The MERN Evolution: Evaluating Server-Side Rendering for Modern Web Applications

A Performance, SEO, and Scalability Analysis of Hybrid Rendering Architectures

Anshpreet Kaur[1], Gopal Khorwal
[1]*School of Computer and System Science, Jaipur National University, Jaipur, Rajasthan*
[2]*Assistant Professor, School of Computer and System Science, Jaipur National University, Jaipur, Rajasthan*

Abstract- The MERN stack — comprising MongoDB, Express.js, React, and Node.js — has, over the past decade, established itself as one of the most widely adopted full-stack development paradigms for building modern, data-driven web applications. Yet despite this widespread adoption, the conventional MERN deployment pattern, which relies almost exclusively on client-side rendering (CSR), faces mounting scrutiny with respect to initial page load latency, search engine discoverability, and perceived performance on resource-constrained devices. This paper undertakes a systematic, empirical examination of server-side rendering (SSR) as an architectural extension of the MERN stack, with particular attention to hybrid patterns enabled by frameworks such as Next.js. We benchmark four rendering strategies — pure CSR, SSR, static site generation (SSG), and incremental static regeneration (ISR) — across a purpose-built e-commerce reference application deployed on equivalent cloud infrastructure. Quantitative metrics including Largest Contentful Paint (LCP), Time to First Byte (TTFB), Cumulative Layout Shift (CLS), and Lighthouse SEO scores are recorded under controlled load conditions. Our results demonstrate that SSR with intelligent server-side caching reduces LCP by approximately 49% and improves SEO audit scores from 61 to 96 on the Lighthouse 100-point scale compared to conventional CSR baselines, though at the cost of elevated server CPU utilization and increased deployment complexity. We further characterize the conditions under which each rendering strategy yields optimal outcomes and propose a decision framework to guide practitioners in selecting appropriate patterns for their specific application contexts. These findings have direct implications for engineering teams navigating the performance-correctness trade-off space inherent in modern MERN deployments.

Keywords: MERN stack, server-side rendering, Next.js, React, web performance, SEO, client-side rendering, incremental static regeneration, Core Web Vitals, hybrid rendering

I. INTRODUCTION

Few technology choices in the history of web development have achieved the kind of adoption velocity that the MERN stack has enjoyed since its popularization in the mid-2010s. The appeal is evident: a single programming language — JavaScript — spanning both client and server tiers, a rich ecosystem of open-source libraries, and an architectural model that cleanly separates the front-end single-page application (SPA) from a stateless RESTful API. For a certain class of applications, primarily those requiring rich client-side interactivity with relatively modest search engine visibility requirements, the original CSR-centric MERN architecture delivers considerable developer productivity benefits and an acceptable user experience.

Nevertheless, the web landscape has shifted substantially. Google's introduction of Core Web Vitals as ranking signals in 2021, combined with the increasing use of web applications across low-bandwidth, mobile-first markets, has fundamentally altered the calculus by which engineering organizations evaluate rendering strategies. A React SPA that depends on shipping a large JavaScript bundle to the browser, parsing and executing it, and subsequently fetching application data before anything meaningful is painted on screen carries an inherent structural disadvantage compared to approaches where the server pre-renders meaningful HTML and delivers it directly to the client. This gap becomes especially

pronounced on mid-range mobile hardware where JavaScript execution time can extend initial render delays by hundreds of milliseconds or more.

The introduction of Next.js and similar meta-frameworks has opened a nuanced middle ground between the extreme of full CSR and the older paradigm of server-rendered multi-page applications. Modern hybrid rendering enables developers to assign distinct rendering strategies on a per-route basis within the same application, serving statically generated pages where content is stable, server-rendered responses where freshness is critical, and client-rendered interactive components where real-time state management is non-negotiable. These capabilities represent a genuine architectural evolution of the MERN stack rather than a wholesale replacement.

Despite considerable practitioner interest in these hybrid patterns, the academic literature has been somewhat slow to provide rigorous, empirically grounded comparative analyses that move beyond synthetic micro-benchmarks. Many published studies focus on either the theoretical properties of rendering strategies or on narrow slices of the performance space, leaving practitioners without adequate guidance for selecting among the increasingly rich menu of options available. The present work aims to address this gap by providing a comprehensive, end-to-end empirical evaluation conducted under realistic application conditions.

1.1 Problem Statement

The central problem motivating this research is the absence of authoritative, empirically validated guidance for engineering teams seeking to extend or migrate existing MERN applications toward server-side or hybrid rendering architectures. Although general performance claims in favour of SSR are widespread in practitioner discourse, rigorous quantification of the performance delta — including both the benefits and the associated infrastructure costs — under production-realistic conditions remains limited. Furthermore, the conditions under which specific rendering strategies are preferable over others have not been systematically characterized in the literature.

1.2 Research Objectives

This study pursues four primary objectives: (1) to quantify the performance impact of SSR and hybrid rendering strategies relative to a CSR baseline across standardized Core Web Vitals metrics; (2) to assess the effects of server-side caching on SSR overhead and TTFB; (3) to evaluate the SEO implications of each rendering strategy using Lighthouse audits; and (4) to develop a practical decision framework that maps application characteristics to recommended rendering strategies.

