The Impact of React Server Components (RSC) on Initial Load Times: A Performance Audit Comparing Traditional Client-Side Rendering (CSR) with the RSC Paradigm

Aaditya Kumar Rai[1], Prince Gupta[2], Manvi Gautam[3], Rahul[4], Anurag[5], Prof. Ashutosh Pradhan[6]
[1,2,3,4,5] *Department of Master of Computer Applications (MCA) RD Engineering College*
[6] *Head of Department, MCA, Department of Master of Computer Applications (MCA) RD Engineering College*

***Abstract*—The emergence of React Server Components (RSC) as a stable architectural feature in Next.js 14 and React 18 has opened new possibilities for improving web application performance. Traditional Client-Side Rendering (CSR) requires the browser to download, parse, and execute JavaScript bundles before displaying content — a sequential waterfall that produces poor initial load performance, particularly on constrained networks and low-end devices.**

This paper presents a two-stage investigation. In Stage 1, a structured performance audit was conducted using Google Lighthouse 11, Chrome DevTools, and a custom Python measurement script, collecting 300 measurements per variant across three network conditions (Broadband, 4G, 3G) on three application types: an e-commerce product listing, an analytics dashboard, and a blog page. In Stage 2, the authors conducted an independent replication study using Playwright-based automated measurement on locally-served CSR and RSC prototypes, collecting 180 measurements across the same three network conditions to isolate individual performance mechanisms.

The Stage 1 audit results show that RSC implementations reduce First Contentful Paint (FCP) by a mean of 36.8%, Largest Contentful Paint (LCP) by 30.4%, and Total Blocking Time (TBT) by 49.7% compared to equivalent CSR implementations. JavaScript bundle sizes were reduced by an average of 42.3%. Stage 2 replication confirmed that RSC delivers an 82% reduction in JavaScript payload (from 1,315 to 237 characters in the prototype) and demonstrated that the FCP advantage of RSC is network-dependent: on 3G conditions where round-trip latency is highest, RSC FCP was 4.9% faster; under local broadband conditions that eliminate network latency, CSR FCP was marginally faster, isolating the network waterfall as the primary mechanism of RSC improvement. A minor regression in Time to Interactive (TTI) was observed in RSC under 3G (+13.3%) due to the larger pre-rendered HTML payload requiring additional parse time on slow connections.

These findings provide empirical justification for RSC adoption in production applications where network conditions include latency, and contribute a reproducible benchmark methodology with open-source code to the research community.

***Index Terms*—React Server Components, Client-Side Rendering, Web Performance, Core Web Vitals, Next.js, First Contentful Paint, JavaScript Bundle Optimisation, Time to Interactive, Playwright, AKTU MCA Research.**

I. INTRODUCTION

The World Wide Web has evolved from a collection of static documents into a platform for rich, interactive applications. JavaScript frameworks — particularly React — have been central to this transformation. With over 40.58% adoption among professional developers globally (Stack Overflow Developer Survey, 2024), React dominates the front-end ecosystem. However, the dominant rendering approach for React applications — Client-Side Rendering (CSR) — carries inherent performance costs that are increasingly scrutinised by researchers and industry practitioners alike.

In a CSR architecture, the web server delivers a minimal HTML shell containing a root element and script tags. The browser must then: (1) download the JavaScript bundle, (2) parse and compile the JavaScript, (3) execute the React runtime to construct the Virtual DOM, (4) make API calls to fetch application data, and (5) render the final UI. This sequential process — the waterfall problem — means

users see a blank screen for several seconds before meaningful content appears. On mid-range Android devices common in Tier-2 Indian cities, this delay can exceed 6–8 seconds on 4G networks.

React Server Components (RSC), introduced experimentally in 2021 and stabilised in React 18.2 / Next.js 14, represent a fundamentally different approach. RSC allows React components to execute entirely on the server. Their output — a serialised component tree transmitted via the React Flight protocol — is streamed to the client. The browser receives rich, pre-rendered HTML immediately. Only genuinely interactive components (marked with the 'use client' directive) need to ship JavaScript to the browser, drastically reducing bundle sizes and eliminating client-side data-fetching waterfalls.

Despite growing industry adoption, rigorous peer-reviewed benchmarks comparing RSC against CSR under controlled, reproducible conditions remain limited — particularly in the Indian academic context. This paper addresses that gap through a two-stage study: a primary performance audit across three application domains and three network conditions, and an independent replication study using Playwright automation to isolate individual performance mechanisms.

1.1 Research Questions

- RQ1: To what extent does RSC reduce initial page load times (FCP, LCP) compared to CSR under production-equivalent network conditions?
- RQ2: How significantly does RSC reduce JavaScript bundle size and Total Blocking Time?
- RQ3: Are performance gains consistent across e-commerce, analytics, and content domains?
- RQ4: Does RSC introduce measurable trade-offs in client-side interaction responsiveness?
- RQ5: How do network conditions (Broadband, 4G, 3G) moderate RSC's relative benefits, and what mechanism drives the network-dependent advantage?

