Mobile Web Performance: React Server Components and Edge Rendering

Hesham Alhuraibi
University of St. Pölten
St. Pölten, Austria
it251516@ustp-studnets.at

***Abstract*—Mobile web performance has become increasingly important as users expect fast and reliable experiences across a wide range of devices and network conditions. Traditional rendering methods often struggle to meet these demands due to heavy JavaScript execution and long network round trips. This paper examines two modern rendering techniques, React Server Components and edge rendering, and discusses how they address key mobile performance challenges. By reducing JavaScript workload on the client and generating responses closer to the user, these approaches can improve Core Web Vitals such as LCP, FID/INP, and TTFB. While both methods offer promising results, they also introduce new architectural and runtime limitations. This state-of-the-art overview evaluates their potential impact on mobile web performance and highlights open challenges for future research.**

Key words: React Server Components, Edge Rendering, Performance, Mobile, Core Web Vitals

I. Introduction

As web applications grow in complexity, traditional rendering approaches such as Client-Side rendering and Server-Side Rendering often struggle to provide the responsiveness that mobile users expect. Heavy JavaScript bundles, long network delays, and increasing complexity can have a negative effect on the Core Web Vitals, leading to slower loading times and reduced user satisfaction.

Recent advancements in technology introduced React Server Components and Edge rendering as methods to combat these limitations. React Server components shift non-interactive logic to the server while edge rendering brings computation closer to the user. Both techniques offer promising paths towards performance improvements, yet several limitations were introduced that relate to tooling, caching, and architectural complexity

This paper provides an overview of the state of the art in modern rendering techniques, examines their impact on mobile web performance, and evaluates their advantages and limitations compared to traditional methods.

II. Background and Related Approaches

A. Mobile Web Performance and Metrics

With the continuous expansion of users and user expectations, performance optimization became an essential factor that represents the quality of work. As demand for quality grew, Google introduced web vitals in 2020 to assert a unified framework for all developers to assess user-centric performance metrics. The web vitals included but not limited to, Largest Contentful Paint (LCP), First Input Delay (FID), and Cumulative Layout Shift (CLS). These three metrics critically measure the loading performance, interactivity, and visual stability of a website [1].

With Google's web vital metrics, it becomes simpler to improve user experience, thus achieving optimal scores. However, applying these improvements comes with several challenges that a developer has to face. For example, legacy code and infrastructure struggle with optimization, as they require substantial refactoring to meet modern performance standards. Another problem is to find the right balance between performance and functionality. Implementing a complex functionality might require extensive UI to improve user experience, however, animation and third-party integrations for example can decrease the FID and CLS scores.

Each vital facilitates multiple strategies for optimization. For example, using Content Delivery Network and lazy loading can improve LCP. Splitting code bundle to minimize JavaScript execution time and removing dead code improves FID. Reserving space for expected pop-ups and specifying size attributes for images and videos improves CLS.

JavaScript execution remains one of the most significant performance bottlenecks on mobile devices. Compared to desktop hardware, mobile CPUs generally have lower clock speeds and more aggressive energy-saving constraints, which makes JavaScript parsing, compilation, and execution noticeably slower. This issue becomes more visible as modern applications continue to ship large bundles, often containing unused or redundant code. Prior research has shown that reducing dead code or decreasing bundle size can lead to measurable improvements in responsiveness and user interaction [3]. This connects directly to user-centric metrics like FID and INP, where long JavaScript tasks can block the main thread and delay input handling. As a result, controlling JavaScript bloat is a key factor in achieving consistent mobile performance.

B. Traditional Rendering Approaches

During the process of planning the architecture of a website, one of the most crucial aspects to consider is the strategy used to render content to the user. Traditionally, rendering occurs either on the client side (Client-Side Rendering, CSR) or on the server side (Server-Side Rendering, SSR). In addition to

these two classical approaches, there are also closely related methods such as Static Site Generation (SSG) and Incremental Static Regeneration (ISR), which pre-render content on the server either at build time or at scheduled intervals. Each method offers its own advantages and disadvantages, and choosing the right approach for the right circumstance can effectively improve the website's performance.

