

Higher Education Diploma in Software Engineering, with Emphasis
in Web Programming
May 2025
urn:nbn:se:bth-28018



Evaluating Server-Side and Client-Side Rendering for Content-Heavy Web Applications

Teophil Ricardo Simao

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Higher Education Diploma in Software Engineering, with Emphasis in Web Programming. The thesis is equivalent to 10 weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author(s):

Teophil Ricardo Simao

E-mail: tesi23@student.bth.se

University advisor:

Victor KEBANDE

Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

This thesis researches the performance and SEO implications of Server-Side Rendering (SSR) and Client-Side Rendering (CSR) in content-heavy web applications. Using the Next.js 15 framework, two identical applications, one rendered via SSR and the other via CSR, were developed and tested across key performance metrics, Time to First Byte (TTFB), Largest Contentful Paint (LCP), and Interaction to Next Paint (INP). Tests were conducted under both fast (WiFi) and throttling (3G) network conditions. SEO performance was evaluated using both automated tools, Lighthouse and Screaming Frog, and manual inspection.

Results showed that while CSR achieved slightly better average LCP under ideal conditions, it exhibited high variability and severe performance degradation on slower networks. SSR demonstrated more consistent performance and significantly better LCP under throttled conditions. In terms of SEO, both approaches achieved perfect Lighthouse scores, but only SSR pre-rendered full content and media assets in HTML, enabling more complete indexing by search engines. CSR's reliance on JavaScript delayed content visibility to crawlers, potentially hindering SEO.

The study concludes that SSR is more suitable for content-heavy applications where consistent performance and robust SEO are priorities, whereas CSR is more viable in controlled, app-like environments with stable network conditions.

Keywords: Web performance, SEO, Server-side rendering (SSR), Client-side rendering (CSR), Core Web Vitals.

Contents

Abstract	i
1 Introduction	1
1.1 Background	1
1.2 Scope	2
1.3 Outline	3
2 Related Work	5
3 Method	7
3.1 Research questions	7
3.2 Research method	8
3.3 Validity and reliability	9
3.3.1 Data Collection and Working Conditions	9
3.3.2 Validity	9
3.3.3 Reliability	10
3.3.4 Limitations	10
4 Architecture	13
4.1 Structure	13
4.2 Rendering	13
4.2.1 Server-Side Rendering	13
4.2.2 Client-Side Rendering	14
4.3 Component Design	14
4.3.1 AlbumCard	14
4.3.2 MetricScript	15
4.3.3 GetAlbums	16
4.3.4 InteractionButton	16
4.4 Site visuals	17
4.5 Network conditions	17
5 Results and Analysis	19
5.1 Results	19
5.1.1 Web -Vitals Performance Results	19
5.1.2 SEO results	21
5.2 Analysis	23
5.2.1 Performance Analysis (RQ1)	23
5.2.2 SEO and Indexing Implications (RQ2)	24

5.2.3	When to Choose SSR over CSR (RQ3)	24
6	Discussion	27
6.1	Architectural Impact on Performance	27
6.2	Research Questions	27
6.3	Comparison to Prior Work	28
6.4	Validity of Results	29
6.5	Limitations	29
6.6	Significance and Impact	30
6.7	Ethical Considerations	31
6.8	Summary	31
7	Conclusions and Future Work	33
7.1	Conclusions	33
7.1.1	Performance conclusion	33
7.1.2	SEO conclusion	34
7.1.3	General Conclusion	34
7.2	Future Work	34
	References	37
A	Supplemental Information	39

1.1 Background

Web development has evolved greatly over the years, especially in how websites are rendered and displayed to users. Traditionally, most websites used Server-Side Rendering (SSR), where the server generates the full HTML page and sends it to the browser [15]. More recently, with the rise of JavaScript frameworks like React, Client-Side Rendering (CSR) has become common. In CSR, the browser receives a minimal HTML page and uses JavaScript to build the content dynamically on the client side [13]. These two rendering strategies have a significant impact on website performance, SEO, and user experience [1, 16].

Content-heavy websites, such as news sites, blogs, e-commerce platforms, and documentation pages, face specific challenges. These include the need to deliver content quickly, appear clearly in search engine results, and provide smooth user interactions [1, 16]. The choice between SSR and CSR can directly affect these factors, so it is important to compare their effectiveness for such use cases.

In recent years, SSR has gained popularity again with frameworks like Next.js, which supports both SSR and CSR [14]. When using SSR in Next.js, the server sends fully rendered HTML to the browser. This improves initial load performance, especially metrics such as Largest Contentful Paint (LCP), and allows search engines to crawl content more easily [6, 8, 15]. Next.js's App Router lets developers choose different rendering methods for different pages or components. When deployed on hosting platforms such as Vercel, SSR benefits from edge caching and serverless functions to further enhance speed and scalability [15, 19].

On the other hand, CSR is handled entirely in the browser (client side). This method allows for dynamic, interactive interfaces, ideal for dashboards or apps with frequent updates, but can lead to slower initial loads (e.g., higher Time to First Byte (TTFB) and delayed LCP) [8, 9, 13, 19]. CSR may also pose SEO challenges because content is not immediately available for crawlers. However, techniques like progressive hydration in Next.js are designed to reduce these issues by rendering parts of the page incrementally [19].

To fairly compare SSR and CSR, developers use Core Web Vitals, a set of metrics developed by Google to assess user experience and performance [6]. The main

metrics included in this research are:

- **Time to First Byte (TTFB)**: Measures how quickly the server starts sending data [9].
- **Largest Contentful Paint (LCP)**: Measures how fast the main content appears on screen [8].
- **Interaction to Next Paint (INP)**: Measures responsiveness based on input delay (e.g., clicks or taps) [7].

These metrics are useful for evaluating content-heavy websites, where fast performance and responsiveness are critical.

Finally, SEO plays a major role for such websites. SSR has traditionally been better for SEO, since it delivers full HTML to search engines [15]. However, CSR has improved in recent years thanks to modern tools like Lighthouse, Next.js SEO features, and better crawler support for JavaScript content [3, 4, 13]. Hosting platforms like Vercel further help by ensuring fast delivery and caching, contributing to both performance and SEO [19].

1.2 Scope

The scope of this research is defined to ensure relevance, focus, and applicability. The study exclusively targets content-heavy web applications, which are characterized by a large volume of text- and media-rich content that users must access quickly and reliably. These applications include news websites, documentation portals, and blog platforms. Scenarios where content discoverability, fast load times, and SEO performance are important for content-heavy web applications [1, 13].

To maintain technical consistency, Next.js version 15 with the App Router will be used for all implementations [14], deployed on Vercel, a platform optimized for Next.js that provides edge caching and serverless functions for production-grade performance [19]. This version of Next.js is chosen for its support of advanced rendering patterns, including SSR [15], CSR [13], and hybrid features like selective rendering and progressive hydration [19]. The use of this modern framework ensures that the research findings are up to date and applicable to the tools commonly used in industry settings today.

The study will focus on three key web performance metrics as defined by Google's Core Web Vitals initiative [6]:

- **Time to First Byte (TTFB)**: Indicates the server's responsiveness and measures how long it takes to receive the first byte of data [9].
- **Largest Contentful Paint (LCP)**: Measures the time taken for the largest visible element to render on the page, reflecting perceived load speed [8].

- **Interaction to Next Paint (INP)**: Captures the latency of user interactions, measuring how responsive an application feels during use [7].

Each metric has direct connection for SEO, user satisfaction, and perceived performance [6]. The study will test these metrics across both SSR and CSR implementations to determine clear, actionable performance differences.