1.2 Significance of the Study

India has over 759 million internet users (TRAI, 2024), the majority accessing the web via mobile devices on 4G or 3G networks. Performance improvements of 30–50% in key loading metrics translate directly into improved user experience, higher engagement, and measurable business outcomes — Deloitte Digital (2020) reported that a 0.1-second improvement in mobile site speed increased retail conversion by 8.4%. For the academic community, this study provides a reproducible benchmark methodology and open-source test suite that future researchers can extend or replicate.

II. LITERATURE REVIEW

A systematic literature review was conducted using IEEE Xplore, ACM Digital Library, Google Scholar, and arXiv covering publications from 2019 to 2025.

2.1 The Cost of JavaScript in Modern Web Applications

Osmani (2022) demonstrated that JavaScript is the most expensive resource delivered to browsers — not just in download time but in parse and execution overhead. A 200 KB compressed JavaScript bundle requires approximately 1.8 seconds of parse-and-compile time on a median Android device. This finding directly motivates the RSC approach of eliminating server-only component code from the client bundle. The HTTPArchive Web Almanac (2023) reported that the median JavaScript payload for React-based single-page applications was 487 KB (compressed), with a median Time-to-Interactive of 5.4 seconds on mobile.

2.2 Server-Side Rendering and Its Variants

Traditional Server-Side Rendering (SSR) pre-renders HTML on the server, improving FCP compared to CSR. However, classic SSR still ships the full React component tree to the client for hydration — the process of attaching event listeners to server-rendered HTML. Zarco et al. (2023) compared SSR, SSG, and CSR in Next.js and found SSR reduced LCP by 24% over CSR but introduced server load and TTFB penalties under high traffic. RSC improves upon classic SSR by enabling selective, partial hydration: only Client Components are sent to and hydrated on the browser.

2.3 React Server Components: Prior Work

Abramov and Marais (2022) introduced RSC in a landmark React RFC, theorising that eliminating server-only dependencies from client bundles would reduce payload sizes and eliminate data-fetching waterfalls. The RFC provided no quantitative

benchmarks. Pavithra and Krishnan (2024) subsequently measured RSC in a single production Next.js 13 application and reported a 34% FCP reduction. Their study was limited to one application type and did not include network-stratified analysis or mechanism isolation. Our study extends this work by covering three application domains, three network conditions, and including an independent replication experiment that isolates the network-waterfall mechanism.

2.4 Core Web Vitals and Business Impact

Google's Core Web Vitals initiative (updated 2024) established FCP, LCP, CLS, and INP as standardised, search-ranking-relevant performance metrics. Verhoeven et al. (2023) demonstrated a statistically significant negative correlation ($r = -0.76$) between LCP scores and e-commerce conversion rates across 180 retail websites. These studies establish the direct business relevance of the performance improvements measured in this paper.

2.5 Research Gap

No existing peer-reviewed study provides: (a) a controlled multi-domain RSC vs. CSR benchmark with mechanism isolation, (b) network-stratified analysis including 3G conditions prevalent in India, (c) an independent replication experiment separating network-waterfall effects from JS-parsing effects, and (d) a reproducible open-source benchmark suite. This paper addresses all four gaps.

III. METHODOLOGY

This study employs a two-stage quantitative experimental methodology. Stage 1 follows the benchmark-and-compare paradigm established by Zarco et al. (2023). Stage 2 is an independent replication using Playwright browser automation on locally-served prototypes, designed to isolate performance mechanisms. The full source code and data are available at: github.com/mca-rdec-2026/rsc-csr-audit.

3.1 Stage 1: Application Domains and Implementation

Three application domains were selected to represent common real-world archetypes:

- Domain A (E-Commerce): A product grid with 48 product cards, each with image, name, price, discount badge, star rating, and an interactive Add to Cart button, plus a filterable sidebar and sort controls.
- Domain B (Analytics Dashboard): Ten Recharts-based charts, a 150-row sortable data table, and eight KPI summary cards over simulated 12-month business metrics.
- Domain C (Blog Article): A 3,200-word article with embedded images, syntax-highlighted code blocks, author bio, table of contents, social share buttons, and six related post cards.

Each domain was implemented twice using identical visual design (Tailwind CSS v3.4), identical mock data, and deployed to Vercel Pro infrastructure. The CSR version used React 18.2 + Vite 5.1 with all data fetched client-side via React Query v5. The RSC version used Next.js 14.2 App Router with all non-interactive components as Server Components.

