UNIVERSITY OF PIRAEUS
SCHOOL OF INFORMATION TECHNOLOGIES
DEPARTMENT OF INFORMATICS

Thesis

Thesis Title: Τίτλος Πτυχιακής Εργασίας:	Application for Travel Planning with Points of Interest Εφαρμογή για Οργάνωση Ταξιδιών με Σημεία Ενδιαφέροντος
Student's name-surname: Ονοματεπώνυμο φοιτητή:	Angelos Sakellariou Άγγελος Σακελλαρίου
Father's name: Όνομα Πατρός:	Agamemnon Αγαμέμνων
Student's ID No: Αριθμός Μητρώου:	Π21147
Supervisor: Επιβλέπων:	Efthimios Alepis, Professor Ευθύμιος Αλέπης, Καθηγητής

February 2026 / Φεβρουάριος 2026

Copyright ©

Acknowledgments

I would like to begin by expressing my deepest gratitude to my supervisor, Professor Efthimios Alepis, for his invaluable guidance, continuous support, and the opportunity to conduct this research. His expertise and insights were fundamental to the successful completion of this thesis, and the knowledge I gained from his and Aristea Kontogianni's university courses was vital to my academic growth and understanding of the field. I would also like to extend my sincere thanks to Aristea Kontogianni for her valuable advice and assistance throughout this process.

On a personal note, I am profoundly grateful to my family for their constant encouragement and belief in me. A special thank you goes to my sister, whose help and inspiration were instrumental, especially during the development and realization of this thesis.

Lastly, I want to express my deepest appreciation to my girlfriend, whose unwavering support and constant encouragement during the demanding hours of development and writing were my greatest motivation. She stood by me through every technical challenge and every late night, providing the balance I needed to see this project through to its conclusion. I also thank my friends for being there to share both the trials and the successes of this journey.

Abstract

The tourism industry's evolution within the context of smart tourism has empowered modern travelers to curate their own experiences independently, yet this shift has introduced significant cognitive burdens. Travelers must navigate a fragmented ecosystem of tools for discovery, logistics, and navigation. To address this fragmentation, this thesis details the design and implementation of a web-based Collaborative Travel Planning Application. The proposed system acts as a unified "Single Source of Truth," integrating dynamic itinerary management, drag-and-drop chronological scheduling, and a context-aware geospatial map engine powered by Mapbox. It supports real-time multi-user synchronization, intelligent automation such as automated weather forecasting via Edge Functions, and interactive points of interest (POI) voting to facilitate group decision-making.

Architecturally, the frontend is built using Next.js 16 (App Router) and TypeScript, strictly adhering to a schema-first methodology and a defensive programming strategy to ensure end-to-end type safety. The backend leverages the Supabase ecosystem (PostgreSQL) to manage complex relational data. Security is fundamentally embedded at the database level using a Zero Trust architecture enforced by Row-Level Security (RLS) policies. Combined with a granular Role-Based Access Control (RBAC) model distinguishing between Owners, Editors, and Viewers, the platform ensures strict data privacy and isolation. The final outcome is a highly responsive, full-stack solution that effectively bridges the geospatial and temporal gaps found in contemporary smart tourism and collaborative travel platforms.

Key Words: Smart Tourism, Collaborative Software, Web Applications, Edge Computing, Geospatial Analysis, Next.js, Supabase, Row Level Security (RLS).

Περίληψη

Η εξέλιξη της τουριστικής βιομηχανίας στο πλαίσιο του έξυπνου τουρισμού έχει δώσει τη δυνατότητα στους σύγχρονους ταξιδιώτες να οργανώνουν ανεξάρτητα τις εμπειρίες τους, ωστόσο αυτή η στροφή έχει επιφέρει σημαντικό γνωστικό φόρτο. Οι ταξιδιώτες αναγκάζονται να περιηγούνται σε κατακερματισμένα εργαλεία για ανακάλυψη, οργάνωση και πλοήγηση. Για την αντιμετώπιση αυτού του κατακερματισμού, η παρούσα εργασία περιγράφει τον σχεδιασμό και την υλοποίηση μιας διαδικτυακής Συνεργατικής Εφαρμογής Προγραμματισμού Ταξιδιών. Το προτεινόμενο σύστημα λειτουργεί ως μια ενιαία «Πηγή Αλήθειας» (Single Source of Truth), ενσωματώνοντας δυναμική διαχείριση δρομολογίων με δυνατότητα drag-and-drop και μια γεωχωρική μηχανή (Mapbox) με επίγνωση του εκάστοτε πλαισίου πλοήγησης. Υποστηρίζει συγχρονισμό πολλαπλών χρηστών σε πραγματικό χρόνο, έξυπνους αυτοματισμούς, όπως η αυτόματη ενσωμάτωση μετεωρολογικών προβλέψεων μέσω Edge Functions, και συστήματα ψηφοφορίας για σημεία ενδιαφέροντος (POI) για τη διευκόλυνση ομαδικών αποφάσεων.

Αρχιτεκτονικά, το frontend έχει αναπτυχθεί με Next.js 16 (App Router) και Typescript, ακολουθώντας αυστηρά μια μεθοδολογία schema-first και μια στρατηγική αμυντικού προγραμματισμού για την εξασφάλιση type-safety σε όλο το σύστημα. Το backend αξιοποιεί το οικοσύστημα του Supabase (PostgeSQL) για τη διαχείριση πολύπλοκων σχεσιακών δεδομένων. Η ασφάλεια είναι θεμελιωμένη σε επίπεδο βάσης δεδομένων με χρήση αρχιτεκτονικής Zero Trust, η οποία επιβάλλεται από πολιτικές Row-Level Security (RLS). Σε συνδυασμό με ένα λεπτομερές μοντέλο ελέγχου πρόσβασης βάσει ρόλων (RBAC), το οποίο διαχωρίζει τους χρήστες σε Ιδιοκτήτες (Owners), Συντάκτες (Editors) και Θεατές (Viewers), η πλατφόρμα διασφαλίζει αυστηρή απομόνωση και ιδιωτικότητα δεδομένων. Το τελικό αποτέλεσμα είναι μια full-stack λύση υψηλής απόκρισης, η οποία γεφυρώνει αποτελεσματικά τα γεωχωρικά και χρονικά κενά που εντοπίζονται στις σύγχρονες πλατφόρμες έξυπνου τουρισμού και συνεργατικού προγραμματισμού.

Λέξεις Κλειδιά: Έξυπνος Τουρισμός (Smart Tourism), Συνεργατικό Λογισμικό, Διαδικτυακές Εφαρμογές, Edge Computing, Γεωχωρική Ανάλυση, Next.js, Supabase, Row Level Security (RLS).

Table of Contents

Table of Figures

Table of Tables

Table of Diagrams

1. Introduction

1.1 Description and Problem Statement

The tourism industry has undergone a paradigm shift over the past two decades, moving from a model dominated by travel agencies to one characterized by independent planning and the broader evolution of smart tourism [2],[7]. Empowered by the internet, modern travelers prefer to curate their own experiences, booking flights, accommodations, and activities independently. While this shift has democratized travel, it has also introduced a significant cognitive burden on users, particularly during the planning phase of group trips. Solving the "Tourist Trip Design Problem", particularly for groups attempting to balance diverse logistical constraints and personal preferences—remains a complex challenge that requires a dedicated, user-centered approach [9].

The Problem of Fragmentation

Currently, the workflow for planning a group trip is disjointed and inefficient. Travelers must navigate a fragmented ecosystem of tools to achieve a single goal. A typical user journey includes:

1. **Discovery:** Using platforms such as TripAdvisor, Instagram, or TikTok to POIs.
2. **Logistics:** Manually entering data into spreadsheets (Excel or Google Sheets) to track budgets and timelines.
3. **Navigation:** Pinning locations on Google Maps to visualize distances.
4. **Communication:** Coordinating via chat applications such as WhatsApp or Messenger to make decisions.

This fragmentation leads to "information silos," in which geospatial data (maps) are disconnected from temporal data (schedules). Furthermore, existing tools often lack robust collaboration features. Spreadsheets are excellent for data entry but poor for visualizing geographic information; map applications are excellent for navigation but lack the structure to manage complex, multi-day itineraries with specific time slots.

The growing complexity of modern travel necessitates a unified solution. Group travelers require a system that serves as a "Single Source of Truth," combining the structural flexibility of a spreadsheet with the visual utility of a map, all underpinned by real-time collaboration.

1.2 Objectives & Methodology

The primary objective of this thesis is to design and implement a web-based Collaborative Travel Planning Application that integrates itinerary management, geospatial analysis, and real-time collaboration. The proposed system aims to address fragmentation by providing a unified interface that enables multiple users to co-edit a trip in real time.

Key specific objectives include:

- **Interactive Itinerary Management:** Developing a dynamic dashboard that lets users organize their trip structure intuitively. This includes a drag-and-drop interface for reordering days and activities, as well as tools for managing specific schedule times directly on the timeline.

- **Unified Geospatial Context:** Synchronizing the timeline view with a map view, ensuring that changes in the schedule are immediately reflected spatially to aid in logistical planning**.**
- **Granular Collaboration:** Implementing a Role-Based Access Control (RBAC) system that distinguishes between Owners (the trip's creator), Editors (contributors), and Viewers.
- **Data Security:** Establishing a Zero Trust security architecture using RLS to ensure data isolation at the database level.
- **Intelligent Automation:** Reducing manual entry by automating calendar generation from trip dates and integrating location-specific weather forecasts directly into the daily itinerary headers.

The application's development followed an Agile Software Development methodology [18], adapted specifically for a single-developer academic project. Rather than coordinating with a traditional development team and a corporate client, the process was guided by regular feedback sessions with the supervising professor. This approach was characterized by iterative cycles of design, implementation, and testing, enabling the continuous refinement of features in accordance with technical constraints, usability requirements, and academic objectives.

From an architectural perspective, the project adopted a "Schema-First" methodology. This meant defining strict data contracts and validation schemas before writing any user interface code. The schema-first methodology ensures type safety across the full stack. This approach builds on established web service discovery principles for distributed collaborative systems [3], [5]. By prioritizing structured data models and database types, the development process ensured type safety and data integrity across the full stack, from the Next.js frontend and server layers to the Supabase database. The final outcome of this methodology is a fully functional web application deployed to a production environment. The live application is available for access and evaluation at: https://vengo.vercel.app.

1.3 Thesis Structure

The remainder of this thesis is organized as follows:

- **Chapter 2: Theoretical Background & Related Work** reviews the evolution of web architectures from monolithic to serverless, examines the theoretical foundations of Computer-Supported Cooperative Work (CSCW), and presents a gap analysis of existing market solutions.
- **Chapter 3: App Overview** presents the application’s core functional modules and user interface, followed by a detailed breakdown of the Role-Based Access Control (RBAC) system and the mobile-first adaptive design philosophy.
- **Chapter 4: System Architecture** details the high-level three-tier architecture, the architectural patterns used, and the non-functional requirements that govern the system's infrastructure and deployment.
- **Chapter 5: Implementation Technologies** describes the core frameworks and languages, the BaaS infrastructure, and specialized APIs for geospatial and meteorological data.
- **Chapter 6: Database Design** offers an in-depth look at the entity-relationship model, database triggers, and the Zero Trust security architecture enforced via PostgreSQL RLS.
- **Chapter 7: Software Design & Code Structure** examines the project’s directory structure and design patterns, then presents a detailed analysis of the caching architecture, real-time synchronization channels, and URL-driven state management.
- **Chapter 8: Geospatial Module Architecture** offers a deep dive into the complex logic of the interactive map implementation, including hybrid search algorithms and context-aware filtering.
- **Chapter 9: Functionality & User Workflow** presents the final user experience, walking through the core flows, including onboarding, trip creation, and live collaborative planning sessions.
- **Chapter 10: Testing & Validation** discusses the dual-layer testing strategy using Jest and Playwright to verify security policies and validate system performance.

- **Chapter 11: Conclusions and Future Work** summarizes the thesis's achievements, discusses technical limitations, and outlines potential directions for future research.

2. Theoretical Background & Related Work

2.1 Evolution of Web Architectures

The landscape of web application development has shifted dramatically over the past decade, moving away from monolithic server-side architectures toward decoupled, serverless, and edge-first paradigms [20]. This transition has been strongly supported by advances in dynamic web service discovery and peer-based overlay networks, which enable discrete services to communicate seamlessly [3], [5]. Understanding this evolution is crucial to appreciating the architectural choices in this thesis, particularly the adoption of Next.js and Supabase.

Traditionally, web applications followed a monolithic architecture in which the frontend, backend logic, and database connectivity were tightly coupled within a single server environment. While simple to deploy initially, monoliths suffer from scalability issues and complex maintenance cycles. The industry has since trended toward serverless architectures and Backend-as-a-Service (BaaS) models [21]. In this paradigm, developers rely on managed third-party services for infrastructure-intensive tasks, such as authentication, database hosting, and file storage. This allows developers to focus entirely on business logic and user interface code. This project uses Supabase, an open-source BaaS built on PostgreSQL. By offloading identity management and database scaling to Supabase, the development process remains agile, and the application benefits from enterprise-grade security features, such as RLS, out of the box.

As web applications grow in complexity, the risk of runtime errors from mismatched data structures increases. This has led to the rise of schema-first development. This methodology defines the API contract or database schema as the single source of truth before any UI code is written [22]. This project uses TypeScript with the Zod validation library. By defining Zod schemas that mirror the PostgreSQL tables, the application achieves end-to-end type safety. This ensures that changes to the database structure trigger compile-time errors in the frontend, preventing data inconsistencies at runtime.

2.2 Collaborative Systems and CSCW

The application's design is deeply rooted in the principles of Computer-Supported Cooperative Work (CSCW) [17]. Effective collaborative software must support real-time awareness, allowing users to see who is editing what to avoid conflicts when solving the trip design problem for groups [9]. Furthermore, while complex conflict-resolution algorithms are widely studied, this application deliberately adopts a simpler, highly efficient "Last-Write-Wins" strategy. This approach preserves data integrity without introducing the processing overhead of complex algorithmic merging, which is vital for maintaining fluid performance during rapid, real-time group itinerary planning.