The research will be experimental and comparative, involving the development of a real world like application containing at least 100 content items. Both SSR and CSR versions of this application will be created, using identical content and layout. The application data will be sourced from the MusicBrainz API [12], together with album artwork from the Cover Art Archive API [2]. Tests will be run on a MacBook Pro 2019 Intel, as well as under WiFi and simulated 3G conditions, to replicate realistic usage scenarios using the Google Chrome browser [3,4].

For SEO evaluation, tools like Lighthouse [4] and Screaming Frog SEO Spider [18] will be used to assess crawlability, structured data interpretation, and mobile readiness [5]. These assessments are important for content-heavy applications that rely heavily on organic search traffic.

Importantly, this study excludes several aspects to maintain a focused approach. These include:

- Marketing strategy implications of SEO
- Security concerns related to rendering strategies
- Backend specific performance considerations
- Broader UX metrics beyond those chosen for this study

Additionally, while the test application will be designed to reflect realistic usage patterns, it will not be connected to a live backend service. Instead, standardized datasets will simulate dynamic content delivery to allow for data reproduction and control.

By defining the scope in this way, the study ensures that the findings are technically strict, relevant to industry needs, and applicable to similar development environments in many companies. Especially those leveraging Next.js and Vercel for content-heavy applications.

1.3 Outline

This thesis is organized into seven chapters, each building upon the previous to examine the comparative performance, SEO capabilities, and practical compromises between Server-Side Rendering (SSR) and Client-Side Rendering (CSR) for content-heavy websites.

- **Chapter 1: Introduction** – Introduces the topic of rendering strategies in modern web development, outlines the scope of the study, and describes the motivation for comparing SSR and CSR in performance and SEO contexts.
- **Chapter 2: Related Work** – Examine prior academic and industry studies related to web performance, SEO, and rendering approaches, and identifying existing gaps this study addresses.
- **Chapter 3: Method** – Describes the research methodology used in the study, including the design science research approach, research questions, testing procedures, and strategies to ensure validity and reliability.
- **Chapter 4: Architecture** – Details the technical structure of the test applications, explaining how SSR and CSR were implemented using Next.js. It includes descriptions of component design, rendering logic, and network simulation.
- **Chapter 5: Results and Analysis** – Presents the results of performance and SEO testing under different network conditions. The chapter analyzes key metrics including TTFB, LCP, and INP, and compares SSR and CSR implementations using data tables, tools, and visual inspection.
- **Chapter 6: Discussion** – Interprets the results in the context of the research questions. Discusses implications for developers, compares findings to existing literature, considers limitations, ethical dimensions, and broader relevance in the field of web development.
- **Chapter 7: Conclusions and Future Work** – Summarizes the main findings of the study regarding performance and SEO trade-offs between SSR and CSR. Proposes directions for future research, including hybrid rendering approaches, extended testing conditions, and long-term SEO tracking.
- **References and Appendix** – Provides a comprehensive list of cited sources and additional materials including source code links, deployment URLs, and supporting datasets used in the study.

The performance and SEO implications of server-side rendering (SSR) versus client-side rendering (CSR) have been widely explored in both academic literature and industry documentation. This research builds on several key studies and practical findings to establish a clear understanding of where existing knowledge stands and how this project contributes further. BTH’s library tool and Google Scholar were used to find relevant literature for this study, the main focus in the literature search was to find other studies that compares SSR and CSR with regard to performance, SEO and user experience for content-heavy websites.

A foundational comparison of SSR and CSR approaches is presented by Nordström and Dixelius [16], who analyzed load time and performance variations between the two examples. Their study highlights that SSR often results in better perceived load performance, especially in content-dense applications. Similarly, Beke [1] evaluated software quality attributes like performance, maintainability, and SEO friendliness, concluding that SSR generally offers advantages for SEO and initial content visibility, although CSR may provide better client-side interactivity.

Iskandar et al. [11] contributed further to this discourse by experimentally comparing CSR and SSR using real-world testing setups. Their findings reinforce the idea that rendering strategy must be chosen based on application context—CSR may excel in highly interactive single-page applications (SPAs), while SSR is more suitable for content-heavy platforms requiring optimal crawlability and fast first paints.

Google’s Web Vitals initiative formalized the performance metrics most relevant to user experience, namely Time to First Byte (TTFB), Largest Contentful Paint (LCP), and Interaction to Next Paint (INP) [6–9]. These metrics are now standard across SEO and performance optimization tools such as Lighthouse [4] and WebPageTest [20], and are central to this study’s benchmarking approach.

From a technical implementation standpoint, the Next.js framework (version 15) plays a critical role in enabling controlled experiments. Its documentation outlines native support for SSR [15], CSR [13], and hybrid approaches such as progressive hydration and selective rendering [19]. Vercel, the hosting platform used in this study, further optimizes rendering strategies by offering features like edge caching and serverless function execution, ensuring real-world deployment scenarios are reflected [19].

Unlike previous studies that often relied on synthetic content or theoretical models, this research distinguishes itself by leveraging real-world data sourced from the MusicBrainz API [12] and enriched via the Cover Art Archive API [2]. This introduces dynamic, media-rich content into the experiment, aligning closely with the target category of content-heavy applications such as news portals and documentation platforms.

Additionally, while most prior comparisons were limited to performance metrics, this research also incorporates SEO evaluations using both Lighthouse [4] and Screaming Frog SEO Spider [18], giving a broader view of web visibility and discoverability in rendering contexts.

Together, these prior works and toolsets form the backbone of the methodological and technical choices in this study. By integrating and extending these foundations, this research aims to provide more context-specific and actionable insights for developers and organizations building content-heavy web applications.

3.1 Research questions

As established in the introduction, this study aims to address three primary research questions that guide the investigation of rendering techniques for content-heavy web applications:

RQ1: How does the performance of content-heavy web applications differ between SSR and CSR implementations in Next.js, specifically regarding TTFB, LCP, and INP metrics?

This question focuses on quantifying performance differences between SSR and CSR using standardized web performance metrics. Time to First Byte (TTFB) measures server response time [9], Largest Contentful Paint (LCP) reflects perceived loading speed [8], and Interaction to Next Paint (INP) captures input responsiveness [7]. These metrics, as defined by Google's Core Web Vitals [6], offer a comprehensive view of user experience and are widely recognized by the industry. By analyzing these indicators across both rendering strategies, the study seeks to establish measurable, performance characteristics under real world conditions.

RQ2: How do SSR and CSR impact SEO and indexing for content-heavy web applications?

This question investigates how different rendering methods affect search engine visibility and content indexing. Although SSR is often assumed to offer superior SEO due to pre-rendered HTML content [15], modern search engines are increasingly capable of parsing JavaScript-rendered content [3]. The research will test indexing speed, structured data accuracy, and crawlability using tools such as Lighthouse [4] and Screaming Frog SEO Spider [18], providing data to support or contradict these assumptions.

RQ3: What specific performance thresholds should trigger the selection of SSR over CSR for content-heavy websites?

This question aims to produce concrete guidelines for when SSR should be favored over CSR. It will identify both absolute thresholds, based on Core Web Vitals benchmarks [6], and relative comparisons derived from experimental results. For example,

if SSR consistently reduces LCP by a significant percentage under slower network conditions, it may be recommended for similar use cases. This approach is intended to help developers make informed rendering decisions based on measured evidence.

Together, these research questions aim to close an existing gap in the literature by offering a comparative and practical evaluation of rendering strategies, grounded in real world performance, SEO effectiveness, and technical guidance relevant to modern frameworks such as Next.js.