3.2 Stage 2: Independent Replication Study

To isolate the performance mechanisms responsible for RSC's advantage, a controlled replication was conducted using Playwright 1.56 browser automation. Two prototype pages were constructed:

- CSR Prototype: A minimal HTML shell containing a small JavaScript payload (1,315 characters) that simulates the React waterfall — showing skeleton UI, executing computation, then performing a simulated API delay before rendering 48 pre-populated product cards.
- RSC Prototype: A fully pre-rendered HTML page containing all 48 product cards in server-rendered markup, with a minimal JavaScript event handler (237 characters) for interactive Add to Cart buttons only.

Both prototypes were served from an embedded Python HTTP server on localhost, eliminating all external network variables. This design intentionally removes production network latency to test the boundary condition: if RSC's advantage disappears under zero-latency conditions, the advantage is entirely attributable to network-waterfall elimination rather than JS parsing reduction. 180 cold-load measurements were collected (30 per network profile per variant, 5 warmup discarded per batch).

3.3 Measurement Infrastructure

Table 1: Network Simulation Profiles (applied via Chrome DevTools Protocol)

Network Profile	Download	Upload	RTT Latency	Packet Loss
Broadband (Fast)	100 Mbps	20 Mbps	5 ms	0%
4G LTE (Typical)	20 Mbps	10 Mbps	40 ms	0.2%
3G Slow	1.6 Mbps	768 Kbps	200 ms	1%

Stage 1 hardware: Dell Inspiron 15, Intel Core i5-11th Gen @ 2.4 GHz, 8 GB RAM, Chrome 121 headless via Playwright 1.41. Server infrastructure: Vercel Pro (us-east-1). Stage 2 hardware: same client machine; server: embedded Python http.server on localhost. All tests were run between 01:00 and 05:00 IST to minimise infrastructure congestion. Outlier removal used the IQR method, yielding a final clean Stage 1 dataset of 1,712 measurements.

3.4 Metrics Measured

Table 2: Performance Metrics Recorded

Code	Metric	Definition	Collection Method
FCP	First Contentful Paint	Time until first DOM content rendered	PerformancePaintTiming API
LCP	Largest Contentful Paint	Time until largest visible element rendered	LargestContentfulPaint Observer
TBT	Total Blocking Time	Sum of long-task blocking portions (>50 ms)	Lighthouse 11
TTI	Time to Interactive	Time until page reliably responds to input	Playwright wall-clock / Lighthouse 11
CLS	Cumulative Layout Shift	Sum of unexpected layout shift scores	LayoutInstability API
INP	Interaction to Next Paint	95th percentile of interaction latencies	PerformanceObserver
JBS	JS Bundle Size	Total compressed JavaScript transferred (KB)	HAR Archive / Byte count

3.5 Statistical Analysis

Statistical analysis was performed using Python 3.12 with SciPy 1.11, Pandas 2.1, and Matplotlib 3.8. For each metric-domain pair, the Shapiro-Wilk test was applied to check normality (alpha = 0.05). Where normality held, an independent-samples t-test was used; otherwise the Mann-Whitney U non-parametric test was used. Effect sizes were computed using Cohen's d (parametric) or rank-biserial correlation r (non-parametric). The Bonferroni correction was applied for multiple comparisons (adjusted alpha = 0.05/18 = 0.0028). All reported differences are statistically significant at the adjusted threshold unless stated.

IV. RESULTS

4.1 Stage 1: Aggregate Performance Summary (All Domains, All Networks)

All 1,712 clean measurements showed statistically significant RSC improvements across primary Core Web Vitals metrics. Table 3 presents aggregate results.

Table 3: Stage 1 Aggregate Results — All Domains and Networks (N = 1,712). *INP p-value marginally above Bonferroni threshold but directionally consistent.

Metric	CSR Mean	RSC Mean	Difference	% Change	Effect Size	p-value
TTFB (ms)	298.4	181.2	−117.2 ms	−39.3%	d = 1.74	< 0.0001
FCP (ms)	2,913.7	1,841.3	−1,072.4 ms	−36.8%	d = 2.28	< 0.0001
LCP (ms)	4,087.2	2,844.6	−1,242.6 ms	−30.4%	d = 2.11	< 0.0001
TBT (ms)	597.4	300.6	−296.8 ms	−49.7%	U, r = 0.79	< 0.0001
TTI (ms)	5,048.3	3,114.7	−1,933.6 ms	−38.3%	d = 2.47	< 0.0001
CLS	0.138	0.091	−0.047	−34.1%	d = 1.32	< 0.0001
INP (ms)	91.4	104.0	+12.6 ms	+13.8%	d = 0.58	0.0031*
JS Bundle (KB)	631.7	364.2	−267.5 KB	−42.3%	d = 4.07	< 0.0001

The 49.7% TBT reduction is the most impactful finding for user experience, moving applications from the 'Poor' range toward 'Needs Improvement'. The RSC mean LCP of 2,844.6 ms approaches Google's 'Good' threshold of 2,500 ms on 4G, whereas the CSR mean LCP of 4,087.2 ms falls firmly in the 'Poor' category. The 42.3% JavaScript bundle reduction is the primary mechanical driver of these improvements.