A critical challenge in collaborative web apps is network latency. If the interface waits for a server response before updating, the application feels sluggish. To address this, this thesis implements Optimistic UI [38]. When a user takes an action, such as adding an activity, the interface updates immediately, as if the request had already succeeded. The actual network request runs in the background. If the request fails, the UI creates a rollback transaction to revert the change and notify the user. This pattern is essential for maintaining flow state during rapid itinerary planning.

2.3 Comparative Analysis of Existing Solutions

To contextualize this thesis's contribution, it is necessary to analyze modern market solutions for group travel planning. These specialized web applications attempt to unify the itinerary and geospatial experience, yet they often prioritize commercial aesthetics over technical transparency [2], [7].

2.3.1 Full-Service Market Solutions

This category includes platforms that aim to be all-in-one environments but often struggle with rigid interfaces or proprietary limitations.

Stippl

Stippl positions itself as an "all-in-one" travel platform with a strong focus on the visual and social aspects of planning [62]. It integrates transportation, budgeting, and accommodation management into a single interface. While its aesthetic design is notable, the platform's rigid user interface forces users into specific workflows. It lacks the dynamic, context-aware map filtering presented in this thesis and often displays an overwhelming amount of data on the map without accounting for the user's specific temporal context. Furthermore, its collaborative features lack the granular, role-based controls needed to manage complex group dynamics effectively.

Wanderlog

Wanderlog is a widely used itinerary-mapping tool that emphasizes discovery and visual synchronization between maps and lists [74]. However, the platform operates as a proprietary ecosystem that severely limits technical extensibility for custom collaborative logic. Many essential planning features are locked behind aggressive subscription models, creating barriers for users seeking a full logistical experience.

Lambus

Lambus addresses the logistical chaos of large groups by integrating document management and expense tracking [75]. It is a robust utility for general logistics, but its interface often becomes cluttered due to high feature density. Like other commercial solutions, the map in Lambus serves as a static overview rather than a context-sensitive tool. It does not dynamically respond to the specific day or activity currently being edited, which can lead to cognitive overload when planning dense, multi-activity itineraries.

2.3.2 Passive and Single-Purpose Utilities

These tools are either industry standards for specific tasks or leverage existing data ecosystems, yet they fail to provide an active, unified workspace.

Splitwise

Splitwise is the industry standard for managing group finances and cost-sharing during travel [61]. It provides a robust ledger for tracking debts and simplifies settling up among large groups. However, Splitwise is a single-purpose utility that exists entirely outside the temporal and geospatial context of a trip. Users must manually reconcile their itinerary with their expenses, resulting in fragmented record-keeping. It cannot associate costs with specific scheduled activities or visualize spending on a map, limiting its utility as a comprehensive planning tool.

Google Travel

Google Travel leverages the extensive data from Google Maps and Search to generate automated trip summaries from confirmation emails [69]. Despite its massive data advantage, it remains a passive viewing tool rather than an active planning workspace. It lacks the collaborative "whiteboard" flexibility for real-time group editing and does not offer granular permissions. This makes it difficult to manage complex collaborative roles on the platform.

Table 1. Comparative analysis of Market-Leading itinerary planning tools

Feature	Real-time Collaboration	Geospatial Context	Itinerary Flexibility	Role-Based Access
Stippl	High	Static Overview	Rigid Workflow	Basic (Invite Only)
Wanderlog	High	Integrated	Medium	Proprietary/Locked
Splitwise	High	None	None (Expense Only)	Financial Only
Lambus	Medium	Static Overview	Medium	Basic
Google Travel	Medium	High	Low (Static)	Yes (Historical/Forecast)
Vengo	High	Context-Aware	High (Drag-and-Drop)	Granular (Owner/Editor/Viewer)

2.3.3 Gap Analysis

The comparative analysis highlights a significant technical gap in the current market, particularly when viewed through smart tourism research. Smart tourism shows social data enables personalized recommendations through user modeling and network analysis, but lacks real-time multi-user synchronization for group itinerary planning [4], [8]. This thesis addresses these deficiencies by developing a web application that provides structural flexibility equivalent to spreadsheets, context-aware geospatial visualization that dynamically adapts to the selected day or activity, and granular role-based management where authorization is enforced as a fundamental property of the data architecture rather than a secondary UI feature. Unified geospatial collaboration with RBAC directly addresses the collaboration gaps identified in contemporary smart tourism systems [2].

3. App Overview

The application is designed as a comprehensive platform for collaborative travel planning. By unifying fragmented logistical data into a centralized dashboard, the system enables groups to manage itineraries and geospatial data in a single, high-performance environment.

3.1 Core Functionality Modules

The application journey begins on a dedicated Landing Page and proceeds through a secure Authentication Flow, which serves as the entry point for identity management. Once authenticated, the User Dashboard serves as the central analytical command center. This interface provides an immediate overview of travel activity through a personalized Interactive Travel Map that dynamically highlights visited countries and specific destinations. While concise metric cards summarize statistics such as trip counts and active partners, the primary emphasis is on the geospatial visualization, which provides immediate context for the user's global travel history and plans.

Centralized navigation is provided by a Persistent Navigation Sidebar that anchors the application's layout and provides immediate access to the Dashboard, Trips, and Map modules. The sidebar also provides entry points to the global Expenses and Collaborators pages, which aggregate data across the user's entire travel portfolio. This global view is functionally separate from the specialized, trip-specific collaborator lists found within individual trip planner workspaces. The sidebar also houses the application's Theme Switcher, which allows users to toggle between high-contrast Dark and Light modes. These preferences persist across the entire application state to ensure visual comfort in diverse lighting environments. Additionally, the sidebar serves as the primary entry point for the Account page, a centralized hub where users can manage their data by updating personal details, refreshing their avatar, and changing their password to maintain account security. This interface also grants users full control over their data, including the option to permanently delete their account. Finally, the sidebar includes a dedicated Sign Out button for securely managing user sessions.

As the core of the application experience, the Itinerary serves as the primary timeline, enabling users to manage daily sequencing through an intuitive drag-and-drop interface. This is complemented by Automated Itinerary Generation, which creates the day-by-day calendar structure immediately upon trip creation. For logistical precision, a synchronized sidebar integrated into the planning view enables users to instantly verify distances and locations, ensuring the planned itinerary is geographically feasible. Hybrid search capabilities enable the discovery of both global locations via geocoding and local trip assets stored in the database. Smart tourism research shows that multi-source data integration, combining geospatial APIs with user-generated content, enables context-aware geospatial planning, which is essential for modern collaborative travel applications [2], [7]. The collaborative POI voting system enables real-time group preference aggregation, allowing multiple users to collectively prioritize destinations and activities through structured social decision-making mechanisms [8].

A dedicated Global Discovery Map page offers a broader view of the user’s entire travel portfolio and supports broader geographical exploration. This interface uses a 3D globe projection powered by Mapbox, offering a visual "passport" of the user's global footprint. The core of this module is the Map Sidebar, which serves as the functional control center for the geospatial engine. The sidebar enables users to discover new locations or browse existing trip assets, with real-time filtering by category and trip status to manage high volumes of geospatial data efficiently.

To support group decision-making and logistical planning, the application includes Collaborative POI Voting and Intelligent Weather Forecasting. Collaborators can "like" or "dislike" saved points of interest to prioritize the schedule asynchronously. Meanwhile, the weather engine automatically fetches 16-day forecasts for upcoming dates or historical meteorological data for dates further in the future.

3.2 User Roles and Access Control

The application implements a strict Role-Based Access Control system to manage user privileges for trips. These roles are defined in the database using the collaborator_role enum and enforced via RLS policies to ensure data integrity and privacy across the platform. The system distinguishes three primary roles: Owner, Editor, and Viewer.

The Owner is the user who initially creates the trip and possesses full control over the entire trip lifecycle. This role includes the exclusive ability to delete the trip, manage sensitive settings, and maintain the collaborator list by inviting new users or removing existing ones.

An Editor is a collaborator with permission to contribute actively to the planning process. In this role, users can create, update, and delete trip content, including Activities, POIs, and Destinations. Editors can manage the itinerary and logistical details, but are strictly prohibited from deleting the trip or modifying the access rights of other collaborators.

The Viewer role is for users invited with read-only access to the itinerary. Viewers can browse daily schedules, interact with the geospatial map, and view specific activity details, but the system prevents them from modifying any data.

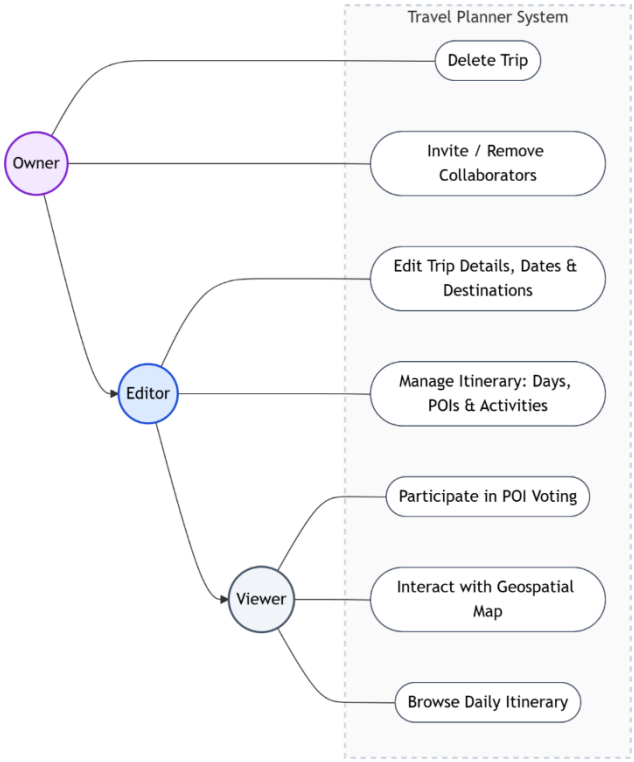

Diagram 1. Use Case Diagram Illustrating RBAC and System Permissions.

3.3 Application Interface

The user interface (UI) is designed to be responsive and data-centric. Drawing on foundational definitions and modern principles of user experience design [16], it implements a comprehensive Adaptive Design System that seamlessly supports both Light and Dark themes. This ensures optimal readability in various lighting conditions and accommodates user preferences. The theme preference is persisted locally and applied consistently across all components, from the navigation sidebar to the interactive map tiles.

To demonstrate the system's versatility, the following figures present a side-by-side comparison of the themes across different device layouts.

The central hub shows travel statistics and upcoming plans. The desktop view illustrates the Dark theme with high-contrast surfaces, while the mobile view showcases the Light theme and the responsive single-column layout.

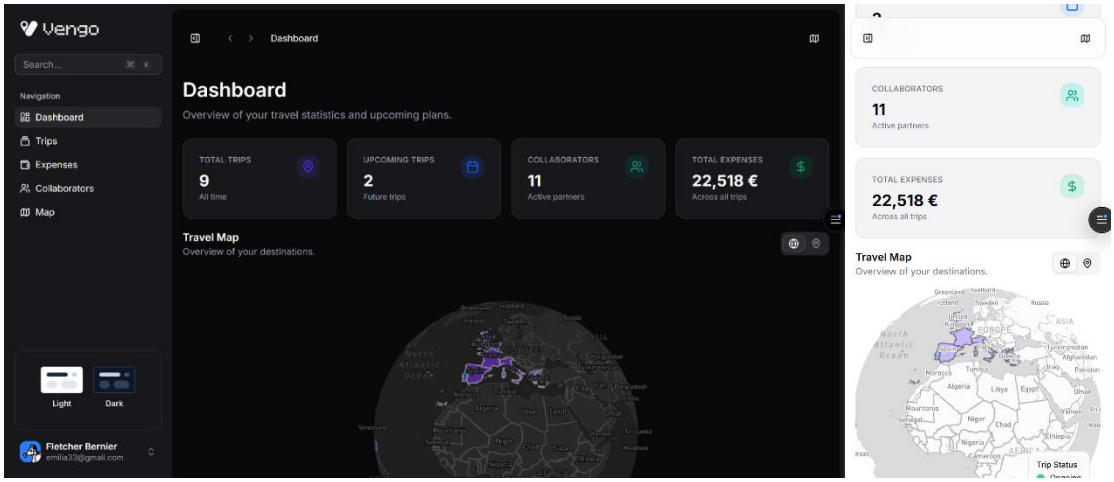

Figure 1. The Personal Dashboard (Desktop Dark vs. Mobile Light)

This interface provides a card-based grid view of all user trips. The desktop view is presented in Dark mode, highlighting depth and neon-like status indicators, while the mobile view uses the Light theme to showcase the cards' clean, airy aesthetics.

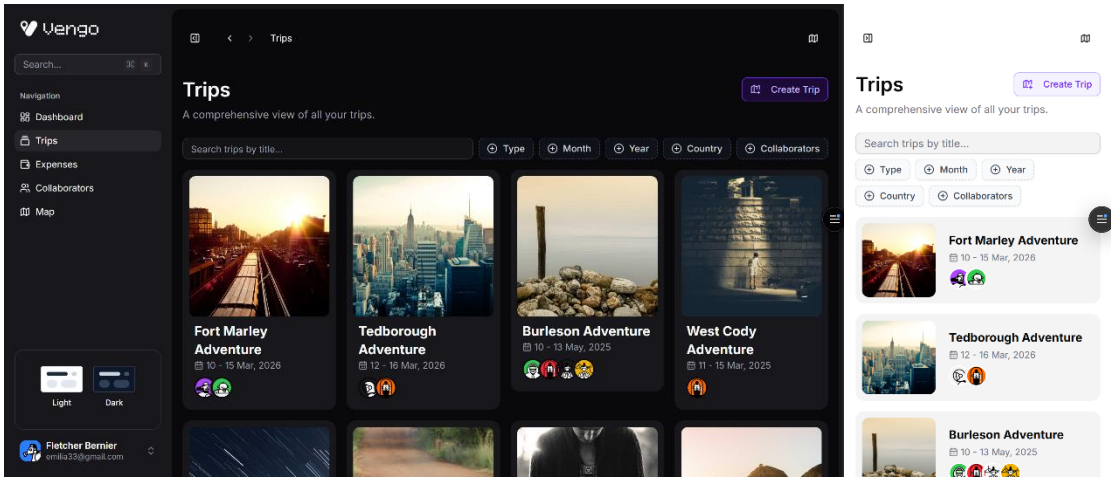

Figure 2. The Trips Management Grid (Desktop Dark vs. Mobile Light)

Serving as the landing page for a specific trip, this view demonstrates how analytics charts and expense summaries adjust their color palettes and transparency levels to maintain visual clarity in both themes.