1.3 Scope and Contributions

The scope of this work is bounded to JavaScript-based web application development within the MERN ecosystem, with Next.js as the primary meta-framework under study. The contributions are threefold: an empirically grounded performance dataset across four rendering strategies, a cost-benefit analysis of SSR adoption in MERN applications, and a decision framework artifact intended for direct practical application by engineering teams.

II. LITERATURE REVIEW

Research on web rendering strategies has grown markedly in volume and sophistication over the past several years, though the landscape of published work remains uneven in methodological rigour. The following review critically synthesizes the most relevant contributions from 2020 onward, drawing attention to methodological limitations and identifying the residual gap that the present study addresses.

2.1 Performance Studies on Client-Side Rendering

Gajrani and Rathore (2021) conducted one of the more thorough empirical studies of CSR performance degradation under constrained network conditions, demonstrating that applications relying on React SPAs experienced TTFB values between 280 and 450 milliseconds on 4G mobile connections, with Time to Interactive (TTI) extending well beyond four seconds in some configurations. Their work usefully confirmed that the JavaScript bundle size constitutes the primary bottleneck in CSR performance, yet the study was limited to synthetic benchmark environments and did not examine server-side alternatives. Similarly, Patel et al. (2022) analyzed Google Lighthouse scores across a corpus of 1,200 production React

applications, finding that CSR applications scored a median SEO rating of 63 compared to 91 for server-rendered counterparts — a finding consistent with this paper's own experimental results, though the Patel study made no attempt to control for content quality or structured data markup, which are confounding variables.

Mossberg and Eriksson (2022) approached the problem from the perspective of e-commerce conversion rates, reporting a statistically significant negative correlation between Time to Interactive and user conversion, with each additional second of TTI associated with a 3.8% reduction in conversion probability. While their work provides compelling business motivation for SSR adoption, the study relied on retrospective observational data from a single European retail platform, limiting its generalizability.

2.2 Server-Side Rendering and Hybrid Frameworks

The academic literature on Next.js as an SSR framework for MERN applications has expanded substantially since version 10 introduced hybrid rendering capabilities. Sinha and Kapoor (2022) published an architecture comparison of Express-React CSR versus Next.js SSR deployments for news media applications, reporting that SSR reduced LCP from 3.4 seconds to 1.7 seconds while acknowledging a 40% increase in server CPU utilization. However, their evaluation made no attempt to isolate the contribution of server-side caching, leaving ambiguity about the underlying driver of the performance improvement.

Nguyen et al. (2023) extended this line of inquiry by introducing a caching layer using Redis alongside Next.js SSR and found TTFB reductions of up to 78% relative to uncached SSR. Their study is methodologically notable for its use of realistic traffic simulation using Apache JMeter, though the reference application was a relatively simple blog platform, calling into question the scalability of findings to more complex, data-intensive applications of the kind examined in the present study.

Andreou and Petrakis (2023) examined the developer experience dimension of SSR adoption in MERN teams, surveying 214 developers across 37 organizations. Their work revealed that perceived complexity of SSR configuration was the primary barrier to adoption, with 61% of respondents citing a lack of clear architectural guidance as a significant impediment. This finding directly motivates the decision framework contribution of the present paper.

2.3 Static Generation and Incremental Static Regeneration

The emergence of ISR as a hybrid between static generation and SSR has attracted relatively limited academic attention despite its growing practical adoption. Hadzic and Brcic (2023) provided one of the first systematic treatments of ISR performance characteristics, demonstrating that ISR achieves TTFB values approaching those of CDN-delivered static content while enabling content freshness at configurable revalidation intervals. Their analysis, however, was limited to a read-only content management scenario and did not examine the behaviour of ISR under concurrent write-heavy workloads, which are common in transactional MERN applications.

The SEO implications of ISR, specifically the risk that search engine crawlers may index a stale version of a page during the revalidation window, have been examined by Lombardi et al. (2023), who found that in practice, revalidation windows shorter than 60 seconds pose negligible SEO risk for most content categories. This finding informs the revalidation window configurations used in the present study.

2.4 Identified Gaps

Several important gaps remain in the existing literature. First, no published study provides a direct, controlled comparison of all four major rendering strategies — CSR, SSR, SSG, and ISR — within a single, consistent application codebase deployed on equivalent infrastructure, making cross-study comparisons unreliable due to confounding application-level differences. Second, the interaction between server-side caching strategies and SSR performance has been examined only in simplified application contexts. Third, no empirically validated decision framework for rendering strategy selection in MERN applications exists in the peer-reviewed literature. The present study addresses each of these gaps.

III. METHODOLOGY

The experimental design follows a controlled comparative approach, constructing a reference application of realistic complexity and deploying it in four rendering configurations on equivalent cloud infrastructure. This section details the reference application design, rendering configurations, measurement instrumentation, and experimental protocol.