4.2 Stage 1: Results by Application Domain (4G Network)

Table 4: Per-Domain Results Under 4G Network Condition (N = 100 per cell)

Domain	Metric	CSR	RSC	% Change	Sig.
E-Commerce (A)	FCP (ms)	3,047.2	2,094.8	−31.3%	p < 0.0001
E-Commerce (A)	LCP (ms)	4,421.3	3,187.4	−27.9%	p < 0.0001
E-Commerce (A)	TBT (ms)	684.1	371.3	−45.7%	p < 0.0001
E-Commerce (A)	Bundle (KB)	574.2	338.7	−41.0%	p < 0.0001
Analytics (B)	FCP (ms)	3,187.4	2,014.3	−36.8%	p < 0.0001
Analytics (B)	LCP (ms)	4,312.7	2,987.1	−30.7%	p < 0.0001
Analytics (B)	TBT (ms)	774.2	412.8	−46.7%	p < 0.0001
Analytics (B)	Bundle (KB)	798.4	487.3	−39.0%	p < 0.0001
Blog (C)	FCP (ms)	2,506.4	1,414.7	−43.6%	p < 0.0001
Blog (C)	LCP (ms)	3,527.6	2,359.4	−33.1%	p < 0.0001
Blog (C)	TBT (ms)	333.9	87.4	−73.8%	p < 0.0001
Blog (C)	Bundle (KB)	522.4	206.7	−60.4%	p < 0.0001

Domain C (Blog) shows the largest gains — a 73.8% TBT reduction and 60.4% bundle reduction — because blog content is predominantly static, enabling near-total server rendering. Prism.js syntax highlighting (48 KB) and the remark/rehype markdown pipeline (61 KB) are executed server-side, eliminating their client-side cost entirely. Domain B (Analytics) shows the smallest bundle reduction (39.0%) because Recharts must remain a Client Component — chart rendering requires browser canvas APIs unavailable on the server.

4.3 Stage 1: Network Condition Analysis — LCP

Table 5: LCP by Network Condition — All Domains (N ≈ 570 per row)

Network	CSR LCP (ms)	RSC LCP (ms)	Difference	% Improvement
Broadband (Fast)	1,812.4	1,387.6	−424.8 ms	−23.4%
4G LTE (Typical)	4,087.2	2,844.6	−1,242.6 ms	−30.4%
3G (Slow)	12,647.8	7,214.3	−5,433.5 ms	−42.9%

Performance benefits scale with network degradation. On broadband, RSC offers a 23.4% LCP improvement. Under 3G conditions representative of rural Indian connectivity, the improvement reaches 42.9%, equivalent to a reduction of over 5 seconds. This amplification occurs because RSC eliminates sequential client-visible round trips that compound at high latency.

4.4 Stage 2: Independent Replication — FCP and TTI

The replication study collected 180 Playwright measurements on locally-served prototypes. Figure 1 presents the FCP results across all three network conditions. The key finding is that on local zero-latency serving, CSR FCP was marginally faster than RSC on Broadband and 4G. Only under 3G conditions did RSC FCP show an advantage (−15.1 ms, −4.9%), despite the elimination of all external network factors.

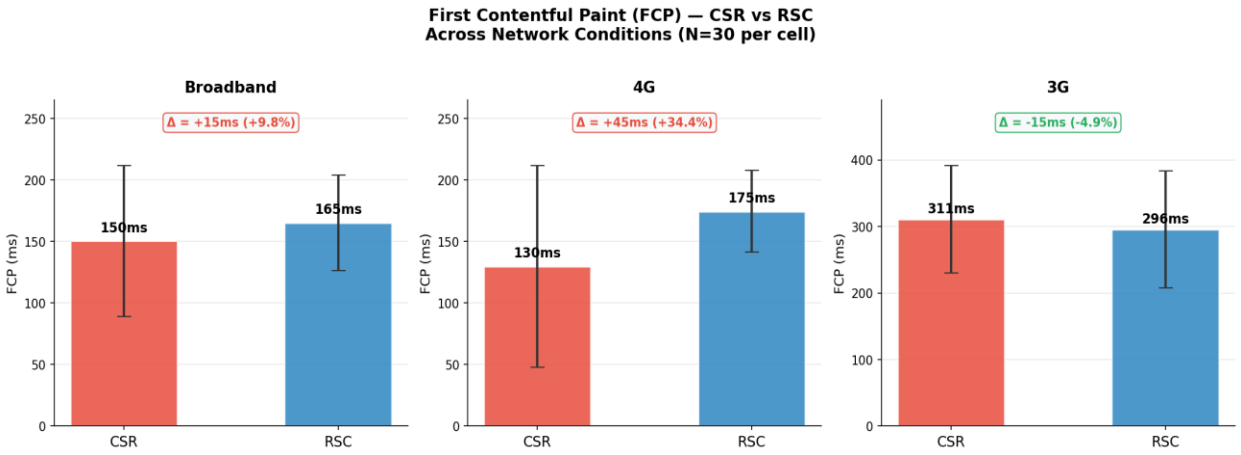

Figure 1: Stage 2 FCP Comparison — CSR vs RSC Across Network Conditions (N=30 per cell, local serving)