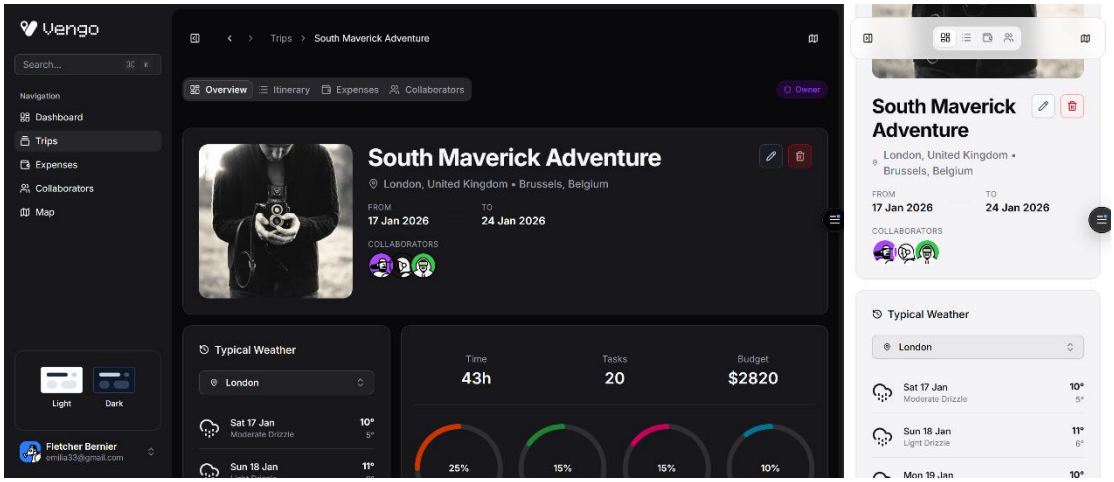

Figure 3. Trip Overview & Analytics (Desktop Dark vs. Mobile Light)

The core planning interface centers on the drag-and-drop itinerary as the primary functional focus of the workspace. This view enables intuitive organization of daily activities while also showcasing the integrated split-view map. Although the map is a global component available throughout the application, it is used here to visualize POIs and aid in itinerary organization.

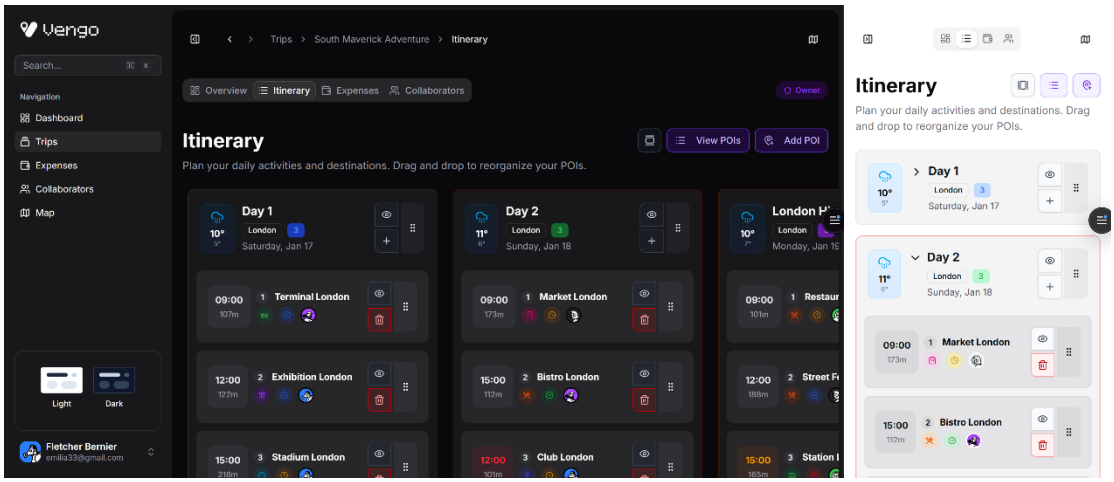

Figure 4. The Collaborative Itinerary Workspace (Desktop Dark vs. Mobile Light)

This full-screen geospatial interface is designed for exploring destinations. The markers utilize clustering for performance, and the interface includes floating controls for filtering by category or status.

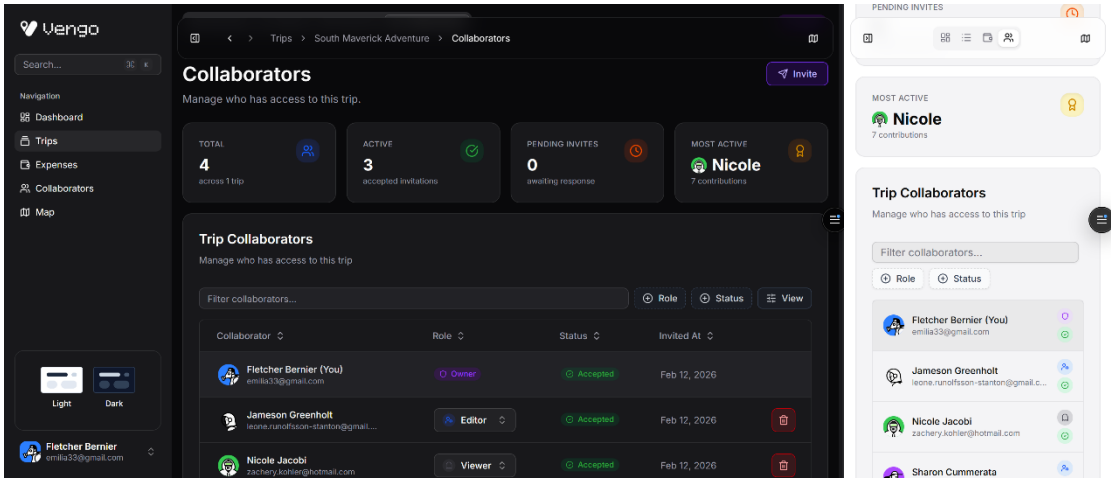

Figure 5. The Global Exploration Map (Desktop Dark vs. Mobile Light

3.4 User Experience & Adaptive Design

The application prioritizes a "Mobile-First" philosophy [48], ensuring that complex planning tasks, traditionally reserved for desktop environments, are fully accessible and performant on devices of any size. Using Responsive Layouts, the interface dynamically adjusts its structure, shifting from a multi-pane grid on desktop monitors to a simplified single-column stack on mobile devices. This fluid transition ensures that data remains legible and interactive regardless of the hardware used.

To achieve a native-like feel in a web environment, the system uses optimized Touch Interactions. Key features, such as drag-and-drop itinerary reordering and map panning, are tuned for touch events to ensure responsiveness on mobile browsers. The application also employs Adaptive Dialogs, which render as standard Modals on desktop viewports but automatically transform into ergonomic Bottom-Sheet Drawers on mobile. This keeps critical input forms and controls within the "thumb zone," significantly improving the application's usability for mobile users.

4. System Architecture

The application is built on a modern full-stack architecture that leverages the Vercel Edge Network and the Supabase BaaS ecosystem. This decoupled approach keeps the frontend highly performant while offloading complex infrastructure management to specialized serverless providers.

4.1 System Architecture Overview

The system architecture defines the relationship between the client environment and the cloud infrastructure. At the top level, the User or Browser communicates with the application over HTTPS using JSON requests. These requests are first received by the Next.js App Router, hosted on the Vercel Edge Network, which ensures that initial routing and rendering occur as close to the user as possible, minimizing latency.

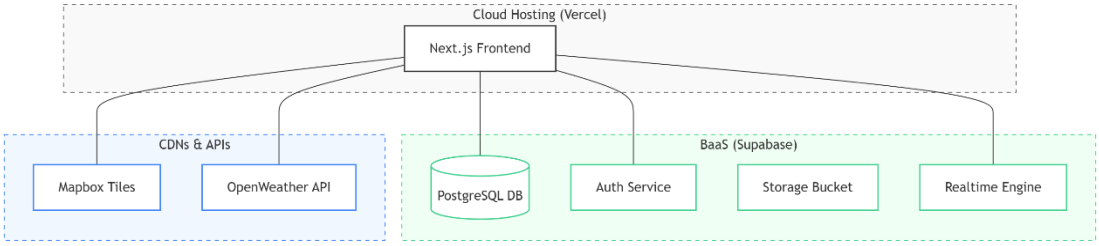

Diagram 2. High-Level Service Orchestration across Vercel, Supabase, and External APIs.

Within the cloud environment, the application coordinates several core internal logic pathways. For data-intensive rendering, the router uses React Server Components to fetch data directly from the backend. For user-initiated modifications, the system employs Server Actions. Both pathways are strictly gated by an RLS engine that serves as a database-level firewall, ensuring every query includes an implicit filter based on the user's authenticated identity.

The Supabase BaaS layer manages the application's core infrastructure. The Next.js App Router communicates directly with Supabase Auth for identity management and with Supabase Storage for binary assets such as trip images. For background automation, the system uses Edge Functions running on the Deno runtime. A specialized Cron Trigger within the PostgreSQL database periodically wakes these functions to perform tasks such as fetching weather data from the Open-Meteo API. Finally, the frontend maintains a direct connection to the Mapbox GL API to receive vector tiles for the geospatial module.

4.2 Architectural Pattern

The application is built on a modern Three-Tier Architecture optimized for the serverless paradigm. The Presentation Tier (Next.js/React) manages the UI and state [56]. The Application Logic Tier (Server Actions/Edge Functions) handles secure mutations and automation, while the Data Tier (Supabase/PostgreSQL) serves as the authoritative source. This layered approach keeps the user interface decoupled from backend logic, allowing each layer of the stack to scale independently.

A defining characteristic of this project is the implementation of a Lightweight CQRS pattern to optimize performance. By separating "Read" and "Write" pathways, the system processes data through specialized channels. Reads are handled by React Server Components that fetch data directly via lib/data, while secure Server Actions in app/actions manage Writes. This structure is supported by the BaaS integration. By offloading authentication, storage, and database management to Supabase, the system leverages enterprise-grade infrastructure while allowing the application to focus exclusively on business-specific logic.

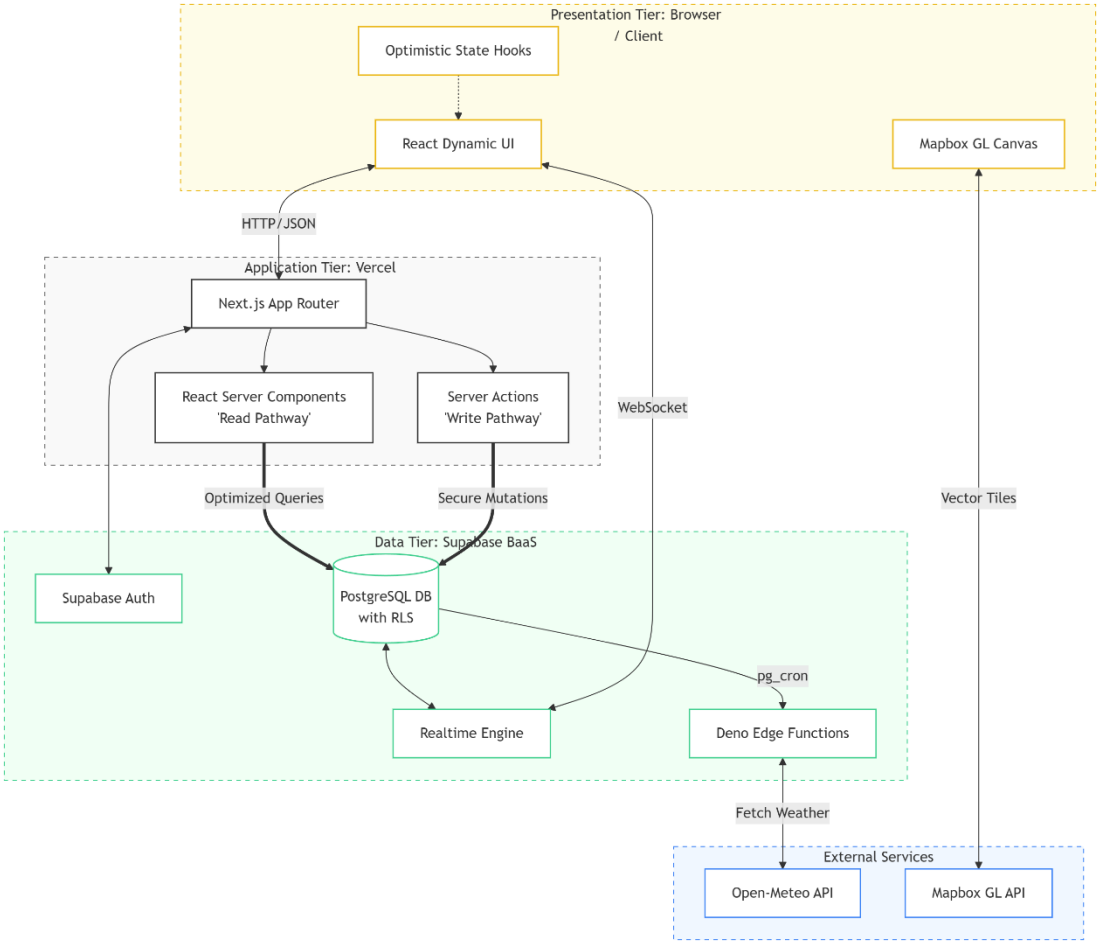

Diagram 3. UML Component Diagram for the Decoupled Three-Tier Architecture and Data Flow.

4.3 Data Flow & Logic

The movement of data through the system follows a Zero Trust and Optimistic flow designed to maximize responsiveness. When a user performs an action, such as reordering an activity, the client-side state is updated immediately to provide instant visual feedback. Privacy challenges in relational data publishing for multi-user collaborative environments directly inform the secure data flows implemented through Row Level Security and optimistic UI patterns [1]. During database interactions, the PostgreSQL engine intercepts the query and applies Row-Level Security policies. This adds a filter based on the user's identity to verify they have the correct role for that trip. Once the database confirms the transaction, the system triggers a revalidation of the current path.