3.2 Research method

The DSR methodology, as introduced by Peffers et al. [17] and further developed by Wieringa [21] and Gregor & Hevner [10], follows an iterative process composed of six main stages. This study implements the methodology as follows:

1. **Problem Identification and Motivation:** The research addresses the challenge of selecting the appropriate rendering technique for content-heavy web applications. This issue is relevant to many modern web development teams seeking to optimize performance and SEO in high content environments.
2. **Definition of Solution Objectives:** The study aims to:
 - Compare the performance of SSR and CSR using Core Web Vitals metrics.
 - Assess the SEO and indexing behavior of each approach.
 - Define practical threshold values for choosing one rendering strategy over the other.
3. **Design and Development:** A content-heavy application is developed twice—once using SSR and once using CSR—both built with the Next.js App Router framework [14]. Each version uses the same data, layout, and UI components to ensure comparability.
4. **Demonstration:** Both implementations are deployed to Vercel and tested in controlled environments. This stage validates that the applications are functional and meet baseline usability and performance expectations [19].
5. **Evaluation:** The SSR and CSR implementations are measured and analyzed using metrics TTFB, LCP, and INP [6–9]. SEO performance is assessed using tools like Lighthouse [4] and Screaming Frog [18], focusing on crawlability, indexing speed, and structured data interpretation.
6. **Communication:** Findings from the research are presented in this thesis and can serve as practical guidance for web developers and teams making architectural decisions in content-heavy projects.

This structured methodology ensures scientific rigor while maintaining practical relevance, supporting both academic contribution and real-world application.

3.3 Validity and reliability

Ensuring the validity and reliability of this study was a central concern throughout the research design and execution phases. The chosen methodology, Design Science Research (DSR), supports methodological clarity by enforcing a systematic structure that aligns research activities with clearly defined goals [10, 17, 21]. Methodological decisions were further shaped by the practicalities of evaluating real world web applications under controlled, repeatable conditions.

3.3.1 Data Collection and Working Conditions

Data was collected through multiple rounds of testing using standardized, browser based tools such as Chrome DevTools [3], Lighthouse [4], and Web-vitals [6]. By automating tests for each rendering implementation (SSR and CSR), the study minimized human error and ensured consistent measurement. Tests were conducted using a MacBook Pro (2019) under stable WiFi and throttled 3G conditions, reflecting realistic user environments for desktop users.

To isolate rendering technique as the independent variable, both application versions used identical datasets (from MusicBrainz API [12] and Cover Art Archive [2]), component structures, and layouts. Both were deployed to the same hosting platform (Vercel) to control for server-side differences. This setup ensured that differences in key performance metrics, Time to First Byte (TTFB), Largest Contentful Paint (LCP), and Interaction to Next Paint (INP) [7–9], could be attributed to the rendering strategy rather than to unrelated variations.

3.3.2 Validity

- Performance metrics were selected from Google’s Core Web Vitals [6], which are industry standard and validated indicators of page speed and responsiveness.
- SEO was evaluated using tools that simulate realistic search engine behavior, such as Screaming Frog [18] and Lighthouse [4], and findings were manually verified to ensure crawlability and indexing accuracy.
- Multiple test runs under consistent conditions (same device, network settings, and test environment) reduced random variance and increased the trustworthiness of average values reported. While no formal statistical testing was conducted, consistency across repeated trials supports the validity of observed trends.
- TTFB was used as a measure of server responsiveness [9].
- LCP indicated perceived loading speed for above-the-fold content [8].
- INP represented interactivity and responsiveness to user input [7].

This alignment ensured that the evaluation accurately captured meaningful performance differences in ways that reflect real-world implications.

Although the experiment was limited to a single test application, its design was intentionally modeled on real world content-heavy platforms such as media libraries or documentation portals. As a result, the findings are highly applicable to web applications where SEO performance and fast content delivery are mission-critical.

However, the use of a specific framework (Next.js 15 with App Router) and modern features like lazy loading and hydration [19] may limit the direct applicability to other frameworks such as Angular Universal or Nuxt. The test setup also does not include a lot of dynamic user interactions or live backend updates, which would be present in some production scenarios.

3.3.3 Reliability

To maximize reliability this study employed:

- Identical application logic, dataset, and hosting environment for both rendering strategies
- Standardized scripts and toolchains for automated performance testing
- Repeated test runs (minimum 10) under the same conditions, with outliers removed before averaging
- Public APIs with consistent, open-access content [2,12]

Because the website is content-heavy, longer load times are expected. However, to focus on the typical user experience and avoid extreme cases (like server issues or network errors), we excluded TTFB and LCP values above 10 seconds without throttling, and above 30 seconds under 3G. These generous limits help keep the analysis realistic while reducing the impact of outliers.

For INP, the thesis follows the standard threshold defined by Google, where values above 500 milliseconds are considered poor.

While real-world deployments may introduce additional variability (e.g., content freshness, client-side caching), the experiment's configuration can be reliably reproduced by other researchers or developers using the provided conditions. The setup and tooling used in this study can be replicated using publicly available datasets and standard testing tools, making it accessible for further research or validation.

3.3.4 Limitations

Despite the methodological rigor, several limitations are acknowledged:

- The application was static and lacked realistic interactive or user-generated content, which could affect performance under dynamic load.

- SEO evaluations do not account for long-term ranking behavior, which depends on factors like domain authority or backlink profiles.
- Although statistical testing was used, the small sample size limits the statistical power of some findings.
- Performance results are inherently tool-dependent; using only Chromium-based tools may miss behaviors unique to other browsers.

Despite these constraints, the study is still valuable insights into SSR and CSR trade-offs, especially for developers and teams evaluating rendering strategies in content-heavy, SEO-relevant content-heavy web applications.

This chapter describes the technical architecture and implementations details of the web applications used in this study. Both the SSR and CSR were created in Next.js 15 and deployed on Vercel. The applications have identical content and layout, fetching 100 album data from MusicBrainz API and album covers from the Cover Art Archive API. Except for the albums shown on the page there is also a button that allows interactivity, when the user presses the button a picture appear top of the page and the script that gathers Web Vital gets triggered.

4.1 Structure

```
components/  
  AlbumCard.tsx  
  getAlbums.ts  
  InteractionButton.tsx  
  MetricsScript.tsx  
csr/  
  page.tsx  
ssr/  
  page.tsx
```

Components contains all the components that are reusable and used in the respective page.tsx folders under csr and ssr folders.

4.2 Rendering

4.2.1 Server-Side Rendering

In the SSR version, the application fetches data on the server before rendering the HTML response. This is implemented using an asynchronous function (`getAlbums()`) directly within the SSR component. The full page, including the album list, is rendered before being sent to the browser, ensuring that content is immediately visible to users and search engine crawlers.

```
const SSRPage = async () => {  
  let albums: Album[] = []  
  
  try {
```

```

    albums = await getAlbums();
  } catch (error) {
    console.error('Error fetching albums:', error);
  }

```

4.2.2 Client-Side Rendering

In the CSR version, the page initially renders a basic layout. After the component mounts, it uses a `useEffect` hook to asynchronously fetch the same album data via `getAlbums()`. The content is then populated on the client, requiring JavaScript execution to become visible.

```

const CSRpage = () => {
  const [albums, setAlbums] = useState<Album[]>([]);

  useEffect(() => {
    async function fetchData() {
      try {
        const albumsData = await getAlbums();
        setAlbums(albumsData);
      } catch (error) {
        console.error('Error fetching albums:', error);
        setAlbums([]);
      }
    }
  }, []);

  fetchData();
}, []);

