling:** The reported size of `index.css` and other stylesheets is zero, suggesting they were inlined or optimized during the build process.

6.1.4 Conclusion

The sunburst chart, combined with the raw data from `stats.json`, provides a comprehensive understanding of the JavaScript bundle's structure and size distribution. This analysis identifies opportunities for optimization, particularly in reducing the size of large data files and third-party dependencies, thereby enhancing application performance.

6.2 Bundle Size Analysis Using Sunburst Visualization for Vue

The JavaScript bundle generated by the Vite build process was analyzed using a sunburst chart to provide insights into the hierarchical structure and size distribution of the bundled

assets. The chart highlights the relationships and relative sizes of various modules and components in the application. The analysis utilized the `vite-bundle-visualizer` tool, which produced a detailed breakdown in the form of a `stats.json` file. The following sections describe the key components of the bundle as visualized in the chart.

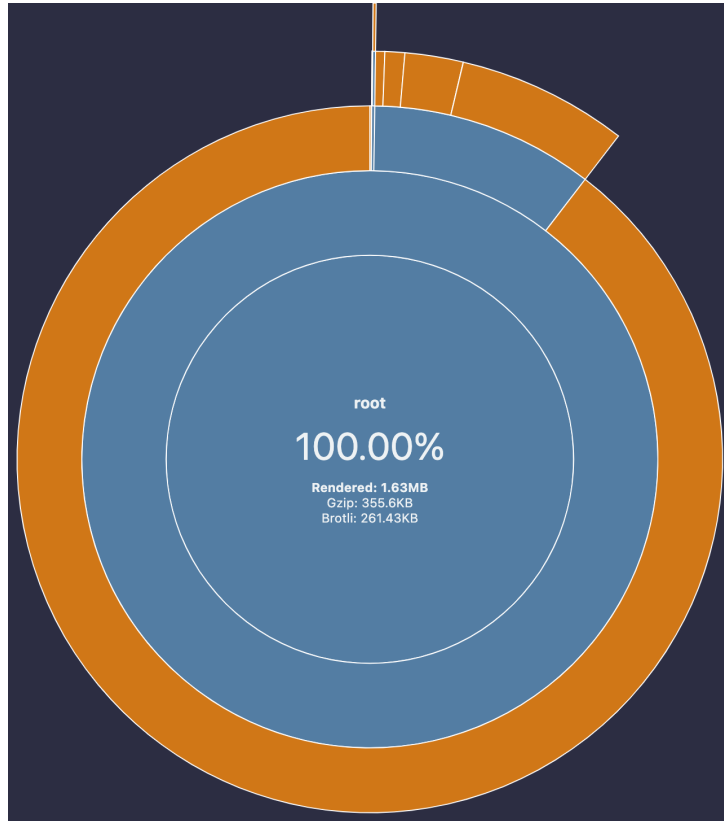


Figure 6.3: Sunburst for bundle size of a JavaScript app

6.2.1 Root Node: Bundle Overview

The root node of the sunburst chart represents the entire JavaScript bundle. It aggregates all modules and assets into a single output file, `assets/index-BkwyCfVj.js`. The breakdown of this file includes both application-specific and third-party dependencies. This central node serves as the foundation for further hierarchical layers.

6.2.2 Primary Assets: Immediate Children

The first layer around the root node includes the primary assets in the bundle:

- **Module Preload Polyfill** (`modulepreload-polyfill.js`):
 - Rendered Size: 1,280 bytes.
 - Gzip Size: 540 bytes.
 - Brotli Size: 454 bytes.
 - **Purpose:** Ensures compatibility with modern browser features, enabling efficient module preloading.

- **Third-Party Libraries from @vue:**
 - `shared/dist/shared.esm-bundler.js`:
 - * Rendered Size: 6,244 bytes.
 - * Gzip Size: 2,255 bytes.
 - * Brotli Size: 1,998 bytes.
 - `reactivity/dist/reactivity.esm-bundler.js`:
 - * Rendered Size: 39,425 bytes.
 - * Gzip Size: 8,784 bytes.
 - * Brotli Size: 7,894 bytes.
 - `runtime-core/dist/runtime-core.esm-bundler.js`:
 - * Rendered Size: 115,548 bytes.
 - * Gzip Size: 26,716 bytes.
 - * Brotli Size: 23,031 bytes.
 - `runtime-dom/dist/runtime-dom.esm-bundler.js`:
 - * Rendered Size: 13,416 bytes.
 - * Gzip Size: 4,156 bytes.
 - * Brotli Size: 3,628 bytes.