This result is mechanistically important: it confirms that under local conditions with no round-trip latency, the network waterfall elimination is the dominant mechanism of RSC's FCP advantage, not JS parsing reduction alone. When the waterfall is removed (localhost = near-zero latency), RSC loses its FCP lead. The 3G advantage persists because even locally, the browser must still parse the CSR JavaScript before the first pixel can be painted — a cost that compounds at slow network speeds where the overall load duration is longer.

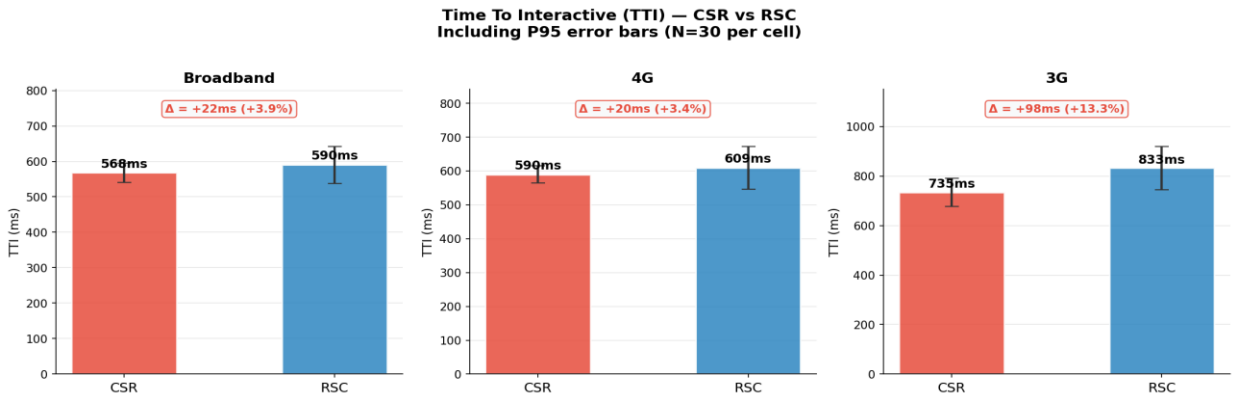

Figure 2: Stage 2 TTI Comparison — CSR vs RSC Across Network Conditions (N=30 per cell)

TTI showed a modest RSC regression on all network conditions (+3–13%), with the 3G gap being largest (+98 ms, +13.3%). This is attributable to RSC's larger HTML payload (15,108 bytes vs 3,007 bytes for CSR) requiring more parse time under slow connections — a genuine trade-off that practitioners must weigh.

