

Design and Optimization of an Intercity Travel Booking Platform: A Case Study of FlyNayTrip

Hemanshu Patel

B.E. Computer Science Engineering

Abstract

The rapid proliferation of mobile internet access across tier-2 and tier-3 cities in India has fundamentally altered the digital travel landscape, creating both an opportunity and a technical challenge for intercity booking platforms. This paper presents the design, development, and systematic optimization of FlyNayTrip, a mobile-first intercity travel booking platform developed as part of an industry internship engagement. The system addresses the unique performance constraints imposed by 2G and 3G network conditions, limited device memory, and heterogeneous browser environments prevalent among its target user base.

The platform integrates a React-based frontend, RESTful API communication via Axios, and a structured backend service layer. The core contribution of this work lies in the identification and remediation of critical performance bottlenecks through techniques including route-level code splitting, lazy loading of non-critical components, in-memory and HTTP caching of API responses, and strategic image optimization. Empirical measurements obtained through Lighthouse auditing and field data collection demonstrate significant improvements across Core Web Vitals: Largest Contentful Paint (LCP) decreased from 5.1 seconds to 2.3 seconds, First Input Delay (FID) reduced from 210 ms to 78 ms, and Cumulative Layout Shift (CLS) improved from 0.31 to 0.07. Overall initial page load time was reduced by approximately 57%, and the JavaScript bundle size decreased from 1.82 MB to 0.64 MB following tree-shaking and selective vendor chunk splitting. The booking completion rate improved from 54% to 73%, representing a statistically meaningful uplift in user conversion. The paper further discusses the architectural trade-offs encountered during implementation, the inherent limitations of a student-scale deployment, and prospective directions for future engineering work.

Keywords: Mobile-First Design, Web Performance Optimization, Core Web Vitals, React, Lazy Loading, Code Splitting, API Caching, Intercity Travel, Axios, LCP, FID, CLS, Low-Bandwidth Networks, User Experience

1. Introduction

The Indian domestic travel market has experienced structural growth over the past decade, with the National Payments Corporation of India reporting a consistent year-on-year increase in digital ticket transactions. Unlike air or rail travel, the intercity bus and shared-cab segment has historically been fragmented and poorly digitized, with a significant portion of bookings occurring at physical counters. The emergence of affordable smartphones and prepaid data plans, even at suboptimal 2G/3G speeds, has created a viable channel for digital booking adoption in markets that were previously considered unreachable.

However, building for this demographic requires an engineering approach that differs substantially from building for urban, broadband-connected users. Industry benchmarks published by Google indicate that 53% of mobile users abandon a site that takes longer than three seconds to load (Google, 2022). For users on 3G networks, where typical throughput ranges between 0.5 Mbps and 2 Mbps, an unoptimized single-page application can take upwards of seven to ten seconds to reach interactivity — a threshold at which abandonment rates approach 70% (Akamai Technologies, 2021).

FlyNayTrip was conceived as a pragmatic response to these constraints. Developed during a structured six-month internship, the platform enables users to search available intercity routes, view operator details and seat availability, and complete bookings through an integrated payment gateway. The system was designed with a mobile-first philosophy from the outset, acknowledging that over 80% of target users access the service via a mobile browser rather than a desktop client.

This paper documents the complete engineering arc of FlyNayTrip: from initial architecture decisions through to iterative performance tuning, concluding with an empirical analysis of outcomes. The paper is organized as follows: Section 2 reviews relevant prior work and industry standards; Section 3 articulates the problem statement; Sections 4 and 5 describe the system architecture and methodology; Sections 6 and 7 detail implementation and optimization techniques; Section 8 presents results; and Sections 9 through 11 provide discussion, conclusions, and future directions.

2. Literature Review

2.1 Web Performance and Core Web Vitals

Google's introduction of Core Web Vitals (CWV) in 2020 formalized a framework for quantifying user-centric web performance. The three primary metrics — Largest Contentful Paint (LCP), First Input Delay (FID), and Cumulative Layout Shift (CLS) — each correspond to a discrete dimension of perceived user experience: loading, interactivity, and visual stability respectively (Google Developers, 2023). LCP measures the render time of the largest visible content element, with a threshold of 2.5 seconds or less considered 'good'. FID captures the latency between user interaction and browser response, with 100 ms as the upper bound for a good score. CLS quantifies unexpected layout shifts using an aggregate impact fraction score, with values below 0.1 considered acceptable. As of 2023, CWV signals directly influence Google Search ranking, making their optimization commercially as well as technically significant.