While standard mutations follow a secure HTTPS request-response cycle, the system enables synchronous multi-user collaboration through a persistent WebSocket connection. This real-time data flow works alongside the Optimistic UI pattern. When a collaborator mutates data (e.g., dragging an activity to a new time slot), the database enforces the Row-Level Security (RLS) policies. On success, the Supabase Realtime engine broadcasts the payload to all other authenticated clients subscribed to that trip channel. This hybrid flow ensures the primary user experiences zero latency while secondary users see changes appear in their interface instantly without a manual refresh.

4.4 Frontend Engineering

The frontend is built using the Next.js App Router architecture and adheres to a strict mobile-first design philosophy. This approach ensures a seamless transition from handheld devices to high-resolution desktop monitors. React Server Components are a core technology in this project, bridging server-side data and client-side interactivity [33]. By moving data-intensive logic to the server, the application achieves a zero-bundle size for its data-fetching logic, significantly improving performance.

This architectural choice directly impacts performance metrics by reducing the amount of JavaScript sent to the browser. This results in a faster First Contentful Paint [31] and a significantly lower Cumulative Layout Shift. Furthermore, the use of Server Components eliminates traditional request waterfalls. In standard single-page applications, the browser must download and execute JavaScript before making multiple serial network requests. In this system, these requests execute in parallel on the server, often within the same data center as the database, thereby reducing latency. Additionally, large libraries used for data parsing or complex formatting remain on the server, keeping the client-side bundle lightweight.

4.5 Infrastructure & Deployment

The application is deployed as a unified full-stack application on the Vercel Cloud Platform [72]. The live production environment is publicly accessible at https://vengo.vercel.app. The deployment process is fully automated via GitHub integration, where every push to the main repository triggers a build and deployment pipeline. This workflow includes preview deployments for pull requests, which allow isolated testing of new features before they are merged into the production environment.

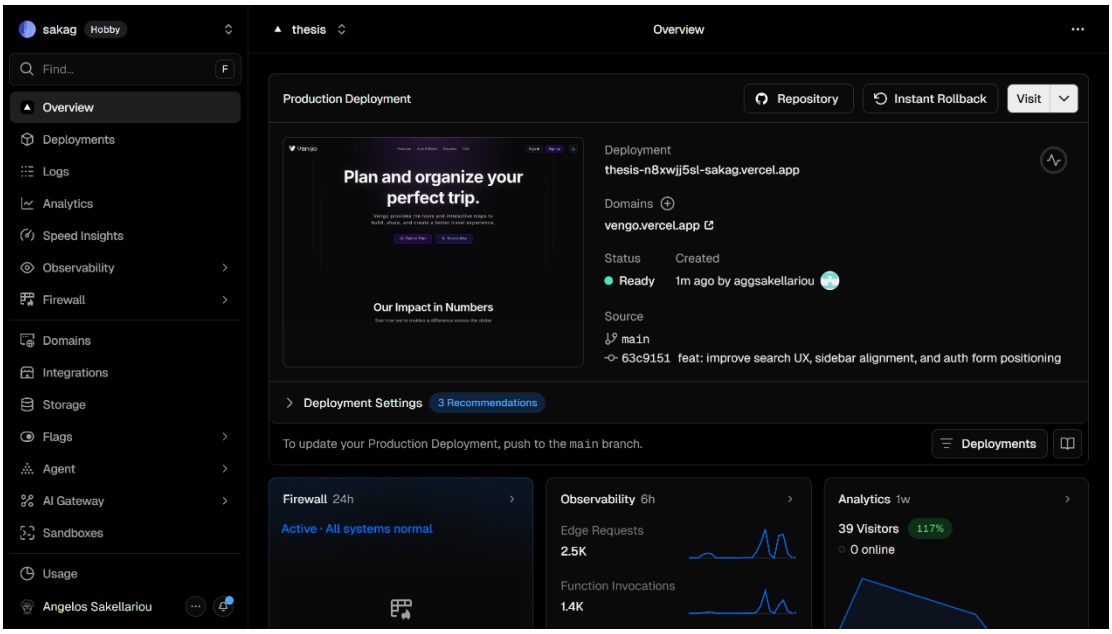

Figure 6. Vercel production deployment and analytics dashboard.

The application is hosted on the Vercel Edge Network, a global Content Delivery Network. This infrastructure delivers static assets and renders Server Components at the edge node closest to the user, minimizing round-trip latency. The system operates within two distinct runtime environments. The Node.js runtime executes Server Actions and performs server-side rendering, while the Edge Runtime handles Supabase Edge Functions for external API integrations. All communication between the client, the Vercel server, and the database is encrypted using SSL/TLS.

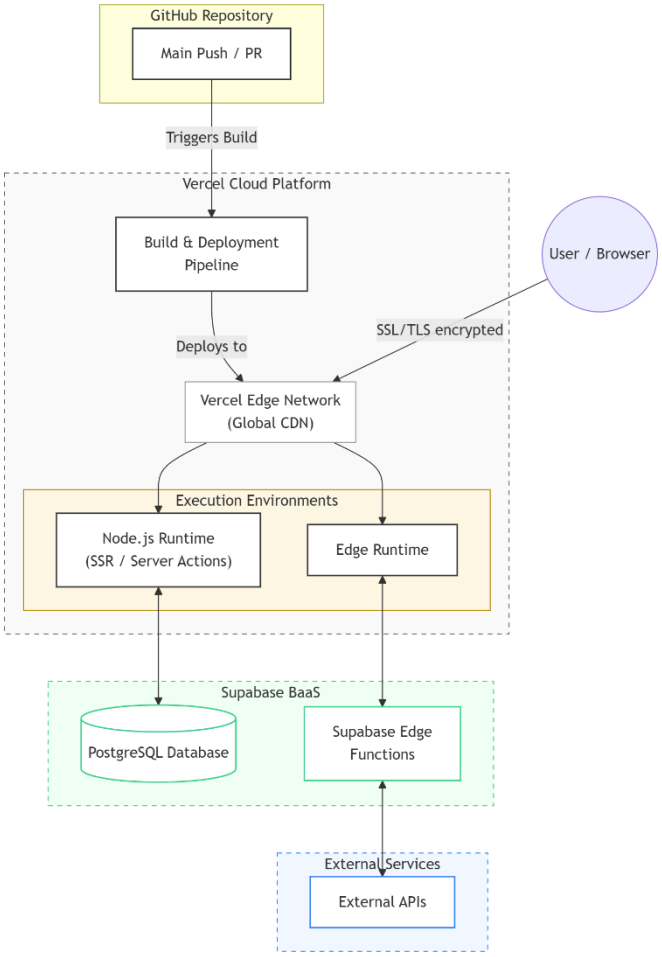

Diagram 4. Deployment Infrastructure and Architecture

4.6 Non-Functional Requirements

The system's non-functional requirements are met through specialized protocols designed for scalability and security. The serverless architecture enables horizontal scaling, with logic instances initializing automatically in response to traffic volume. Data is stored in multi-zone PostgreSQL clusters, ensuring high availability even during regional hardware failures. Reliability is further reinforced through end-to-end type safety using TypeScript and Zod schemas across the entire deployment.

Security is managed through several layers of protection. Authentication is handled via JWT-based sessions managed by Supabase Auth, while authorization is enforced through granular RLS policies that act as a database-level firewall. These policies ensure that users can access only the data they are explicitly authorized to view. Finally, the system provides auditability through automatic triggers that maintain a historical record of all modifications within collaborative trips.

5. Implementation Technologies

This chapter provides a detailed analysis of the technical stack used to develop the collaborative travel planning application. For each technology, the section outlines its definition, rationale for selection, and specific application within the project.

5.1 Core Frameworks and Languages

The application is built on Next.js 16, a full-stack React framework that supports essential features such as Server-Side Rendering and static site generation [50]. It was selected primarily for its App Router architecture and robust support for React Server Components. These features enable high-performance server-side data fetching, significantly reducing the JavaScript payload sent to the client and improving both SEO and initial load times. In this project, Next.js serves as the primary backbone, managing routing, server-side logic through Server Actions, and the rendering of dynamic user interface components. To further optimize the development experience and build performance, the project uses Turbopack, an incremental bundler optimized for Next.js that significantly reduces compile times and accelerates hot module replacement (HMR) [24].

Complementing the framework is TypeScript, a strongly typed superset of JavaScript that adds optional static typing [41]. TypeScript was chosen for its robust compile-time error checking and superior developer tooling, including IntelliSense. This ensures that data structures remain consistent across the frontend and backend layers, which is critical for reducing runtime bugs in a complex collaborative environment. The entire codebase is written in TypeScript to ensure end-to-end type safety, from database schemas through React component props. This adherence to strict typing forms the foundation of a broader defensive programming strategy employed throughout the codebase. This methodology anticipates potential failures, whether from malformed user input, network latency, or unexpected API responses, and mitigates them proactively. Alongside TypeScript, this strategy is enforced through strict schema validation using Zod at the application boundaries and comprehensive error handling boundaries within the Next.js routing layer [32], ensuring that the system fails gracefully rather than exposing raw errors or corrupting the database state.

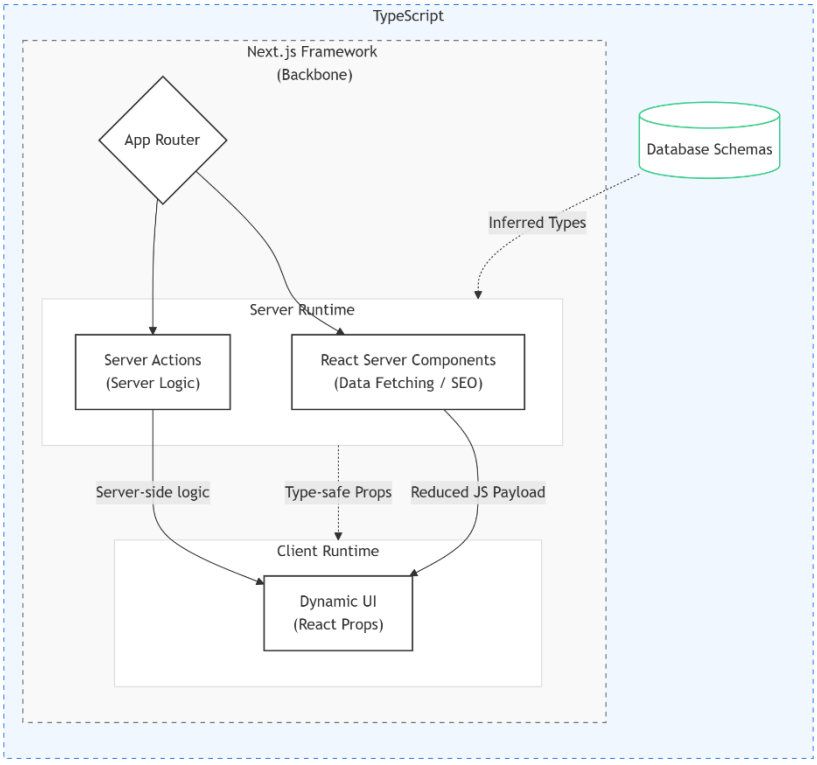

Diagram 5. End-to-End TypeScript Safety Diagram

5.2 Backend-as-a-Service Infrastructure

The application uses Supabase as its primary Backend-as-a-Service provider. Built on top of a relational database like PostgreSQL, Supabase provides enterprise-grade features such as Row-Level Security, automated triggers, and real-time capabilities without the overhead of manual infrastructure management. Its relational nature is ideally suited to the complex data hierarchies of travel itineraries, where trips, days, and activities must remain strictly organized. Beyond simple data storage, Supabase implements the application's Zero Trust security layer through its native policy engine.

Identity management is handled by Supabase Auth, an integrated service that supports secure user registration, authentication, and session management. It was selected for its seamless integration with PostgreSQL and support for both traditional email-based logins and OAuth flows. The service manages JWT-based sessions, which Row-Level Security uses to verify user identities at the database level. This ensures that only authenticated users can access or modify trip data. Furthermore, Supabase Storage is used to manage large files such as trip cover images and profile avatars [63]. This service integrates directly with the database's security policies, allowing the application to restrict file access using the same logic applied to database records.

5.3 Design and User Interface

The application's visual layer is meticulously crafted to prioritize a minimalist, clean aesthetic centered on whitespace and subtle borders. It follows a strict mobile-first approach [48], ensuring accessibility and responsiveness across all devices.

Styling and Design System

The design system is built on Tailwind CSS v4, using the modern OKLCH color space to ensure perceivably uniform colors across the application [65]. The interface natively supports both Light and Dark modes via next-themes, using a "Zinc" base theme characterized by neutral grays with a slight cool tint.

Component Architecture

To maintain high standards of accessibility (WAI-ARIA compliance) and visual consistency, the project uses Shadcn UI [68]. Rather than serving as a traditional monolithic dependency, it uses a modern component registry pattern (components.json) to pull unstyled, accessible primitives from Radix UI and style them directly in the codebase with Tailwind [55]. This registry pattern is extended to include high-quality community components, including:

- @supabase: UI components optimized for database integration [64].
- @reui [76], @ncdai [26], kibo-ui [43], coss-ui [77], and @diceui [29] and more: Specialized component collections for advanced interactive elements.

Specific UI requirements are managed by dedicated libraries:

- **Icons:** Lucide React offers a consistent, clean SVG icon set [46].
- **Overlays:** Vaul powers highly ergonomic, mobile-friendly bottom-sheet drawers, while Sonner handles opinionated, non-blocking toast notifications.
- **Data Visualization & Dates:** Recharts is used for analytical graphics, while date-fns and react-day-picker handle complex calendar interactions.

Animation and Motion

Fluid motion and interactive feedback are essential to the application's perceived performance. Framer Motion [49] is the primary engine for complex gestures and layout transitions, while Animate UI (tw-animate-css) provides performant CSS animations via utility classes [37].

5.4 Specialized Libraries and APIs

Data integrity at the application boundaries is enforced by Zod, a TypeScript-first schema declaration and validation library [40]. Zod enables the definition of strict Data Transfer Objects that validate data before it enters the application logic. This ensures the system never processes malformed data from APIs or user forms. It is used extensively across all Server Actions for input validation and in the Data Layer to parse database responses.

The project's geospatial requirements are met by Mapbox GL JS, a powerful library for interactive vector maps [25], [47]. It delivers superior performance when rendering large marker datasets and offers advanced camera controls for animating transitions between trip destinations. Mapbox powers the entire geospatial module and handles the visualization of POIs and search results.