```

4.3 Component Design

4.3.1 AlbumCard

This component is responsible for displaying each album. It receives an album object and renders the cover image, title, artist, date, country, track count, and label. The cover image is rendered using Next.js's `Image` component for optimized loading.

```

const AlbumCard = ({ album }: { album: Album }) => (
  <div className="border rounded-lg overflow-hidden shadow-lg bg-white min-h-96">
    <div className="h-48 bg-gray-200 relative">
      {album.coverUrl && (
        <Image src={album.coverUrl} alt={`${album.title} cover`} fill
          className="object-cover" loading="lazy" />
      )}
    </div>
    <div className="p-4">
      <h2 className="text-xl font-bold line-clamp-2">{album.title}</h2>

```

```
    <p className="text-gray-700">Artist: {album.artist}</p>
    <p className="text-gray-600">Released: {album.date}</p>
    <p className="text-gray-600">Country: {album.country}</p>
    <p className="text-sm text-gray-800 mt-4"><strong>Track Count:</strong>
      <strong>{album.trackCount}</strong></p>
    <p className="text-sm text-gray-800"><strong>Label:</strong> {album.
      label}</p>
  </div>
</div>
);
```

4.3.2 MetricScript

The `MetricsScript` component uses the `web-vitals` library to measure TTFB, LCP, INP, and other performance metrics. It listens for user interactions and collect the metrics.

```
import { onCLS, onFCP, onFID, onINP, onLCP, onTTFB } from 'web-vitals';
...
onTTFB(({ value }) => {
  metrics.ttfb = value;
});

onFCP(({ value }) => {
  metrics.fcp = value;
});

onLCP(({ value }) => {
  metrics.lcp = value;
});

onCLS(({ value }) => {
  metrics.cls = value;
});

onFID(({ value }) => {
  metrics.fid = value;
});

let inpReported = false;
onINP(({ value }) => {
  metrics.inp = value;
  inpReported = true;
  console.log('INP reported:', value);

  if (interactionOccurred) {
    setTimeout(() => sendMetrics(), 1000);
  }
});
```

```
...
```

4.3.3 GetAlbums

This component fetches album data from MusicBrainz API and after that it uses the id to fetch album covers from the Cover Art Archive API. If the album does not have a cover it uses a placeholder image.

```
...
export async function getAlbums(): Promise<AlbumDetails[]> {
  const res = await fetch(
    'https://musicbrainz.org/ws/2/release?query=primarytype:album&fmt=json&
      limit=100',
    {
      cache: 'no-store',
      headers: {
        'User-Agent': 'TechPhoenixComparisonProject/1.0 (teophilsimao@gmail.
          com)',
      },
    },
  );

  if (!res.ok) throw new Error('Failed to fetch album data');
  const data = await res.json();

  const albumsWithDetails = await Promise.all(
    data.releases.map(async (album: Album) => {
      try {

        const coverRes = await fetch(
          'https://coverartarchive.org/release/${album.id}',
          {
            cache: 'no-store',
            redirect: 'follow',
          }
        );

        ...

```

4.3.4 InteractionButton

This component uses client side rendering, "use client", on both pages and allows for user interaction. When interacting with the component it shows a placeholder picture on the top of the page and triggers the MetricScript component.

```
'use client';
...
const InteractionButton = () => {
  const [showImage, setShowImage] = useState(false);

```

```
const handleClick = () => {  
  setShowImage(true);  
  console.log('Button clicked - interaction detected collecting metrics');  
};
```

4.4 Site visuals

This is the page before pressing the button:

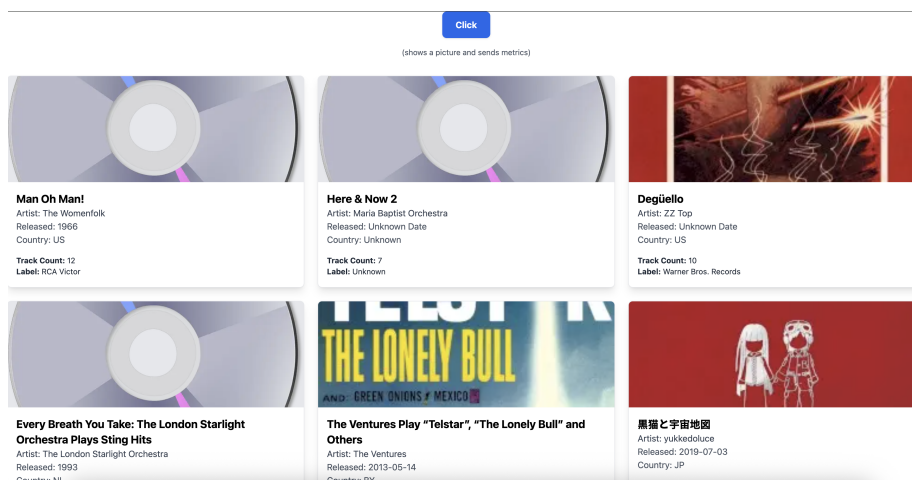


Figure 4.1: Page before pressing the button

This is the page after pressing the button:

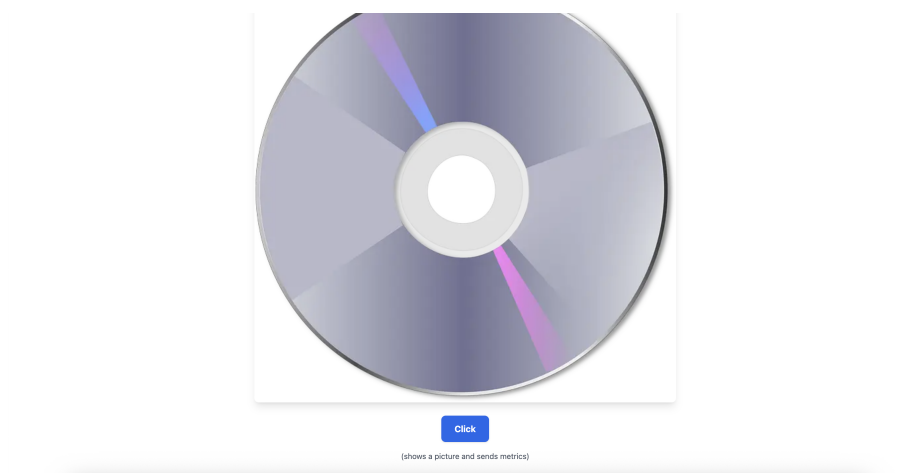


Figure 4.2: Page after pressing the button

4.5 Network conditions

This is the network conditions without Throttling:

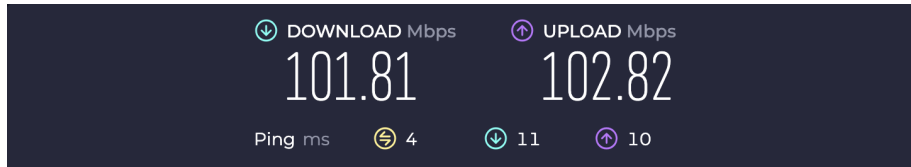


Figure 4.3: network conditions without Throttling

When changing the network condition to 3G, The Google Chrome Dev Tools were used to change the environment.