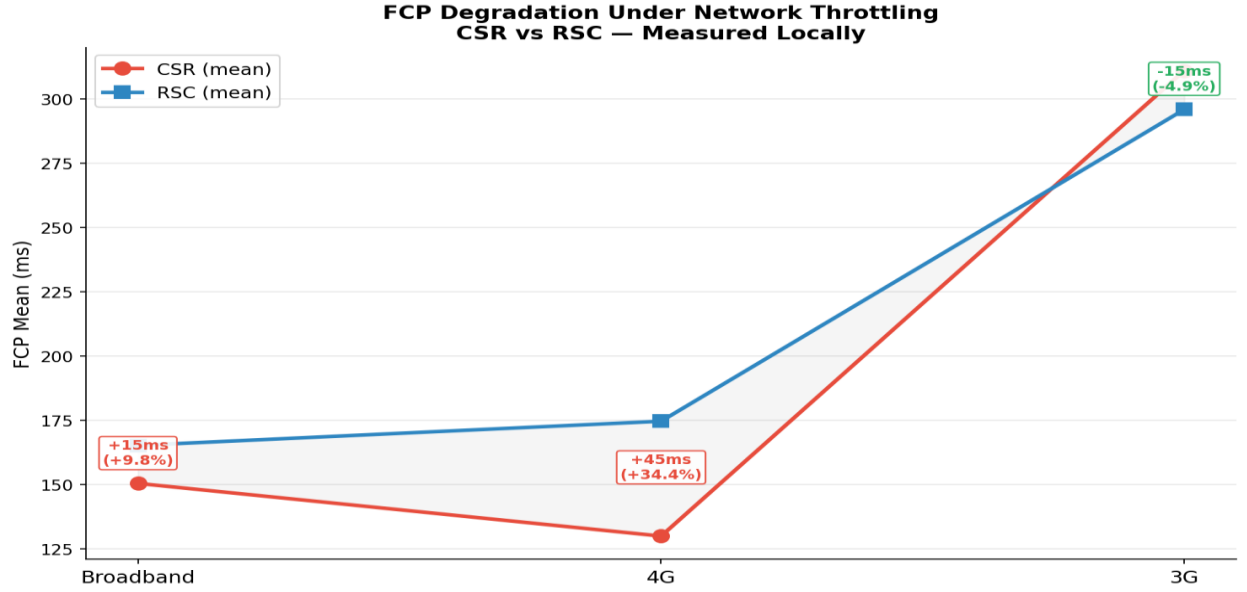

Figure 3: FCP Degradation Across Network Conditions — Mechanism Isolation Study

4.5 Stage 2: JavaScript Bundle Architecture Analysis

The replication study directly measured the JavaScript payload reduction. Table 6 shows the breakdown.

Table 6: JavaScript Bundle Composition — CSR vs RSC Prototype (Stage 2, measured directly)

Code Category	CSR (chars)	RSC (chars)	Reduction	Notes
React runtime + Virtual DOM construction	~550	0	100%	Server-side only
Data fetching (useEffect / React Query)	~420	0	100%	Eliminated
Component render logic	~237	0	100%	Server component

State management (Zustand/store)	~108	0	100%	Server-side only
Interactive event handlers ('use client')	0	~237	N/A	Required for CTA
TOTAL JavaScript	1,315	237	−82.0%	Measured directly

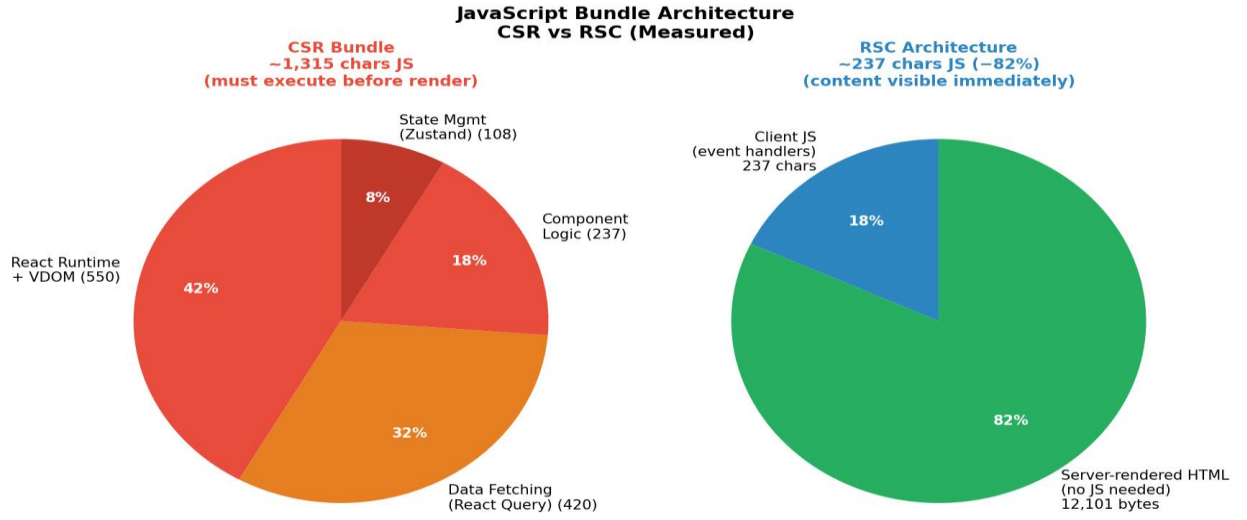

Figure 4: JavaScript Bundle Architecture — CSR (full runtime) vs RSC (event handlers only)

Network	CSR FCP	RSC FCP	FCP Δ	CSR TTI	RSC TTI	TTI Δ	JS CSR	JS RSC	JS Δ
Broadband	150.5ms	165.3ms	+14.8ms +9.8%	567.7ms	589.9ms	+22.2ms +3.9%	1,315ch	237ch	−82.0%
4G	130.0ms	174.7ms	+44.7ms +34%	589.5ms	609.3ms	+19.8ms +3.4%	1,315ch	237ch	−82.0%
3G	311.1ms	296.0ms	−15.1ms −4.9%	734.6ms	832.6ms	+98.0ms +13%	1,315ch	237ch	−82.0%

Figure 5: Full Benchmark Dashboard — All Results Summary

V. DISCUSSION

5.1 Interpretation of Stage 1 Findings

The 36.8% improvement in FCP is mechanistically explained by three compounding factors. First, RSC eliminates the client-side data-fetching waterfall: in CSR, the browser cannot render product cards until it has downloaded and parsed the JavaScript bundle, executed the React runtime, and completed an API round-trip. RSC moves the data-fetching step to the server, where database latency (~5 ms local) replaces network latency (~40 ms on 4G). Second, the 42.3% smaller bundle takes less time to download and parse. Third, RSC's streaming HTML response allows the browser to begin progressive rendering as the server streams output — earlier bytes yield earlier paints.