2.2 Mobile UX and Bandwidth Constraints

Extensive research from Deloitte Digital and the Aberdeen Group has established a direct correlation between page load speed and commercial performance indicators. A one-second delay in mobile page load time is associated with a 7% reduction in conversions and an 11% reduction in page views (Aberdeen Group, 2019). Nah (2004) established that users typically begin abandoning tasks if response delays exceed 1.5 to 2 seconds for interactive operations, a finding that remains directionally valid in mobile web contexts despite shifting user expectations.

The W3C Web Performance Working Group has developed a suite of browser APIs — including the Navigation Timing API, Resource Timing API, and the Long Tasks API — that allow web applications to instrument their own performance characteristics in production environments. These tools were employed in the FlyNayTrip instrumentation strategy.

2.3 Single-Page Application Performance

The proliferation of JavaScript-heavy single-page application (SPA) frameworks such as React, Vue, and Angular has introduced well-documented performance challenges. Wagner (2019) identifies bundle bloat, uncontrolled third-party script loading, and render-blocking resources as primary contributors to poor LCP scores in SPAs. Code splitting, pioneered as a pattern by the webpack bundler and further simplified by React's native `lazy()` API and Suspense component, provides a mechanism to defer the loading of route-specific JavaScript until it is required, substantially reducing the initial download payload. Grigorik (2013) provides a foundational treatment of network performance optimization from an HTTP/TCP perspective, establishing the case for reducing round-trip latency through resource caching, connection reuse, and payload compression — all of which were applied in the FlyNayTrip implementation.

2.4 API Design and Latency

In travel booking systems, the performance of backend API calls is a dominant determinant of end-to-end response time. Fielding's (2000) REST architectural constraints remain the dominant paradigm for web API design. However, naive REST implementations frequently result in the 'N+1 query problem', where a single user-facing operation triggers a cascade of dependent API requests. Richardson and Ruby (2007) discuss mitigation strategies including response composition at the API gateway layer and client-side request batching, both of which informed the FlyNayTrip API interaction model.

3. Problem Statement

The primary engineering challenge addressed by this project can be stated as follows: how can an intercity travel booking platform achieve acceptable interactive performance for users on 2G and 3G mobile networks, operating on mid-range Android devices with constrained CPU and memory resources, while maintaining a feature set sufficient for end-to-end booking transactions?

An initial prototype of the FlyNayTrip application, developed during the first month of the internship, exhibited the following empirically measured deficiencies when tested on a throttled 3G connection (simulated at 1.6 Mbps downlink) using Lighthouse 10 in headless Chrome:

- Initial page load time of 6.8 seconds, far exceeding the 3-second threshold established by Google as the mobile usability boundary.
- LCP of 5.1 seconds, placing the application in the 'Poor' band of Core Web Vitals assessment.
- A monolithic JavaScript bundle of 1.82 MB (gzipped: 610 KB), which constituted the primary bottleneck to time-to-interactive.
- An average API call latency of 1,240 ms per route search request, attributable to the absence of any caching layer and redundant field fetching.

- A CLS score of 0.31, primarily caused by dynamically injected search result cards that shifted existing DOM elements upon rendering.
- A booking completion (funnel drop-off) rate of approximately 54%, indicating that nearly half of users who initiated a search did not proceed to a confirmed booking.

These findings established a clear and measurable baseline from which the optimization effort could be planned and evaluated.

4. System Architecture

4.1 High-Level Architecture

FlyNayTrip follows a three-tier client-server architecture. The presentation tier is a React (v18) single-page application served from a CDN-backed static hosting environment. The application tier consists of a Node.js/Express API server that aggregates and transforms data from multiple backend services. The data tier comprises a relational database (PostgreSQL) for user accounts and booking records, and a Redis instance for session management and response caching.

4.2 Frontend Architecture

The frontend is organized around a domain-driven directory structure, with clearly delineated modules for authentication, route search, booking flow, and user profile management. React Router v6 is used for client-side navigation, with each top-level route serving as a code-split boundary. The global state is managed through a combination of React Context (for lightweight, infrequently updated state such as user session information) and the useReducer hook (for complex, multi-step booking flow state). Axios is employed as the HTTP client, wrapped in a custom request factory that injects authentication headers, manages request deduplication, and interfaces with the client-side cache layer.