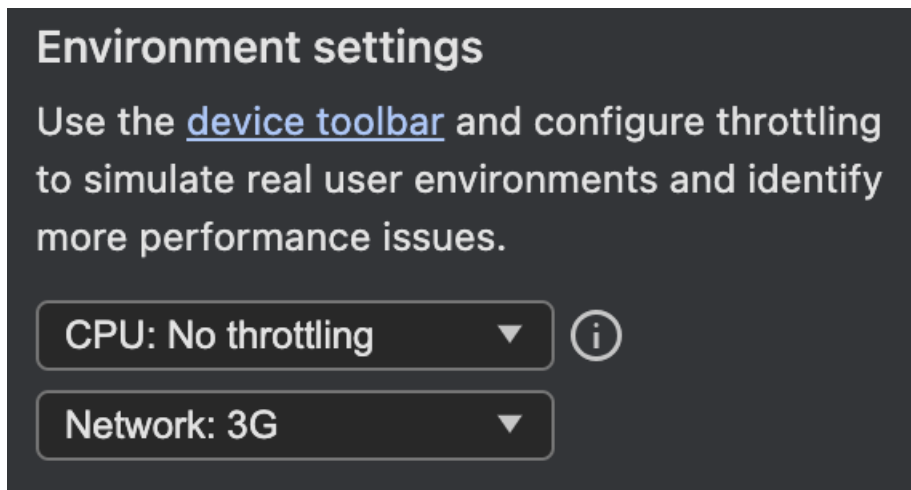


Figure 4.4: 3G network conditions

5.1 Results

This chapter presents the findings derived from performance testing and evaluates the differences between Server-Side Rendering (SSR) and Client-Side Rendering (CSR) implementations. The results are structured around key web performance indicators, Time to First Byte (TTFB), Largest Contentful Paint (LCP), and Interaction to Next Paint (INP), as well as SEO related metrics and qualitative assessments of user experience.

5.1.1 Web -Vitals Performance Results

To evaluate and compare the runtime performance of the SSR and CSR implementations, a series of 10 tests were conducted under two network conditions: no throttling (WiFi) and throttled 3G. Each run measured Time to First Byte (TTFB), Largest Contentful Paint (LCP), and Interaction to Next Paint (INP). The following tables present the mean and standard deviation for each metric across all runs, collected using the Web Vitals library and Google Chrome browser-based tools.

5.1.1.1 Client-Side Rendering (CSR) and Server-Side Rendering – No Throttling

These results represents web-vitals results for the metrics, without throttling, in focus for this research.

Metric	Mean	Std Dev
LCP (s)	5.076000	1.431965
TTFB (s)	0.016270	0.005656
INP (ms)	40.8	5.6

Table 5.1: Web-vitals results for CSR without throttling. LCP and TTFB in seconds and INP in milliseconds.

Metric	Mean	Std Dev
LCP (s)	5.462400	0.275711
TTFB (s)	0.016950	0.010135
INP (ms)	44	8.944272

Table 5.2: Web-vitals results for SSR without throttling. LCP and TTFB in seconds and INP in milliseconds.

Under no-throttling (WiFi) conditions, both Client-Side Rendering (CSR) and Server-Side Rendering (SSR) demonstrated relatively similar performance in terms of Time to First Byte (TTFB), indicating comparable server responsiveness. CSR achieved a slightly lower average Interaction to Next Paint (INP), suggesting marginally faster interactivity, though the difference was minor. The Largest Contentful Paint (LCP), which reflects perceived loading speed, was also lower for CSR, but with a higher standard deviation, indicating more variability across test runs. SSR, on the other hand, offered more consistent LCP results. Overall, CSR performed slightly better in terms of speed, while SSR delivered more stable outcomes.

5.1.1.2 Client-Side Rendering (CSR) and Server-Side Rendering (SSR) – 3G Throttling

These results represents web-vitals results for the metrics, regular 3G, in focus for this research.

Metric	Mean	Std Dev
LCP (s)	27.2864	0.2089
TTFB (s)	0.01301	0.00299
INP (ms)	42.4	5.122499

Table 5.3: Web-vitals results for CSR under 3G. LCP and TTFB in seconds and INP in milliseconds.

Metric	Mean	Std Dev
LCP (s)	8.03	0.197247
TTFB (s)	0.015020	0.002214
INP (ms)	40.8	5.6

Table 5.4: Web-vitals results for SSR under 3G throttling. LCP and TTFB in seconds and INP in milliseconds.

Under 3G throttling conditions, the performance gap between Client-Side Rendering (CSR) and Server-Side Rendering (SSR) became more noticeable. CSR displayed significantly higher LCP values, averaging over 27 seconds, compared to SSR's average of around 8 seconds, highlighting CSR's struggle with slow network conditions due to its reliance on client-side loading and rendering. INP values remained similar between the two, suggesting that interactivity timing was unaffected by the rendering strategy. TTFB was slightly better for CSR, but the difference was minimal. Overall, SSR proved far more resilient under constrained network conditions, delivering faster and more consistent content loading than CSR.

5.1.2 SEO results

To address RQ2 , *How do SSR and CSR impact SEO and indexing for content-heavy web applications?* , this study evaluated the SEO performance of the Server-Side Rendering (SSR) and Client-Side Rendering (CSR) implementations using industry-standard tools: Screaming Frog SEO Spider and Google Lighthouse. These tools simulate search engine behavior to assess critical SEO factors such as crawlability, indexability, and resource discoverability. All important for content-heavy applications dependent on organic search traffic. Manual verification was also performed to validate the automated findings and capture nuances missed by the tools. The following subsections present the results from Screaming Frog, Lighthouse, and manual checks, offering a clear assessment of how rendering strategies influence search engine visibility and indexing completeness.

5.1.2.1 Screaming frog SEO Spider results

For the CSR implementation, the crawl identified a total of 9 URLs. These included the main HTML page (<https://ssr-vs-csr.vercel.app/csr>, `text/html`, 33 bytes), 7 JavaScript files (`application/javascript`), and 1 CSS file (`text/css`). All resources returned a 200 OK status code and were marked as indexable, indicating that search engine crawlers could access and process them.

The main HTML page had a unique hash (5dcdb078145f44771dc2dae7c0efb0bf), confirming it contained non-duplicate content. However, no image resources were detected during the crawl, suggesting that visual content, such as album artwork, is loaded dynamically via JavaScript. This means that search engines would need to execute JavaScript in order to render and index the full content, which introduces additional complexity and may lead to incomplete indexing.

On the other hand, the SSR implementation yielded 100 URLs. These included the main page (<https://ssr-vs-csr.vercel.app/ssr>, `text/html`, 33 bytes), 88 JPEG images (`image/jpeg`), 1 PNG image (`image/png`), 1 CSS file, and 9 JavaScript files. All URLs returned a 200 OK status and were indexable. The main page had a unique hash (4ab8370aad3f72c1eaa8e1e3b5f6a630), indicating distinct content.

A big difference is that the SSR page preloaded 88 image resources, sourced from the Cover Art Archive API, making them immediately available to crawlers without

requiring JavaScript execution. This significantly improves content visibility and indexing efficiency, especially for visual assets central to the application.

While both implementations are technically crawlable and indexable, the SSR approach offers clear advantages for SEO, particularly in content-heavy applications. Its server-rendered HTML and embedded image resources provide immediate content visibility to crawlers, facilitating faster and more complete indexing. In contrast, the CSR version relies heavily on client-side rendering, requiring multiple JavaScript executions to expose core content. This can delay indexing or result in incomplete content discovery, particularly in less capable or resource-constrained crawlers.

5.1.2.2 Results from lighthouse and manually checking

To evaluate the SEO effectiveness of the SSR and CSR versions of the application, Lighthouse audits were conducted using Chrome, with all browser extensions disabled and network throttling turned off. The focus was on the SEO score, not performance.