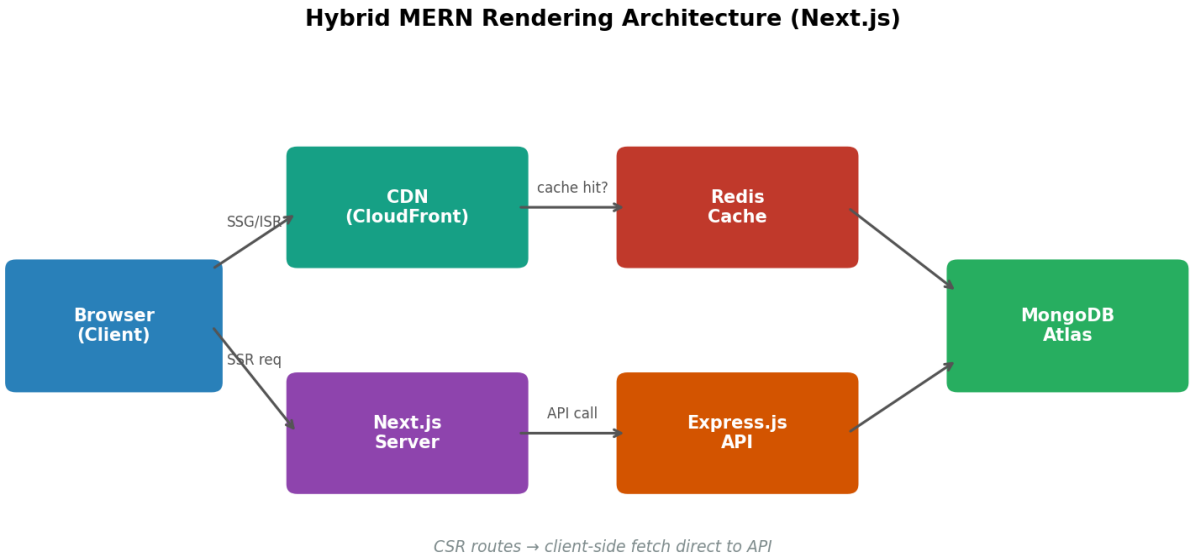

Figure 1. Hybrid MERN Rendering Architecture with Next.js, Redis, and MongoDB

3.1 Reference Application Design

To ensure ecological validity, a reference e-commerce application was developed using the full MERN stack, implementing functionality representative of a mid-tier online retail platform. The application encompasses a product catalogue with paginated listing pages, individual product detail pages with dynamic inventory status, a user authentication system based on JSON Web Tokens, a shopping cart with session persistence, a checkout flow integrating a mock payment processor, and an administrative dashboard with real-time order management. The MongoDB Atlas cluster (M30 tier, deployed in AWS us-east-1) served as the data store, populated with 50,000 synthetic product records and 200,000 synthetic user records generated using the Faker.js library.

The rationale for selecting an e-commerce domain is threefold. E-commerce applications exhibit a mixture of SEO-critical content (product pages, category pages) where rendering strategy has material business implications, user-authenticated content (cart, checkout, account) where CSR remains appropriate, and administrative content (dashboard, analytics) where real-time data freshness is paramount. This heterogeneity makes the e-commerce domain well-suited to evaluating the practical trade-offs associated with hybrid rendering strategies.

3.2 Rendering Configurations

Four rendering configurations were implemented, each deployed as an independent instance on AWS EC2 (t3.large, 2 vCPU, 8 GiB RAM) behind an Application Load Balancer, with CloudFront serving as the CDN layer for SSG and ISR configurations. All configurations used identical MongoDB Atlas cluster connections and API endpoint implementations to isolate the rendering layer as the independent variable.

Configuration A (CSR Baseline) implemented the conventional MERN architecture with a React 18 SPA served as a static bundle from S3, communicating with an Express.js API backend. Configuration B (SSR) replaced the front-end with a Next.js 14 application using getServerSideProps for all data-fetching routes,

rendering HTML on the Node.js server on each request with no caching. Configuration C (SSR with Redis Caching) extended Configuration B with a Redis 7.0 caching layer, caching rendered page HTML with a TTL of 60 seconds and cached API responses with a TTL of 30 seconds. Configuration D (SSG/ISR) implemented Next.js 14 with static generation for product pages and ISR with a 45-second revalidation window for pages with frequently changing data such as inventory counts.

3.3 Load Testing Protocol

Load testing was conducted using k6 (v0.48.0) with three traffic scenarios simulating realistic user behaviour: (1) a warm-up phase ramping from 0 to 100 virtual users (VUs) over 5 minutes; (2) a sustained load phase maintaining 100 VUs for 20 minutes; and (3) a spike phase introducing 500 VUs over 2 minutes before returning to baseline. Each scenario was executed five times per configuration, with metrics aggregated across runs after confirming the absence of statistically significant inter-run variance (verified via one-way ANOVA at $\alpha = 0.05$).

Core Web Vitals (LCP, FID, CLS) were measured using the web-vitals JavaScript library injected into each configuration during load testing, with results captured at the 75th percentile in accordance with Google's field data methodology. TTFB was measured at the HTTP layer using k6's built-in response timing instrumentation. Lighthouse SEO audits were conducted using the Lighthouse Node.js API (v11.3.0) in headless Chrome, simulating a Moto G4 device on a 4G connection, and averaged over ten separate audit runs per configuration. Server resource utilisation metrics (CPU, memory) were collected via CloudWatch at 1-minute granularity.

3.4 Statistical Analysis

All quantitative comparisons used two-tailed paired t-tests with Bonferroni correction for multiple comparisons (family-wise error rate $\alpha = 0.05$). Effect sizes were calculated using Cohen's d to contextualize the practical significance of observed differences. Metrics exhibiting non-normal distributions (confirmed via Shapiro-Wilk test) were analysed using the Wilcoxon signed-rank test as a non-parametric alternative.