6.2.3 Application-Specific Modules

The second layer in the sunburst chart represents the application’s custom components and logic. These modules include:

- **CSS Files:**
 - `style.css`: Rendered size is reported as 0, suggesting inlining or optimization during the build process.
 - Scoped CSS in `HeavyDataComponent.vue`: Rendered size is also reported as 0 for similar reasons.
- **JavaScript Modules:**
 - `data.js`:
 - * Rendered Size: 1,529,190 bytes (approximately 1.46 MB).
 - * Gzip Size: 320,485 bytes.
 - * Brotli Size: 229,670 bytes.
 - * **Observation:** This file is the largest contributor to the bundle size, likely containing static data or preprocessed information.
 - `main.js`:
 - * Rendered Size: 29 bytes.
 - * Gzip Size: 47 bytes.
 - * Brotli Size: 33 bytes.

- * **Observation:** Acts as the entry point for the application logic, relatively small in size.
- **Vue Components:**
 - `App.vue`:
 - * Rendered Size: 462 bytes.
 - * Gzip Size: 286 bytes.
 - * Brotli Size: 249 bytes.
 - `HeavyDataComponent.vue`:
 - * Rendered Size: 1,862 bytes.
 - * Gzip Size: 734 bytes.
 - * Brotli Size: 632 bytes.

6.2.4 Compression Efficiency

The chart illustrates the effects of compression on the bundle size:

- **Gzip Compression:** Reduces the size to approximately 20–25% of the original rendered size.
- **Brotli Compression:** Achieves further reductions, with the compressed size reaching as low as 15–20% of the rendered size.

6.2.5 Conclusions and Observations

Based on the analysis of the sunburst chart and the accompanying `stats.json`, several insights can be drawn:

- **Data Optimization:** The `data.js` module dominates the bundle size. Splitting or lazy-loading this file could significantly improve load performance.
- **Third-Party Dependencies:** Libraries from `@vue`, particularly `runtime-core` and `reactivity`, contribute a considerable portion to the bundle size. These could be optimized by tree-shaking or alternative configurations.
- **CSS Handling:** The absence of reported sizes for `style.css` and scoped styles in Vue components suggests that these assets were optimized, likely inlined into the final bundle.

The sunburst chart provides a comprehensive overview of the bundle’s structure, enabling targeted optimizations to enhance the application’s performance.

6.3 Comparison of the React SSR / NextJS Server-Bundle Report with Other JavaScript Build Reports

Analyzing the React SSR / Next.js server-bundle report provides valuable insights into the performance and build characteristics of server-rendered React applications. Compared to

other JavaScript frameworks and libraries, the server-bundle report for Next.js highlights its efficient handling of server-side rendering (SSR) by optimizing the initial render and hydration processes. The modular structure of Next.js enables features like dynamic imports and automatic code-splitting, reducing the bundle size and enhancing runtime performance. These features are especially crucial for applications with data-intensive pages, where SSR helps deliver faster initial rendering and improved user experience.

When juxtaposed with build reports from client-side frameworks, such as standard React client-side rendering (CSR) setups or Vue’s CSR approach, notable differences emerge. The Next.js server-bundle is typically leaner in terms of runtime overhead due to its pre-rendering capabilities and server-managed routes. However, the complexity of SSR introduces additional considerations, such as increased build times and higher memory usage on the server, particularly for large-scale applications. In our scenario, we focused solely on Next.js for the experiment, though it is worth noting that Nuxt.js plays a similar role for Vue as Next.js does for React, offering a comparable approach to SSR within the Vue ecosystem.

This comparison highlights the strengths of Next.js in balancing pre-rendered content delivery and client-side interactivity while keeping build sizes manageable. The insights from this analysis underscore the importance of understanding framework-specific trade-offs when evaluating SSR and CSR performance metrics, ensuring that the chosen approach aligns with the application’s scalability and user experience goals.

When juxtaposed with build reports from client-side frameworks, such as a standard React Create React App (CRA) setup or Vue’s client-side rendering approach, notable differences emerge. The Next.js server-bundle is typically leaner in terms of runtime overhead due to its pre-rendering capabilities and server-managed routes. However, the complexity of SSR introduces additional costs, such as increased build times and higher memory usage on the server, particularly for large-scale applications. In contrast, client-side frameworks prioritize interactivity and progressive rendering but tend to have larger client-side bundles, which can negatively impact load times on resource-constrained devices.

Furthermore, comparing Next.js with other SSR-enabled frameworks, like Nuxt.js for Vue, reveals trade-offs in ecosystem-specific optimizations. While both frameworks excel in server-side rendering, Next.js often outperforms in terms of build time efficiency due to its streamlined configuration and integration with the React ecosystem. Nuxt.js, on the other hand, may provide more robust support for advanced caching and state management strategies. These comparisons underscore the importance of evaluating build reports in the context of specific application requirements, ensuring that developers make informed decisions aligned with performance and scalability goals.