Lighthouse SEO Scores

- **SSR SEO Score:** 100
- **CSR SEO Score:** 100

Both the SSR and CSR implementations received a perfect SEO score from Lighthouse. This indicates that, according to automated auditing tools, both versions met key SEO best practices, such as:

- Proper use of meta tags
- Valid heading structure
- Crawlable links and readable text
- Mobile compatibility

Although the Lighthouse scores were identical, manual inspection of the page source revealed important differences:

- **SSR:** The full content (e.g. album titles and metadata) was visible in the HTML source as delivered from the server. This means the data is immediately available to search engine crawlers without requiring JavaScript execution.
- **CSR:** The HTML source only contained the static layout without any actual album content. The data was loaded and rendered by JavaScript after the initial page load. While modern search engines are generally able to render JavaScript, this introduces an extra dependency that could delay indexing or affect completeness, especially for lightweight crawlers or under limited crawl budgets.

This shows that while CSR can technically be SEO-friendly, SSR provides a more reliable approach for ensuring that content is immediately visible to all crawlers, regardless of their JavaScript capabilities.

5.1.2.3 Summary of Key Findings: SSR vs. CSR

Aspect	SSR	CSR
LCP (WiFi)	5.46s (more consistent)	5.08s (faster, but more variable)
TTFB (WiFi)	0.017s (slightly slower)	0.016s (slightly faster)
INP (WiFi)	44ms	41ms
LCP (3G)	8.0s (much faster)	27.3s (significantly worse)
TTFB (3G)	0.015s	0.013s
INP (3G)	40.8ms	42.4ms
SEO – Screaming Frog	All content (incl. images) visible to crawlers	Dynamic content hidden without JS
SEO – Lighthouse	100 (perfect score)	100 (perfect score)
SEO – Manual Check	Content pre-rendered in HTML	Only layout pre-rendered; data loaded via JS

Table 5.5: Key findings summary: SSR vs. CSR

The performance tests revealed that under normal (WiFi) conditions, Client-Side Rendering (CSR) achieved slightly faster load times (LCP and INP) than Server-Side Rendering (SSR), though with greater variability. However, under 3G throttling, SSR significantly outperformed CSR—particularly in LCP, which was more than three times faster. Highlighting SSR’s superior resilience under constrained networks. Both approaches received perfect Lighthouse SEO scores, but manual inspection and Screaming Frog analysis showed that SSR offered greater visibility to search engines by delivering fully rendered content, including images, in the initial HTML. In contrast, CSR required JavaScript execution to expose critical data, which may hinder indexing, especially for lightweight or non-JS crawlers.

5.2 Analysis

This section interprets the results presented in Section 5.1, addressing the research questions by analyzing the implications of the observed differences between Server-Side Rendering (SSR) and Client-Side Rendering (CSR). The analysis is divided into three parts, representing performance (RQ1), SEO impact (RQ2), and the identification of thresholds for selecting SSR over CSR or the other way around (RQ3).

5.2.1 Performance Analysis (RQ1)

The performance results show a detailed comparison between SSR and CSR depending on network conditions. Under no-throttling (WiFi) conditions, CSR demonstrated a marginal advantage in average LCP (5.08s vs. SSR’s 5.46s) and INP (40.8ms vs. 44ms). However, CSR exhibited a higher standard deviation for LCP, suggesting

greater variability in load performance. This implies that while CSR may appear faster on average, user experience can vary more across sessions and devices.

In contrast, under 3G network throttling, SSR significantly outperformed CSR in LCP (8.03s vs. 27.29s), showing a much more consistent and responsive experience in constrained network environments. The CSR approach struggled under these conditions due to its reliance on JavaScript execution and client-side data fetching, which is slower over limited bandwidth. INP remained relatively stable across both rendering strategies and network conditions, suggesting that interactivity is less affected by rendering method than content loading.

These findings highlight an important insight: while CSR may perform effectively in ideal conditions, SSR provides more consistent and reliable performance, especially in scenarios where users may have slower or unreliable connections. The ability of SSR to deliver pre-rendered HTML and preload critical assets gives it a significant advantage in environments where network speed is limited.

5.2.2 SEO and Indexing Implications (RQ2)

Both CSR and SSR implementations received a perfect SEO score (100) from Lighthouse, which may initially suggest equivalent SEO performance. However, manual inspection of the HTML source revealed critical differences that automated tools like Lighthouse do not capture.

The SSR version delivered complete content, including album titles and images, directly in the server-rendered HTML. This allows search engine crawlers to access all meaningful content immediately without executing JavaScript. In contrast, the CSR version rendered only a simple basic layout in the HTML response, with all actual content populated client-side via JavaScript. This adds an extra step for crawlers and increases the risk of incomplete indexing, particularly for simpler bots or under constrained crawl budgets.

Furthermore, the Screaming Frog SEO Spider identified significantly more discoverable resources in the SSR version (100 URLs vs. 9 in CSR), including all 88 images used in the interface. These images were preloaded server-side in SSR but not visible at all during CSR crawling. This suggests that for content-heavy sites, SSR ensures a more complete indexing experience, which is crucial for search visibility.

The combined automated and manual evaluations showed that SSR offers better reliability and completeness for SEO, especially in scenarios where timely and comprehensive indexing is important.

5.2.3 When to Choose SSR over CSR (RQ3)

Based on the findings above, it is possible to establish both absolute and relative thresholds that help inform when SSR should be favored over CSR in content-heavy web applications.

Thresholds:

- If Largest Contentful Paint (LCP) consistently exceeds 2.5 seconds, especially under real-world or throttled conditions, SSR should be preferred to improve loading speed and meet Core Web Vitals benchmarks and offer consistency.
- If search engine visibility is important, particularly for sites with a large volume of dynamic or image-based content, SSR provides a more reliable mechanism for content discovery, since it exposes the complete content directly in the HTML source.

Use-case recommendations.:

- In this study, SSR reduced LCP under 3G by approximately **70%** compared to CSR (8s vs. 27s). Such a large improvement suggests that in similar conditions, SSR should be chosen to ensure a good user experience and faster perceived load times.
- CSR may still be viable when content is personalized per user, SEO is not a primary concern, or users are expected to access the site primarily on fast networks.

These thresholds provide developers with concrete, evidence-based guidance. SSR should be strongly considered when:

- Network conditions are unpredictable or commonly slow.
- The application is content-heavy and SEO-dependent.
- Stability in performance is more critical than initial loading speed under ideal conditions.

CSR may be acceptable when:

- Fast interactivity is prioritized for repeat visits.
- The application is highly dynamic or user-specific.
- SEO is less of a concern, or JavaScript-based indexing is sufficient.

This chapter discusses the findings of this thesis in relation to the three research questions. It provides interpretations of the results, reflects on their implications for web development practice and theory, considers possible alternative explanations, and addresses both the limitations and broader significance of the study.

6.1 Architectural Impact on Performance

The observed performance differences between SSR and CSR stem largely from how each architecture handles data fetching and rendering.

In the SSR version, data from the MusicBrainz API and 100 cover images from the Cover Art Archive are fetched server-side before the page is sent to the client. This results in a fully rendered HTML response with all content immediately available, leading to faster and more consistent **Largest Contentful Paint (LCP)**, especially under 3G conditions.

In contrast, the CSR version performs all data fetching on the client after the page loads. The initial HTML is nearly empty, and JavaScript must first load and execute before any content appears. The browser must then make 101 API calls (1 for album data, 100 for images), causing significant delays. This explains the much higher LCP for CSR under throttling (27s vs. 8s for SSR).

Additionally, SSR improves SEO by embedding all content directly in HTML, allowing crawlers to index it immediately. CSR relies on JavaScript for rendering content, which can hinder indexing if crawlers do not fully execute scripts.