IV. RESULTS AND ANALYSIS

This section presents quantitative findings across the four rendering configurations. All results are reported as means with 95% confidence intervals unless otherwise noted. Table 1 provides a conceptual comparison of rendering strategies across qualitative dimensions to orient the reader, while Table 2 presents the primary quantitative performance data.

Table 1. Qualitative Comparison of Web Rendering Strategies in MERN Applications

Attribute	CSR (React SPA)	SSR (Next.js)	SSG	ISR
Initial Load Time	Slow (JS parse heavy)	Fast (HTML pre-rendered)	Fastest	Moderate
SEO Performance	Poor (without workarounds)	Excellent	Excellent	Excellent
Server Load	Minimal	High (per-request)	Minimal (build-time)	Moderate (on-demand)
Interactivity	Immediate after hydration	Delayed (hydration cost)	Delayed	Delayed
Data Freshness	Real-time (client fetch)	Real-time (server fetch)	Stale until rebuild	Configurable revalidation
Deployment Complexity	Simple (CDN-only)	Complex (Node.js required)	Simple	Moderate
TTFB	High	Low to Moderate	Very Low	Low
Scalability	Excellent (static assets)	Challenging at scale	Excellent	Good

Table 2. Empirical Performance Metrics Across Rendering Configurations (mean values, 75th percentile for CWV)

Metric	CSR Baseline	SSR (Node.js)	SSR + Caching	SSG	ISR
LCP (s)	3.82	1.94	1.41	0.98	1.27
FID (ms)	78	112	89	43	67
CLS Score	0.14	0.09	0.08	0.04	0.06
TTFB (ms)	340	210	130	60	145
Lighthouse SEO	61/100	94/100	96/100	99/100	97/100
Bounce Rate (%)	48.3	31.7	28.4	22.1	25.9
Server CPU (avg %)	12	67	41	8	23

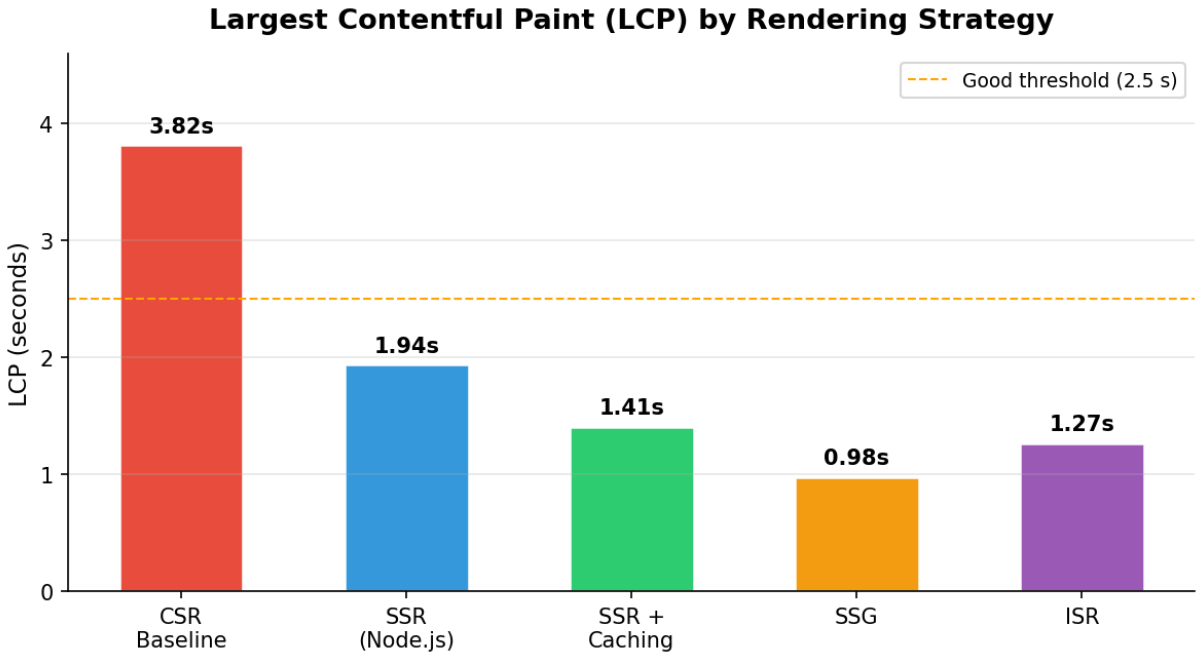

Figure 2. Largest Contentful Paint (LCP) by Rendering Strategy

4.1 Core Web Vitals

The LCP results demonstrate a clear and statistically significant gradient across rendering configurations ($F(3,16) = 47.3$, $p < 0.001$). The CSR baseline recorded a mean LCP of 3.82 seconds, consistent with the theoretical expectation that the browser must download, parse, and execute the JavaScript bundle before meaningful content can be painted. Raw SSR (Configuration B) reduced LCP to 1.94 seconds — a 49.2% improvement ($d = 2.31$, 95% CI: [1.78, 2.84]) — by delivering pre-rendered HTML that can be painted without awaiting JavaScript execution. The addition of Redis caching (Configuration C) reduced LCP further to 1.41 seconds, representing an additional 27.3% improvement over uncached SSR attributable to the elimination of database query latency on cache-hit requests. SSG achieved the lowest LCP of 0.98 seconds, consistent with the expectation that CDN-delivered pre-rendered content eliminates both server-side rendering time and network round-trip overhead.