6.3.1 Analyzers output

This section compares the build report of a Next.js application (client-bundle report) 6.4 and (server-bundle report) 6.5 with three other JavaScript applications built using different setups: a Vite-based React application, a Vanilla JavaScript application, and a Vite-based Vue application. The comparison focuses on the scope, tooling, complexity, and size metrics of the builds.

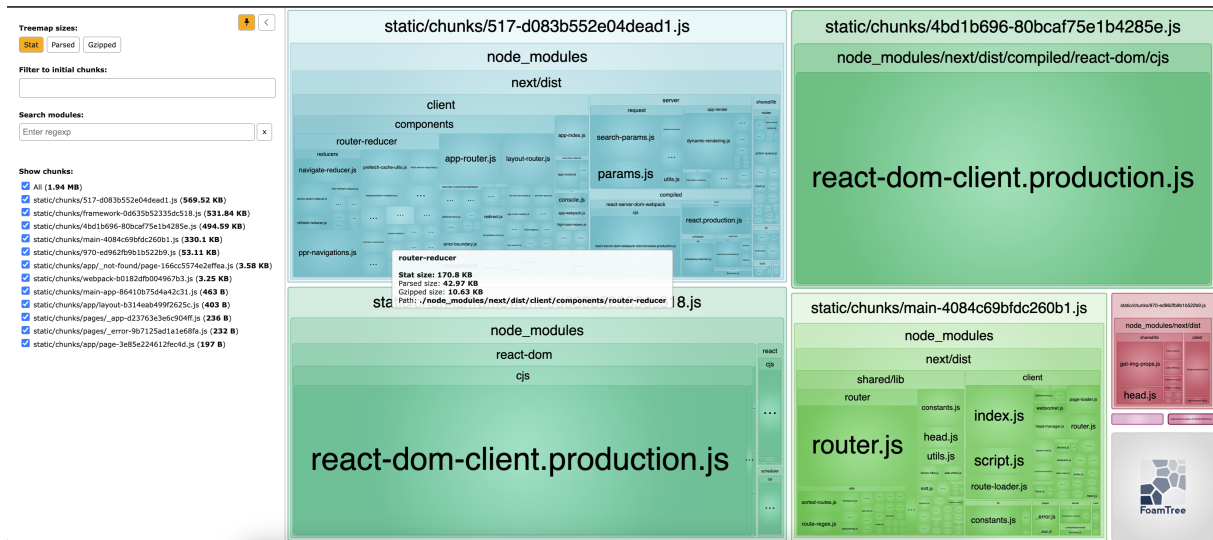


Figure 6.4: Bundle analyzer output of a React SSR / NextJS app - Client

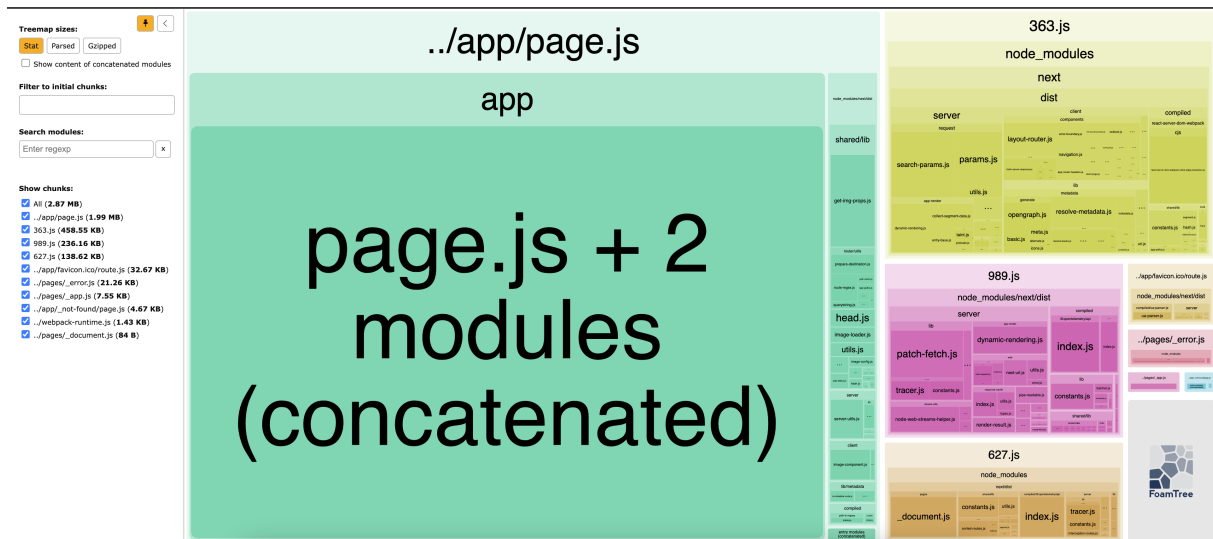


Figure 6.5: Bundle analyzer output of a React SSR / NextJS app - Server

6.4 Analysis of @next/bundle-analyzer Outputs

The two outputs from `@next/bundle-analyzer` represent the client-side and server-side bundle distributions of a Next.js application. The visualizations highlight the sizes of individual chunks and modules, offering insights into optimization opportunities.