6.2 Research Questions

1. How does the performance of content-heavy web applications differ between SSR and CSR implementations in Next.js, specifically regarding TTFB, LCP, and INP metrics?

The results present a Detailed picture. Under fast network conditions (WiFi), CSR exhibited slightly better average LCP than SSR (5.08s vs. 5.46s), indicating a marginal advantage in perceived speed. However, CSR's performance was notably

more variable, with a standard deviation more than five times greater than SSR. In contrast, SSR delivered more consistent LCP outcomes, a valuable trait for user experience reliability.

Under constrained (3G) conditions, the differences became more significant. SSR drastically outperformed CSR in LCP (8.03s vs. 27.29s), showing that SSR scales better in slower environments. While TTFB differences remained minor across all tests, SSR provided more consistent and favorable INP values under load, suggesting more stable interactivity. Thus, SSR proves more dependable in content delivery, especially under suboptimal conditions, where CSR, while occasionally faster, struggles with consistency and predictability.

2. How do SSR and CSR impact SEO and indexing for content-heavy web applications?

SEO analysis favored SSR. While both implementations achieved perfect Lighthouse SEO scores, manual and Screaming Frog inspections revealed important differences. SSR exposed all meaningful content in pre-rendered HTML, ensuring visibility to crawlers even without JavaScript execution. In contrast, CSR delayed the rendering of dynamic content until after hydration, potentially hindering discoverability by search engines that do not execute JavaScript or do so incompletely. Therefore, SSR offers superior default SEO behavior, making it a more robust choice for search-sensitive content.

3. What specific performance thresholds should trigger the selection of SSR over CSR for content-heavy websites?

Based on Core Web Vitals and the results, practical thresholds emerged:

- **If LCP exceeds 2.5 seconds**, particularly under mobile or 3G-like conditions, SSR is the safer choice.
- **If TTFB nears or exceeds 800ms**, SSR helps reduce latency.
- **If SEO visibility is critical**, especially for pages containing dynamic or content-heavy content, SSR should be favored to ensure immediate crawlability.

These findings support a decision-making framework grounded in observable performance and content exposure metrics.

6.3 Comparison to Prior Work

The results of this study generally support prior research suggesting that Server-Side Rendering (SSR) offers performance advantages in delivering the initial view, particularly under constrained or real-world network conditions [8, 15, 16]. While Client-Side Rendering (CSR) demonstrated a slightly better average LCP under

ideal (WiFi) conditions in this study, SSR delivered more consistent performance and significantly outperformed CSR under 3G, reinforcing its strengths in environments with limited bandwidth or latency sensitivity.

On the other hand, CSR's strengths lies in flexibility and interactivity, as emphasized in previous studies [1, 13]. These benefits are more applicable to application-like interfaces where client-side state and dynamic updates are central. However, for content-heavy pages like those examined in this study, CSR's dependence on JavaScript and runtime data fetching significantly drew down LCP under slow network conditions. This suggests that CSR's advantages are highly dependent on ideal circumstances, modern devices and strong network connectivity.

It is also important to note that the performance gap between SSR and CSR may narrow in the future. Modern developments such as partial hydration, React Server Components, and streaming SSR aim to combine the initial load benefits of SSR with the dynamic capabilities of CSR. Although these emerging techniques were outside the scope of this study, they represent promising directions for future research and may alter the current trade offs between SSR and CSR [19].

6.4 Validity of Results

This study was conducted in a controlled environment using consistent codebases, hosting configurations, testing tools, and datasets across both rendering strategies. Performance metrics were collected using industry-standard tools, especially those aligned with Google's Web Vitals. Multiple test runs under each network condition were performed to minimize the influence of anomalies or outliers, enhancing validity.

However, some limitations remain. The analysis focused on a single application with a specific architecture and dataset, which may limit generalizability. Results might differ when applied to other frameworks, larger applications, or alternative rendering approaches such as static site generation or edge rendering. Furthermore, the experiments relied on Chrome-based tools, and performance across other browsers, such as Safari or Firefox, was not evaluated. Which could introduce browser-specific bias.

Despite these constraints, the findings align well with existing literature, real-world developer experience, and Google's Core Web Vitals guidelines. This consistency supports the external validity of the results and suggests that the observed trends are likely applicable to other content-heavy web applications operating under similar conditions.

6.5 Limitations

The findings of this study should be interpreted within the context of the following limitations:

- **Scope and Use Case:** The test application represented a content-heavy website but did not incorporate personalized, user-specific, or highly interactive content. In such cases, Client-Side Rendering (CSR) or hybrid strategies, such as progressive or selective hydration, may offer more suitable performance characteristics.
- **Browser and Tooling Bias:** All performance and SEO metrics were gathered using Chromium-based tools (e.g., Lighthouse, Web Vitals). Rendering performance and crawl behavior may differ in alternative browser engines such as WebKit (Safari) or Gecko (Firefox), potentially affecting generalizability.
- **SEO Proxy Limitations:** The SEO evaluation focused on technical crawlability and content discoverability using tools like Screaming Frog and Lighthouse. However, true search performance in the real world (e.g., SERP rankings) depends on broader factors such as backlink profiles, content relevance, and domain authority, which were beyond the scope of this study.
- **larger sample size:** The performance evaluation used 10 test runs per configuration. While this provides a general indication of behavior, a larger sample size would strengthen the statistical reliability of the findings. Future studies are encouraged to use at least 30 or more measurements and apply formal statistical testing to better account for variability.
- **Rendering Pipeline Complexity:** The significant LCP difference between CSR and SSR under 3G network conditions may, in part, be due to the hydration overhead associated with Next.js. Future improvements in frameworks, such as streaming, partial hydration, or React Server Components, may narrow this performance gap.
- **Variation Within Content-Heavy Website Types:** While the study focuses on content-heavy web applications, this category includes a wide range of website types with different performance characteristics and user interaction models. For example, static news sites primarily deliver read-only content, while interactive documentation platforms may involve dynamic navigation, real-time search, or embedded media. The specific composition and behavior of such applications can influence how rendering strategies perform in practice. Therefore, the generalizability of the results may vary depending on the exact nature of the content-heavy site being considered.

Despite these limitations, the consistency of the results across network conditions and rendering strategies lends confidence to the practical insights derived. The trends observed are in line with broader industry findings and offer meaningful guidance for selecting rendering approaches in content-heavy web development contexts.

6.6 Significance and Impact

This study underscores the importance of evidence-based rendering decisions. While Client-Side Rendering (CSR) may be appealing due to its flexibility and rich ecosystem, Server-Side Rendering (SSR) is often the more appropriate choice for public

facing, content-heavy websites. Especially when performance and SEO are important.

These findings are especially relevant for content-heavy websites, where fast performance and full crawlability are essential for user experience and visibility. SSR's advantages in ensuring predictable loading times and better content exposure can significantly impact user engagement, search engine indexing, and long-term accessibility. Especially in bandwidth restricted or resource limited environments.

By adopting SSR strategically, developers can enhance user experience across a wider range of conditions and improve the discoverability of rich content, thereby supporting the goals of websites that prioritize content visibility, accessibility, and reach.

6.7 Ethical Considerations

Rendering strategies have real world inclusivity implications. SSR contributes to digital accessibility by delivering readable, structured content even before JavaScript execution. Benefiting users with older devices, poor connectivity, or limited browser capabilities.

On the other hand, CSR risks excluding vulnerable users if JavaScript fails or network conditions prevent timely content delivery. Given the web's global reach, rendering decisions must balance performance, inclusivity, and technological advancement.

