Journal of Software Engineering Research and Development, 2026, 14:30, doi: 10.5753/jserd.2026.8015
This work is licensed under a Creative Commons Attribution 4.0 International License.

Evaluating Quality Attributes in Web Development: A Comparative Study of Rendering Strategies

Joey Clapton [**Pontifical Catholic University of Minas Gerais (PUC Minas)** | ***joeyclapton42@gmail.com***]
João Pedro Oliveira Batisteli [**Pontifical Catholic University of Minas Gerais (PUC Minas)** | ***joaobatisteli@pucminas.br***]
Hugo Bastos de Paula [**Pontifical Catholic University of Minas Gerais (PUC Minas)** | ***hugo@pucminas.br***]
Danilo Boechat Seufitelli [**Federal Center for Technological Education of Minas Gerais (CEFET-MG)** | ***danilobs@cefetmg.br***]
Cleiton Tavares [**Pontifical Catholic University of Minas Gerais (PUC Minas)** | ***cleitontavares@pucminas.br***]

Abstract

Modern web development identifies the rendering strategy as a critical architectural choice, directly impacting performance and user experience. This study compares the performance of Client-Side Rendering (CSR) and Server-Side Rendering (SSR) in web applications under varying JSON payload sizes. We evaluated performance based on Google's Core Web Vitals (CWV) metrics, including Largest Contentful Paint (LCP), First Contentful Paint (FCP), Speed Index (SI), Time to Interactive (TTI), and JS Heap Size. We implemented a controlled environment using a photo gallery application and automated measurements via Lighthouse and Puppeteer across five data sizes: Extra Small (XS), Small (S), Medium (M), Large (L), and Extra Large (XL). The experiment was expanded to include six experimental conditions, incorporating JSON minification and Gzip compression as optimization techniques. The statistical analysis involved the Kruskal–Wallis test, pairwise Mann–Whitney U tests with Holm correction, and Cliff's Delta as the effect size measure. Results indicate that SSR outperformed CSR in LCP and TTI across all data sizes. Conversely, CSR maintained a stable FCP regardless of the data volume. Gzip compression was the most impactful optimization. It reduced the median LCP by up to 86.7% for CSR and 93.0% for SSR on the Extra Large payload. SSR with Gzip was up to 7.9× faster than the optimized CSR in terms of LCP for the largest payload. These findings characterize the trade-offs between a controlled implementation of client-side JSON rendering and server-side pre-rendered HTML, and should not be interpreted as a complete evaluation of modern production frameworks (e.g., Next.js, Nuxt, SvelteKit, React, Vue) that employ hydration, streaming SSR, or partial/island architectures. Within this scope, these findings guide software engineers in choosing the most appropriate rendering strategy based on performance objectives and application data volume.

Keywords: *Client-Side Rendering, Server-Side Rendering, Performance Analysis, User Experience, Web Applications*

1 Introduction

Modern web development considers the rendering strategy a pivotal decision, directly impacting user experience and performance (Verma and Aland, 2024). Server-Side Rendering (SSR) processes JavaScript and assembles HTML content directly on the server. This approach benefits low-power devices by offloading computational tasks and improves search engine indexing (Iskandar et al., 2020). Conversely, Client-Side Rendering (CSR) delivers a minimal HTML structure and executes JavaScript locally to build the interface. This strategy favors highly interactive Single Page Applications (SPAs) (Gangishetti and Jain, 2026). The choice between these methods directly impacts performance, computational costs, and user experience (Sikder, 2016).

The digitization of daily activities has significantly increased the volume of data processed by these web applications, such as e-commerce marketplaces and streaming platforms (Gangishetti and Jain, 2026). JavaScript Object Notation (JSON) has emerged as the standard format for data exchange due to its flexibility. However, large JSON payloads introduce challenges regarding data parsing (Li et al., 2017) and memory usage (Gil and Trezentos, 2011). These computational costs also have environmental implications. The Information and Communication Technology (ICT) sector contributes significantly to global CO_2 emissions, making resource efficiency a priority in software engineering (Huber et al., 2022).

The choice between CSR and SSR is a recurring decision in web projects, with a direct impact on performance and user experience rather than being merely an academic comparison. The currently dominant frameworks (Next.js, Nuxt, SvelteKit, Remix) require development teams to define the rendering strategy on a per-route basis. This decision is made early in the development process, propagates throughout the application architecture, and becomes costly to reverse as the application grows. Its consequences are reflected in business outcomes rather than solely in laboratory metrics. High loading times increase user abandonment and reduce conversion rates (Sikder, 2016). Since the incorporation of *Core Web Vitals* (CWV) into Google's ranking signals, LCP and other loading metrics have directly influenced an application's organic visibility (Developers, 2024), particularly affecting CSR, whose content is only available after JavaScript execution (Iskandar et al., 2020). There is also an accessibility impact: shifting the rendering cost to the client disadvantages users with lower-performance devices or limited network connections, thereby increasing inequality in access to web applications (Verma and Aland, 2024). Treating this decision as merely a technical preference therefore overlooks its implications for revenue, reach, and inclusivity.

This trade-off becomes more pronounced as data volume increases. Modern e-commerce applications, analytical dash-

boards, and media platforms routinely transfer catalogs, time series, and metadata collections whose sizes are orders of magnitude larger than those of the web pages for which these rendering strategies were originally conceived (Zhu et al., 2022; Gangishetti and Jain, 2026). Under these conditions, the differences between client-side and server-side rendering are no longer marginal, motivating the systematic investigation of payload size effects conducted in this study.

Previous research has explored data interaction formats and rendering performance (Gil and Trezentos, 2011; Ollila et al., 2022; Gangishetti and Jain, 2026). Studies show that SSR typically offers faster initial loading, while CSR provides smoother transitions in interactive scenarios (Verma and Aland, 2024). Furthermore, rendering costs grow significantly with data volume, potentially creating bottlenecks in real-world applications (Ollila et al., 2022; Gangishetti and Jain, 2026). Despite these insights, a systematic investigation into how JSON payload sizes (combined with optimization techniques) affect standardized quality metrics remains missing.

This paper expands our previous work (Clapton et al., 2025), increasing the scope of the original study by evaluating the impact of varying JSON payload sizes on Google's CWV metrics. This version also updates the experimental analysis to include data optimization techniques and a more robust statistical methodology. These additions provide a comprehensive understanding of the trade-offs between CSR and SSR in data-intensive scenarios.

The goal of this study is to investigate how JSON payload size and transport optimizations, namely JSON minification and Gzip compression, influence the performance trade-offs between client-side JSON rendering (CSR) and server-side HTML pre-rendering (SSR) in a controlled web application. To achieve this objective, the study is guided by the following research questions:

- **RQ1**: What is the impact of increasing JSON payload size on Largest Contentful Paint (LCP), First Contentful Paint (FCP), Speed Index (SI), Time to Interactive (TTI), and JavaScript heap memory usage in client-side JSON rendering and server-side HTML pre-rendering?
- **RQ2**: What are the effects of JSON minification and Gzip compression on the evaluated performance metrics?
- **RQ3**: Under optimized transport conditions, which rendering strategy achieves lower LCP and TTI in the controlled web application?

We designed and implemented a controlled experimental environment based on a photo gallery web application to enable a fair comparison between client-side rendering (CSR) and server-side rendering (SSR). Performance measurements were automated using Lighthouse and Puppeteer across five JSON payload sizes. The experimental results show that SSR consistently outperformed CSR in terms of Largest Contentful Paint (LCP) and Time to Interactive (TTI), whereas CSR achieved better First Contentful Paint (FCP) and Speed Index (SI), particularly for smaller payloads. As payload size increased, SSR became increasingly advantageous for user-perceived loading performance. Among the evaluated optimizations, Gzip compression had the greatest impact, reducing LCP by more than 90% for the largest payloads. These findings provide empirical evidence to support software engineers in selecting the rendering strategy that best aligns with application performance requirements and data volume.

The main contributions of this study are as follows:

(i) a systematic evaluation of the impact of JSON payload size on the performance of client-side rendering and server-side HTML pre-rendering;
(ii) a comparative analysis of the computational costs associated with CSR and SSR across multiple performance metrics;
(iii) an assessment of the effectiveness of transport optimizations, namely JSON minification and Gzip compression, on rendering performance; and
(iv) a statistical analysis of the observed differences using the Kruskal–Wallis test, pairwise Mann–Whitney U tests with Holm correction, and Cliff's Delta effect size.

The remainder of this study is organized as follows. Section 2 provides the theoretical foundation for web rendering and data exchange. Section 3 reviews related literature to contextualize our findings within the state of the art. Section 4 describes the experimental methodology, including the tools used for automated auditing and the data optimization techniques. Section 5 presents the collected experimental data, while Section 6 offers a critical analysis of the observed performance trade-offs. Section 7 examines the potential limitations and the validity of the research. Finally, Section 8 summarizes the study's main contributions and suggests directions for future research.

2 Background

This section presents the key concepts related to the study. Section 2.1 describes the SSR method. Section 2.2 discusses the CSR method. Section 2.3 addresses software performance aspects, and Section 2.4 covers quality indicators in web applications based on Core Web Vitals (CWV) metrics.

2.1 Server-Side Rendering (SSR)

Server-Side Rendering is a web application development methodology in which pages are rendered on the server and sent to the browser (Verma and Aland, 2024). When a request is made, the server processes it and returns the page already rendered in HTML to the user's browser.

In applications that use SSR, the rendering process is handled on the server, enabling the page to be fully loaded when accessed by the user (Iskandar et al., 2020; Verma and Aland, 2024). This approach improves the user experience, particularly on low-performance devices, as the server performs the more computationally intensive tasks. Moreover, SSR provides advantages in Search Engine Optimization (SEO), as it facilitates the indexing of page content by search engines.

2.2 Client-Side Rendering (CSR)

Client-Side Rendering is a web page rendering mechanism in which the rendering process takes place in the user's browser

(Verma and Aland, 2024). The server sends a basic HTML structure with JavaScript responsible for dynamically rendering the content on the client side.