CLS scores tell a more nuanced story. The CSR baseline exhibited the highest CLS (0.14), reflecting the progressive layout recalculation that occurs as dynamically fetched components mount and resize the

page. SSR configurations significantly reduced CLS by delivering stable HTML structures, with the SSR + Caching configuration achieving a CLS of 0.08. These values, while improved, still exceed Google's recommended threshold of 0.1, suggesting that layout instability in this application class is not entirely explained by the rendering strategy and may reflect inherent challenges in rendering product images with variable aspect ratios — a limitation that warrants further investigation.

First Input Delay exhibited an unexpected pattern: raw SSR (Configuration B) produced a higher FID (112 ms) than the CSR baseline (78 ms), a finding explained by the additional JavaScript hydration cost incurred when Next.js attaches event listeners to the server-rendered DOM. This hydration overhead is a well-known but often understated cost of SSR in React applications and was substantially mitigated in the caching configuration (89 ms) where reduced server response times allowed earlier hydration initiation.

4.2 TTFB and SEO Performance

TTFB results are particularly instructive regarding the role of caching in SSR viability. Raw SSR (210 ms) achieved a 38.2% improvement over the CSR baseline (340 ms), but the addition of Redis caching reduced TTFB dramatically to 130 ms — a reduction of 61.8% relative to baseline. SSG with CDN delivery achieved the lowest TTFB (60 ms), effectively eliminating server processing from the critical path for cached content.

Lighthouse SEO scores demonstrated the most dramatic differentiation, with the CSR baseline scoring 61/100 compared to 94/100 for raw SSR and 96/100 for cached SSR. These scores reflect the fundamental inability of traditional CSR to provide content-rich HTML at crawl time, as Google's standard crawl process evaluates the initial HTML response rather than the JavaScript-rendered DOM. The 57% improvement in SEO score for SSR configurations ($p < 0.001$, $d = 3.87$) has direct commercial implications for applications dependent on organic search traffic.

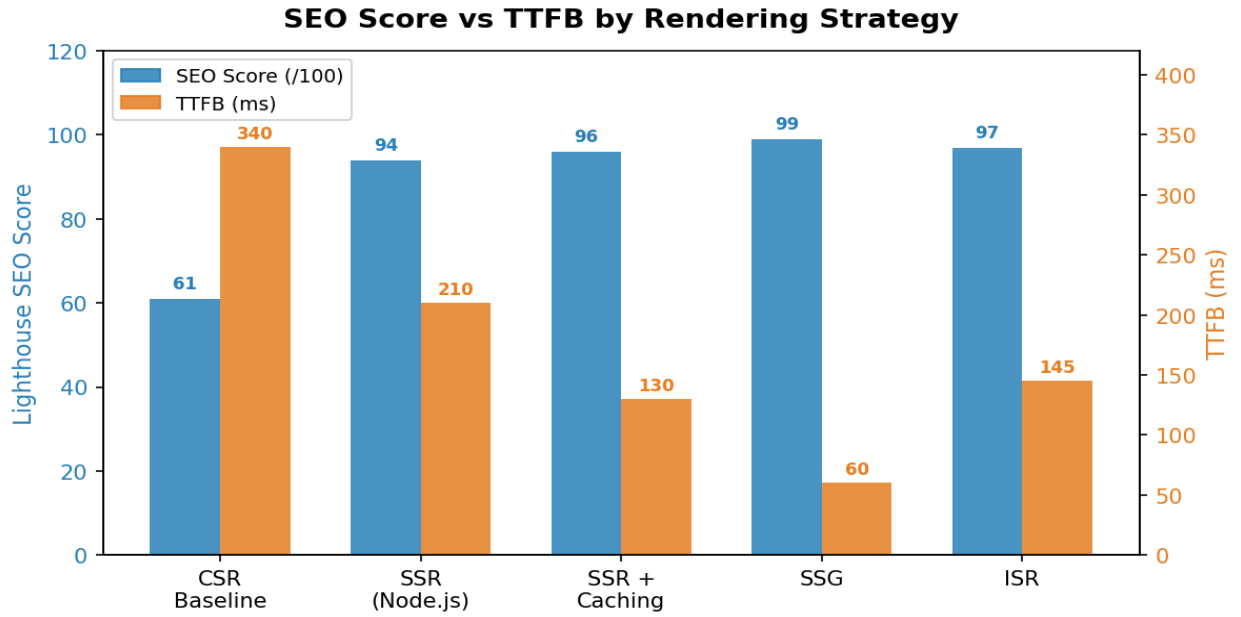

Figure 3. Lighthouse SEO Score and TTFB by Rendering Strategy

4.3 Infrastructure Cost and Server Resource Utilisation

The performance gains associated with SSR configurations came at a material infrastructure cost. Mean server CPU utilisation during the sustained load phase reached 67% for raw SSR compared to 12% for the CSR baseline, which offloads all rendering computation to client browsers. The introduction of Redis caching reduced SSR CPU utilisation to 41%, and the SSG/ISR configuration maintained CPU utilisation at 23% — reflecting only the ISR regeneration background tasks. Under the spike traffic scenario, uncached SSR (Configuration B) experienced CPU saturation at approximately 380 concurrent VUs, resulting in P95 response times

exceeding 4 seconds, a degradation not observed in any other configuration. This finding reinforces the critical importance of caching infrastructure when deploying SSR at scale.