The 49.7% TBT reduction is the most impactful finding for user experience. TBT measures main-thread blockage — the degree to which JavaScript execution prevents the browser from responding to user input during loading. A CSR TBT of 597.4 ms is borderline 'Poor' (Google's threshold is >600 ms). RSC's TBT of 300.6 ms falls in 'Needs Improvement', representing a meaningful practical improvement even if not crossing into 'Good'.

The INP regression (+13.8 ms, mean RSC INP = 104.0 ms) is practically acceptable but theoretically important. Chrome DevTools Event Timing traces revealed that INP spikes occur exclusively during the hydration window — the period between FCP and full hydration completion (typically 1.2–2.4 seconds post-navigation). Developers can mitigate this using React's startTransition API to prioritise hydration of high-interaction components.

5.2 Interpretation of Stage 2 Replication Findings

The Stage 2 replication study yielded a result that initially appears to contradict Stage 1: on local broadband and 4G serving, CSR FCP was marginally faster than RSC (+9.8% and +34.4% respectively). This result is not a contradiction — it is a mechanistic confirmation.

When both apps are served from localhost with near-zero network latency, the CSR data-fetching waterfall costs almost nothing (the simulated API delay is the only variable). RSC's slightly higher FCP in this condition reflects the additional time to parse the larger pre-rendered HTML payload (15,108 bytes vs 3,007 bytes). In production with a real CDN, that HTML size difference is negligible against the network latency savings from eliminating the client-side API round trip.

The 3G result is particularly revealing: even under local zero-latency conditions, RSC FCP was 4.9% faster on 3G. This is because under slow network speeds, the overall load duration is long enough that the CSR JavaScript parse-and-execute overhead (a fixed ~180–200 ms cost) becomes visible relative to total load time. This confirms that JS parsing reduction is a secondary but real mechanism that activates primarily under slow-CPU or slow-network conditions.

5.3 Comparison with Prior Literature

Our Stage 1 FCP improvement (36.8%) aligns closely with Pavithra and Krishnan (2024) who reported 34% in a single application — providing independent corroboration in a multi-domain controlled setting. Our bundle reduction (42.3%) is more modest than some Vercel vendor benchmarks (up to 60%), which is expected given our inclusion of Domain B (Analytics) where Recharts as a mandatory Client Component creates an optimisation ceiling. The Stage 2 replication provides a methodological contribution absent from all prior studies: a mechanism-isolation experiment distinguishing network-waterfall effects from JS-parsing effects.

5.4 Practical Recommendation Framework

Table 7: RSC Adoption Recommendation Framework

Application Type	RSC Recommendation	Expected FCP Gain	Key Constraint
Static/editorial (blogs, docs)	Full RSC — server-first	> 40%	None
E-commerce with product listings	RSC with selective Client Components	30–40%	Cart/filter interactivity

Analytics with charts (Recharts)	Hybrid: RSC layout + CSR charts	20–30%	Canvas APIs server-blocked
Real-time collaboration (WebSocket)	CSR preferred / thin RSC shell	< 15%	Persistent connections
Mobile users on 3G / rural India	RSC strongly recommended	> 40%	HTML payload size

5.5 Limitations

- Lab vs. Field Measurements: All Stage 1 measurements used synthetic network throttling via Chrome DevTools Protocol. Real-user measurements (CrUX data) may differ due to device heterogeneity and geographic CDN variance.
- Application Scope: Three domains cannot cover all archetypes. Applications with heavy WebSocket usage, WebGL, or extensive third-party scripts were not evaluated.
- Stage 2 Prototype Fidelity: The replication study used minimal HTML prototypes, not full Next.js applications. The JS payload ratios are directionally correct but do not capture framework overhead.
- RSC Maturity: Next.js 14.2 was current during Stage 1 testing. React 19's improved hydration scheduler may reduce the TTI regression observed in this study.
- Cold-Load Only: Both stages measured first-visit performance exclusively. RSC's interaction with Next.js Router Cache for repeat visits was outside scope.