Finally, the application integrates the Open-Meteo API to deliver high-resolution meteorological data [54]. This free, open-source API provides both historical and forecast data without complex API keys, making it an ideal choice for a privacy-focused planning tool. The API is accessed via Supabase Edge Functions to fetch and synchronize weather data for itineraries, ensuring users always have access to the latest environmental context for their trips.

Table 2. Comprehensive Implementation of Technology Stack

Category	Technology	Purpose
Framework	Next.js 16 (App Router)	Core application framework, SSR, and Routing.
Language	Typescript	Type-safe development and improved maintainability.
Backend (BaaS)	Supabase	Database, Authentication, Storage, Real-Time, and Edge Functions.
Database	PostgreSQL	Relational data management with RLS support.
Styling	Tailwind CSS	Utility-first CSS framework for responsive design.
UI Components	Shadcn/UI (Radix UI)	Accessible, customizable UI component library.
State Management	React Context & Hooks	Local and shared application state handling.
Maps	Mapbox GL JS	Interactive geospatial visualization and geocoding.
Validation	Zod	Schema-first validation for DTOs and Form data.
Animations	Framer Motion	High-performance UI transitions and animations.

UI Components / Data Grid	TanStack Table v8	Headless UI logic for responsive, state-driven data tables.

6. Database Design

The backend infrastructure is powered by Supabase, an open-source Firebase alternative built on PostgreSQL [28]. This choice was driven by the need for a robust relational database that supports complex joins and RLS. Unlike NoSQL alternatives, Supabase offers the power of a traditional relational engine with modern features such as real-time subscriptions, managed authentication, and serverless edge functions.

The importance of Supabase in this project lies in its "Database-First" philosophy. By centering logic and security within PostgreSQL, the application ensures data integrity regardless of the client-side implementation. This architectural decision enables complex relationship mapping (such as linking trips to multiple days and days to specific activities) while maintaining high performance and enterprise-grade security through its native RLS engine.

6.1 Entity Relationship Model

The system architecture is built around several core entities that define the planning workflow. The Profiles table stores user-specific metadata linked to the Supabase Auth system. This includes essential information such as the user's email address, first and last name, and a URL pointing to their avatar. The Trips table serves as the root entity of the entire planning ecosystem. It contains metadata such as the trip title, description, and date range, along with the owner's identifier and a reference to the cover image. To ensure a clean separation between core relational data and binary assets, a separate Trip Images table manages file storage references, including paths and MIME types.

The Destinations table serves as a critical bridge between the trip and its geographical regions. It stores location names, country codes, and precise coordinates. This table serves as the primary anchor for fetching meteorological data through the weather synchronization engine, ensuring each location is uniquely identified for forecasting. The itinerary's logistical structure is defined by the Days table, where each record is linked to a parent trip and associated with a specific destination entry. These records manage the trip's chronological sequence by storing the date and a numerical sort order.

The POIs table serves as a persistent geospatial library. It stores location data, including coordinates, addresses, and categories. The Activities table represents the planner's operational layer. It uses junction entries to link specific POIs to a Day, maintaining the scheduled time and a sort order that supports drag-and-drop reordering in the user interface.

The relational schema supports the complex collaboration patterns essential for smart tourism applications, where multiple stakeholders require structured access to shared geospatial and temporal data [4]. This entity-relationship model accommodates the Owner-Editor-Viewer hierarchy through foreign key relationships and enum constraints that mirror RBAC principles at the database level.

Multi-user collaboration is managed through the Trip Collaborators and Trip Invites tables. The Collaborators table tracks user roles and the current status of their membership, while the Invites table manages token-based access for onboarding new members. Finally, the Weather table enriches the application by storing temperature ranges and weather codes. This data is indexed by destination and date to support informed travel planning.

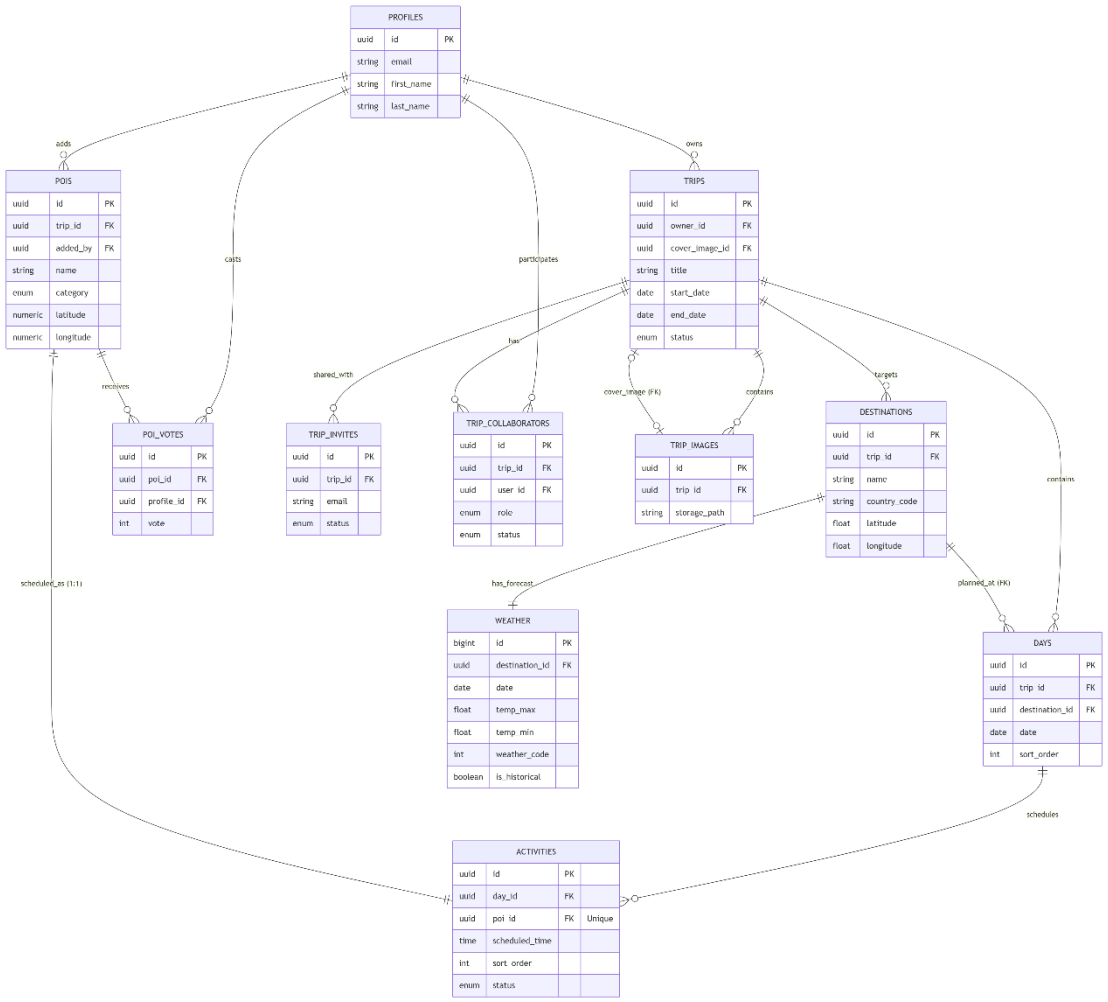

Diagram 6. Entity Relationships Diagram

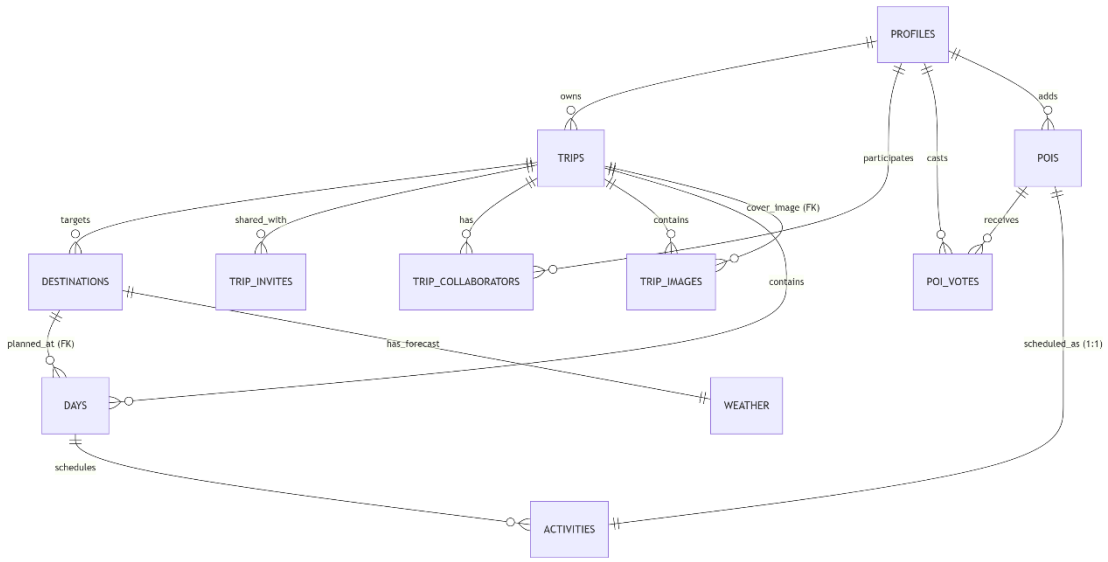

Diagram 7. Conceptual Data Entity Relationship Diagram

6.2 Supabase Authentication Architecture

The project uses Supabase Auth as a comprehensive service for user identity and session management. The system primarily uses the built-in email provider for registration and login, leveraging the Supabase infrastructure to manage password recovery and sign-up confirmations.

To ensure a cohesive User Experience (UX), the default Supabase transactional emails were redesigned to align with the application's minimalist UI guidelines. Custom HTML templates were developed for the "Confirm Signup" and "Reset Password" workflows. These templates feature:

- A clean, centralized card-based layout that visually mirrors the application's core web interface.
- Consistent typography using native system fonts to ensure legibility across various email clients.
- High-contrast Call-to-Action (CTA) buttons paired with fallback URLs to ensure accessibility and reliable navigation, even when HTML elements are stripped by the user's email provider.

To improve the onboarding experience, the application also supports third-party OAuth via Google [45]. This lets users authenticate with their existing accounts while Supabase handles the secure token exchange [53]. The entire authentication system is configured to be fully functional in the local development environment, enabling rigorous testing of all identity flows before deployment to the production network.

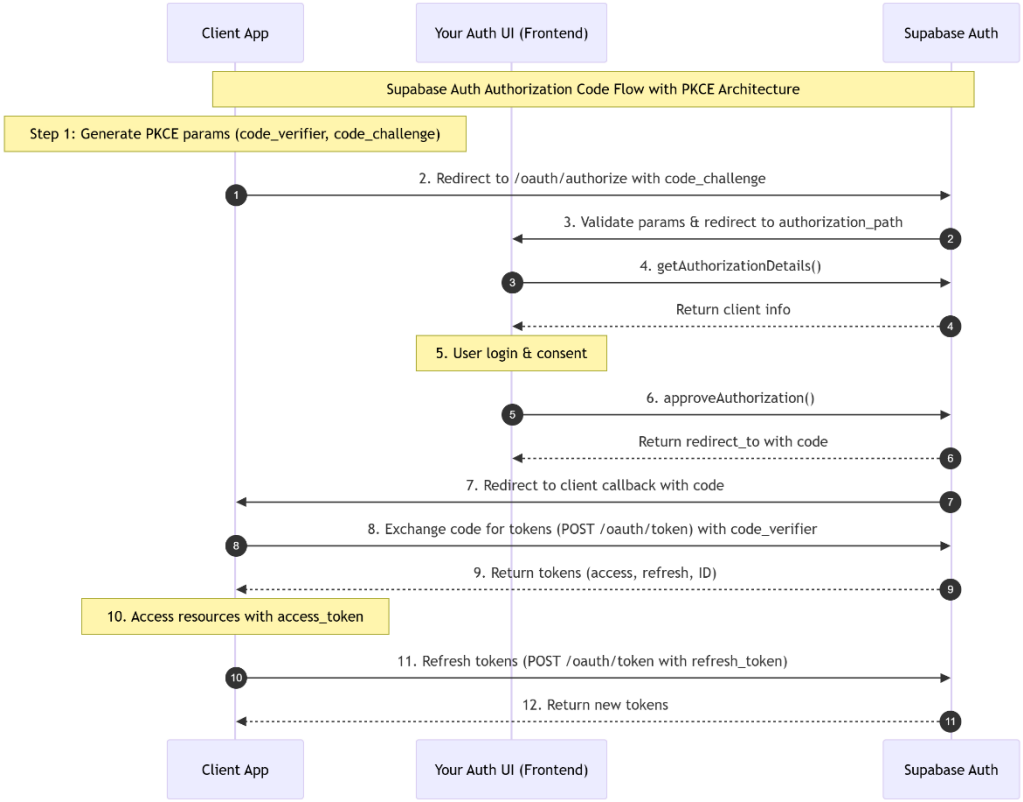

Diagram 8. Supabase Auth Flow using PKCE Architecture

6.3 Security Architecture (Row-Level Security)

Security is enforced through a Zero Trust model using PostgreSQL Row-Level Security [19], [60]. This architectural paradigm shifts security responsibility from the application code to the database engine. By defining granular access policies directly on the tables, the system ensures that data remains isolated even if the application layer is compromised, satisfying modern privacy properties required for publishing and managing relational data [1].

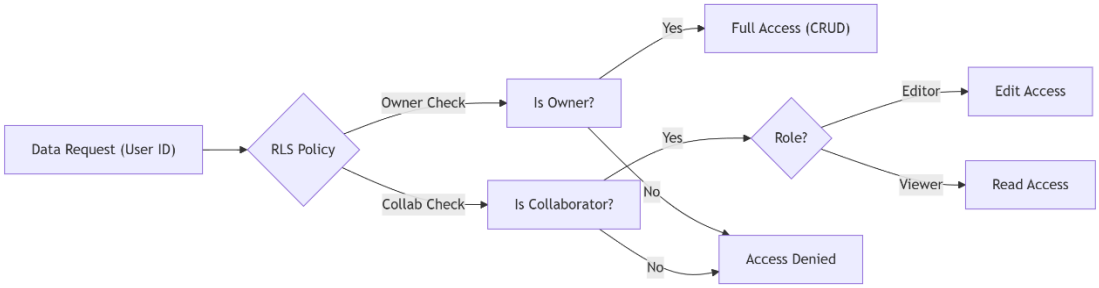

Diagram 9. Data Access Control through RLS Policies

The security architecture categorizes users into the three distinct roles outlined in Section 3.1: Owner, Editor, and Viewer. At the database level, these roles map directly to strict data privileges. Owners have full CRUD (Create, Read, Update, Delete) access to manage the trip and its collaborators. Editors are granted read and content-write permissions, allowing them to modify itineraries and POIs, while Viewers are restricted to read-only access, preventing any data mutations.