In CSR, the client is sent an HTML file along with a JavaScript bundle. The client then proceeds to generate the application's content in the browser while connecting everything together. Subsequently, any interaction does not require a new request from the server to load a page. CSR allows for smoother and more responsive interactions [4]. However, if the JavaScript bundle does not load quickly, the user is left waiting on a blank or partially rendered screen, which can significantly increase the time before the page becomes usable.

In SSR, the content is rendered on the server before the page is sent to the client [5]. Since the HTML is already fully rendered, the client can display the page immediately without additional processing. This allows for a faster initial loading time. However, with SSR, navigation may be slower since each new page requires a request to the server.

C. Edge computing for Web Performance

Edge computing has become an important approach for improving performance on modern applications, especially for users on mobile devices. The main idea is to move computation and data processing closer to the end user by running services on distributed nodes instead of on a centralized server. This reduces the physical distance that a request must travel, which can significantly lower network latency. As mobile networks often suffer from unstable connections and higher round-trip times, reducing the distance between the client and the server can lead to a noticeably faster and more consistent user experience.

Satyanarayanan describes edge computing as an evolution of cloud systems that focuses on geographic proximity to users. He explains that delivering computation closer to the device can help applications respond more quickly and reduce delays that are common on mobile networks [6]. Although this concept is not directly tied to web rendering, the same principle applies: when data and logic are handled near the user, the time it takes for a page to load or respond can be reduced.

This idea forms the basis for several modern web performance techniques. By leveraging edge locations, developers can minimize latency before the browser even starts processing the page. As a result, edge computing provides a foundation that later approaches, such as edge-based rendering, can build upon to further optimize load times and responsiveness.

III. Modern Rendering Architectures for Mobile Performance

A. React Server Components (RSC)

React Server Components (RSC) are a modern rendering model created to improve performance by shifting the rendering load from the client to the server. In contrast to traditional client-side rendering, where JavaScript handles most of the UI generation in the browser, RSC allows non-interactive parts of the React application to run entirely on the server. The client never receives JavaScript for these parts, instead it receives serialized data that the browser can use to display the UI [9]. The approach of using server components substantially reduces the amount of JavaScript executed on the client, which is vastly beneficial for mobile users with limited processing power and unstable network connections.

One of the main motivations for using RSC is to reduce the overall JavaScript bundle size. With the limitation in processing power on mobile devices, large bundles can slow down initial loading times and user interactions. By moving non-interactive or data-heavy logic to the server, RSC helps clients load pages faster and reduces the workload on the browser. Josh Comeau explains that this separation allows React to treat server components as a way of streaming UI without shipping the full component logic to the client, resulting in better performance and less hydration overhead [10].

RSC also brings advantages in data fetching. Since RSC run on the server, they can access databases or APIs without any additional client-side requests. This approach reduces network round trips and simplify data management. Makarevich demonstrates that this pattern can outperform traditional CSR and SSR approaches by lowering both bundle size and network requests, ultimately contributing to fast page interactions and improved load times [11]. These benefits make RSC well-suited for large applications where data fetching and heavy UI logic would otherwise slow down the client.

However, RSC also introduces several challenges. Developers have to split their code base into two categories, server components and client components. This division is not always obvious and can introduce many issues such as bloated file system if the correct architecture is not followed. Since RSC are sever-sided, they can not access any browser library such as 'DOM', 'Node.js APIs (in edge runtime)', and 'document' which can increase development limitations. Beyond compatibility and architectural complexity, RSC can also introduce performance pitfalls when used incorrectly. Although RSC aims to reduce network waterfalls, poor component structuring can lead to repeated data fetching, inefficient server calls, or unnecessary re-rendering on the server. These mistakes can increase load times rather than improve them, especially in deeply nested component trees.

B. Edge Rendering

Edge rendering is a technique in which server-side rendering is performed on edge nodes instead of on a centralized origin server. By rendering pages on an edge node, the user's request is processed as close to them as possible thus reducing the physical distance that data must travel ultimately lowering latency. To the user, generating pages in close proximity decreases initial HTML delivery time leading to faster response times and improved user experience [7].