6.8 Summary

In conclusion, SSR remains the more suitable rendering strategy for content-heavy applications that prioritize performance consistency and SEO. While CSR continues to evolve and holds promise in certain interactive or app-like contexts, its variability and dependence on optimal conditions make it less reliable for content delivery at scale.

7.1 Conclusions

This thesis compared Server-Side Rendering (SSR) and Client-Side Rendering (CSR) in the context of content-heavy web applications, focusing on performance and SEO. Using two identical implementations in Next.js 15, one rendered via SSR and the other via CSR, the study examined Time to First Byte (TTFB), Largest Contentful Paint (LCP), and Interaction to Next Paint (INP), under both unthrottled and 3G network conditions. SEO capabilities were also evaluated using automated tools and manual inspection.

7.1.1 Performance conclusion

Under WiFi conditions, CSR showed slightly better average LCP (5.08s vs. SSR's 5.46s), suggesting faster perceived load time. However, CSR's LCP results were significantly more variable (Std Dev = 1.43s) compared to SSR (Std Dev = 0.28s), indicating that SSR was more consistent in delivering content quickly. INP and TTFB were nearly identical, with CSR marginally ahead.

Under 3G throttling, SSR significantly outperformed CSR in LCP (8.03s vs. 27.29s), clearly highlighting its advantage under constrained network conditions. CSR struggled to load meaningful content in a timely manner when bandwidth was limited. TTFB differences remained minor, though CSR had a slight edge in both network scenarios. INP was slightly better for SSR overall, especially under 3G, where interaction delays were minimized.

- **CSR offers better average LCP performance** under optimal network conditions, but with increased variability.
- **SSR provides more consistent performance** across the board and significantly better LCP under poor network conditions.
- **TTFB differences are negligible**, though CSR is marginally faster.
- **INP is comparable** between the two, with slight favorability to SSR on 3G and to CSR on WiFi.

7.1.2 SEO conclusion

SEO analysis via Screaming Frog and manual inspection showed that SSR made all content, including dynamic data and images, immediately visible to search engine crawlers, thanks to pre-rendered HTML. In contrast, CSR hid dynamic content until JavaScript execution, reducing initial indexability. Lighthouse scores were identical (100) for both, but SSR had an advantage in real-world SEO readiness due to better crawlability and pre-rendered content availability.

7.1.3 General Conclusion

There is no universally superior rendering strategy; the optimal choice depends on application needs. For performance important, content-heavy applications that must serve users across diverse devices and networks, SSR is the more reliable choice. Its consistent loading behavior and superior SEO friendliness make it ideal for public-facing content sites. On the other hand, CSR is well-suited for dynamic, app-like experiences under stable network conditions, offering faster LCP in ideal cases.

Ultimately, modern development frameworks like Next.js support hybrid rendering, and this study's results support the idea that a hybrid approach could combine the strengths of both SSR and CSR.

7.2 Future Work

Several areas of future investigation were identified during the course of this study:

1. **Hybrid Rendering Techniques:** Investigate advanced rendering strategies such as Incremental Static Regeneration (ISR) and selective hydration, which combine SSR and CSR advantages.
2. **Wider Test Scope:** Expand performance testing across more devices (especially mobile), browsers (including Safari and Firefox), and global locations to improve generalizability.
3. **Scalability and Interactivity:** Explore how each rendering strategy performs when dealing with highly interactive components, real-time updates, or larger-scale applications.
4. **User Experience Research:** Supplement Core Web Vitals with user-centric studies that capture subjective performance perceptions and usability differences.
5. **Long-term SEO Impact:** Track how rendering strategy influences real search engine ranking and indexation over time, beyond initial crawl tests.
6. **Sustainability Considerations:** Examine energy consumption on client devices, particularly for CSR's reliance on heavier JavaScript execution.

7. **Accessibility:** Assess how rendering mode affects screen reader support and accessibility standards, which are increasingly tied to both user experience and compliance.

The field continues to evolve rapidly, and future studies will benefit from emerging techniques, tools, and community practices aimed at optimizing rendering strategies.

References

- [1] M. Beke, “On the comparison of software quality attributes for client-side and server-side rendering,” Ph.D. dissertation, University of Antwerp, 2018.
- [2] Cover Art Archive. (2025) Cover Art Archive API. <https://coverartarchive.org/>. Accessed: Feb. 20, 2025.
- [3] Google Developers. (2025) Chrome devtools. Accessed: Feb. 7, 2025. [Online]. Available: <https://developer.chrome.com/docs/devtools/>
- [4] ——. (2025) Lighthouse performance measurement tool. Accessed: Feb. 7, 2025. [Online]. Available: <https://developer.chrome.com/docs/lighthouse/>
- [5] ——. (2025) Mobile-friendly test. Accessed: Feb. 7, 2025. [Online]. Available: <https://search.google.com/test/mobile-friendly>
- [6] Google Web.dev. (2025) Core web vitals. Accessed: Feb. 15, 2025. [Online]. Available: <https://web.dev/vitals/>
- [7] ——. (2025) Interaction to next paint (inp). Accessed: Feb. 15, 2025. [Online]. Available: <https://web.dev/inp/>
- [8] ——. (2025) Largest contentful paint (lcp). Accessed: Feb. 12, 2025. [Online]. Available: <https://web.dev/lcp/>
- [9] ——. (2025) Time to first byte (ttfb). Accessed: Feb. 12, 2025. [Online]. Available: <https://web.dev/ttfb/>
- [10] S. Gregor and A. R. Hevner, “Positioning and presenting design science research for maximum impact,” *MIS Quarterly*, vol. 37, no. 2, pp. 337–355, 2013.
- [11] T. F. Iskandar *et al.*, “Comparison between client-side and server-side rendering in the web development,” in *IOP Conference Series: Materials Science and Engineering*. IOP Publishing, 2020, p. 012136.
- [12] MusicBrainz. (2025) MusicBrainz API. https://musicbrainz.org/doc/MusicBrainz_API. Accessed: Feb. 20, 2025.
- [13] Next.js. (2025) Client-side rendering (csr). Accessed: Feb. 7, 2025. [Online]. Available: <https://nextjs.org/docs/pages/building-your-application/rendering/client-side-rendering>
- [14] ——. (2025) Next.js documentation. Accessed: Feb. 7, 2025. [Online]. Available: <https://nextjs.org/docs>
- [15] ——. (2025) Server-side rendering (ssr). Accessed: Feb. 7, 2025. [Online]. Available: <https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering>

- [16] C. Nordström and A. Dixelius, “Comparisons of server-side rendering and client-side rendering for web pages,” Uppsala University, Tech. Rep., 2023.
- [17] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, 2007.
- [18] Screaming Frog. (2025) Seo spider tool. Accessed: Feb. 7, 2025. [Online]. Available: <https://www.screamingfrog.co.uk/seo-spider/>
- [19] Vercel. (2024) Optimizing: Lazy loading and hydration. Accessed: Mar. 11, 2025. [Online]. Available: <https://nextjs.org/docs/app/building-your-application/optimizing/lazy-loading#progressive-hydration>
- [20] WebPageTest. (2025) Performance testing tool. Accessed: Feb. 9, 2025. [Online]. Available: <https://www.webpagetest.org/>
- [21] R. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.

The following resources support the implementation and evaluation discussed in the thesis.

- **SSR Page:** <https://ssr-vs-csr.vercel.app/ssr> – A server-side rendered version of the content-heavy application.
- **CSR Page:** <https://ssr-vs-csr.vercel.app/csr> – A client-side rendered version for comparison.
- **Code Repository:** <https://github.com/teophilsimao/SSR-vs.-CSR-> – Contains the full source code, setup instructions, and performance monitoring script.