Security for the trips table is enforced through ownership and collaboration checks. Access is granted only to owners and accepted collaborators, preventing pending invitees from viewing trip details until they formally join the project. The update logic is restricted to owners and editors, while the delete operation is reserved exclusively for the owner to protect the project's integrity.

For content-related tables such as POIs and activities, the application uses a cascading check pattern. Because these entities do not always store a direct reference to the user, the database engine traverses the relationship tree to verify authorization. A request to view an activity triggers a check on the parent day, which in turn validates the user's access to the trip. Permissions for all write operations are validated by confirming the user has the required status on the parent trip. If a user is removed from a trip, their access rights are immediately revoked at the database layer.

The storage layer uses a policy system that synchronizes the trip identifier from the folder path and validates it against the user's permissions. This ensures that cover images can be viewed or modified only by authorized trip members. Finally, the system uses optimized indexes on critical columns such as owner_id and trip_id to ensure that these security checks do not introduce significant latency during high-volume operations.

6.4 Storage Strategy

Binary assets are stored in Supabase Storage buckets, which are tightly integrated with the database to enforce strict access control. The trip_images bucket is a private container used exclusively for trip cover photos. Files are organized into folders named after the unique trip identifier. Access to these images is governed by policies that grant upload and delete permissions only to the owners and editors of that specific trip. To maintain security, the application generates signed URLs that expire after one hour, ensuring that direct access to the files is temporary and authorized. In contrast, the avatars bucket is a public container pre-populated with system-provided avatars that users can select for their profiles.

6.5 Automation via Database Triggers

To improve the user experience and ensure database consistency, the system offloads critical logic to PostgreSQL triggers. These functions execute automatically within the database engine to handle repetitive tasks. When a new record is inserted into the trips table, a specialized function calculates the trip's temporal range and populates the days table. This automation makes the itinerary structure available to the user immediately upon trip creation.

The system also uses automation to streamline the initial setup of simple trips via a destination-linking trigger. If a trip is created or updated with only a single destination, the trigger automatically links that destination to every itinerary day generated for that trip. This step is critical for maintaining synchronized weather data, as meteorological information is fetched for each destination-day pair. When a trip involves multiple destinations, the trigger remains inactive, allowing the user to manually assign destinations to specific days to maintain the correct geographical context for each portion of the journey.

Identity management is further supported by a trigger that synchronizes metadata from Supabase Auth into the public profiles table whenever a new user signs up. This ensures that details such as names and avatars are consistently available throughout the application. Additionally, automated timestamp triggers are implemented across all primary tables to maintain a reliable audit trail and ensure data integrity during collaborative editing sessions. The complete sequence of this background automation, from the initial trip insertion to the conditional linking of destinations, is mapped out visually. Diagram 10 shows how these PostgreSQL triggers execute entirely within the database engine, seamlessly preparing the itinerary structure without client-side intervention.

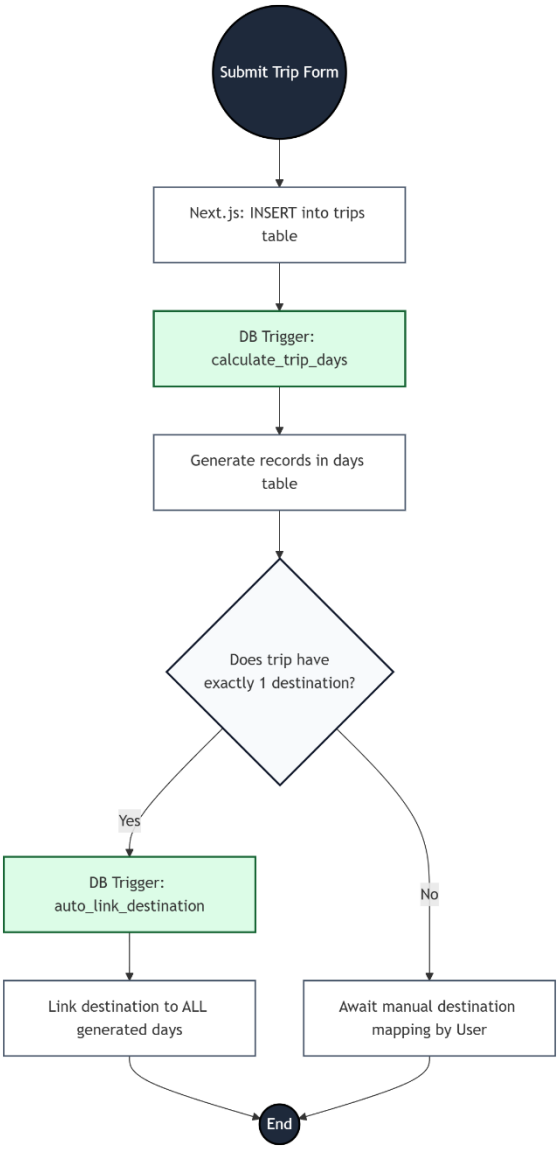

Diagram 10. Activity Diagram for PostgreSQL Trigger Automation during Trip Initialization.

6.6 Advanced Database Operations and Optimization

To maximize performance and guarantee data integrity, the application architecture offloads complex logic and search optimizations directly to the PostgreSQL database engine. By using custom PostgreSQL extensions, Remote Procedure Calls (RPCs), and automated triggers, the system minimizes client-side computational overhead and prevents network-induced race conditions.

6.6.1 Automated Profile Hydration via Triggers

A critical challenge in decoupled authentication systems is ensuring that the private authentication schema (auth.users) remains perfectly synchronized with the public application schema (public.profiles). To address this without relying on asynchronous frontend API calls, the database uses a server-side trigger (handle_new_user).

As soon as a user registers, the PostgreSQL engine intercepts the insertion and automatically creates a corresponding profile record. This PL/pgSQL function extracts the user's first and last name from the raw metadata, assigns a default profile image, and dynamically selects a random background color from a custom database ENUM (avatar_bg_color). This architectural pattern guarantees strict data consistency; the frontend application can safely assume a valid profile exists the exact millisecond a user signs in, completely eliminating "missing profile" UI errors.

6.6.2 Atomic Transactions for Complex Mutations (RPCs)

While standard CRUD operations are handled via Supabase's REST API and protected by RLS, the application requires advanced orchestration for specific workflows. To support this, the system uses Supabase RPC (Remote Procedure Call) functions [75]. These PostgreSQL functions are executed directly on the database server and are reserved for operations that require atomic transactions, batched updates, or controlled bypasses of RLS policies. The RPC architecture is deployed across critical domains to handle complex database interactions:

- **Activities:** Manages the chronological reordering and deletion of activities during drag-and-drop operations. By executing these shifts within an atomic transaction (e.g., reorder_activities), the system prevents state fragmentation and maintains a contiguous timeline.
- **Trip Management:** Automates background processes, including asynchronous weather fetching and atomic synchronization of itinerary days when dates are changed.
- **Invitations:** Safely bypasses standard RLS to validate invite tokens and retrieve required metadata, allowing users to securely join collaborative workspaces without granting premature read access.
- **Authentication:** Manages sensitive, cascading operations, such as secure account deletion that completely removes a user's identity and associated relational data.
- **Data & Analytics:** Aggregates platform-wide metrics for the application's public landing page. Because visitors are unauthenticated, the RPC function securely bypasses RLS to return high-level statistics in a single query without exposing private row data.

6.6.3 Full-Text Search Optimization (GIN Indexes)

To support the rapid, application-wide navigation provided by the Global Search (Command Palette) feature, the database must perform complex string matching across multiple tables. Traditional sequential scans using LIKE operators become severe performance bottlenecks as user-generated data (such as trip titles and POI names) grows.

To achieve sub-millisecond search query resolution, the database enables the pg_trgm (trigram) extension. Trigrams break strings into three-character sequences, allowing the engine to compute similarity scores efficiently. This is paired with a Generalized Inverted Index (GIN), applied to textual data such as the idx_trips_search index on the trip table. By mapping words directly to their row locations, the GIN index enables highly optimized, fuzzy text searches, delivering immediate results to the user's command palette without degrading the server's overall transaction throughput.

6.7 Real-Time Logical Replication

To enable the synchronous multi-user collaboration described in the data flow architecture, the database layer implements logical replication. The backend infrastructure, powered by Supabase and PostgreSQL, uses the native Supabase Realtime engine to broadcast changes. Instead of broadcasting the entire database state indiscriminately, the system configures a PostgreSQL publication to listen for data mutations (INSERT, UPDATE, DELETE) across the application's core entities.

To ensure complete synchronization of the collaborative workspace, this logical replication monitors a comprehensive suite of tables:

- **Core Trip Data:** trips, destinations, and trip_images.
- **Logistics & Itinerary:** days and activities.
- **Geospatial & Social:** pois and poi_votes.
- **Access Control:** trip_collaborators.

Crucially, this real-time broadcasting mechanism is tightly integrated with the system's Zero Trust security model and uses two distinct filtering strategies to optimize network bandwidth and enforce privacy.

For primary tables that directly reference the trip, such as destinations, days, pois, trip_collaborators, and trip_images, the real-time subscriptions explicitly use direct foreign key filtering. In this scenario, the database filters events using the trip_id parameter, ensuring it dispatches only events that match the specific channel identifier.

Conversely, for deeply nested relational tables such as activities and poi_votes, which do not natively store a trip_id column, the architecture relies exclusively on implicit Row-Level Security (RLS) filtering. Before dispatching any payload, the Supabase Realtime engine evaluates the table's RLS policies against the connected WebSocket client's JWT session. The database dynamically verifies the user's access rights through the relationship tree, ensuring that payloads are dispatched only to users with an active, verified role in the parent trip.

This dual-layered approach ensures that unauthorized users cannot intercept or eavesdrop on itinerary modifications over socket connections, maintaining strict data isolation while enabling seamless real-time updates across all facets of the travel plan.

6.8 Edge Functions & Database Automation

While most business logic resides in Next.js Server Actions, specialized tasks are offloaded to Supabase Edge Functions [30]. These serverless functions, written in TypeScript, execute on the Deno runtime at the "edge" of the network, closer to the user. This ensures low-latency execution and prevents long-running network requests from blocking the main application thread.

6.8.1 Case Study: Weather Synchronization

The fetch-trip-weather function is a primary example of edge computing in this project. It aggregates meteorological data for multiple trip destinations by calling the Open-Meteo API.

Unlike many commercial weather providers, Open-Meteo offers a high-resolution, open-access API that does not require an API key for standard use, simplifying the edge function's logic and improving reliability. The implementation intelligently switches between two specialized APIs to provide accurate estimates regardless of the trip dates:

- **Forecast API:** Used for dates within 16 days of the current date to provide real-time weather predictions.
- **Archive/Historical API:** Used for dates further in the future or the past. For future dates beyond the forecast window, the system retrieves historical averages for those dates from previous years to provide travelers with a statistically accurate "estimated" weather condition.

The automated decision-making logic governing this synchronization is shown below. Diagram 11 details the activity flow of the pg_cron trigger and the Edge Function as they evaluate the trip's temporal proximity and route requests to the appropriate meteorological endpoint.

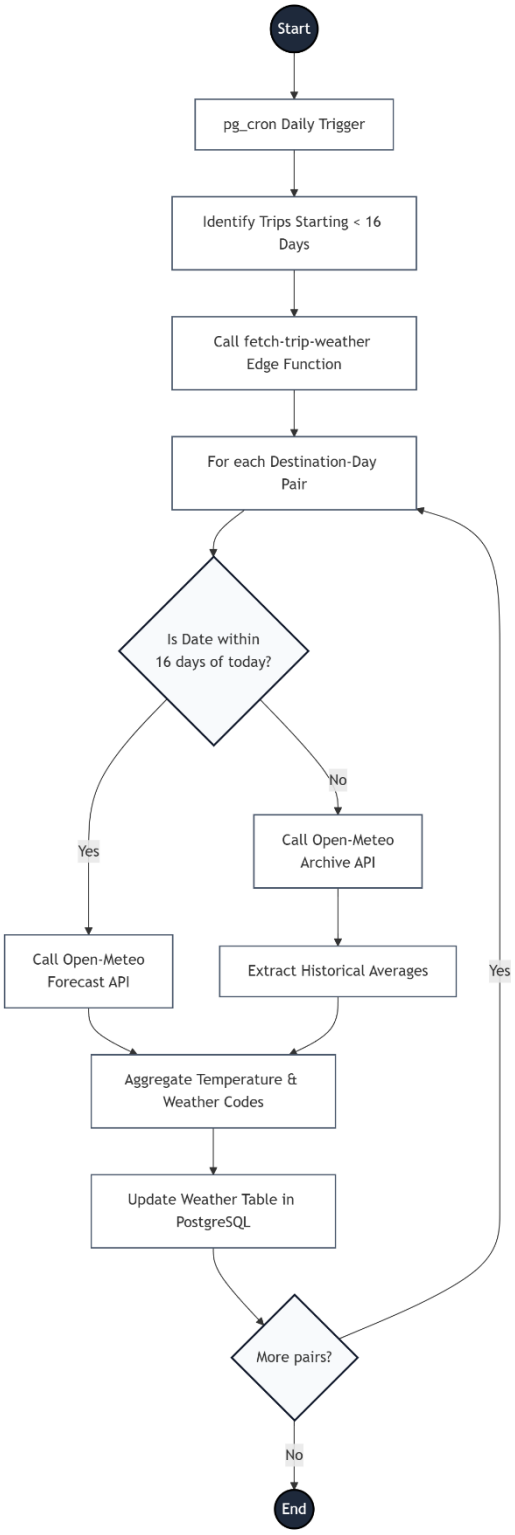

Diagram 11. Activity Diagram of the Automated Background Weather Synchronization Process.