CSR enables the development of more dynamic and interactive applications, as both the rendering logic and Document Object Model (DOM) manipulation are executed directly within the user's browser (Gangishetti and Jain, 2026). This approach can deliver a smoother and more responsive user experience, particularly in highly interactive scenarios such as Single Page Applications (SPAs), where navigation between views occurs without the need for full page reloads.

2.3 Software Performance Aspects

A software performance aspect refers to a non-functional characteristic that describes the operational efficiency of a software system under various execution conditions (Moghadam, 2023). These aspects are essential to ensure that the software meets the expected quality and efficiency demanded by users. Key performance metrics include response time, which measures how long the system takes to respond to a request, and resource usage, which evaluates the memory and CPU consumption during application operations.

Performance aspects are critical to user experience, user retention, and the overall success of a web application. Pages that load slowly or respond poorly can frustrate users and cause them to abandon the site. Good performance is also important for Search Engine Optimization (SEO), as search engines tend to favor fast-loading websites (Sikder, 2016). Optimizing performance aspects has become a crucial part of developing and maintaining websites, requiring an integrated approach that considers both operational efficiency and end-user experience.

2.4 Quality Indicators in Web Applications Based on Core Web Vitals Metrics

Core Web Vitals (CWV) is a set of performance metrics introduced by Google[1] to help optimize and evaluate the user experience of web pages, guiding improvements in the user experience. The CWV includes:

- **Largest Contentful Paint (LCP)**: measures the loading time of the largest content element visible to the user.
- **First Input Delay (FID)**: measures the time from the user's first interaction to the browser's response.
- **Cumulative Layout Shift (CLS)**: evaluates layout shifts during page loading.
- **First Contentful Paint (FCP)**: measures how quickly content is rendered to the screen, ensuring users see something useful sooner.
- **Speed Index (SI)**: measures how quickly visible content is displayed during page loading, influencing user experience and search engine ranking.
- **Time to Interactive (TTI)**: evaluates page usability. A faster TTI means users can interact with the page sooner.
- **JS Heap Size**: measures the memory space allocated for storing JavaScript objects during application execution.[2] Although not part of Google's official Core Web Vitals, it was included in this study as a computational cost indicator, collected via Puppeteer.[3]

3 Related Work

The related work in web application engineering spans rendering paradigms, data interchange overheads, and performance optimization strategies. To contextualize our empirical evaluation, we organize existing research into three key thematic areas: (i) baseline rendering comparisons and framework performance, (ii) data processing and network optimization bottlenecks, and (iii) hybrid, isomorphic, and progressive rendering architectures.

3.1 Rendering Strategies and Framework Performance

Evaluating the empirical trade-offs between CSR and SSR is an established topic in software engineering. Iskandar et al. (2020) evaluated both approaches using a photo-sharing application, demonstrating that static SSR yielded superior First Contentful Paint (FCP), Speed Index (SI), and Time to Interactive (TTI) compared to monolithic CSR. Building upon these baseline comparisons, Dymora et al. (2023) evaluated modern client-side frontend frameworks (Angular, React, and Vue), emphasizing that client-side rendering performance is heavily constrained by browser execution threads and DOM reconciliation overheads. While these studies establish fundamental trade-offs, they primarily evaluate fixed data payloads, leaving open questions regarding how performance scales as backend payload sizes increase.

3.2 Data Transfer and Parsing Bottlenecks

As web applications consume increasingly large structured datasets, network data exchange and data parsing emerge as primary performance bottlenecks. Langdale and Lemire (2019) highlighted that traditional client-side JSON parsers struggle with large payloads, introducing *simdjson* to mitigate parsing latencies through SIMD instruction parallelism. On the client side, parsing large JSON documents directly consumes CPU cycles and increases JavaScript Heap memory allocations. Complementing data processing research, van Riet et al. (2023) conducted an industrial case study using Google Lighthouse to measure web performance engineering interventions, achieving up to 98% faster load times on dashboard applications. Furthermore, Korobeinikova et al. (2022) emphasized that managing application state and payload volume is crucial for maintaining scalability and responsiveness under high traffic demands. However, these studies do not explicitly evaluate how payload reduction techniques, such as minification and

[1] `https://developers.google.com/web/tools/lighthouse`

[2] `https://developer.chrome.com/docs/devtools/memory-problems`

[3] `https://pptr.dev/`

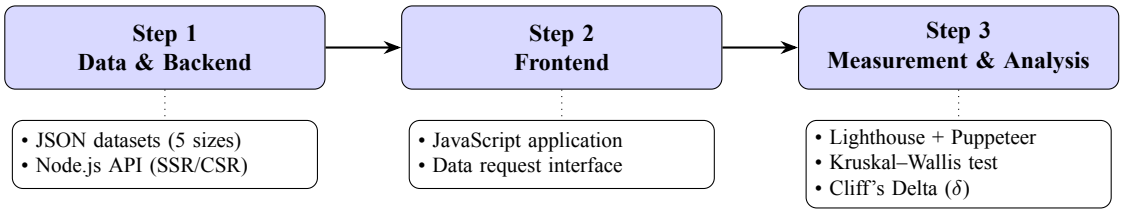


Figure 1. Overview of the study steps and their sub-activities.

Gzip compression, interact with distinct rendering paradigms (CSR vs. SSR) across varying data volumes.

3.3 Hybrid Paradigms and Progressive SSR

To address the traditional "hydration gap"—where pre-rendered HTML appears visually complete but remains non-interactive until client-side JavaScript finishes executing—recent literature explores hybrid and streaming architectures. Early efforts by da Silva and Farah (2018) combined pre-rendering and isomorphic JavaScript within Single-Page Application (SPA) architectures, demonstrating performance improvements during initial page loading. Pushing this boundary further, Carvalho and Fialho (2023) introduced a Progressive Server-Side Rendering (PSSR) framework designed for low-thread web servers, enabling concurrent streaming of HTML chunks as asynchronous backend requests resolve. Carvalho (2024) expanded this approach by introducing suspendable web templates, which allow servers to defer unrendered content blocks without blocking the initial response stream.

3.4 Positioning of This Study

While progressive streaming architectures (e.g., PSSR) address server-side I/O concurrency and rehydration delays, many production web systems continue to rely on transferring fully assembled HTML responses (SSR) or delivering complete API datasets for client-side assembly (CSR). Our study bridges this gap by systematically quantifying the architectural trade-offs of these standard data transfers across five scalable JSON payload sizes (XS to XL) and evaluating the practical impact of baseline data optimizations (JSON minification and Gzip compression) on Core Web Vitals metrics and memory consumption.

4 Methodology

This study adopts a quantitative experimental approach to investigate and compare the performance, computational cost, and CWV metrics of SSR and CSR for data presentation in web applications. The following subsections describe the development environment, which consists of a web application. Subsequently, the materials and methods used in the experiment are detailed, including performance measurement tools, computational cost metrics, and the different data sizes analyzed.

4.1 Study Steps

The study was conducted in three steps, as illustrated in Figure 1, which summarizes each stage and its corresponding sub-activities. During the first step, the datasets were defined and generated in JSON format. Five dataset sizes were created: 498 kB, 2 MB, 7.9 MB, 16.8 MB, and 26 MB. Each dataset consisted of repeated combinations of 13 distinct images, resulting in collections of 370 images (498 kB), 1,500 images (2 MB), 6,000 images (7.9 MB), 12,000 images (16.8 MB), and 19,000 images (26 MB). These dataset sizes were selected based on the study by Nurseitov et al. (2009), which evaluated the performance of data interchange formats across a range of application scenarios.

The five dataset sizes were selected to progressively span the range between typical API responses and payloads large enough to stress browser resources. The smallest payload (498 kB) is representative of a conventional paginated API response, such as a search results page in an e-commerce catalog. The intermediate payloads (2 MB and 7.9 MB) approximate production scenarios commonly found in modern web applications, including analytical dashboards that retrieve complete time-series datasets in a single request and media feeds that deliver metadata for thousands of items without pagination. Such designs are frequently adopted to enable client-side filtering and sorting after a single data transfer, thereby reducing the need for subsequent network requests (Gangishetti and Jain, 2026; Zhu et al., 2022).

The two largest payloads (16.8 MB and 26 MB) were included as stress-test conditions rather than as representations of typical production workloads. Their purpose within the experimental design is to reveal the asymptotic behavior of each rendering strategy. Because rendering costs increase nonlinearly with data volume (Ollila et al., 2022), experiments restricted to moderate payload sizes may conceal performance differences that become critical as applications scale. Including these stress conditions enables the characterization of the complete performance curve, rather than only a limited portion of it, and helps identify the payload size beyond which each rendering strategy becomes impractical. Consequently, the effect magnitudes reported for the largest payloads should be interpreted as upper-bound estimates of the performance trade-offs, rather than as performance differences expected in typical web applications.

For the experiment, a data structure was designed to simulate a photo gallery system using JSON (Figure 2). The choice of this data structure aims to represent a real-world use case where user information is frequently exchanged between systems. Furthermore, a back-end application was developed using the Node.js platform. An Application Programming Interface (API) was created to handle two param-

eters: data size and rendering type (SSR or CSR). Based on these parameters, the API returns either the HTML rendered from the JSON or the raw JSON itself to the client. The data is stored locally.


```
{
  "user_id": 101,
  "username": "user6",
  "avatar_url": "https://example.com/profiles/user6.jpg",
  "posts": [
    {
      "post_id": 2,
      "image_url": "https://example.com/photos/photo2.jpg",
      "caption": "Sunset vibes",
      "timestamp": "2023-10-02T18:30:00Z",
      "likes": [
        {
          "user_id": 107,
          "username": "user7"
        },
        {
          "user_id": 108,
          "username": "user8"
        }
      ]
    }
  ]
}
```


Figure 2. JSON data structure used in the application.

The second step involved the development of a JavaScript-based front-end web application. As illustrated in Figure 3, the application allows users to select both the payload size and the response format to be requested. After the request is submitted to the back-end, the returned content is displayed in the browser. Depending on the selected rendering strategy, the response consists of either server-side pre-rendered HTML or raw JSON, which is subsequently processed on the client to dynamically construct the web page.