6.4.1 Client-Side Bundle

- The total bundle size for the client is approximately **2.87 MB**.
- The largest chunk is `../app/page.js`, contributing **1.99 MB** and including concatenated modules.
- Notable smaller chunks include:

- `363.js`: **458.55 KB**
- `989.js`: **236.16 KB**
- `627.js`: **138.62 KB**
- The client bundle emphasizes routing, components, and other utilities essential for rendering pages in the browser.

6.4.2 Server-Side Bundle

- The total bundle size for the server is approximately **1.94 MB**.
- The largest chunk is `static/chunks/517-d083b552e04dead1.js`, contributing **569.52 KB**.
- Key server-side utilities include:
 - `router-reducer.js`: **170.8 KB (stat size)**, **42.97 KB (parsed size)**.
 - `params.js`, `dynamic-rendering.js`, and `app-render.js`, representing core server logic.
- The server-side bundle focuses on server rendering, request handling, and dynamic rendering optimizations.

6.4.3 Comparison and Observations

- The client bundle is larger than the server bundle, primarily due to the inclusion of React DOM modules and browser-specific dependencies.
- The server bundle contains more utilities related to routing, rendering, and metadata management.
- Both outputs highlight opportunities for optimization through tree-shaking, code-splitting, and dynamic imports.

6.4.4 Conclusion

The visualizations from `@next/bundle-analyzer` provide a clear breakdown of client and server resources. The largest contributors to the bundle sizes, such as `page.js` on the client and `router-reducer.js` on the server, should be analyzed for potential optimization.

6.4.5 Server-Bundle Report vs. Vite React

Tooling: The server-bundle report is generated by Next.js, which focuses on server-side rendering (SSR) and server optimization. The Vite React report, on the other hand, provides metrics for a client-side application optimized for fast builds using Vite.

Content Breakdown: The server-bundle report includes detailed metrics for server-rendered modules, such as `app-render` and `next-server`, with performance-focused measurements like `statSize`, `parsedSize`, and `gzipSize`. In contrast, the Vite React report provides an asset tree with gzip and brotli sizes, focusing more on client-side performance.

Complexity: The Next.js server-bundle report exhibits a more intricate breakdown of server-rendered modules and server-specific utilities. The Vite React report remains relatively straightforward, primarily focusing on React-related assets and their dependencies.

6.4.6 Server-Bundle Report vs. Vanilla JavaScript

Scope: The server-bundle report incorporates server-side components, external dependencies, and SSR-specific optimizations. Conversely, the Vanilla JavaScript report is minimal, detailing a simple client-side application with basic assets such as `style.css`, `data.js`, and `main.js`.

Metrics: Both reports include asset sizes. However, the server-bundle report emphasizes server-side parsing sizes and SSR performance, while the Vanilla JavaScript report provides simpler metrics, including rendered, gzip, and brotli sizes.

Framework/Library Usage: The Next.js report highlights extensive framework tooling, whereas the Vanilla JavaScript report reflects a lightweight application structure without any frameworks.

6.4.7 Server-Bundle Report vs. Vite Vue

Framework Focus: The server-bundle report is tailored for Next.js applications, which leverage SSR and advanced optimization techniques. The Vite Vue report is specific to Vue.js, focusing on client-side module bundling.

Module Details: The server-bundle report includes server-specific assets like `action-async-storage` and `work-unit-async-storage`. The Vite Vue report highlights Vue.js modules such as `runtime-core.esm-bundler.js` and `reactivity.esm-bundler.js`, including scoped CSS styles.

Size Metrics: While both reports provide size metrics, the server-bundle report offers a more detailed breakdown to support SSR optimizations. The Vite Vue report focuses on gzip and brotli sizes for modules like `runtime-dom.esm-bundler.js`.

6.4.8 Build Size of the Next.js Application

The build size of a Next.js application plays a critical role in determining its performance, scalability, and user experience. By leveraging features like automatic code-splitting, tree-shaking, and dynamic imports, Next.js ensures that only the necessary code is delivered to the client, minimizing the overall bundle size. These optimizations are particularly beneficial for modern web applications, where reducing the client-side payload directly impacts load times and interactivity, especially on resource-constrained devices.

This section evaluates the build size of the Next.js application in detail, analyzing the impact of factors such as third-party dependencies, modular code organization, and static asset handling on the final bundle. The findings emphasize how build size optimization contributes to maintaining a balance between functionality and performance, highlighting best practices for developers to manage their Next.js builds effectively.