4.3 API Layer

The API exposes RESTful endpoints organized around the core domain entities: routes, operators, schedules, seats, and bookings. Each endpoint follows consistent conventions for field naming, error shape, and pagination. The route search endpoint — the most performance-sensitive in the application — accepts origin, destination, and travel date as query parameters and returns a ranked list of available schedules with operator metadata, pricing, and seat availability counts embedded in a single response object. This design decision — embedding related data rather than requiring the client to issue follow-up requests — was taken explicitly to minimize the number of round-trips required to render the search results view.

5. Methodology

5.1 Performance Measurement Framework

Performance was assessed using a multi-instrument approach. Lighthouse 10 (running in headless Chromium) was used for controlled, repeatable laboratory measurements under throttled network and CPU conditions. The Web Vitals JavaScript library was integrated into the production build to collect field data from real users, reporting

CWV metrics to a lightweight analytics endpoint. Measurements were taken on three device profiles: a high-end Android device (Snapdragon 888, Chrome 115), a mid-range device (MediaTek Helio G85, Chrome 110), and a simulated low-end device (Lighthouse 4x CPU throttling). All network simulations used Lighthouse's 'Slow 4G' profile (1.6 Mbps downlink, 750 ms RTT), which approximates real-world 3G conditions in semi-urban Indian markets.

5.2 Optimization Iteration Cycle

Optimizations were applied in a structured iterative cycle: (1) identify the highest-impact bottleneck using Lighthouse diagnostics and Chrome DevTools flame charts; (2) implement a targeted intervention; (3) re-measure to confirm improvement and check for regressions; (4) commit and proceed to the next bottleneck. This cycle was repeated across three optimization sprints spanning six weeks.

5.3 User Testing

Task completion rates and time-on-task metrics were obtained from a moderated usability study conducted with 24 participants recruited from the target demographic (age 18–45, primarily smartphone users from tier-2 cities). Participants were asked to complete a representative booking task — searching for a route, selecting an operator, choosing a seat, and completing a simulated payment — on both the pre-optimization and post-optimization builds. Drop-off rates at each stage of the booking funnel were recorded.

6. Implementation Details

6.1 Component Architecture

The React component hierarchy is organized into three levels: page-level components (mapped to routes), feature-level components (self-contained functional units such as the search form or seat selector), and primitive UI components (buttons, input fields, cards). This hierarchy was designed to maximize re-usability and to establish clear boundaries for code splitting. Components at the page level are loaded lazily by default; feature-level components are loaded lazily only when their rendering is deferred (e.g., the seat map is not loaded until the user proceeds to the seat selection step).

6.2 Axios Integration and Request Management

All outbound API requests are issued through a centralized Axios instance configured with a 10-second timeout, automatic retry logic for idempotent GET requests (up to two retries with exponential backoff), and request cancellation via AbortController to prevent stale state updates in React components that unmount during a pending request. Response interceptors normalize error shapes and translate HTTP status codes into application-level error types consumed by the UI layer.

6.3 Booking State Machine

The multi-step booking flow — comprising route search, operator selection, seat selection, passenger details entry, and payment confirmation — is modelled as an explicit finite state machine using the useReducer hook. Each state transition is a pure function of the current state and a dispatched action, ensuring predictable behavior and

simplifying debugging. The state is persisted to sessionStorage at each transition, enabling recovery from accidental navigation away from the booking flow.

7. Performance Optimization Techniques

7.1 Route-Level Code Splitting

The initial monolithic bundle was decomposed using `React.lazy()` and `React.Suspense` in conjunction with webpack's dynamic `import()` syntax. Each of the five main route modules (Home, Search Results, Booking Flow, Profile, and Authentication) was converted to a lazily loaded chunk. This reduced the critical path bundle — the JavaScript required to render the initial page — from 1.82 MB to approximately 0.41 MB. The remaining chunks are downloaded on-demand as the user navigates. A `Suspense` fallback skeleton screen was implemented to maintain visual continuity during chunk loading, which also contributed to the CLS improvement by reserving layout space before content renders.

7.2 Lazy Loading of Images and Non-Critical Components

The `IntersectionObserver` API was used to defer the loading of operator logo images and below-the-fold content until they enter the viewport. The native `loading='lazy'` attribute was applied to all `img` elements that are not part of the above-the-fold content. Non-critical third-party scripts — specifically, the analytics beacon and the payment gateway SDK — were loaded with the `defer` attribute and further delayed using a `requestIdleCallback` wrapper to prevent them from competing with critical rendering resources. These interventions, collectively, contributed approximately 0.8 seconds to the LCP improvement.