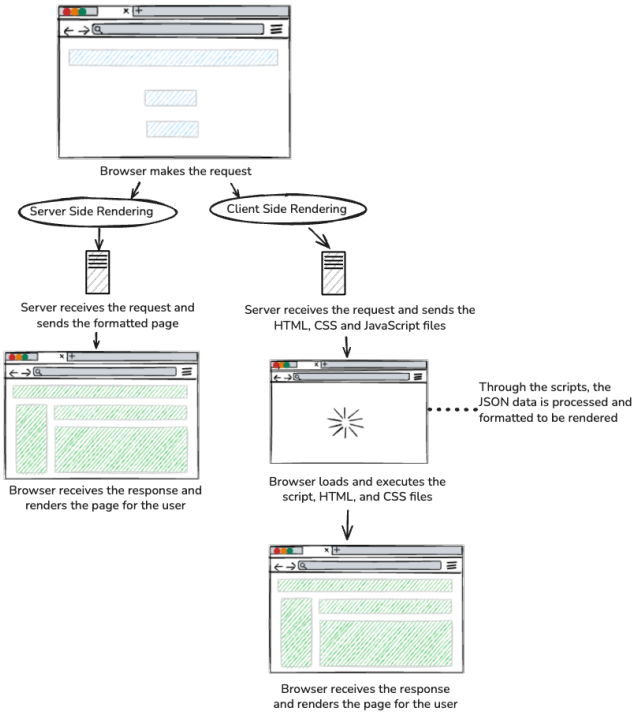


Figure 3. Application operation flow.

Subsequently, a JavaScript-based benchmarking script was developed to evaluate application performance. Using the Lighthouse library,[4] the script automatically collected the selected web performance metrics. To ensure consistent experimental conditions, the `chrome-launcher` library was used to launch a fresh instance of the Chrome browser for each execution. In addition, Puppeteer was employed to collect browser profiling data and computational cost metrics. Each experimental configuration (JSON and HTML) was executed 10 times, and the median and interquartile range (IQR) were reported, as these statistics are robust to outliers. The evaluated metrics were: *(i)* First Contentful Paint (FCP), *(ii)* Speed Index (SI), *(iii)* Time to Interactive (TTI), *(iv)* Largest Contentful Paint (LCP), and *(v)* JavaScript Heap Size (JS Heap).

Each evaluated metric captures a distinct stage of the page loading process at which CSR and SSR are expected to differ. Moreover, the temporal metrics have well-established thresholds and documented correlations with user experience and behavior (Developers, 2024; Sikder, 2016). FCP measures the moment when the first visual content is rendered, marking the end of the initial blank screen. This is the metric in which CSR may benefit from delivering the page structure before the application data. LCP measures when the primary page content becomes visible. In CSR, this occurs only after the JSON payload has been downloaded, parsed, and rendered, making LCP the metric most strongly associated with perceived loading speed and user abandonment (Google, 2024b).

TTI indicates when the page becomes fully responsive to user input and is particularly sensitive to main-thread blocking caused by processing large client-side payloads. Finally, JS Heap is not a temporal metric but quantifies the memory overhead that fundamentally distinguishes the two rendering strategies, as CSR must retain the parsed dataset in browser memory, whereas SSR performs the rendering on the server and therefore avoids this client-side memory cost. Collectively, these metrics capture four complementary dimensions of web application performance (initial rendering, primary content rendering, interactivity, and memory consumption) through which the practical effects of choosing between CSR and SSR can be observed (Verma and Aland, 2024; Iskandar et al., 2020).

4.2 Measurement Configuration and Data Collection Protocol

All measurements were collected using Lighthouse 12.2.0 orchestrated through Puppeteer 23.2.1 (Chromium) in a single uninterrupted experimental batch covering all 30 experimental conditions (300 executions in total, with no failed runs). For reproducibility, Lighthouse was executed with its default laboratory configuration: mobile device emulation (Moto G Power profile with a 412×823 viewport) and *simulated* throttling based on the Lantern model, using a reference network profile of approximately 1.47 Mbps download bandwidth, 150 ms round-trip latency, and a $4\times$ CPU slowdown multiplier. Under simulated throttling, the page is loaded without artificial network or CPU constraints, after which Lighthouse *estimates* the performance metrics under the standardized reference profile from the recorded execution trace. Consequently, the reported metrics represent model-based estimates of page-loading performance under a fixed laboratory profile rather than field measurements. The

[4] `https://developer.chrome.com/docs/lighthouse/`

implications of this experimental design are further discussed in Section 7.

To ensure that every execution represented a cold-page load, the browser cache and cookies were cleared through the Chrome DevTools Protocol (CDP), and page caching was disabled before each of the 10 executions for every experimental condition. Lighthouse performs an independent navigation for each measurement; therefore, no warm-up navigation was performed before evaluating the target URL. JS Heap was measured on the evaluated page using the CDP metric `JSHeapUsedSize` after the page reached the network-idle state and is reported in both kilobytes (kB) and megabytes (MB). Executions in which the measurement instrumentation failed were excluded from the dataset and recorded separately rather than being assigned zero values. In the final experimental batch, no execution failures occurred. The complete data collection pipeline is available in the replication package.

4.3 Collected but Excluded Metrics

The measurement pipeline recorded additional metrics beyond those analyzed in this study. This subsection documents the rationale for excluding them from the statistical analysis, as an earlier version of this manuscript referenced these metrics in the discussion of validity without reporting their results.

Speed Index (SI) was identical to FCP in all 300 executions. This outcome was not caused by a measurement error but rather by the way SI is estimated under the simulated throttling configuration described in Section 4.2. The Lantern model computes SI as a weighted average of simulated *layout* events, truncating each event time to be no smaller than the FCP and applying a second truncation to the final estimate. Because all experiments were conducted on *localhost*, the actual page load completed almost instantaneously, causing every observed *layout* event to occur before the modeled FCP. Consequently, both truncation steps were triggered, and SI converged exactly to the FCP value. Under these conditions, SI provides no additional information beyond FCP and was therefore excluded from the analysis.

Cumulative Layout Shift (CLS) was equal to 0.00 in all 300 executions, regardless of rendering strategy or payload size. The evaluated application renders its content in a single pass, without asynchronously inserting elements above the fold, and explicitly defines image dimensions. Consequently, no layout shifts occurred during page loading. Because the metric exhibited no variation across experimental conditions, it could not support a meaningful statistical comparison. Nevertheless, the consistently null result indicates that neither rendering strategy introduced visual instability in the evaluated scenario.

First Input Delay (FID) was collected to maintain compatibility with the conference version of this work (Clapton et al., 2025) but was not included in the analysis for two reasons. First, laboratory experiments report the *Maximum Potential FID*, which is an upper-bound estimate derived from the longest task executed on the browser's main thread rather than the field-measured FID, which depends on actual user interactions. Second, Google replaced FID with *Interaction to Next Paint* (INP) as a Core Web Vitals metric in March 2024 because INP captures the latency of all user interactions throughout a session instead of only the first one (Developers, 2024). Reporting FID as a primary outcome would therefore rely on a deprecated metric.

Finally, the **Lighthouse Performance Score** was not analyzed because it is a composite metric computed as a weighted aggregation of the individual performance metrics already reported in this study. Since our objective is to attribute observed performance differences to specific stages of the page loading process, analyzing the aggregated score would obscure these stage-specific effects while providing little additional insight. Nevertheless, all metrics described in this subsection are included in the replication package, allowing independent reanalysis by other researchers.

4.4 Statistical Analysis

In the third step, statistical tests were conducted to assess the significance and practical magnitude of the observed differences. The Shapiro–Wilk test confirmed the non-normality of the data, justifying the use of non-parametric tests. The Kruskal–Wallis test was applied to compare the metrics between the five data sizes (XS, S, M, L, and XL), while the Mann–Whitney U test was used for pairwise comparison between CSR and SSR, as it is more suitable for two independent groups (Kitchenham et al., 2017; Kitchenham and Madeyski, 2024). Since the p-value alone does not indicate the practical magnitude of the differences (Torkar et al., 2022), Cliff's Delta was adopted.

We adopted Cliff's Delta (δ) as an effect size measure to quantify the magnitude of the observed differences between the experimental groups. This non-parametric statistic calculates the probability that a randomly selected observation from one group is larger than an observation from another group (Cliff, 1993). This method is suitable for non-normal data distributions, as confirmed by the Shapiro–Wilk tests conducted in this study. The δ values range from -1 to $+1$. We interpreted these values according to the thresholds proposed by Romano et al. (2006): $|\delta| < 0.147$ (*negligible*), $0.147 \leq |\delta| < 0.33$ (*small*), $0.33 \leq |\delta| < 0.474$ (*medium*), and $|\delta| \geq 0.474$ (*large*).

We justify the use of effect size because statistical significance alone does not quantify the practical importance of the observed differences. Relying exclusively on p-values may lead to overstated conclusions, particularly in controlled experimental settings (Kitchenham et al., 2017; Torkar et al., 2022). In addition, 95% confidence intervals were estimated using the bootstrap method with 1,000 resamples (Efron and Tibshirani, 1994). Pairwise post-hoc comparisons following the Kruskal–Wallis test were performed using Mann–Whitney U tests with Holm correction to account for multiple comparisons (Kitchenham and Madeyski, 2024). We note that this procedure differs from Dunn's test, which is based on a single joint ranking of all groups.

Finally, the applicability of inferential statistics is constrained by the measurement model described in Section 4.2. Because Lighthouse simulated throttling estimates performance metrics from a deterministic model under a fixed network and CPU profile, experimental conditions dominated

by payload transfer time exhibited virtually no variability across the 10 executions (coefficient of variation below 1% in 23 of the 30 experimental conditions). Under these circumstances, hypothesis tests and effect size estimates become saturated ($|\delta| = 1$) by construction and therefore do not support meaningful statistical inference. Accordingly, these comparisons are reported descriptively in terms of differences between median values, whereas inferential claims are restricted to the experimental conditions that exhibited genuine execution-to-execution variability.