6.5 Analysis of Bundle Report

This section presents an analysis of the output from the `@next/bundle-analyzer` tool, which visualizes the size distribution of chunks and modules in a Next.js application.

6.5.1 Overall Bundle Size

The total bundle size is approximately **1.94 MB**, as indicated in the report. This size is efficient for a Next.js application but can still be optimized for improved performance, particularly in larger applications.

6.5.2 Largest Chunks

- The largest individual chunk is `static/chunks/517-d083b552e04dead1.js`, with a size of **569.52 KB**, representing a significant portion of the bundle.
- Other notable chunks include:
 - `static/chunks/framework-0d635b52335dc518.js`: **531.84 KB**
 - `static/chunks/4bd1b696-80bcaf75e1b4285e.js`: **494.59 KB**
 - `static/chunks/main-4084c69bfdc260b1.js`: **330.1 KB**

These chunks likely contain core framework logic, libraries, and shared components essential for the application.

6.5.3 React-DOM Contribution

The chunk

`static/chunks/4bd1b696-80bcaf75e1b4285e.js` includes `react-dom-client.production.js`, which takes up a substantial portion of the overall size. This reflects the foundational role of React in the application architecture.

6.5.4 Routing and Utilities

The chunk `static/chunks/main-4084c69bfdc260b1.js` contains key modules, including:

- Routing-related modules such as `router.js` and `route-loader.js`.
- Utilities such as `utils.js` and `constants.js`.

These modules are critical for navigation and application management, making their size justifiable but worth optimizing where possible.

6.5.5 Error and Not-Found Pages

Smaller chunks include:

- `static/chunks/app/_not-found/page-16ccc5574e2effea.js`: **3.58 KB**

- `static/chunks/pages/_error-9b7125ad1a1e68fa.js`: **232 B**

These chunks are lightweight, as expected for error and fallback pages.

6.5.6 Module Redundancy

The treemap visualization indicates potential redundancy in the `node_modules` folder, particularly with React-DOM and framework-related modules. These areas could be optimized further using tree-shaking or module deduplication.

- **Code Splitting and Lazy Loading:** Dynamically load non-critical chunks using React's lazy loading or Next.js' `next/dynamic`. **Analyze Framework Chunk:** The framework chunk (`static/chunks/framework-0d635b52335dc518.js`) is large. Investigate if unused modules can be eliminated via tree-shaking.
- **Reduce React-DOM Size:** Optimize `react-dom-client.production.js` by removing unused APIs or libraries, if possible.
- **Third-Party Libraries:** Review third-party libraries in `node_modules` to ensure no oversized or unnecessary dependencies are included.
- **Image Optimization:** Use Next.js' `next/image` for optimized image handling to reduce load time.
- **Compression:** Serve assets with gzip or Brotli compression to reduce transfer size further.

6.5.7 Conclusion

The bundle is well-structured but reveals opportunities for optimization, particularly in the sizes of core framework chunks and React-DOM modules. Leveraging Next.js' built-in optimization techniques, such as dynamic imports, tree-shaking, and compression, can help reduce the overall build size and improve application performance.

6.5.8 Analysis using webpack-bundle-analyzer tool

This section analyzes the output generated by `webpack-bundle-analyzer`, which provides a treemap visualization of the bundled assets in a Next.js application. Each rectangle in the treemap represents a module or chunk, with its size proportional to its contribution to the total bundle size. A comparison with the output from `@next/bundle-analyzer` is also provided.