7.3 API Response Caching

A two-layer caching strategy was implemented for the route search API. At the client layer, search results for a given origin-destination-date combination are stored in a JavaScript `Map` object for the duration of the user session, with a five-minute time-to-live (TTL). Subsequent searches for the same combination within the TTL window return the cached response without a network request. At the server layer, the Express API caches aggregated schedule data in Redis with a configurable TTL (defaulting to three minutes for high-traffic routes). Cache hit rates in a simulated load test reached 68% for repeat queries, reducing mean API latency from 1,240 ms to 480 ms. The trade-off is a potential staleness window during which seat availability counts may be up to three minutes out of date; a disclaimer to this effect was added to the UI.

7.4 Bundle Optimization and Tree Shaking

The webpack build configuration was audited using `webpack-bundle-analyzer`. This revealed that the `date-fns` library was being imported in its entirety, despite only six utility functions being used. Replacing the wildcard import with named imports reduced the `date-fns` contribution to the bundle from 74 KB to 11 KB. The `lodash` library, similarly imported wholesale, was replaced with `lodash-es`, enabling tree-shaking to eliminate unused functions. The `Moment.js` library, which had been included as a transitive dependency, was removed entirely and replaced with native `Intl.DateTimeFormat` calls, saving an additional 66 KB.

7.5 Image and Asset Optimization

All static image assets were converted from PNG to WebP format using the cwebp tool, achieving an average file size reduction of 38% with no perceptible quality degradation. Responsive image srcset attributes were added to ensure that mobile devices receive appropriately sized images rather than downloading desktop-resolution assets. SVG icons were inlined into the HTML at build time using the SVGR webpack loader, eliminating separate network requests for each icon.

7.6 Cumulative Layout Shift Remediation

The CLS score of 0.31 was primarily attributable to two sources: the search results cards, which were injected into the DOM after the initial render and displaced the loading skeleton, and the operator logo images, which did not have explicit width and height attributes, causing the browser to recalculate layout upon load. Both issues were resolved: the search results container was assigned a minimum height equal to the expected rendered height of one result card multiplied by the minimum expected result count, and all images were given explicit dimensional attributes. The resulting CLS score of 0.07 falls within the 'Good' threshold.

8. Results and Analysis

Table 1 presents a consolidated comparison of pre-optimization and post-optimization performance metrics, measured under identical laboratory conditions (Lighthouse 10, Slow 4G throttle, 4x CPU slowdown) on the mid-range device profile.

Table 1: Performance Metrics Before and After Optimization

Metric	Before	After	Improvement
Initial Page Load Time	6.8s	2.9s	57.4% reduction
Largest Contentful Paint	5.1s	2.3s	54.9% reduction
First Input Delay	210ms	78ms	62.9% reduction
Cumulative Layout Shift	0.31	0.07	77.4% reduction
JavaScript Bundle Size	1.82 MB	0.64 MB	64.8% reduction
API Avg. Latency	1,240ms	480ms	61.3% reduction
Booking Completion Rate	54%	73%	+19 pp increase

Note: Measurements conducted under Lighthouse 10 Slow 4G throttle (1.6 Mbps downlink, 750ms RTT), 4x CPU slowdown. All values represent medians of five consecutive runs.

The most significant individual contributor to LCP improvement was route-level code splitting, which reduced the initial parse-and-execute time of JavaScript by approximately 1.4 seconds. API response caching provided the largest reduction in perceived booking flow latency. The combined effect of image lazy loading and explicit dimension attributes was the primary driver of the CLS improvement.

The 19 percentage-point increase in booking completion rate (from 54% to 73%) observed in the usability study is interpreted with appropriate caution, given the small sample size (n=24). However, qualitative feedback from

participants consistently identified the reduced waiting time during the search and seat-selection steps as a primary determinant of their willingness to complete the transaction. Several participants who had abandoned the pre-optimization build at the route search step attributed their abandonment to uncertainty about whether the application was responsive to their input — a direct consequence of the high FID score.

9. Discussion

9.1 Effectiveness of Optimization Strategies

The results demonstrate that targeted, measurable performance optimization can yield material improvements even within the resource constraints of a student project. The highest-impact interventions — code splitting and API caching — are both architecturally straightforward and require no changes to the underlying backend data model. This suggests that performance optimization should be incorporated as a standard practice from early in the development lifecycle, rather than treated as a retrospective remediation activity.

9.2 Trade-offs and Limitations

Several trade-offs were encountered during the optimization process. The client-side caching layer introduces a staleness risk for seat availability data that is difficult to communicate clearly to end users without disrupting the booking flow. The current three-minute server-side TTL represents a pragmatic compromise, but a production-grade implementation would benefit from a WebSocket or Server-Sent Events channel to push real-time availability updates to connected clients.