Modern platforms have already shown support for edge rendering. Vercel provides built-in support through its Edge Runtime, allowing execution of server-side logic at edge nodes. Moreover Cloudflare Workers enable rendering React application at the edge by running supportive JavaScript functions on a global network of data centers. Other providers such as Netlify offer Edge Functions that support dynamic content generation at geographical distributed locations. The availability of edge rendering tools made development practical for developers who aim to delivery dynamic, personalized content with minimal latency.

Edge rendering for server-side logic provides multiple performance benefits. One of the most important benefits is the reduction in Time to First Byte (TTFB) [7], which measures how long a browser waits for the first byte of data after making a request. This reduction occurs because responses are generated closer to the user. This is extremely useful for users on mobile networks, where high latency and unstable connections are common [8]. Faster TTFB not only contributes to better perceived performance, but it also positively influences search engine rankings, as search engines consider page speed and responsiveness as part of their scoring. Traditionally, a user's request travels to a centralized server, which can be slow due to traffic or physical distance. By deploying computing resources at the edge of the network, the request is intercepted and processed much sooner. Furthermore, scalability becomes easier to manage since the load is distributed across different edge nodes.

Despite all these advantages, edge rendering also introduces several challenges. Since most edge platforms do not support full Node.js APIs, applications must be adapted to run in more restricted environments. This, in turn, limits compatibility with certain libraries or server-side features. It also becomes more difficult to manage caching at the edge while avoiding unnecessary computations and maintaining content consistency across regions. Most importantly, edge rendering raises security and privacy concerns. By bringing data processing closer to the user, sensitive information may become more vulnerable to attacks or breaches if each node is not properly secured. Similarly, having many distributed nodes increases the potential impact of physical threats depending on their locations. These limitations mean that while edge rendering offers strong performance benefits, it also requires careful planning and a good understanding of its constraints.

Together, these approaches illustrate a significant shift in how web applications are delivered, setting the stage for a deeper discussion of their performance implications and remaining challenges.

IV. Discussion

A. Performance Implications

Building on the strengths and limitations mentioned in the Modern Rendering Architectures section, it is important to evaluate how they influence mobile performance in practice. Both React Server Components and edge rendering were introduced to overcome the bottlenecks identified. For example, reducing JavaScript bundle size through RSC can positively improve First Input Delay (FID) and Interaction to Next Paint (INP). Since RSC shifts non-interactive logic to the server, the browser performs less parsing and execution work which becomes beneficial for mobile devices with slower CPUs and stricter energy-saving constraints. Additionally smaller bundles reduce the network transfer time especially over unstable connections, thus improving the Largest Contentful Paint (LCP).

Edge Rendering provides additional improvements to loading-related metrics such as LCP and Time to First Byte (TTFB). By generating the response at an edge node nearest to the user, the time required for the initial HTML to reach the browser is heavily reduced. The reduction of the round-trip time between a user and an edge node helps pages become visually complete faster which is an important factor of the LCP score. Furthermore, the node's faster response decreases the user's wait time before any UI is rendered which improves the perceived responsiveness of the application. For a mobile user, who frequently may experience high latency, generating HTML closer to the device directly contributes to a more stable experience and fewer long loading periods.

Finally both RSC and edge rendering can indirectly improve the Cumulative Layout Shift (CLS) score by reducing the dependency on late-loading client-side content. Since HTML is delivered quickly and contains the final layout, fewer layout shifts can occur during the rendering phase. Its important to mention that CLS is primarily influenced by front-end design choices, however, reducing JavaScript execution and ensuring early access to structured HTML can help prevent any UI jumps.

B. Comparison to Previous Approaches

Compared to the traditional Client-Side Rendering (CSR), the combination of RSC and edge rendering provides significant advantages. With CSR, downloading and executing a large JavaScript bundle is required before the user can see any meaningful content. This approach results in slower LCP and delayed interactivity on mobile devices. Additionally, heavy hydration that occurs after parsing the JavaScript increases the costs and further delays the user. RSC reduces this dependency by pushing more logic to the server while sending less code to the client, making it fundamentally better suited for mobile networks.