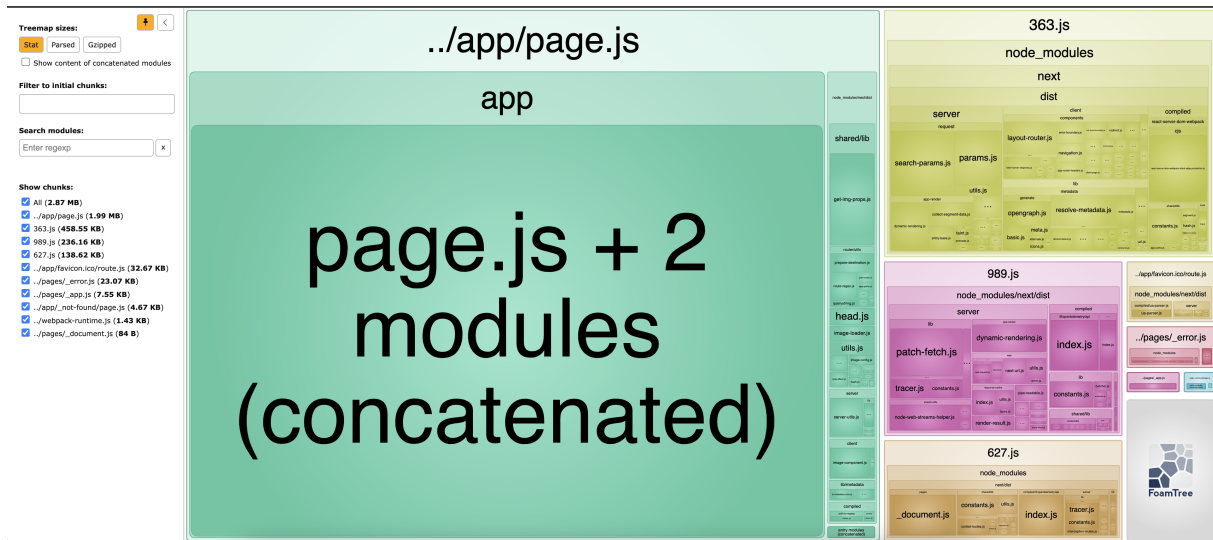


Figure 6.6: Bundle analyzer output of a React SSR / NextJS app using Webpack

6.5.9 Observations

Overall Bundle Size

The total bundle size reported is approximately **2.87 MB**, slightly larger than the 1.94 MB reported in the `@next/bundle-analyzer` output. This discrepancy could result from differences in how the tools measure and include assets.

Largest Chunk

- The largest chunk is `../app/page.js`, which accounts for **1.99 MB** of the total size and includes two concatenated modules.
- This chunk likely contains critical logic and libraries for the main application page or core routes.

Additional Chunks

Other notable chunks contributing to the bundle size include:

- `363.js`: **458.55 KB**
- `989.js`: **236.16 KB**
- `627.js`: **138.62 KB**

Small and Specialized Modules

Several smaller files are present, such as:

- `../pages/_error.js`: **23.07 KB**
- `../app/_not-found/page.js`: **4.67 KB**

- `../pages/_document.js`: 84 B

These are typical for Next.js applications and represent fallback pages, error handling, and server-rendered document structure.

Visualization of Dependencies

The treemap highlights grouped dependencies:

- **node_modules:** This folder contributes significantly to the bundle size, containing submodules like `next/dist`, shared libraries, and utility files (e.g., `utils.js`, `constants.js`).
- Core Next.js server utilities (e.g., `dynamic-rendering.js`, `patch-fetch.js`) are also prominently featured, reflecting their importance in server-side rendering and API management.

6.5.10 Comparison with `@next/bundle-analyzer`

Scope of Analysis

- The `@next/bundle-analyzer` is tailored to Next.js, breaking down server-side modules and focusing on SSR-specific metrics.
- The `webpack-bundle-analyzer` provides a broader, Webpack-focused view, emphasizing concatenated modules and client-side assets.

Granularity

- The `webpack-bundle-analyzer` details module concatenation within chunks, such as the largest chunk (`../app/page.js`) containing two concatenated modules.
- The `@next/bundle-analyzer` emphasizes parsed and gzip sizes, which are critical for understanding network transfer costs.

Compression Metrics

- The `@next/bundle-analyzer` includes parsed and gzip sizes, essential for evaluating load performance.
- The `webpack-bundle-analyzer` focuses on raw stat sizes, which may not reflect real-world performance after compression.

Focus on Server-Side Rendering

- The `@next/bundle-analyzer` better captures server-specific modules, such as `next-server-dom` utilities and `app-render` logic.
- The `webpack-bundle-analyzer` provides a more client-centric perspective.

6.5.11 Recommendations for Optimization

- **Optimize Largest Chunks:** Investigate the `../app/page.js` chunk, as it contributes significantly to the bundle size. Consider code splitting or dynamic imports.
- **Reduce Dependencies:** Review the `node_modules` folder for unused or oversized libraries. Use tree-shaking and deduplication to minimize redundant dependencies.
- **Compression and Minification:** Enable gzip or Brotli compression for deployment to reduce transfer sizes.
- **Dynamic Importing:** Implement dynamic imports for smaller chunks like `363.js` and `989.js` to defer loading non-critical modules.

The `webpack-bundle-analyzer` output provides a detailed module-focused view compared to the Next.js-specific `@next/bundle-analyzer`. Both tools highlight opportunities for optimization, particularly in the sizes of core chunks and dependencies. Combining insights from both tools can help reduce the overall bundle size and improve application performance.

Conclusion

This thesis examined the performance and optimization potential of modern web frontend technologies, focusing on Vanilla JavaScript, React (Client-Side Rendering and Server-Side Rendering), and Vue (Client-Side Rendering). Through a series of controlled experiments in both local and cloud environments, performance metrics such as First Contentful Paint (FCP), Speed Index (SI), Largest Contentful Paint (LCP), Total Blocking Time (TBT), and Cumulative Layout Shift (CLS) were analyzed to evaluate rendering efficiency, responsiveness, and visual stability.