6.8.2 Database Automation & Cron Jobs

The application maintains data freshness without manual user intervention by using a built-in cron scheduler powered by the pg_cron extension [27]. This system enables recurring tasks to run directly

within the PostgreSQL database. A specific daily task is configured to identify all trips starting within a sixteen-day window and automatically trigger the fetch-trip-weather synchronization function.

This proactive automation ensures that weather icons and temperature data are updated in the background, making accurate information immediately available when a user opens their itinerary. By leveraging the pg_net extension with the cron scheduler, the database makes non-blocking HTTP POST requests to the Edge Functions, ensuring the main application's performance is never compromised by background maintenance tasks.

7. Software Design & Code Structure

7.1 Project Directory Structure

The application uses a highly modular directory structure designed to isolate the user interface from business logic, ensuring scalability and ease of maintenance [51]. The organization clearly separates the Next.js routing layer from the underlying feature modules.

7.1.1 Routing Architecture (The app/ Directory)

The application leverages the Next.js App Router with advanced routing patterns to manage state and enforce access control through the URL structure [73]. This hierarchy is organized through a series of nested layouts that progressively initialize the application environment.

The Root Layout is the foundational entry point for the entire application. It initializes global providers such as the ThemeProvider for visual consistency and the AuthProvider for identity management. A defining technical feature of this layout is the server-side initialization of the Supabase client to fetch the current user session before rendering the component tree. By passing this initial user data to the client-side authentication provider, the system prevents the rendering of unauthenticated states and ensures a smooth hydration process. Furthermore, the root layout also includes global utilities, such as the ScrollToTop component and a GlobalInviteListener wrapped in a Suspense boundary. This listener monitors the application state for incoming trip invitations, allowing the system to respond to shared links regardless of the user's current route.

The Protected Layout, located within the (authenticated) route group, serves as the primary workspace for logged-in users. It uses a SidebarProvider to manage a dual-sidebar architecture comprising the main application navigation and a specialized geospatial sidebar. Within this shell, a BreadcrumbProvider works with the NavBreadcrumbLayout to give users a consistent sense of location. This layout also integrates Vercel Analytics at the top level to monitor application performance and user engagement in production. Notably, this analytics integration is strictly cookieless and anonymized, aggregating usage trends without storing Personally Identifiable Information (PII) to align with the platform's broader privacy-first routing architecture.

In addition to dynamic collaborative workspaces, this architecture includes dedicated static segments for operational compliance, specifically the /privacy and /terms routes (app/privacy/page.tsx and app/terms/page.tsx). These pages outline the platform's privacy protocols—which rely exclusively on essential authentication cookies and eliminate the need for third-party tracking or disruptive consent banners. Furthermore, the Terms of Service establish an Acceptable Use Policy that explicitly prohibits malicious activity and automated data scraping, while transferring liability for user-generated content directly to users. By implementing these as static routes within the Next.js App Router, the application efficiently delivers these essential legal frameworks without unnecessary server-side rendering overhead.

Finally, the Trip Detail Layout is a dynamic segment that manages context for specific travel projects. It performs a critical server-side existence check by attempting to fetch the trip title; if the record is missing, the layout invokes the notFound() utility to trigger a graceful 404 state. Beyond validation, this

layout determines the user's permissions by fetching their role for that trip. This data is then injected into a TripProvider, which exposes the trip identifier and role-based flags, such as canEdit and isOwner, to all child components. This ensures that UI elements like the TripTabs and itinerary controls are rendered according to the user's actual permissions. Additionally, a WeatherLoadingProvider is initialized here to coordinate the asynchronous loading of meteorological data across the trip workspace.

7.1.2 Core Modules & Logic

Beyond the routing layer, the project is organized into specialized directories that manage the application's internal state and data flow. The lib/ directory serves as the logic center and contains several critical subdirectories. The data/ folder manages the read-only layer through optimized queries. The mappers/ folder handles data transformation, and the validations/ folder stores Zod schemas. A dedicated supabase/ folder is also located within this directory to house the client factories and configuration needed to instantiate database connections across different execution environments.

The components/ directory is structured to support long-term maintainability and scalability. Within this directory, the ui/ folder hosts all low-level Shadcn components and base primitives. The remaining folders organize and separate each complex domain-specific organism. This architecture keeps the codebase navigable even as the number of features grows. Drawing on modern atomic design methodologies [23], the component structure follows a clear progression, with complexity increasing from small, atomic elements in the UI folder to large, integrated organisms in the feature-specific directories. This separation of concerns enables the independent development of complex interface segments, such as the map sidebar or the interactive itinerary page. Finally, the types/ and hooks/ directories provide centralized definitions and reusable state logic to maintain consistency across the entire codebase.

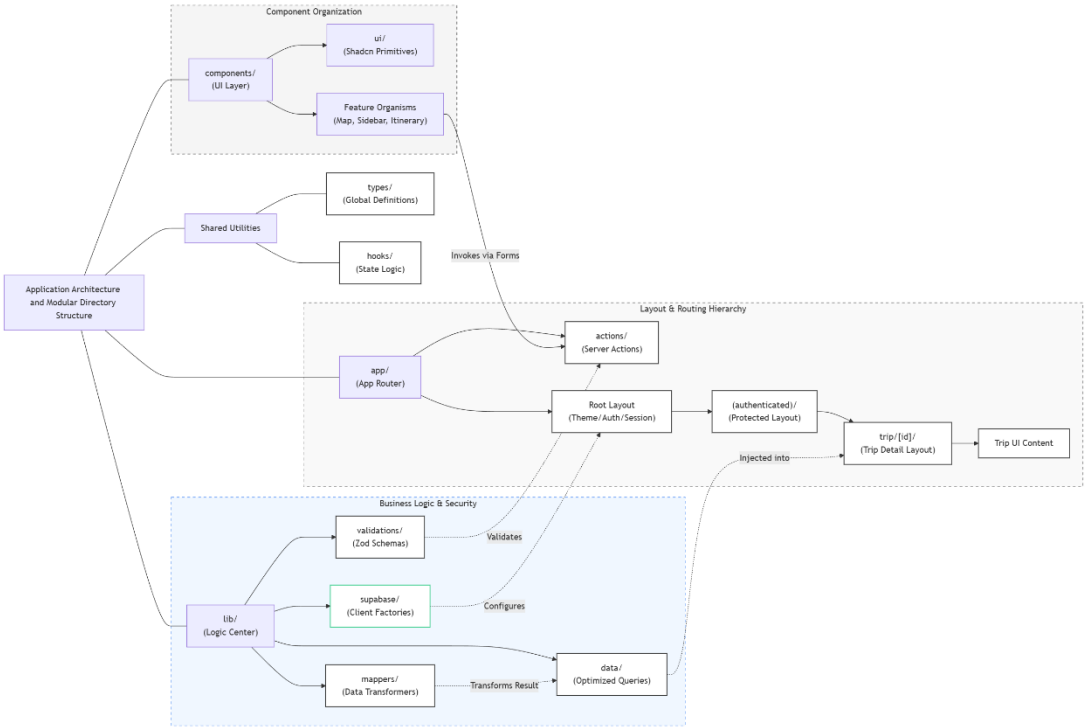

Diagram 12. Project Directory Structure Diagram

7.2 Architectural Design Patterns

The application architecture is defined by a strict separation of read and write operations. Read operations are centralized in the lib/data directory. These modules contain pure functions that execute optimized

Supabase queries, allowing React Server Components to fetch data directly on the server. This decouples the View layer from the database, enabling centralized optimization and easier testing. Conversely, all mutations, including creation, updates, and deletions, are handled exclusively by Server Actions in the app/actions directory [34]. These actions serve as secure gateways that handle authentication, permission checks, and cache invalidation [36]. The standard workflow involves verifying the user's identity, followed by a permission check, before executing the mutation and triggering a revalidation of the relevant path.

To manage complex data requirements efficiently, the application uses asynchronous promises to orchestrate data fetching. Instead of multiple client-side round-trips, a single server-side function handles sequential or parallel requests to build a complete record. This is particularly evident in the trip layout, where the system fetches titles, user sessions, and roles in parallel to minimize total blocking time during navigation. This orchestration ensures the UI receives a fully prepared dataset, reducing the need for cascading loading states on the client.

7.3 Structural Patterns

- **Module Pattern:** Data modules (e.g., data/profile.ts) are implemented as TypeScript modules that export related operations, providing a clear namespace and singleton-like behavior for service logic.
- **Dependency Injection (Method Level):** Data Functions in the lib/data/ folder accept external dependencies (such as the SupabaseClient) as arguments rather than creating them internally. This approach enables easy mocking of dependencies during testing and better control over the client lifecycle.
- **Mapper Pattern:** Raw database rows are transformed into typed DTOs, decoupling the database schema from application types. This ensures that changes to database column names affect only mappers, not the entire codebase.
- **Metadata Pattern**: The application uses Next.js App Router conventions to ensure consistent Search Engine Optimization. Static metadata is used for fixed routes, while dynamic metadata is generated for trip-specific routes to ensure social sharing links display the correct context.
- **StrictContext Pattern**: A helper for creating type-safe React Contexts that throw errors when used outside their Provider, eliminating the need for constant "undefined" checks.
- **Factory Pattern**: The createClient() helper in lib/supabase/ constructs the correct Supabase client instance based on the environment (Browser vs. Server).

7.4 Behavior Patterns

- **Custom Hooks Pattern:** Logic that involves React state or lifecycle methods is encapsulated in hooks such as useUser or useDatabaseAction [59].
- **React Forms Pattern:** The application implements a robust form management system that follows a standardized tripartite approach, using React Hook Form for state management, Zod for schema-based validation, and Shadcn UI for accessible interface components [57]. Rather than relying on raw register calls, which can tightly couple logic to the DOM, the architecture employs the Controller Pattern. This enables deep decoupling between form state and custom UI elements, allowing precise tracking of validation states, such as invalid status, and facilitating graceful error message rendering. To ensure maximum reliability, the system performs Dual-Layer Validation. Validation is enforced first on the client-side to provide users with immediate interactive feedback and then on the server-side within the Server Action to safeguard data integrity against malformed or unauthorized requests.

- **Optimistic UI Pattern:** The "Transition + Toast + Optimistic" pattern makes the UI feel instant [38]. The local state updates immediately, while toast.promise() provides background feedback and automatically rolls back on failure.
- **Empty State Pattern:** The application handles zero-data scenarios with reusable Empty components that replace confusing blank screens with clear visual cues. These components support user onboarding by providing helpful messages and call-to-action buttons that guide users in populating the page. By suggesting immediate steps such as inviting collaborators or adding activities, the system reduces friction during the initial planning phase.
- **Global Feedback (Sonner):** A designated "Toaster" component serves as a global observer, displaying non-blocking notifications in response to application events.
- **Global Navigation Pattern (Command Palette):** To streamline navigation, the application implements a globally accessible command palette (components/share/global-search.tsx). Built on cmdk and shadcn/ui, it is mounted at the root of the authenticated shell (app/(authenticated)/layout.tsx) to ensure global event listeners (Ctrl + K and custom DOM events) persist across all protected routes. To optimize performance, data fetching (getGlobalSearchData()) is lazy-loaded. It runs only on the user's first interaction, using React useRef to memoize the state and prevent redundant database queries. Results are neatly categorized into static pages, active trips, and contextual Points of Interest. A centralized callback handles user selections by closing the dialog and using Next.js useRouter to seamlessly navigate to the targeted route (e.g., /itinerary/pois/[id]).

7.5 Caching Architecture and Data Revalidation

To ensure high performance and strict data consistency across collaborative sessions, the application leverages the advanced caching mechanisms built into the Next.js 16 App Router. By default, the framework aggressively caches data to minimize network latency and reduce database operational costs. However, in a real-time planning environment, this caching must be carefully coordinated with data mutations.

7.5.1 Request Memoization (React Cache)

The first layer of optimization uses Request Memoization with React's cache() function. This technique memoizes data fetches within a single server render pass. It is widely used in the data access layer (e.g., the lib/data directory) to prevent redundant database queries when multiple components across the React tree require the same foundational data, such as a trip's title or a point of interest's metadata.

By wrapping these specific queries in the cache() mechanism, the lifecycle of the fetched data is tightly bound to the duration of the specific server request. This ensures that if a parent layout and a deeply nested child component both request the trip's title, the Supabase database is queried only once, optimizing execution times without risking stale data being delivered to the client.

7.5.2 Persistent Data Cache and On-Demand Revalidation

Beyond per-request memoization, the system uses the Next.js Data Cache to persist fetch results across multiple requests and deployments. To resolve the inherent conflict between persistent caching and dynamic collaborative editing, the architecture implements a strict On-Demand Revalidation strategy.

When a user initiates a data mutation, such as creating an activity, updating a point of interest, or deleting a trip, the operation is processed through a secure Server Action. Upon a successful database transaction, the system explicitly invokes the Next.js revalidatePath() utility, which purges the cached data for the affected routes.

To optimize performance, this revalidation is applied granularly. By targeting specific paths (e.g., /trips/[id]) rather than executing broad layout purges, the application minimizes unnecessary background refetching while guaranteeing that all collaborators immediately receive the most up-to-date and accurate state of the itinerary after any structural modification.

7.6 Real-Time State Synchronization and Channels

To seamlessly bridge backend logical replication and the client-side user interface, the application implements a dedicated real-time listener component (TripRealtime). Operating strictly as a Client Component within the Next.js App Router paradigm, it uses React lifecycle hooks to establish a persistent WebSocket connection through the Supabase client.

The architecture uses a single channel to multiplex events across the monitored tables. When the database broadcasts an INSERT, UPDATE, or DELETE event, the client component intercepts the payload and initiates a synchronization sequence. However, integrating real-time web sockets with Next.js Server Components and an Optimistic UI introduces two significant technical challenges: the "echo" effect (where a user receives a broadcast of their own mutation) and cache invalidation bottlenecks.

To address these challenges, the implementation uses two distinct state-management strategies:

- **Echo Cancellation:** To preserve the fluidity of the Optimistic UI, the system must ignore self-triggered database broadcasts. By applying a brief grace window immediately after a local mutation, the application automatically discards incoming "echo" events. This prevents the interface from flickering or temporarily reverting states due to race conditions between the optimistic local update and the subsequent server broadcast.
- **Debounced Cache Invalidation:** High-frequency collaborative actions, such as rapid drag-and-drop reordering, can generate bursts of real-time payloads. To prevent performance degradation from excessive cache invalidations, the component aggregates incoming updates and applies a short debounce algorithm.