4.5 Photo Application

Two photo gallery web applications were developed: one implementing server-side rendering and the other client-side rendering. In addition, a selection page was created to allow users to choose both the rendering strategy and the payload size before each experiment. To ensure a fair comparison, both applications shared the same HTML structure, CSS styles, and visual design, differing only in their rendering strategy. No additional scripts or interface modifications were introduced that could influence the performance measurements. Figure 4 presents the initial selection page, whereas Figure 5 illustrates the application using client-side rendering with the smallest payload. These figures illustrate the experimental artifact used to generate each evaluated scenario. Figure 4 shows how the payload size and rendering strategy are selected before each measurement, whereas Figure 5 demonstrates that the user interface remains visually identical for CSR and SSR. This design ensures that the performance differences reported in this study are attributable to the rendering strategy itself rather than to differences in layout, styling, or application content.

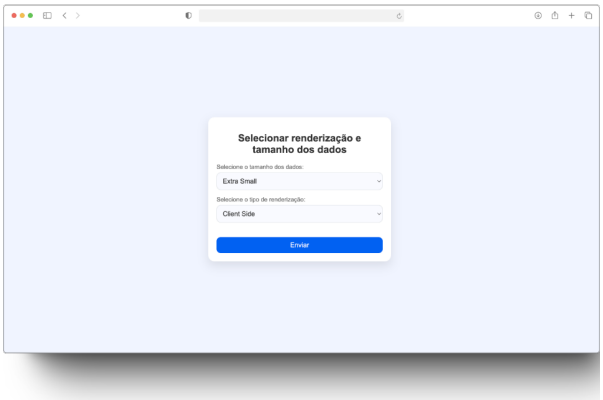


Figure 4. Data size and rendering type selection.

4.6 Performance Optimization Techniques

We selected two performance optimization techniques to examine their effects on application quality metrics. These techniques include JSON data minification and Gzip compression. Their practical relevance in web development justifies their inclusion in this study. Developers widely use these methods in production environments to reduce network data

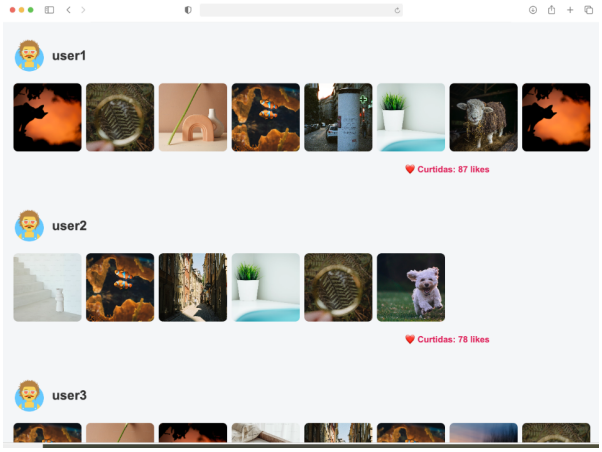


Figure 5. Application using extra small data and CSR.

volume and decrease page load times (Sikder, 2016). Therefore, investigating the impact of these optimizations on rendering approaches contributes to a more comprehensive analysis of factors influencing web application quality (Verma and Aland, 2024; Gangishetti and Jain, 2026).

On the one hand, *Data Minification* involves removing whitespace, line breaks, and other unnecessary characters from a document to reduce the payload size transferred between server and client. In this study, we applied minification to JSON data using the native `JSON.stringify()` function without the spacing parameter (*indent*). This approach results in a compact data representation without altering its structure or content (Tiwary et al., 2021). The method is computationally simple and introduces no external dependencies to the project.

On the other hand, *Compression* reduces data size by applying algorithms to identify and eliminate redundancies in the transferred content. Web servers commonly adopt the Gzip algorithm to compress HTTP responses before sending them to clients. Most modern browsers support this compression. In this study, we implemented Gzip compression in the Node.js server using the native `zlib` module. This module compresses data before returning it as a response to client requests. We used the `Content-Encoding: gzip` HTTP header to indicate that the received content is compressed. This allows the browser to perform decompression transparently (Tiwary et al., 2021; Tusa and Clayman, 2021).

Based on these characteristics, we formulated the following hypotheses for this stage of the study:

- **H1**: JSON minification reduces the transmitted payload size. This reduction positively impacts FCP and SI metrics because it decreases data transfer and initial processing times.
- **H2**: Gzip compression significantly reduces data transfer time between server and client. This reduction improves the evaluated performance metrics.
- **H3**: Combining minification with Gzip compression creates a synergistic effect. Together, they achieve greater reductions in data size and faster loading times than either technique alone.

Evaluating these hypotheses aims to verify whether data optimizations can alter the observed performance patterns between CSR and SSR. This analysis expands the understand-

Table 1. Descriptive statistics for baseline CSR and SSR results across all data sizes (without optimization).

Metric	Measures	Client XS	Server XS	Client S	Server S	Client M	Server M	Client L	Server L	Client XL	Server XL
LCP (ms)	Average	11,263	4,788	20,863	6,832	49,832	10,298	93,662	20,221	136,891	29,994
	Median	9,831	2,178	20,861	3,832	49,830	10,297	93,657	20,220	136,883	29,994
	Std. Dev.	2,328	4,214	6.3	4,848	8.2	1.3	15.8	2.8	24.0	2.0
FCP (ms)	Average	1,578	1,231	1,577	2,952	1,577	9,510	1,577	19,429	1,577	29,199
	Median	1,578	1,231	1,577	2,951	1,577	9,508	1,577	19,427	1,577	29,199
	Std. Dev.	1.7	1.2	0.2	5.4	0.2	5.3	0.2	7.4	0.1	2.8
SI (ms)	Average	1,578	1,231	1,577	2,952	1,577	9,510	1,577	19,429	1,577	29,199
	Median	1,578	1,231	1,577	2,951	1,577	9,508	1,577	19,427	1,577	29,199
	Std. Dev.	1.7	1.2	0.2	5.4	0.2	5.3	0.2	7.4	0.1	2.8
TTI (ms)	Average	11,843	4,971	21,823	6,971	50,792	10,350	94,622	20,347	137,866	30,292
	Median	10,409	2,230	21,821	3,884	50,790	10,350	94,617	20,340	137,858	30,284
	Std. Dev.	2,328	4,422	6.5	4,984	8.3	1.4	15.9	14.9	24.1	20.8
***JS Heap* Used (KB)**	Average	2,256	1,557	5,865	5,877	13,648	3,676	41,293	9,501	48,151	7,398
	Median	2,509	1,533	5,975	1,810	13,357	1,466	40,762	1,464	42,764	1,690
	Std. Dev.	530	515	2,112	6,994	4,072	7,078	19,953	17,193	23,337	18,529

* Client = Client Side, Server = Server Side, XS = Extra Small, S = Small, M = Medium, L = Large, XL = Extra Large, Std. Dev. = Standard Deviation. Values from a single-batch corrected collection; *JS Heap* is now measured on the evaluated page (see Section 4.2).

ing of factors influencing web application quality (Ollila et al., 2022; Gangishetti and Jain, 2026).

5 Results

This section discusses the experimental results obtained by comparing CSR and SSR in five JSON payload sizes (XS, S, M, L, and XL). The performance metrics LCP, FCP, SI, TTI, and JS Heap Size are shown individually, with descriptive statistics and figures. Next, we evaluate the significance of the observed differences through a statistical analysis. Finally, we report the results of the applied optimization techniques, namely minification and Gzip compression.

After experimentation, we computed the mean, median, and standard deviation for each metric and rendering method (shown in Table 1). We note that, for smaller data sizes, the metrics exhibited lower, more consistent values, especially on the server side. On the other hand, as the data size increased, the standard deviation also increased in some measurements, indicating greater variability in the results.

Largest Contentful Paint. LCP measures the time required for the largest visible content element (e.g., an image, text block, or video) to be rendered on the page. According to Google's performance guidelines, an LCP of up to 2.5 s is considered good, values between 2.5 s and 4.0 s indicate that improvement is needed, and values above 4.0 s correspond to a poor user experience (Google, 2024b). Figure 6 presents the median LCP for each dataset size under the six experimental conditions (logarithmic scale on the y-axis). Among the non-optimized configurations, the SSR baseline consistently achieved lower LCP values than the CSR baseline across all payload sizes. For example, the median LCP was 2.18 s for SSR versus 9.83 s for CSR with the XS payload, increasing to 29.99 s and 136.88 s, respectively, for the XL payload (Table 1).

The most substantial performance improvements, however, resulted from transport optimization. SSR combined with Gzip achieved the lowest LCP values for almost all payload sizes and remained below the 2.5 s threshold even for the XL payload. In contrast, the CSR baseline degraded substantially, exceeding 100 s under the same conditions. The only exception occurred for the S payload, where the SSR+Gzip configuration exhibited a bimodal distribution, with five executions clustered around 1.6 s and five around 12 s (median = 6.83 s). This configuration also showed the highest execution-to-execution variability in the experiment, explaining why the corresponding post-hoc comparison did not detect a statistically significant difference.

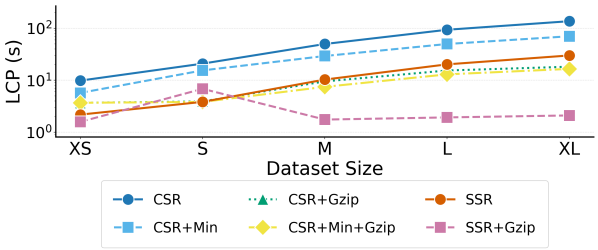


Figure 6. Median LCP values.

First Contentful Paint. FCP measures the time elapsed between the user's navigation request and the moment when the first visual content is rendered on the screen. According to Google's performance guidelines, an FCP of up to 1.8 s is considered good, whereas values above 3.0 s indicate poor user experience (Google, 2024a). Figure 7 presents the median FCP for each payload size under the six experimental conditions (logarithmic scale on the y-axis). CSR maintained a nearly constant FCP of approximately 1.58 s across all payload sizes, regardless of the applied transport optimization. This behavior is expected because the browser renders the initial page structure before fetching the JSON payload through JavaScript. In contrast, the baseline SSR configuration exhibited a progressive increase in FCP, from 1.23 s for the XS payload to 29.20 s for the XL payload (Table 1), as the browser must receive the complete HTML document before rendering the first visible content. Applying Gzip compression substantially reduced this overhead, with the SSR+Gzip configuration achieving an FCP of approximately 1.31 s even for the XL payload.