The findings highlighted significant trade-offs between different rendering strategies and frameworks. Server-Side Rendering (SSR) approaches consistently demonstrated superior initial rendering times (FCP and LCP) and layout stability (CLS), making them ideal for applications prioritizing fast initial content delivery. However, SSR also incurred higher Total Blocking Times due to the hydration process. Conversely, Client-Side Rendering (CSR) excelled in progressive rendering and interactivity but was more affected by JavaScript execution overhead and network conditions, leading to higher FCP and LCP values in data-intensive workloads.

Optimization techniques such as code splitting, caching, and edge rendering were explored, revealing their potential to significantly mitigate identified bottlenecks and improve user experience across diverse deployment environments.

The analysis of performance metrics from various experiments provides concrete evidence supporting the hypotheses regarding the impact of JavaScript libraries, rendering strategies, and bundle sizes on web application efficiency. The findings validate and expand upon the proposed hypotheses as follows:

Impact of JavaScript Libraries on Initial Load Speed

The use of different JavaScript libraries significantly affects the initial load speed of web applications. Libraries with lightweight and optimized architectures, such as Vanilla JavaScript, consistently demonstrate faster First Contentful Paint (FCP) and Largest Contentful Paint (LCP) values compared to heavier frameworks like React and Vue. These results confirm that simpler, less complex libraries reduce JavaScript execution overhead, leading to faster initial rendering. However, the trade-off lies in the lack of higher-level abstractions and ecosystem support that frameworks provide.

Server-Side Rendering (SSR) vs. Client-Side Rendering (CSR)

The comparison between SSR and CSR implementations in React highlights the significant advantage of SSR in reducing initial load times for large-scale applications. Metrics such

as FCP and LCP are consistently lower in SSR setups, reflecting the benefit of pre-rendering content on the server and delivering it as ready-to-display HTML. Conversely, CSR relies heavily on client-side execution, which introduces delays due to JavaScript parsing and hydration. However, SSR's advantage comes at the cost of slightly higher Total Blocking Time (TBT) during hydration, reflecting the computational effort required to attach interactivity to pre-rendered content.

Smaller Bundle Sizes and Performance

Front-end technologies that produce smaller bundle sizes demonstrate better load speed and application efficiency. Smaller bundles reduce the time required to download, parse, and execute JavaScript, as evidenced by the faster FCP and SI metrics in technologies leveraging optimized bundling strategies. Frameworks and tools that support code splitting, tree shaking, and modular development practices significantly contribute to achieving smaller bundles, ultimately enhancing performance, especially in cloud-hosted environments where network latency plays a critical role.

6.6 Lessons learned

Frameworks Suit Different Use Cases

Each frontend technology and rendering strategy has its strengths and limitations. Vanilla JavaScript provides unmatched simplicity and control but requires meticulous manual optimizations. React and Vue offer advanced abstractions and ecosystem support, with SSR being particularly advantageous for applications requiring rapid initial rendering.

Impact of Rendering Strategy

Server-Side Rendering (SSR) reduces perceived load times and improves layout stability but increases client-side workload during hydration. Client-Side Rendering (CSR) provides flexibility and interactivity but may struggle with large-scale data or slower networks.

Performance is Multidimensional

Key performance metrics (FCP, SI, LCP, TBT, and CLS) each provide unique insights into rendering efficiency, responsiveness, and user experience. A balanced approach that considers all metrics is critical for optimizing web applications.

Optimization is Context-Specific

Techniques such as caching, code splitting, and edge rendering yield varying results depending on application complexity and deployment scenarios. Tailored optimizations are essential for maximizing performance.

Cloud Environments Introduce Variability

Network latency and resource delivery significantly impact performance metrics in cloud environments compared to local setups. Leveraging CDNs and edge computing can miti-

gate these effects.

6.7 Future Directions

Exploring Emerging Technologies

- Investigate the performance of newer frameworks such as Svelte and SolidJS to understand how they compare with React and Vue in terms of rendering efficiency and developer experience.
- Evaluate the impact of React Server Components on SSR performance and hydration overhead.

Optimizations for Large Data Workloads

- Develop and benchmark advanced strategies for efficiently rendering large datasets, such as incremental hydration, streaming server-side rendering, and virtualization.

Cross-Browser and Device Testing

- Extend the analysis to include performance testing across different browsers and device types, particularly focusing on resource-constrained environments like mobile devices.

Real-User Monitoring

- Augment synthetic testing with real-user monitoring (RUM) to capture the actual experience of end-users in diverse network and device conditions.

Environmental Impact Analysis

- Investigate the energy consumption and carbon footprint of various frontend technologies, considering the growing importance of sustainable web development.

Comprehensive Cost-Benefit Analysis

- Develop a cost-benefit framework to help organizations choose the most suitable frontend technology based on performance requirements, development complexity, and operational costs.