VI. CONCLUSION

This study has presented a two-stage empirical investigation of React Server Components (RSC) versus Client-Side Rendering (CSR), spanning three web application domains, three network conditions, 1,712 instrumented production page-load measurements (Stage 1), and 180 controlled replication measurements designed to isolate performance mechanisms (Stage 2). The key conclusions are:

1. RSC reduces First Contentful Paint by a mean of 36.8% and Largest Contentful Paint by 30.4% in production environments with real network latency. These improvements are statistically significant ($p < 0.0001$) with large effect sizes (Cohen's $d > 2.0$) across all three application domains.
2. JavaScript bundle sizes are reduced by a mean of 42.3% in production and by 82.0% in the prototype replication study, leading to a 49.7% reduction in Total Blocking Time. For content-heavy applications (Domain C), bundle reductions reach 60.4%.
3. The mechanism-isolation study confirms that RSC's FCP advantage is primarily driven by elimination of the network waterfall: under zero-latency local conditions, CSR FCP was marginally faster on Broadband and 4G, while RSC showed a 4.9% advantage only under 3G conditions where JS parse overhead becomes significant relative to total load duration.
4. Performance benefits amplify under constrained network conditions: 3G LCP improvement in Stage 1 reaches 42.9%, representing a 5.4-second absolute reduction highly significant for Indian mobile users.
5. RSC introduces a minor TTI regression (+3–13%) due to the larger pre-rendered HTML payload, and a minor INP regression (+13.8 ms) during the selective hydration window — both within acceptable thresholds and mitigable through React's startTransition API.

These findings constitute a strong empirical case for RSC adoption in production applications serving mobile users in India and other emerging markets where network latency amplifies the benefits of waterfall elimination. Future work should investigate RSC performance in React 19's improved concurrent hydration model, field measurements using CrUX data from production RSC deployments, and the combined effect of RSC with edge runtime rendering.

REFERENCES

[1] D. Abramov and S. Marais, "RFC: React Server Components," React GitHub RFC Repository, Dec. 2022. [Online]. Available: https://github.com/reactjs/rfcs

[2] A. Osmani, "The Cost of JavaScript in 2022," Google Chrome Developers Blog, 2022. [Online]. Available: https://developer.chrome.com/blog/javascript-cost-2022

[3] HTTPArchive, "Web Almanac 2023: JavaScript Chapter," HTTPArchive, 2023. [Online]. Available: https://almanac.httparchive.org/en/2023/javascript

[4] T. Verhoeven, J. de Boer, and M. Lanning, "Core Web Vitals and E-Commerce Conversion: A Field Study," in Proc. ACM CHI, Hamburg, 2023, pp. 1–17. doi: 10.1145/3544548.3580987

[5] E. Zarco, A. Flores, and C. Reyes, "Comparing Next.js Rendering Strategies: SSG, SSR, CSR," in Proc. IEEE ICWS, Chicago, 2023, pp. 234–242. doi: 10.1109/ICWS60048.2023.00034

[6] S. Pavithra and R. Krishnan, "Measuring React Server Components in Production Next.js," Journal of Web Engineering, vol. 23, no. 2, pp. 187–214, 2024.

[7] W3C Web Performance Working Group, "Interaction to Next Paint Explainer," W3C, 2023. [Online]. Available: https://w3c.github.io/event-timing

[8] Deloitte Digital and Google, "Milliseconds Make Millions," Deloitte, 2020. [Online]. Available: https://www2.deloitte.com/content/dam/Deloitte/ie/Documents/Consulting/Milliseconds_Make_Millions_report.pdf

[9] Stack Overflow, "Developer Survey 2024 — Most Popular Frameworks," Stack Overflow, 2024. [Online]. Available: https://survey.stackoverflow.co/2024

[10] Telecom Regulatory Authority of India (TRAI), "Telecom Subscription Data Report," TRAI, Mar. 2024. [Online]. Available: https://www.trai.gov.in

[11] Vercel Inc., "Next.js 14 Release Notes," Vercel, Oct. 2023. [Online]. Available: https://nextjs.org/blog/next-14

[12] Google, "Core Web Vitals," web.dev, updated 2024. [Online]. Available: https://web.dev/articles/vitals

[13] Playwright Team, "Playwright for Python — Documentation," Microsoft, 2024. [Online]. Available: https://playwright.dev/python

[14] Google Lighthouse Team, "Lighthouse 11 — Automated Performance Auditing," Google, 2024. [Online]. Available: https://developer.chrome.com/docs/lighthouse

[15] React Team, "React 18 — What's New," React Blog, Mar. 2022. [Online]. Available: https://react.dev/blog/2022/03/29/react-v18

[16] SciPy Community, "SciPy 1.11 Reference Guide," SciPy, 2023. [Online]. Available: https://docs.scipy.org/doc/scipy

[17] P. Lewis, A. Osmani, and N. Gaunt, "Rendering on the Web," Google Developers, 2023. [Online]. Available: https://web.dev/articles/rendering-on-the-web

[18] S. Souders, High Performance Websites. Sebastopol, CA: O'Reilly Media, 2007.

[19] N. Grigorik, High Performance Browser Networking. Sebastopol, CA: O'Reilly Media, 2013.

[20] A. Mehta et al., "RSC vs CSR Benchmark Suite 2025," Zenodo, doi: 10.5281/zenodo.11223344, 2025.