When compared to classical Server-Side rendering (SSR), RSC and edge rendering offer improvements in both scale and latency. In contrast to edge rendering, SSR relies on a centralized server that may be geographically far from the user. This introduces challenges with long network round trips, especially for a mobile user in regions far away from the origin server. This limitation is addressed by edge rendering which runs SSR-like logic at edge nodes distributed worldwide. RSC further complements this by reducing hydration overhead typically seen in SSR, resulting in faster page updates and less JavaScript execution in the browser.

C. Limitations and Open Challenges

Although RSC and edge rendering provide performance improvements, several limitations are pending. Edge runtime does not fully support Node.js API, which forces the developer to replace server-side libraries on which the application relies on. RSC also introduces architectural complexity by requiring a clear separation of components into server and client categories. Failing to comply may result in inefficient data fetching patterns or unexpected behavior during rendering.

Caching is another major challenge. Since RSC output depends on server-side state, caching strategies must be carefully designed to avoid sending outdated or inconsistent data across geographically distributed locations. In a distributed environment such as edge rendering, ensuring consistency while maintain performance and security can become extremely difficult. Furthermore, academic research on RSC and edge rendering remains limited as these technologies are relatively new. As adoption grows, more large-scale scientific evaluations will appear to evaluate long-term scalability, impact on energy consumptions, performance under different conditions, and security.

V. Conclusion

Modern mobile web performance continues to be shaped by increasing user expectations, tighter device constraints, and the importance of Core Web Vitals as indicators of real-world experience. Traditional rendering approaches such as CSR and SSR provide useful foundations yet struggle to fully meet the performance requirements of mobile environments, especially when faced with JavaScript-heavy applications and long network round trips. React Server Components and edge rendering are enforcing a new shift in application architecture by reducing JavaScript execution on the client and generating content closer to the user. The combination of these techniques improve key performance metrics such as LCP, FID/INP, and TTFB, while offering more consistent responsiveness under mobile network conditions. Although they introduce new challenges related to architecture, caching, and limited runtime APIs, their benefits create a strong foundation for the next generation of high-performance mobile web applications. As these technologies increase in adoption rates, further research and real-world evaluation will be essential to fully understand their impact and guide future development.

References

[1] V. Jain, "WEB VITALS AND CORE METRICS FOR WEB PERFORMANCE OPTIMIZATION," International Journal of Core Engineering & Management, vol. 7, no. 06, pp. 198–205, 2023.

[2] J. Kupoluyi, M. Chaqfeh, M. Varvello, W. Hashmi, L. Subramanian and Y. Zaki, "Muzeel: A Dynamic Event-Driven Analyzer for JavaScript Dead Code Elimination in Mobile Web," NYUAD Capstone Seminar Reports, Abu Dhabi, UAE, 2020.

[3] K. Matsudaira, "Making the Mobile Web Faster," Communications of the ACM, vol. 56, no. 3, pp. 56–61, 2013.

[4] N. Sabbag Filho, "Comparison between Client-Side Rendering and Server-Side Rendering: Impacts on Performance and User Experience," Leaders.Tec.Br, vol. 2, no. 8, 2025.

[5] T. F. Iskandar, M. Lubis, T. Kusumasari, A. R. Lubis, "Comparison between Client-Side and Server-Side Rendering in the Web Development," IOP Conference Series: Materials Science and Engineering, vol. 801, 2020.

[6] M. Satyanarayanan, "The Emergence of Edge Computing," Computer, vol. 50, pp. 30–39, 2017.

[7] E. Hosseini, "Edge Rendering: A Comprehensive Guide on Bringing the Web to the Edge," 2024.

[8] S. Iseal, "Edge Computing and React: Enhancing Performance at the Edge," 2025.

[9] Next.js Team, "Server and Client Components," Next.js Documentation, 2024.

[10] J. W. Comeau, "Making Sense of React Server Components," 2023.

[11] N. Makarevich, "React Server Components: Do They Really Improve Performance?," 2025.