The insights from this research provide a foundation for understanding the performance trade-offs and optimization opportunities in modern web frontend development. By adopting the lessons learned and exploring the proposed future directions, developers and organizations can create more efficient, scalable, and user-friendly web applications.

Bibliography

- [1] Ioannis Zozas Andreas Kerren Angelos Chatzimpampas, Stamatia Bibi. Analyzing the evolution of javascript applications. *ResearchGate*, 2019.
- [2] Anonymous. Jsrehab: Weaning common web interface components from javascript addiction. *arXiv*, 2203.06955, 2022. Available at <https://arxiv.org/abs/2203.06955>.
- [3] Bangzhong Cao, Minyong Shi, and Chunfang Li. The solution of web front-end performance optimization. In *2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, pages 1–5, 2017.
- [4] Vitalis Salis Diomidis Spinellis Dimitris Mitropoulos, Panos Louridas. Time present and time past: Analyzing the evolution of javascript code in the wild. In *IEEE Conference Proceedings*, 2019.
- [5] Matteo Varvello Russell Coke Waleed Hashmi Lakshmi Subramanian Yasir Zaki-Authors Jesutofunmi Kupoluyi, Moumena Chaqfeh. Performance improvements in modern web applications. *ACM Digital Library*, 2023.
- [6] Gursheen Kaur and Raj Gaurang Tiwari. Comparison and analysis of popular frontend frameworks and libraries: An evaluation of parameters for frontend web development. In *2023 4th International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 1067–1073, 2023.
- [7] Mattias Levlin. Dom benchmark comparison of the front-end. 2022. Available at https://www.doria.fi/bitstream/handle/10024/177433/levlin_mattias.pdf?sequence=2.
- [8] Bernard Vukelić Marin Kaluža, Krešimir Troškot. Comparison of front-end frameworks for web applications development. *Academia.edu*, 2023. Available at https://www.academia.edu/65267307/Comparison_of_front_end_frameworks_for_web_applications_development.
- [9] Waleed Hashmi Russell Coke Lakshmi Subramanian Yasir Zaki Moumena Chaqfeh, Jacinta Hu. Jsanalyzer: A web developer tool for simplifying mobile pages through javascript optimizations. *arXiv*, 2106.14093, 2021. Available at <https://arxiv.org/abs/2106.14093>.
- [10] Risto Ollila. Analysis of javascript frameworks for modern web development. *Helsinki Digital Repository*, 2018.

- [11] MARIUSZ NYCZ Paweł Dymora, MIROŚLAW MAZUREK. Comparison of angular, react, and vue technologies in the process of creating web applications on the user interface side. *ResearchGate*, 2023.
- [12] Suryaansh Rathinam. Analysis and comparison of different frontend frameworks. In *Springer*, 2024. Available at https://link.springer.com/chapter/10.1007/978-981-99-2264-2_19.
- [13] Suryaansh Rathinam. Analysis and comparison of different frontend frameworks. 2024. Accessed from https://www.suryaanshrathinam.com/static/media/SuryaanshRathinam_ComparisonOfFrontendFrameworks.a6d0c12b1b711c4cc8aa.pdf.
- [14] Shashikant Suman Basu Dev Shivahare Vimal Bibhu Amar Deep Gupta Sai Sri Nandan Challapalli, Prakarsh Kaushik. A comparative analysis of modern frontend frameworks for building large-scale web applications. *IEEE Xplore*, 2024. Available at <https://ieeexplore.ieee.org/document/9673464>.
- [15] Pancham Singh, Mili Srivastava, Mrignainy Kansal, Aditya Pratap Singh, Abhay Chauhan, and Adarsh Gaur. A comparative analysis of modern frontend frameworks for building large-scale web applications. In *2023 International Conference on Disruptive Technologies (ICDT)*, pages 531–535, 2023.
- [16] Sajjad Arshad Engin Kirda Tobias Lauinger, Chaabane Abdelberi. Thou shalt not depend on me: Analyzing the use of outdated javascript libraries on the web. *ResearchGate*, 2017.
- [17] Jason Tong, Ricky Rivaldo Jikson, and Alexander Agung Santoso Gunawan. Comparative performance analysis of javascript frontend web frameworks. In *2023 3rd International Conference on Electronic and Electrical Engineering and Intelligent System (ICE3IS)*, pages 81–86, 2023.
- [18] Moritz Steiner Utkarsh Goel. System to identify and elide superfluous javascript code for faster webpage loads. *arXiv*, 2003.07396, 2020. Available at <https://arxiv.org/abs/2003.07396>.
- [19] Yuhua Wei. Big data intelligent analysis based web front-end performance optimization and its application research. In *2022 Fourth International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT)*, pages 1–5, 2022.
- [20] Leah Williams. Bundle size optimization: A practical guide using webpack and statoscope, 2021.