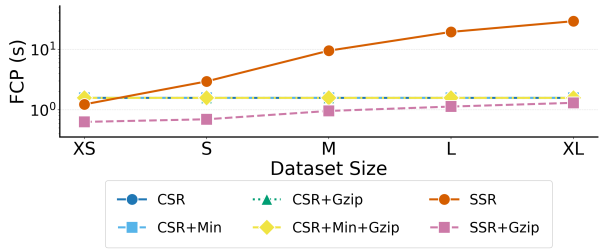


Figure 7. FCP values.

Speed Index. The SI metric measures the speed at which content is visually displayed during page loading. Lighthouse classifies values up to 3.4 s as fast, between 3.4 s and 5.8 s as moderate, and above 5.8 s as slow. Figure 8 presents a line chart with the SI results. For smaller dataset sizes, SI followed the same trend observed for the LCP and FCP metrics, with both rendering methods exhibiting comparable loading times. As dataset size increases, however, SSR shows a marked and progressive performance degradation, with SI values growing substantially faster than those of CSR, which remains comparatively stable across all dataset sizes. This divergence indicates that, unlike client-side rendering, SSR's visual loading speed is highly sensitive to dataset size.

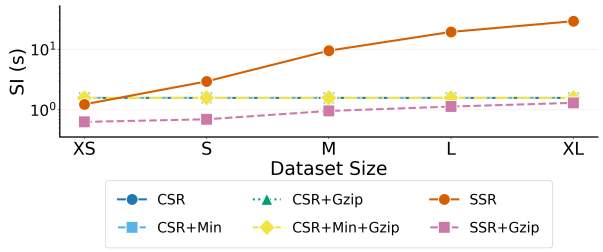


Figure 8. SI values.

Time to Interactive. The TTI metric measures how long it takes for the page to become fully interactive. Lighthouse considers values up to 3.4 s as fast, between 3.4 s and 5.8 s as moderate, and above 5.8 s as slow. Figure 9 presents a line chart with the TTI results. SSR consistently achieved shorter TTI than uncompressed CSR across all dataset sizes, with the gap widening as data volume increases. Notably, applying Gzip compression to CSR substantially narrows this gap: both CSR+Gzip and CSR+Min+Gzip track much closer to the SSR curves, indicating that compression plays a decisive role in bringing CSR's interactivity performance in line with server-side rendering.

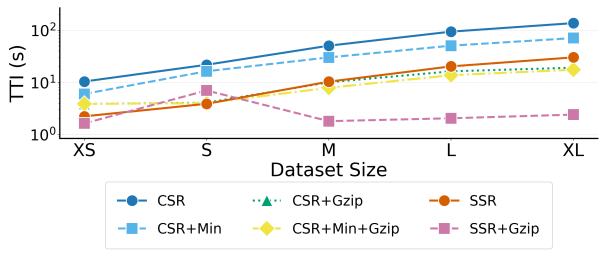


Figure 9. TTI values.

JS Heap Size. The *JS Heap Size* metric refers to the memory space allocated for storing JavaScript objects. There is no standardized quality threshold for this metric. Figure 10 presents a line chart with the JS Heap results. Both strategies allocate comparable amounts of memory for the smallest dataset, but they diverge sharply as data volume grows. CSR memory usage rises by more than an order of magnitude across the dataset sizes, whereas SSR remains essentially flat, staying near its initial level regardless of dataset size. Compression and minification have little effect on this behavior, since the CSR variants follow the same upward trend as plain CSR. This confirms that the memory cost is inherent to retaining the parsed dataset on the client, rather than a consequence of how the payload is transferred.

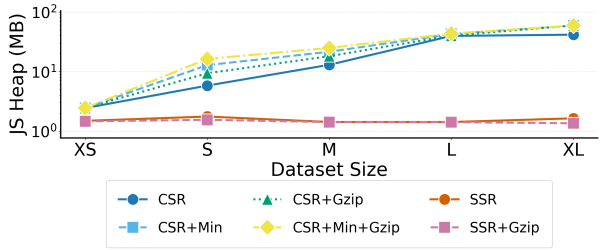


Figure 10. *JS Heap* memory usage.

5.1 Statistical Analysis

All performance metrics discussed in the previous sections were included in the statistical analysis. The Shapiro–Wilk test confirmed that the data did not follow a normal distribution, justifying the use of non-parametric statistical methods. Pairwise comparisons between the baseline CSR and SSR configurations were therefore performed using the Mann–Whitney U test, which is appropriate for two independent groups. Statistical significance was assessed at the 95% confidence level ($p \leq 0.05$). As discussed in Section 4.2, the experimental conditions whose loading time was dominated by payload transfer exhibited nearly deterministic measurements under Lighthouse simulated throttling. Consequently, the extreme p-values observed in these conditions should be interpreted as confirming systematic differences in the model estimates rather than as evidence derived from sampling variability.

Table 2 summarizes the Mann–Whitney U test results for the baseline experiment, comparing CSR and SSR across all payload sizes and evaluated metrics. Statistically significant differences ($p \leq 0.05$) were observed for every metric and payload size except JS Heap under the Small payload ($p = 0.140$). This result suggests that the memory overhead associated with the two rendering strategies becomes consistently distinguishable only as payload size increases.

To complement the significance analysis with a measure of practical importance, Cliff's Delta effect sizes were computed for every metric and payload size. The results are reported in Table 3. Large effect sizes were observed for LCP and TTI across all payload sizes ($|\delta| = 1.0$ from the S payload onward and $\delta = 0.58$–1.0 for the XS payload, where genuine execution-to-execution variability was present), indicating an almost complete separation between the CSR and SSR distributions. FCP also exhibited large effect sizes across all payload sizes, with negative δ values indicating

Table 2. Client Side vs Server Side (Mann–Whitney U) by data size.

Size	Metric	p-value
Extra Small	LCP	3.12e-02
	FCP	1.83e-04
	SI	1.83e-04
	TTI	3.12e-02
	JS Heap	1.11e-02
Small	LCP	1.83e-04
	FCP	1.83e-04
	SI	1.83e-04
	TTI	1.83e-04
	JS Heap	1.40e-01
Medium	LCP	1.83e-04
	FCP	1.83e-04
	SI	1.83e-04
	TTI	1.83e-04
	JS Heap	2.83e-03
Large	LCP	1.83e-04
	FCP	1.83e-04
	SI	1.83e-04
	TTI	1.83e-04
	JS Heap	4.59e-03
Extra Large	LCP	1.83e-04
	FCP	1.83e-04
	SI	1.83e-04
	TTI	1.83e-04
	JS Heap	1.01e-03

that SSR consistently achieved lower (i.e., faster) FCP values than CSR. For JS Heap, the effect size was large for all payload sizes except S, where a medium effect was observed ($\delta = 0.40$), consistent with the corresponding Mann–Whitney U test results.

5.2 Optimization Results

To evaluate the impact of the optimization techniques described in Section 4.6, the experiment was extended to include six experimental configurations: CSR without optimization (*baseline*), CSR with minification, CSR with Gzip compression, CSR with combined minification and Gzip compression, SSR without optimization (*baseline*), and SSR with Gzip compression. Table 4 presents the median values of LCP, FCP, TTI, and JavaScript Heap Size (JS Heap) for each configuration and payload size. Minification was not applied to SSR because the generated HTML output is already delivered in a compact representation and does not contain the same structural redundancies found in raw JSON payloads.

The results indicate that Gzip compression provided the largest improvements for LCP and TTI, particularly for the CSR approach. For the XL payload, the median LCP decreased from 136.88 s in the CSR baseline configuration to 18.22 s with Gzip compression (an 86.7% reduction) and to 16.54 s when combined with minification (an 87.9% reduction). Minification alone also produced a substantial improvement, reducing the median LCP from 136.88 s to 70.19 s (a 48.7% reduction). For SSR, Gzip compression reduced the median LCP from 29.99 s to 2.10 s for the XL payload, corresponding to a 93.0% reduction.

CSR optimizations produced no noticeable changes in the FCP metric. All CSR configurations maintained a stable FCP of approximately 1.58 s across all payload sizes, regardless of the applied optimization technique. This behavior is expected because CSR renders the initial page structure before requesting and processing application data through JavaScript. In contrast, payload size directly influenced SSR FCP because the browser must receive the generated HTML document before rendering the first visible content. Without Gzip compression, SSR reached an FCP of 29.28 s for the XL payload, whereas Gzip reduced this value to 1.40 s.

JavaScript Heap Size, measured on the evaluated page as described in Section 4.2, revealed a clear distinction between the two rendering strategies as payload size increased. SSR configurations remained nearly constant, ranging from 1.4 to 1.8 MB across all payload sizes, since the browser only maintains the rendered DOM and does not store the application dataset. Conversely, CSR configurations must retain the parsed JSON payload in memory, increasing from approximately 2.5 MB for the XS payload to 41.8–59.5 MB for the XL payload. Therefore, for the largest payload, CSR consumed approximately $30\times$ more JavaScript memory than SSR. This difference is substantially larger than in the previous version of this study because heap usage is now measured on the evaluated page rather than within the measurement process.

Table 5 presents the results of the Kruskal–Wallis test for the extended experiment involving the six experimental configurations. Statistically significant differences ($p < 0.05$) were observed for all metrics and payload sizes, including JavaScript Heap Size. This differs from the previous version of the study, in which JS Heap did not exhibit significant differences for smaller payloads. This change results from the correction in the heap measurement procedure: by measuring memory usage directly on the evaluated page, the systematic difference between configurations that retain the parsed JSON payload and those that do not becomes detectable across all payload sizes.

Pairwise Mann–Whitney U tests with Holm correction were applied to identify which configurations differed significantly. Tables 6–10 present the corresponding results. The most relevant comparisons (CSR vs. CSR+Gzip, CSR vs. CSR+Min+Gzip, and SSR vs. SSR+Gzip) showed statistically significant differences ($p < 0.05$) for LCP and TTI in nearly all payload sizes. The only exception was SSR vs. SSR+Gzip for the S payload, which was not statistically significant ($p = 0.819$) due to the bimodal distribution observed in this configuration, as previously discussed. For FCP, CSR variants differed significantly only for smaller payloads, where transfer time does not dominate and execution variability is observable. From the M payload onward, FCP comparisons among CSR configurations were not statistically significant, confirming the stability of this metric for the client-side approach.