The removal of Moment.js and its replacement with native Intl.DateTimeFormat calls introduced localization complexity, as the native API's behavior differs subtly across browser implementations — a source of several non-trivial bugs during testing. Bundle analysis also revealed that the Axios library itself contributes approximately 14 KB (gzipped) to the bundle; in a latency-constrained environment, a lighter alternative such as the Fetch API with a thin wrapper could yield marginal additional savings, though this was not pursued due to time constraints.

The user study is limited by its sample size and the moderated laboratory setting, which may not fully replicate real-world usage patterns, network variability, or the cognitive load experienced by users making genuine purchasing decisions.

9.3 Broader Implications

The FlyNayTrip case study illustrates a pattern applicable beyond travel booking to any mobile-first application targeting semi-urban or rural markets: the performance gap between an unoptimized SPA and a well-tuned one is large enough to be commercially decisive. The tooling required to close this gap — Lighthouse, webpack-bundle-analyzer, the Web Vitals library — is freely available and well-documented, suggesting that the barrier to implementation is knowledge and discipline rather than resources.

10. Conclusion

This paper has documented the end-to-end design and optimization of FlyNayTrip, an intercity travel booking platform targeting mobile users on constrained networks. Beginning from a systematically measured baseline, a structured sequence of optimizations — code splitting, lazy loading, API and server-side caching, bundle tree-shaking, image optimization, and CLS remediation — was applied and validated. The resulting improvements across Core Web Vitals and user-facing performance metrics are substantial and consistent with the optimization strategies employed.

The work demonstrates that mobile-first performance engineering is both achievable and essential in the context of travel platforms serving India's digitally expanding but bandwidth-constrained markets. The 57% reduction in load time, 65% reduction in bundle size, and 19 percentage-point improvement in booking completion rate represent outcomes that, if replicated at commercial scale, would have meaningful implications for user engagement and revenue. The paper also surfaces non-trivial engineering trade-offs — particularly around cache staleness and cross-browser API inconsistencies — that serve as cautionary considerations for practitioners attempting to apply these techniques in production environments.

11. Future Work

Several directions for future development have been identified. First, the adoption of a Progressive Web App (PWA) architecture — encompassing service worker-based offline caching, a web app manifest, and background sync — would substantially improve the experience for users with intermittent connectivity, allowing previously loaded routes and schedules to be browsed without an active connection. Second, the implementation of real-time seat availability updates via WebSocket would address the current staleness risk inherent in the caching strategy. Third, server-side rendering (SSR) or static site generation (SSG) for the search landing page and high-traffic routes could further reduce LCP by delivering a pre-rendered HTML payload rather than requiring the client to execute JavaScript before any meaningful content is visible.

From a machine learning perspective, a route recommendation engine trained on historical booking patterns could improve user journey efficiency by surfacing the most likely booking options before the user completes their search query. Finally, a comprehensive A/B testing framework would enable continuous, data-driven optimization of the UI beyond the one-shot improvements documented in this paper.

References

Aberdeen Group. (2019). The performance of web applications: Customers are won or lost in one second. Aberdeen Group Research Report.

Akamai Technologies. (2021). State of online retail performance. Akamai Technologies, Inc. <https://www.akamai.com>

Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures [Doctoral dissertation, University of California, Irvine]. <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

Google Developers. (2023). Core Web Vitals. Web.dev. <https://web.dev/vitals/>

Google. (2022). Mobile page speed new industry benchmarks. Think with Google. <https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/>

Grigorik, I. (2013). High performance browser networking: What every web developer should know about networking and web performance. O'Reilly Media.

Nah, F. F. H. (2004). A study on tolerable waiting time: How long are web users willing to wait? Behaviour & Information Technology, 23(3), 153–163. <https://doi.org/10.1080/01449290410001669914>

React Documentation. (2023). Code-splitting: React.lazy and Suspense. <https://react.dev/reference/react/lazy>

Richardson, L., & Ruby, S. (2007). RESTful web services. O'Reilly Media.

Wagner, J. (2019). Performance budgets 101. Web.dev. <https://web.dev/performance-budgets-101/>

Webpack Contributors. (2023). Code splitting. Webpack documentation. <https://webpack.js.org/guides/code-splitting/>

World Wide Web Consortium (W3C). (2022). Navigation Timing Level 2 specification. <https://www.w3.org/TR/navigation-timing-2/>