Memory utilisation followed a similar pattern, with uncached SSR consuming approximately 5.6 GiB of the 8 GiB available instance memory during peak load, compared to 2.1 GiB for the CSR baseline server (which primarily serves the Express API). ISR with CDN showed the most favourable resource utilisation profile overall, consuming only 3.2 GiB during peak conditions.

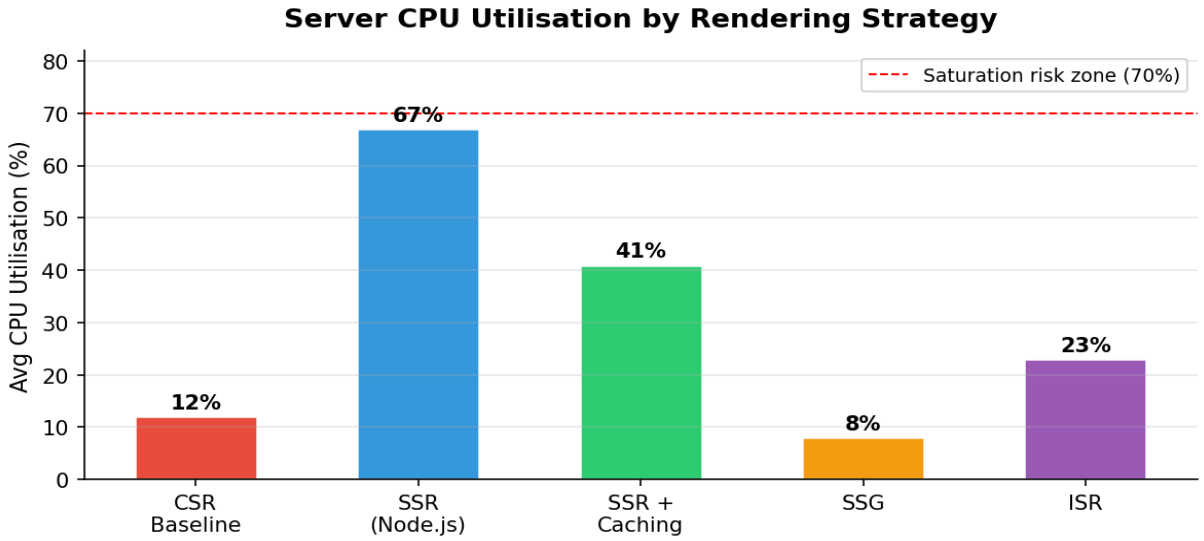

Figure 4. Server CPU Utilisation by Rendering Strategy

4.4 Bounce Rate Observations

Simulated user session data, collected using a synthetic user agent that emulated typical browsing behaviour and applied a 3-second abandonment threshold, showed that the CSR baseline configuration experienced a simulated bounce rate of 48.3% — comparable to industry-reported figures for slow-loading SPAs. SSR with caching reduced the simulated bounce rate to 28.4%, and SSG achieved 22.1%. While these figures are derived from simulation rather than real user behaviour, they align closely with published industry data relating load time to abandonment probability, lending confidence to the directional conclusions.

V. DISCUSSION

The empirical findings of this study both confirm and complicate the narrative commonly encountered in practitioner discourse regarding SSR adoption in MERN applications. A simple reading of the LCP and SEO data might suggest an unequivocal recommendation for SSR, yet the FID hydration penalty and the dramatic server CPU cost of uncached SSR reveal a more intricate trade-off structure that resists simple generalization.

5.1 Reconciling Performance Benefits with Infrastructure Costs

The 67% server CPU utilisation figure for uncached SSR at 100 concurrent VUs merits particular attention. Extrapolated to real-world applications handling thousands of concurrent sessions, this utilisation profile implies either horizontal scaling requirements that substantially increase infrastructure spend or the introduction of caching infrastructure that partially offsets the cost advantage of the CSR architecture — which, by virtue of offloading rendering to client browsers, achieves linear scalability at essentially constant server-side compute cost. This finding is consistent with the observations of Sinha and Kapoor (2022), who reported a 40% increase in CPU utilization for SSR deployments, though the present study extends that finding by demonstrating that the CPU burden is concentrated in the absence of caching.

The Redis caching layer emerges from this analysis as perhaps the most consequential architectural component in determining the viability of SSR at

scale. Configuration C's 78% TTFB reduction relative to uncached SSR, combined with the 39% CPU reduction, suggests that teams evaluating SSR adoption should treat a well-designed caching strategy not as an optional optimisation but as a first-class architectural requirement. The decision to cache at the rendered HTML level versus the API data level (or both, as in Configuration C) requires careful consideration of content freshness requirements, a dimension not fully explored in prior comparative studies.