Table 11 presents Cliff's Delta values for the most relevant comparisons: the impact of optimizations within each rendering strategy (CSR baseline vs. CSR with minification and Gzip; SSR baseline vs. SSR with Gzip) and the comparison between the best-performing configurations of each strategy (CSR with minification and Gzip vs. SSR with Gzip).

Table 3. Cliff's Delta effect sizes for the baseline experiment (CSR vs. SSR). Positive values indicate CSR > SSR. N = negligible ($|\delta| < 0.147$), S = small ($|\delta| < 0.33$), M = medium ($|\delta| < 0.474$), L = large ($|\delta| \geq 0.474$).

Metric	XS	S	M	L	XL
LCP	+0.58 (L)	+1.00 (L)	+1.00 (L)	+1.00 (L)	+1.00 (L)
FCP	+1.00 (L)	−1.00 (L)	−1.00 (L)	−1.00 (L)	−1.00 (L)
SI	+1.00 (L)	−1.00 (L)	−1.00 (L)	−1.00 (L)	−1.00 (L)
TTI	+0.58 (L)	+1.00 (L)	+1.00 (L)	+1.00 (L)	+1.00 (L)
JS Heap	+0.68 (L)	+0.40 (M)	+0.80 (L)	+0.76 (L)	+0.88 (L)

Table 4. Median values for each experimental condition and data size.

Metric	Size	CSR	CSR+min	CSR+gz	CSR+min+gz	SSR	SSR+gz
LCP (s)	XS	9.83	5.71	3.68	3.68	2.18	1.58
	S	20.86	15.46	3.90	3.83	3.83	6.83
	M	49.83	29.28	9.50	7.46	10.30	1.75
	L	93.66	49.85	15.40	12.98	20.22	1.93
	XL	136.88	70.19	18.22	16.54	29.99	2.10
FCP (s)	XS	1.58	1.58	1.58	1.58	1.23	0.63
	S	1.58	1.58	1.58	1.58	2.95	0.70
	M	1.58	1.58	1.58	1.58	9.51	0.96
	L	1.58	1.58	1.58	1.58	19.43	1.13
	XL	1.58	1.58	1.58	1.58	29.20	1.31
TTI (s)	XS	10.41	6.01	3.87	3.87	2.23	1.63
	S	21.82	16.39	4.07	4.00	3.88	7.00
	M	50.79	30.24	10.07	7.89	10.35	1.80
	L	94.62	50.81	16.33	13.77	20.34	2.05
	XL	137.86	71.18	19.21	17.52	30.28	2.41
***JS Heap* (MB)**	XS	2.5	2.5	2.5	2.5	1.5	1.5
	S	5.8	12.9	9.4	16.4	1.8	1.6
	M	13.0	21.6	18.2	25.0	1.4	1.4
	L	39.8	43.0	40.4	43.6	1.4	1.4
	XL	41.8	59.5	59.5	59.5	1.7	1.4

***** CSR = Client-Side, SSR = Server-Side, min = minification, gz = Gzip, XS = Extra Small, S = Small, M = Medium, L = Large, XL = Extra Large. *JS Heap* reported in MB, measured on the evaluated page.

Table 5. Kruskal–Wallis (p-value) for the six experimental conditions.

Metric	Size	p-value	Sig.
LCP	XS	1.50×10^{-7}	Yes
	S	6.83×10^{-9}	Yes
	M	4.23×10^{-11}	Yes
	L	4.23×10^{-11}	Yes
	XL	4.23×10^{-11}	Yes
FCP	XS	3.07×10^{-9}	Yes
	S	2.33×10^{-8}	Yes
	M	6.25×10^{-8}	Yes
	L	7.24×10^{-8}	Yes
	XL	3.81×10^{-8}	Yes
TTI	XS	1.50×10^{-7}	Yes
	S	3.42×10^{-8}	Yes
	M	4.23×10^{-11}	Yes
	L	4.23×10^{-11}	Yes
	XL	4.23×10^{-11}	Yes
JS Heap	XS	2.86×10^{-4}	Yes
	S	3.94×10^{-8}	Yes
	M	9.48×10^{-9}	Yes
	L	6.44×10^{-6}	Yes
	XL	7.76×10^{-7}	Yes

For LCP and TTI, effect sizes were large ($|\delta| \geq 0.86$, predominantly 1.0) for almost all payload sizes, indicating near-complete separation between the corresponding distributions. The main exceptions occurred for the S payload, where the bimodal SSR+Gzip distribution resulted in small to negligible effect sizes. For FCP, optimizations produced large effects for SSR ($\delta = 1.0$ for all payload sizes), whereas the effect of combined optimization on CSR decreased from large for XS to small from M onward, as FCP became stable.

For JavaScript Heap Size, optimizations within each rendering strategy produced negligible to small effects because compression affects transferred data rather than the amount of memory retained by the browser. Conversely, the comparison between optimized CSR and optimized SSR remained large across all payload sizes ($\delta \geq 0.76$), reflecting the structural memory cost of retaining parsed JSON data on the client. As discussed in Section 4.2, saturated effect sizes in payload-transfer-dominated conditions reflect the deterministic nature of the modeled measurements and should therefore be interpreted as magnitude separation rather than sampling-based effects.

The effect size analysis supports the hypotheses formulated in Section 4.6. Regarding **H1**, minification reduced the median CSR LCP by up to 48.7% (XL payload), with large effect sizes observed for LCP and TTI. However, its impact on FCP was limited to smaller payloads, indicating that minification improves overall loading time but does not significantly affect the initial rendering stage of CSR. Regarding **H2**, Gzip compression produced the largest reductions across all temporal metrics. Large effect sizes were observed for both CSR (an 86.7% LCP reduction) and SSR (a 93.0% LCP reduction for the XL payload). **H3** was partially confirmed: combining minification and Gzip in CSR reduced LCP by 87.9% compared with the baseline configuration. This improvement was greater than minification alone (48.7%) and only slightly higher than Gzip compression alone (86.7%).

Table 6. Pairwise Mann–Whitney U tests with Holm correction for LCP in the extended experiment (6 conditions). The asterisk (*) denotes statistical significance ($p < 0.05$).

Comparison	XS	S	M	L	XL
CSR vs CSR+Min	0.187	0.0027^*	0.0027^*	0.0027^*	0.0027^*
CSR vs CSR+Gzip	0.0131^*	0.0027^*	0.0027^*	0.0027^*	0.0027^*
CSR vs CSR+Min+Gzip	0.0131^*	0.0027^*	0.0027^*	0.0027^*	0.0027^*
SSR vs SSR+Gzip	0.0027^*	0.819	0.0027^*	0.0027^*	0.0027^*
CSR+Min+Gzip vs SSR+Gzip	0.0027^*	1.000	0.0027^*	0.0027^*	0.0027^*

Table 7. Pairwise Mann–Whitney U tests with Holm correction for FCP in the extended experiment (6 conditions). The asterisk (*) denotes statistical significance ($p < 0.05$).

Comparison	XS	S	M	L	XL
CSR vs CSR+Min	0.0144^*	0.320	1.000	1.000	0.810
CSR vs CSR+Gzip	0.0085^*	1.000	1.000	1.000	1.000
CSR vs CSR+Min+Gzip	0.0060^*	0.0680	1.000	1.000	1.000
SSR vs SSR+Gzip	0.0027^*	0.0027^*	0.0027^*	0.0027^*	0.0027^*
CSR+Min+Gzip vs SSR+Gzip	0.0027^*	0.0027^*	0.0027^*	0.0027^*	0.0027^*

The marginal benefit of adding minification on top of Gzip was limited (9.3% for the XL payload), as both techniques reduce similar redundancies in the transferred payload.

6 Discussion

While both rendering strategies demonstrated good performance with smaller datasets, significant differences emerged as data volume increased. These variations directly impact application efficiency. Specifically, SSR consistently yielded better results for LCP and TTI. This performance likely results from the server delivering a fully rendered HTML page. This approach allows the browser to display core content and become interactive rapidly without extensive client-side JavaScript processing.

Conversely, CSR exhibited superior performance in FCP and SI, especially with larger datasets. Its ability to dynamically render content once initial scripts are loaded minimizes the time to render visible elements. This distinction suggests that SSR is more advantageous for content-heavy applications where initial load and SEO are paramount. In contrast, highly interactive applications benefit from CSR's dynamic rendering capabilities after initial asset loading. Regarding *JS Heap* memory usage, we found no statistically significant differences in the initial tests, as shown in Table 2.

Cliff's Delta analysis complements the significance test results by revealing the practical magnitude of the differences. As presented in Table 11, the effect sizes for LCP and TTI were consistently large ($|\delta| = 1.0$) across all comparisons and data sizes. This confirms that performance differences between rendering approaches and optimization settings are not only statistically significant but also practically relevant. For the *JS Heap* metric, however, the Cliff's Delta values between variants of the same approach indicated small to medium effects. This occurred despite significant p-values in L and XL sizes (Table 5). This discrepancy between statistical significance and effect size illustrates a scenario where the p-value alone overestimates the practical difference in memory consumption. This finding reinforces the importance of reporting effect size measures in empirical software engineering research (Kitchenham and Madeyski, 2024; Kitchenham et al., 2017).

The optimization results demonstrate that Gzip compression was the most influential technique for improving performance in both rendering approaches. This technique reduced the median LCP for the largest payload by 86.7% in CSR and 93.0% in SSR. These findings are consistent with previous studies by Tiwary et al. (2021) and Tusa and Clayman (2021), which demonstrated the effectiveness of compression techniques in reducing transferred data volume and improving communication efficiency. When comparing the best-performing configurations of each approach (CSR with minification and Gzip versus SSR with Gzip), SSR maintained superior performance in terms of LCP and TTI, with large effect sizes across all payload sizes except S. In this case, the bimodal distribution observed in the measurement condition prevented a clear separation between the configurations (Section 5). However, optimized CSR achieved substantially lower absolute values than the CSR *baseline*, indicating that the combination of minification and compression significantly reduces the performance gap between CSR and SSR when adopting SSR is not feasible.

7 Threats to Validity

This section addresses the potential threats to validity across different dimensions and the strategies adopted to mitigate them (Wohlin et al., 2012).

Internal Validity. For internal validity, factors such as hardware and software influence on the stability of the results were considered. To mitigate these interferences, all tests were executed in a local environment, on the same machine and operating system. Additionally, each experiment was run ten times, ensuring that the observed results are consistent and not merely due to random chance or transient system variations. This repetition enhances the reliability of the findings by providing more robust and representative data averages.

The minification was applied only to the CSR JSON, without any counterpart in the SSR HTML responses. HTML minification involves a different mechanism and is outside the scope of this study, but the asymmetry limits direct comparisons between the minification effects in the two strategies. The conclusions about minification are therefore valid

Table 8. Pairwise Mann–Whitney U tests with Holm correction for SI in the extended experiment (6 conditions). The asterisk (*) denotes statistical significance ($p < 0.05$).

Comparison	XS	S	M	L	XL
CSR vs CSR+Min	0.0144^*	0.320	1.000	1.000	0.810
CSR vs CSR+Gzip	0.0085^*	1.000	1.000	1.000	1.000
CSR vs CSR+Min+Gzip	0.0060^*	0.0680	1.000	1.000	1.000
SSR vs SSR+Gzip	0.0027^*	0.0027^*	0.0027^*	0.0027^*	0.0027^*
CSR+Min+Gzip vs SSR+Gzip	0.0027^*	0.0027^*	0.0027^*	0.0027^*	0.0027^*

Table 9. Pairwise Mann–Whitney U tests with Holm correction for TTI in the extended experiment (6 conditions). The asterisk (*) denotes statistical significance ($p < 0.05$).

Comparison	XS	S	M	L	XL
CSR vs CSR+Min	0.187	0.0027^*	0.0027^*	0.0027^*	0.0027^*
CSR vs CSR+Gzip	0.0131^*	0.0027^*	0.0027^*	0.0027^*	0.0027^*
CSR vs CSR+Min+Gzip	0.0131^*	0.0027^*	0.0027^*	0.0027^*	0.0027^*
SSR vs SSR+Gzip	0.0027^*	0.819	0.0027^*	0.0027^*	0.0027^*
CSR+Min+Gzip vs SSR+Gzip	0.0027^*	1.000	0.0027^*	0.0027^*	0.0027^*

for CSR architectures. The CPU overhead imposed by Gzip on the server was also not measured, which may limit the generalization of latency gains in environments with restricted processing.

External Validity. Regarding external validity, this study faces a recognized limitation: the experiments occurred in a single execution environment. All experiments ran on one hardware configuration (a MacBook Air M2) and in one browser environment (Chromium). This restricted setup limits the generalizability of the findings because performance and behavior vary across hardware platforms, operating systems, and browsers with different rendering engines. Nevertheless, the chosen environment yielded relevant insights for the study proposal, which future work can expand.

The experiments were conducted on *localhost*, without network latency or bandwidth limitations, which likely underestimates the benefit of Gzip in real-world deployments. Ollila et al. (2022) found that performance differences between rendering strategies vary depending on the JavaScript engine, suggesting that the magnitudes reported here may not be replicated with other frameworks. The domain evaluated, a photo gallery with image metadata, represents a static content application; generalization to interactive SPAs, dashboards, or e-commerce is limited (Gangishetti and Jain, 2026). However, the constructed environment uses different images, which helps to approximate real-world applications.

Consequently, the absolute performance metrics obtained in this study cannot be directly extrapolated to real-world production scenarios. In a live environment, variables such as fluctuating mobile network conditions, geographical distance between clients and servers, and concurrent user traffic introduce significant variability that our controlled setup deliberately excluded. While this isolation was necessary to establish a clear baseline for comparing the rendering strategies, we acknowledge that the advantages and drawbacks of each approach might be amplified or mitigated in the wild. Therefore, our findings should be interpreted as an isolated theoretical baseline, highlighting the need for future empirical validations in deployed environments to confirm how these strategies perform under realistic user conditions.

Conclusion Validity. With respect to conclusion validity, several techniques exist to improve application performance, such as data minification and compression, asynchronous loading (lazy loading), the use of Content Delivery Networks (CDNs), image preloading, and data caching. This study directly investigates two of these techniques, minification and Gzip, in a controlled factorial experiment. To ensure that the evaluation focus remained on rendering performance, the application used only HTML, CSS, and JavaScript without any external libraries.

Pairwise Mann–Whitney U tests with Holm correction were applied to control the family-wise error rate in multiple comparisons (Wohlin et al., 2012). Cliff's Delta with bootstrap-based confidence intervals was used to estimate effect sizes (Efron and Tibshirani, 1994; Kitchenham and Madeyski, 2024). With $n = 10$ observations per experimental condition, the statistical power is sufficient only for detecting large effects ($|\delta| \geq 0.474$); therefore, the conclusions drawn from the inferential analysis are restricted to effects of this magnitude.

A more fundamental threat to conclusion validity arises from the measurement model itself (Section 4.2). Under Lighthouse simulated throttling, loading metrics are derived from a deterministic model with a fixed bandwidth constraint. Consequently, in experimental conditions dominated by payload transfer time, the observed LCP converges toward the modeled payload transfer time (e.g., 136.9 s measured versus 139.0 s theoretical transfer time for the 26 MB payload under the simulated 1.47 Mbps downlink), and execution-to-execution variability approaches zero. In these conditions, the near-complete separation between configurations ($|\delta| = 1$) reflects the deterministic nature of the measurement model rather than a sampling-based effect. Therefore, we interpret this experiment as a reproducible laboratory benchmark under a standardized profile: low variability indicates consistency of the modeled estimates, differences between configurations are interpreted primarily as magnitude comparisons, and inferential statistics are emphasized only for conditions exhibiting genuine execution variability (payloads below 2 MB, where transfer time does not dominate the measurements).

Construct Validity. The metrics analyzed in this study (LCP, FCP, TTI, and JS Heap) have documented correlations with user-perceived performance and behavioral outcomes, including abandonment and conversion rates (Sikder, 2016; Developers, 2024). The remaining metrics collected

Table 10. Pairwise Mann–Whitney U tests with Holm correction for JS Heap in the extended experiment (6 conditions). The asterisk (*) denotes statistical significance ($p < 0.05$).

Comparison	XS	S	M	L	XL
CSR vs CSR+Min	1.000	0.0026^*	0.0069^*	1.000	1.000
CSR vs CSR+Gzip	1.000	0.0079^*	0.0364^*	1.000	1.000
CSR vs CSR+Min+Gzip	1.000	0.0026^*	0.0027^*	1.000	1.000
SSR vs SSR+Gzip	1.000	0.484	0.623	1.000	1.000
CSR+Min+Gzip vs SSR+Gzip	0.0582	0.0026^*	0.0027^*	0.0027^*	0.0027^*

Table 11. Cliff's Delta (δ) for key comparisons. Positive values indicate that the first group exhibits larger values. Effect size categories: N = negligible, S = small, M = medium, L = large.

		CSR vs. CSR+min+gz		SSR vs. SSR+gz		CSR+min+gz vs. SSR+gz	
Metric	**Size.**	δ	**Eff.**	δ	**Eff.**	δ	**Eff.**
LCP	XS	0.86	L	1.00	L	1.00	L
	S	1.00	L	0.30	S	0.00	N
	M	1.00	L	1.00	L	1.00	L
	L	1.00	L	1.00	L	1.00	L
	XL	1.00	L	1.00	L	1.00	L
FCP	XS	0.88	L	1.00	L	1.00	L
	S	0.68	L	1.00	L	1.00	L
	M	0.20	S	1.00	L	1.00	L
	L	−0.30	S	1.00	L	1.00	L
	XL	−0.28	S	1.00	L	1.00	L
TTI	XS	0.86	L	1.00	L	1.00	L
	S	1.00	L	0.30	S	0.00	N
	M	1.00	L	1.00	L	1.00	L
	L	1.00	L	1.00	L	1.00	L
	XL	1.00	L	1.00	L	1.00	L
JS Heap	XS	−0.16	S	0.26	S	0.76	L
	S	−1.00	L	0.33	M	1.00	L
	M	−1.00	L	0.14	N	1.00	L
	L	−0.14	N	0.12	N	1.00	L
	XL	−0.18	S	0.32	S	1.00	L

by the auditing pipeline (SI, CLS, FID, and Lighthouse Performance Score) were excluded from the analysis for the reasons discussed in Section 4.3, and no conclusions in this study rely on them. The four analyzed metrics capture loading performance and memory consumption, but they do not characterize interaction quality throughout the user session. Assessing this dimension would require the INP metric, which cannot be reliably obtained under the measurement model adopted in this study. The mapping of `format=html` to SSR and `format=json` to CSR represents a controlled abstraction of these rendering paradigms. Modern *frameworks*, such as Next.js, Nuxt, React, and Svelte, may introduce additional mechanisms, including streaming and partial rendering, resulting in different performance characteristics (Ollila et al., 2022). Therefore, the results should be interpreted as a characterization of the *trade-offs* between server-side prerendered HTML and client-side JSON rendering under controlled conditions, rather than as a direct evaluation of production *frameworks*.

8 Conclusion

This study compared the performance between two types of rendering in web applications, using Google's Core Web Vitals metrics as a quality assessment approach. For the analysis, both CSR and SSR rendering strategies were evaluated. A photo gallery application was developed for each rendering approach, and aspects such as performance, user experience, and memory usage were compared.

For data collection, a Node.js script was created using the Lighthouse performance measurement tool. The Kruskal–Wallis statistical test was applied to compare the results and verify the presence of statistically significant differences. The tests indicated relevant differences. The study also considers two factors: JSON minification and HTTP compression via Gzip. The experiment encompassed 30 treatment conditions and 300 observations, covering the five data size ranges in all evaluated configurations.