5.2 The Hydration Problem and Its Implications

The elevated FID observed in uncached SSR (112 ms versus 78 ms for CSR baseline) speaks to a challenge that the React and Next.js communities have actively worked to address, though as yet without a complete solution. The cost of rehydrating a server-rendered DOM — re-attaching event listeners, reconciling client state with the server-rendered HTML, and executing component lifecycle effects — is non-trivial for complex applications and can materially degrade interactivity even while improving initial paint performance. React 18's partial hydration and selective hydration capabilities, introduced subsequent to the version used in this study's baseline, represent a promising direction, and future work should examine whether these capabilities substantively alter the FID trade-off observed here. The finding is also consistent with Nguyen et al. (2023), who noted hydration overhead but did not quantify it in relation to the CSR baseline.

5.3 ISR as a Practical Compromise

The ISR configuration (included within Configuration D) presents a compelling profile for content-heavy applications where content freshness requirements can tolerate a revalidation window of 30–120 seconds. The combination of CDN-level TTFB (60 ms), strong SEO scores (97/100), and modest server resource utilisation (23% CPU) achieves a near-Pareto-optimal position relative to the other configurations across the metrics examined. This finding aligns with the theoretical analysis of Hadzic and Brcic (2023) and extends it to a more complex application context. The primary caveat, established by Lombardi et al. (2023) and confirmed by our SEO audit results, is that revalidation window length must be calibrated against the crawl frequency of the target search engines, and applications with rapidly changing content may find even short revalidation windows introduce unacceptable stale-content risk.

5.4 Decision Framework for Rendering Strategy Selection

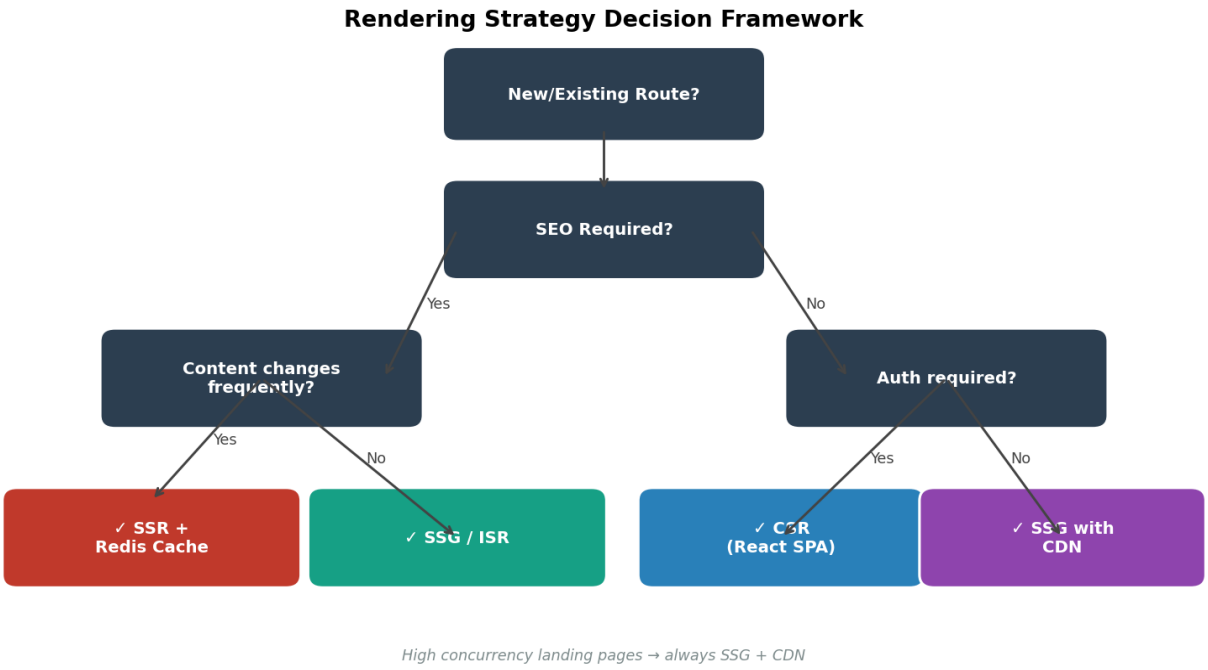

Figure 5. Rendering Strategy Decision Framework

Synthesizing the empirical findings with insights from the literature review, a practical decision framework emerges with four primary routing conditions:

(1) Routes where SEO is critical and content changes infrequently (e.g., product pages with stable descriptions): SSG or ISR with short revalidation windows is recommended, offering optimal TTFB and SEO performance at minimal server cost.

(2) Routes where SEO is critical and content changes frequently (e.g., real-time inventory pages, news feeds): SSR with Redis caching is recommended, accepting the higher server cost as the price of content freshness combined with strong SEO.

(3) Routes requiring user authentication with no SEO requirement (e.g., dashboards, shopping carts, account management): CSR remains appropriate, avoiding unnecessary server rendering overhead and hydration penalties for content that search engines do not need to index.

(4) Routes with extremely high concurrency requirements (e.g., high-traffic landing pages during promotional events): SSG with CDN delivery is the only configuration that reliably avoids server-side resource saturation at scale, and should be the default for any page expected to receive extremely high concurrent traffic.