Gzip compression was the most impactful optimization evaluated in this study. It reduced the median LCP of CSR by 86.7% for the Extra-Large payload (from 136.88 s to 18.22 s) and reduced the median LCP of SSR by 93.0% (from 29.99 s to 2.10 s). JSON minification also provided substantial reductions in CSR LCP, ranging from 41% to 49% depending on payload size. However, the additional benefit of combining minification with Gzip compression was limited, providing an incremental improvement of approximately 9% for the Extra-Large payload. SSR with Gzip consistently outperformed all CSR configurations in terms of LCP and TTI, with the performance gap increasing as payload size grew: it was $2.3\times$ faster than CSR with minification and Gzip for the Extra-Small payload and $7.9\times$ faster for the Extra-Large payload. The S payload was the only exception, due to the bimodal distribution observed in that experimental condition (Section 5). The effect size analysis further supports this separation, with large Cliff's Delta values ($|\delta| \geq 0.86$) for comparisons between SSR+Gzip and CSR configurations (Cliff, 1993; Romano et al., 2006).

The FCP in the CSR remained around 1.58 s regardless of the use of Gzip or minification, indicating that this metric reflects the loading of the initial script and is not sensitive

to data transfer optimizations. To assess the impact of these techniques, LCP and TTI are the appropriate metrics (Verma and Aland, 2024).

Returning to the research questions, regarding **RQ1**, increasing payload size degraded LCP and TTI in both rendering strategies, approximately following the growth in transferred data volume. However, this degradation was substantially more pronounced for CSR: median LCP increased from 9.83 s to 136.88 s between the XS and XL payloads, compared with an increase from 2.18 s to 29.99 s for SSR. In contrast, CSR maintained a stable FCP across payload sizes, while its JavaScript Heap usage increased with payload growth (from 2.5 MB to 41.8 MB), whereas SSR memory consumption remained nearly constant. Regarding **RQ2**, Gzip compression was the dominant optimization technique, reducing LCP by up to 86.7% for CSR and 93.0% for SSR. JSON minification also provided meaningful isolated improvements, but its additional contribution when combined with Gzip was limited. Regarding **RQ3**, under optimized conditions, SSR with Gzip achieved the lowest LCP and TTI values for all payload sizes, except for the S payload anomaly discussed in Section 5. Furthermore, the performance advantage of SSR increased as payload size grew.

Gzip compression should be enabled in any application that uses SSR or CSR, given its significant impact with low implementation cost (Sikder, 2016). JSON minification is useful as a complementary technique in CSR, especially when Gzip is not available. When LCP and TTI are the priority metrics, as in content platforms with larger data volumes (Gangishetti and Jain, 2026), SSR with Gzip offers a consistent advantage over any CSR configuration. In applications with small payloads where FCP is critical, optimized CSR maintains competitive performance.

These recommendations apply to the controlled implementation evaluated in this study (client-side JSON rendering versus server-side pre-rendered HTML) and should not be generalized, without further validation, to modern production frameworks that rely on hydration, streaming SSR, or partial/island rendering architectures, as discussed in Section 7.

As future work, we intend to investigate the impact of HTML minification on SSR performance, an aspect not explored in this study, to provide a more symmetric comparison of the effects of minification across both rendering strategies. Further research should also evaluate alternative rendering approaches commonly adopted in modern web applications, such as Incremental Static Regeneration (ISR) and isomorphic architectures. Since this study relies on Lighthouse simulated throttling under a single reference profile (Section 4.2), reproducing the experiment with applied throttling through browser-level network simulation (e.g., Chrome DevTools) and additional network profiles would provide directly observed measurements rather than model-based estimates. This would enable a more comprehensive analysis of how the performance advantage provided by Gzip compression varies under different connectivity conditions.

Measuring the server-side CPU overhead introduced by Gzip compression under high-concurrency workloads would further complement the cost-benefit analysis of compression techniques. Replicating the study using production frameworks, such as Next.js, Nuxt, SvelteKit, and React, across multiple browsers would also provide additional evidence regarding the applicability of the findings beyond the controlled environment evaluated in this work (Ollila et al., 2022). Finally, future studies should consider a broader range of application domains, including e-commerce platforms, analytical dashboards, and complex single-page applications, to further assess the generalizability of the observed trade-offs.

ARTIFACT AVAILABILITY

The *scripts* and *dataset* are available at: `https://github.com/CleitonSilvaT/jserd-2026-csr-vs-ssr`

References

Carvalho, F. M. (2024). Progressive server-side rendering with suspendable web templates. In *Web Information Systems Engineering – WISE 2024: 25th International Conference, Doha, Qatar, December 2–5, 2024, Proceedings, Part V*, page 458–473, Berlin, Heidelberg. Springer-Verlag.

Carvalho, F. M. and Fialho, P. (2023). Enhancing ssr in low-thread web servers: A comprehensive approach for progressive server-side rendering with any asynchronous api and multiple data models. In *International Conference on Web Information Systems and Technologies*.

Clapton, J., Bastos, H., Batisteli, J. P. O., Seufitelli, D. B., and Tavares, C. (2025). Client-side vs server-side rendering: Impacts on performance and user experience in web applications. In *Simpósio Brasileiro de Qualidade de Software (SBQS)*, pages 165–172. SBC.

Cliff, N. (1993). Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494–509.

da Silva, W. O. and Farah, P. R. (2018). Characteristics and performance assessment of approaches pre-rendering and isomorphic javascript as a complement to spa architecture. In *Proceedings of the XII Brazilian Symposium on Software Components, Architectures, and Reuse*, SBCARS '18, page 63–72, New York, NY, USA. Association for Computing Machinery.

Developers, G. (2024). Explorando e aprendendo sobre os core web vitals. https://web.dev/explore/learn-core-web-vitals?hl=pt-br, Accessed: 2024-11-03.

Dymora, P., MAZUREK, M., and Nycz, M. (2023). Comparison of angular, react, and vue technologies in the process of creating web applications on the user interface side. *Journal of Education, Technology and Computer Science*, 4:210–222.

Efron, B. and Tibshirani, R. J. (1994). *An Introduction to the Bootstrap*. Chapman & Hall/CRC.

Gangishetti, S. and Jain, V. (2026). Comparative analysis of client-side vs. server-side rendering for large-scale content platforms. *International Journal of AI, BigData, Computational and Management Studies*, 7(1):74–80.

Gil, B. and Trezentos, P. (2011). Impacts of data interchange formats on energy consumption and performance in smart-

phones. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication*, OSDOC '11, page 1–6, New York, NY, USA. Association for Computing Machinery.

Google (2024a). First contentful paint (FCP). https://web.dev/articles/fcp, Accessed: 2026-04-05.

Google (2024b). Largest contentful paint (LCP). https://web.dev/articles/lcp, Accessed: 2026-04-05.

Huber, S., Demetz, L., and Felderer, M. (2022). A comparative study on the energy consumption of progressive web apps. *Information Systems*, 108:102017.

Iskandar, T. F., Lubis, M., Kusumasari, T. F., and Lubis, A. R. (2020). Comparison between client-side and server-side rendering in the web development. In *IOP Conference Series: Materials Science and Engineering*, volume 801, page 012136. IOP Publishing.

Kitchenham, B. and Madeyski, L. (2024). Recommendations for analysing and meta-analysing small sample size software engineering experiments. *Empirical Software Engineering*, 29(6):137.

Kitchenham, B., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., and Pohthong, A. (2017). Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, 22(2):579–630.

Korobeinikova, T., Maidaniuk, V., Romanyuk, O., Chekhmestruk, R., Romanyuk, O., and Romanyuk, S. (2022). Web-applications fault tolerance and autoscaling provided by the combined method of databases scaling. In *2022 12th International Conference on Advanced Computer Information Technologies (ACIT)*, pages 27–32. IEEE.

Langdale, G. and Lemire, D. (2019). Parsing gigabytes of json per second. *The VLDB Journal*, 28:941–960.

Li, Y., Katsipoulakis, N. R., Chandramouli, B., Goldstein, J., and Kossmann, D. (2017). Mison: a fast json parser for data analytics. *Proc. VLDB Endow.*, 10(10):1118–1129.

Moghadam, M. H. e. a. (2023). Machine learning to guide performance testing: An autonomous test framework. *Research Institutes of Sweden (RISE) SICS, Sweden; Mälardalen University, Västerås, Sweden*.

Nurseitov, N., Paulson, M., Reynolds, R., and Izurieta, C. (2009). Comparison of json and xml data interchange formats: a case study. *Caine*, 9:157–162.

Ollila, R., Mäkitalo, N., and Mikkonen, T. (2022). Modern web frameworks: A comparison of rendering performance. *Journal of Web Engineering*, 21(3):789–813.

Romano, J., Kromrey, J. D., Coraggio, J., and Skowronek, J. (2006). Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and similar surveys? In *Annual Meeting of the Florida Association of Institutional Research*, Cocoa Beach, FL.

Sikder, A. S. (2016). A pioneering investigation into web performance optimization's impact on user experience and business success. *International Journal of Imminent Science and Technology*, 1:186–198.

Tiwary, G. P., Stroulia, E., and Srivastava, A. (2021). Compression of xml and json api responses. *IEEE Access*, 9:57426–57439.

Torkar, R., Furia, C. A., Feldt, R., Oliveira Neto, F. G. d., Gren, L., Lenberg, P., and Ernst, N. A. (2022). A method to assess and argue for practical significance in software engineering. *IEEE Transactions on Software Engineering*, 48(6):2053–2065.

Tusa, F. and Clayman, S. (2021). The impact of encoding and transport for massive real-time iot data on edge resource consumption. *Journal of Grid Computing*, 19(32).

van Riet, J., Malavolta, I., and Ghaleb, T. A. (2023). Optimize along the way: An industrial case study on web performance. *Journal of Systems and Software*, 198:111593.

Verma, D. and Aland, P. (2024). A comparative review of server rendering and client side rendering in web development. *International Journal of Scientific Research and Engineering Trends*, 10(2):521–530.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

Zhu, F., Xu, L., Ma, G., Ji, S., Wang, J., Wang, G., Zhang, H., Wan, K., Wang, M., Zhang, X., Wang, Y., and Li, J. (2022). An empirical study on quality issues of ebay's big data sql analytics platform. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, page 33–42, New York, NY, USA. Association for Computing Machinery.