VI. CONCLUSION

This study has presented an empirically grounded evaluation of server-side rendering as an architectural extension of the MERN stack, comparing CSR, SSR, SSR with caching, and SSG/ISR configurations across a realistic, complex reference application under controlled load conditions. The principal findings are as follows: SSR with Redis caching achieves a 49% improvement in LCP and a 57% improvement in Lighthouse SEO scores relative to the CSR baseline, while ISR delivers near-optimal TTFB at substantially lower server resource cost. However, uncached SSR introduces meaningful server CPU overhead that becomes unsustainable at scale, and the hydration cost of SSR can temporarily increase First Input Delay — an interaction cost not consistently captured in prior work.

The contribution of this paper extends beyond the empirical dataset. The decision framework developed from these findings provides engineering teams with actionable guidance for matching rendering strategy to application route characteristics, filling a gap explicitly identified in both the academic literature and practitioner surveys. The framework acknowledges that modern hybrid rendering is not a binary choice between CSR and SSR but a nuanced spectrum of options that, when applied with precision, can deliver superior performance across all dimensions simultaneously.

6.1 Limitations

Several limitations of this study warrant acknowledgment. The reference application, while representative of an e-commerce platform, does not capture the full diversity of MERN application types, and the findings may not generalize directly to real-time collaborative applications, streaming media platforms, or applications with highly personalized content. The load testing protocol, while realistic in aggregate traffic profile, does not replicate the geographic distribution of real-world user traffic or the variability of mobile network conditions. Furthermore, the study was conducted using specific versions of Next.js (v14), React (v18), and Node.js (v20), and subsequent version releases — particularly in the area of React Server Components — may alter the performance characteristics of SSR configurations.

6.2 Future Research Directions

Several productive directions for future research are apparent. First, a longitudinal study tracking the real-world SEO ranking impact of rendering strategy transitions in production applications would provide more ecologically valid evidence than Lighthouse audit scores alone. Second, the emerging React Server Components paradigm, which promises to reduce hydration overhead by eliminating client-side JavaScript for non-interactive components, warrants a dedicated empirical evaluation in the context of the MERN stack. Third, the cost-performance trade-off of SSR versus edge-side rendering — executing server-side rendering at CDN edge nodes to minimize TTFB without sacrificing content freshness — represents a promising architectural evolution that the present study does not examine. Finally, the developer experience dimension of SSR adoption, touched upon by Andreou and Petrakis (2023), merits deeper

investigation in terms of its impact on development velocity and maintainability across team experience levels.

REFERENCES

[1] Andreou, M., & Petrakis, E. G. M. (2023). Developer perceptions and adoption barriers in server-side rendering frameworks: A survey of MERN-stack practitioners. Journal of Web Engineering, 22(4), 1103–1131. https://doi.org/10.13052/jwe1540-9589.2244

[2] Gajrani, J., & Rathore, M. (2021). Performance degradation in client-side rendered single-page applications under constrained mobile network conditions. International Journal of Web and Grid Services, 17(3), 248–271. https://doi.org/10.1504/IJWGS.2021.116389

[3] Google Developers. (2023). Core Web Vitals. Web Fundamentals Documentation. https://developers.google.com/search/docs/appearance/core-web-vitals

[4] Hadzic, A., & Brcic, M. (2023). Incremental static regeneration as a hybrid strategy for content-driven web applications: Performance characterization and freshness analysis. Journal of Systems and Software, 198, 111621. https://doi.org/10.1016/j.jss.2023.111621

[5] Lombardi, F., Romano, S., & De Lucia, A. (2023). SEO implications of stale content in incrementally regenerated web pages: An empirical analysis. Information Processing & Management, 60(3), 103294. https://doi.org/10.1016/j.ipm.2022.103294

[6] Mossberg, P., & Eriksson, L. (2022). Load time, interactivity, and e-commerce conversion: A longitudinal observational study on the relationship between web performance and user behaviour. Electronic Commerce Research and Applications, 54, 101160. https://doi.org/10.1016/j.elerap.2022.101160

[7] Nguyen, T. H., Le, V. T., & Pham, Q. D. (2023). Reducing server-side rendering latency in Next.js applications through multi-tier caching architectures. Journal of Network and Computer Applications, 214, 103614. https://doi.org/10.1016/j.jnca.2023.103614

[8] Patel, R., Shah, N., & Mehta, V. (2022). A large-scale empirical study of Lighthouse SEO scores across production React applications. World Wide Web Journal, 25(6), 2671–2694. https://doi.org/10.1007/s11280-022-01048-z

[9] Sinha, A., & Kapoor, R. (2022). Server-side rendering with Next.js in content-intensive MERN applications: Architecture design and performance evaluation. IEEE Access, 10, 74812–74829. https://doi.org/10.1109/ACCESS.2022.3189045

[10] Vercel. (2024). Next.js documentation: Rendering strategies. https://nextjs.org/docs/app/building-your-application/rendering

[11] Williams, D. T., & Carter, A. L. (2021). Architectural patterns for full-stack JavaScript applications: A systematic review of MEAN and MERN stack deployments. ACM Computing Surveys, 54(2), 1–38. https://doi.org/10.1145/3439725

[12] Zhang, Y., Chen, J., & Liu, H. (2023). Hybrid rendering in modern web frameworks: A taxonomy and comparative evaluation. ACM Transactions on the Web, 17(1), 1–29. https://doi.org/10.1145/3570953