After the debounce timer resolves, the system triggers a single background refresh of the affected components. Concurrently, a non-blocking notification alerts the user to external modifications, preserving contextual awareness of the collaborative workspace without disrupting their active workflow.

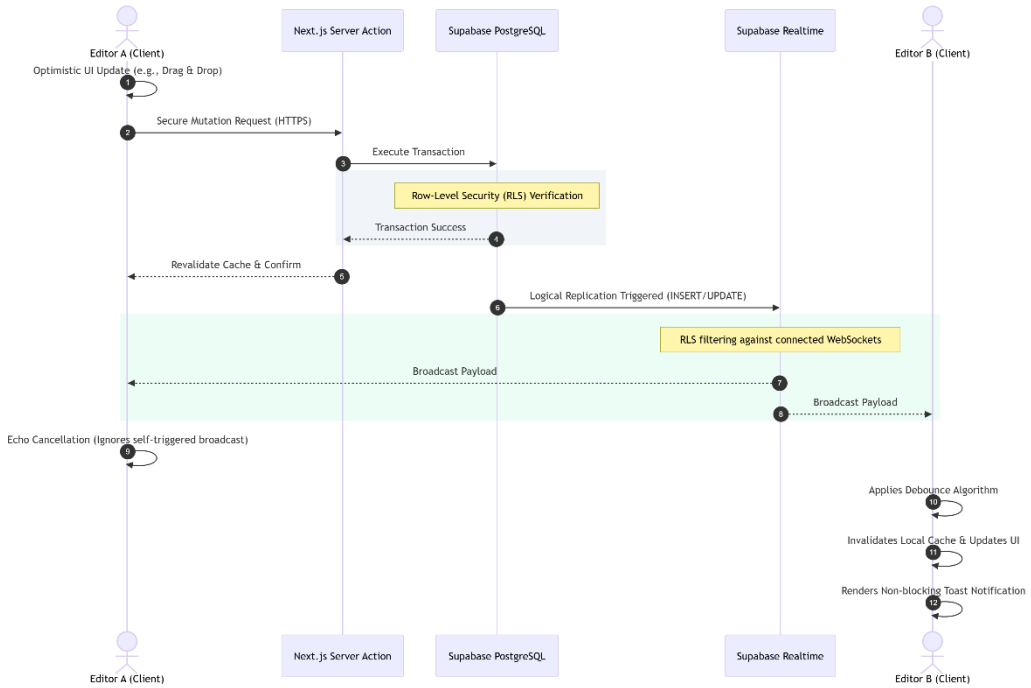

Diagram 13. UML Sequence Diagram of the Real-Time Synchronization Pipeline

7.7 Data Grid Architecture and URL State Management

To manage complex datasets such as expenses and collaborator lists, the application implements a highly modular, responsive data grid built on top of TanStack Table v8. This architecture ensures performant data visualization while providing advanced filtering, sorting, and pagination capabilities.

7.7.1 The Server-to-Client Composition Pattern

The data table architecture follows a strict "Server Wrapper to Client Logic" pattern, aligning with the application's broader React Server Components strategy.

The entry point is a Server Component (data-table.tsx) responsible for executing the secure database query. Rather than blocking the render while waiting for the data, this component passes the data promise down to a Client Component (data-table-client.tsx). The client serves as the operational core of the table, using React's use() hook to unwrap the streaming data while managing local table state. This separation keeps heavy data-fetching logic securely on the server while the client maintains fluid interactivity for sorting and filtering.

7.7.2 Responsive Rendering Strategy

To support the application's mobile-first philosophy, the data tables use a dual-rendering strategy. On desktop viewports, the data is rendered in a traditional, high-density HTML grid layout with customizable view options and column toggles.

Conversely, on mobile devices, the system abandons the traditional table structure, which often suffers from horizontal scrolling on small screens, favoring a card-based layout (e.g., ExpenseCard, CollaboratorCard). Both visual implementations share the exact same underlying TanStack Table instance logic for data manipulation, ensuring functional parity across devices while optimizing touch ergonomics for mobile users.

7.7.3 URL-Driven State Synchronization

A defining technical feature of the data grid architecture is the deep synchronization of the table's state with the browser's URL. This is managed by a custom useDataTableSearchParams hook that persists the following parameters directly in the URL:

- **Search Queries:** Global or column-specific text searches (mapped to ?q=).
- **Faceted Filters:** Multi-select categorical data arrays (mapped to ?f_{id}=).
- **Date Filters:** Start and end date constraints (mapped to ?df_{id}=).
- **Pagination:** The active page index and page size (mapped to ?page= and ?pageSize=).

This URL-first approach to state management delivers significant UX benefits. It lets users bookmark specific data views, share filtered results via links, and ensures that complex search states are preserved exactly across page refreshes.

7.7.4 Silent Updates and Performance

Integrating high-frequency local state updates (such as typing in a search bar) with URL parameters can introduce performance bottlenecks in the Next.js App Router. If the application relied on the standard router.replace() method, each keystroke would trigger a server-side re-render of the route tree, resulting in severe input lag.

To resolve this, the useDataTableSearchParams hook implements "silent updates" using the native browser window.history.replaceState() API. This technique updates the URL in the browser's address bar

instantly without notifying the Next.js router. Consequently, the table gains the benefits of persistent URL state management while maintaining the instantaneous feedback loop of purely client-side React state.

7.8 Data Flow & Responsive Implementation

The application follows a strict mobile-first design philosophy, with base styles written for the smallest screens and progressively enhanced for larger viewports. The project uses a dual-breakpoint system that combines standard Tailwind viewport-based increments with modern Container Queries [58]. These queries allow modular components, such as statistics cards or POIs popups, to adapt to their parent container's width rather than the browser window. This is particularly critical for the draggable map sidebar, where adjusting the panel width dynamically reflows adjacent content, maximizing available workspace. To ensure optimal ergonomics, the system uses media queries to conditionally render dialog modals on desktop and bottom-sheet drawers on mobile devices. This design places critical controls within the natural reach of the user's thumb. On trip-specific pages, the interface further adapts by replacing desktop breadcrumbs with a touch-friendly tab bar for efficient one-handed navigation between the itinerary and map views. Finally, the implementation prioritizes CSS-based responsiveness to minimize main-thread overhead while reserving JavaScript hooks for structural changes that cannot be achieved through styling alone.

The application manages network latency by implementing an Optimistic UI pattern via a custom hook [70]. When a user initiates a data mutation, such as creating or deleting content, the interface immediately updates the local state to provide instant visual feedback. Simultaneously, the system displays a loading toast notification to inform the traveler that the process is running in the background while a secure request is dispatched to the server. This pattern concludes with a confirmation or rollback phase. Upon a successful database response, the server cache is revalidated, and the loading indicator is replaced with a success message. If the operation fails, the system automatically reverts the local state to its previous valid snapshot and displays an error message explaining the issue. This robust error-handling and state-reconciliation process is critical for maintaining a seamless collaborative experience. Diagram 14 illustrates this Optimistic UI lifecycle, showing the exact sequence of client-side predictions, secure server-side validations, and the rollback mechanism.

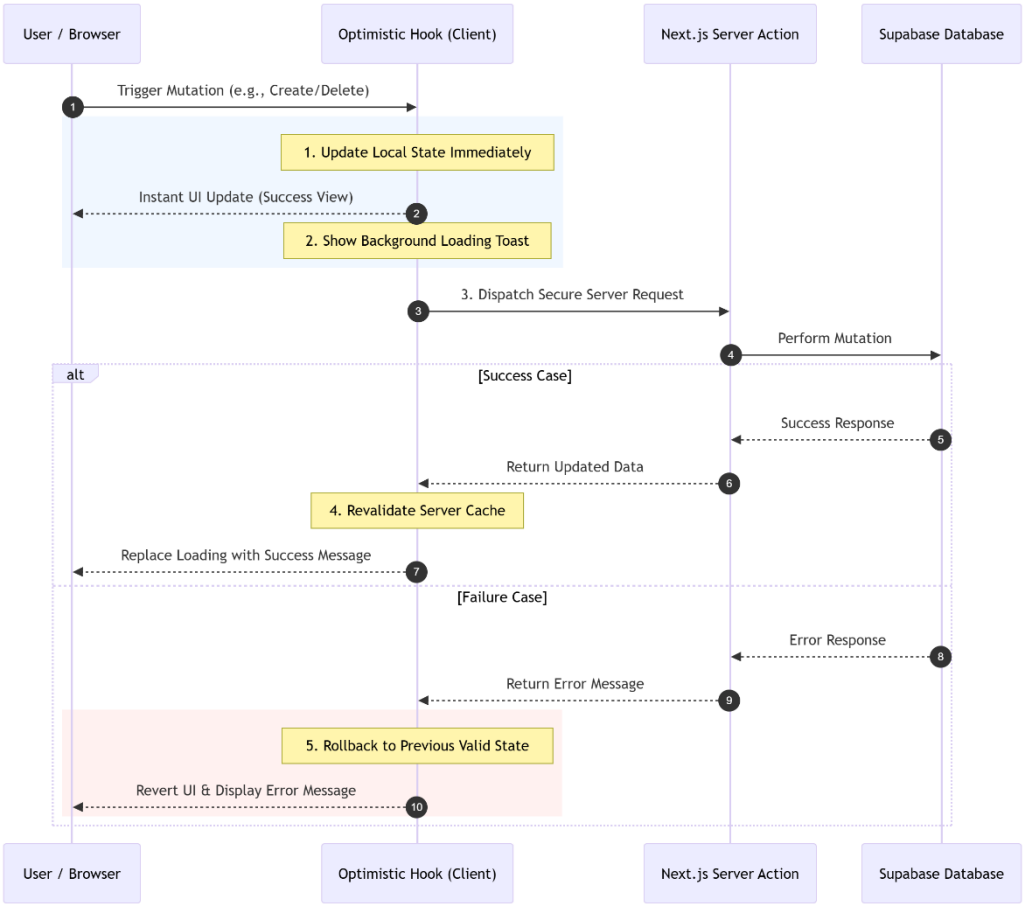

Diagram 14. Optimistic UI lifecycle sequence diagram

To maximize perceived performance, the application implements a universal streaming server-rendering strategy across all primary routes. Each segment provides an immediate layout shell through route-level skeletons that mirror the final page hierarchy. For example, the expenses module uses dedicated skeletons for budget summaries and progress charts to ensure the user perceives the application as loaded before the data settles. The system follows a shell-and-stream architecture, where static elements are sent immediately, while data-intensive components are streamed in parallel. This is achieved through a promise-passing pattern in which server components initiate multiple asynchronous fetches without blocking the initial render. Instead, raw promises are passed as props to client components, which use the React use hook to unwrap the data. This standardized approach ensures that fast-resolving information appears instantly while slower datasets resolve without delaying the rest of the interface.

7.9 Database Environment Management and Seeding Architecture

To ensure the integrity of user data and maintain a reliable development lifecycle, the application enforces strict environment isolation, supported by automated schema management and data seeding scripts.

7.9.1 Environment Strategy and Isolation

The project architecture maintains two completely separate Supabase database environments. This physical separation prevents development or testing operations from inadvertently corrupting live user data.

- **Test / Local Environment:** Dedicated exclusively to local development and the execution of automated testing pipelines (Jest and Playwright). This environment is configured locally using .env.local variables.
- **Production Environment:** The live application database serving real users, secured and accessed via .env.production variables.

7.9.2 Schema Management via Supabase CLI

To maintain structural parity across isolated environments, the project strictly prohibits manual schema mutations via the Supabase Dashboard. Instead, database evolution is managed programmatically using the Supabase CLI [78].

When the schema is modified during development, the changes are extracted from the local or test database into version-controlled migration files using the supabase db pull command. These tracked migration files are then systematically applied to the target remote databases (such as the production environment) using the supabase db push command. This workflow ensures that all environments remain perfectly synchronized and provides a clear audit trail of structural database changes over time.

7.9.3 Automated Data Seeding

To accelerate development and testing, the application includes a specialized seeding engine accessible via dedicated Node package scripts (pnpm db:seed:local and pnpm db:seed:prod).

This script temporarily bypasses Row-Level Security to populate the target environment with high-fidelity datasets, generating thousands of records for users, profiles, trips, destinations, POIs, and activities. To achieve maximum realism, the engine uses the @faker-js/faker library to generate authentic identities and descriptive trip content.

The seeding process validates the system's core logic and automated dependencies. By leveraging existing database triggers to generate calendar days and user profiles, the script continuously tests the backend's reactive logic. Additionally, the seeding algorithm groups destinations and POIs into spatially logical regions, creating a realistic, data-rich environment for evaluating the performance of the geospatial module and map clustering algorithms.

7.9.4 Integration with Testing Pipelines

Environment isolation and seeding strategies are deeply integrated into the project's automated testing workflows. Both the Jest unit testing suite and the Playwright End-to-End testing suite are configured to run exclusively in the Test environment. By running the local seed script before test execution, the pipeline ensures that the exact user profiles and trip states required for complex E2E workflows are present in the database, producing deterministic, reliable test outcomes without compromising the integrity of the production environment.

8. Geospatial Module Architecture (Map Implementation)

The geospatial module serves as the application's primary visual engine, bridging the temporal itinerary and the physical world. Unlike traditional travel applications, where maps often serve as static secondary views, this system integrates Geographic Information Systems (GIS) with web-based architectures [15] and leverages advanced techniques for the interactive visualization of travel itineraries [13]. This treats the map as a dynamic, state-driven component that responds to the user's planning context. This integration allows travelers to visualize the geographical feasibility of their schedules in real time, making the map an essential tool for logistical decision-making.

8.1 Functional Modes and Navigational Context

The mapping engine operates in three distinct modes that adapt to the user's location within the application. On the personal dashboard, the map provides a high-level geographical summary of the traveler's history. In this view, the system supports two primary visualization modes for representing the user's footprint. The first mode applies a fill effect to entire countries to highlight the breadth of the traveler's reach at the national level [35]. The second mode focuses on specific destinations using localized markers. This dashboard interface is primarily designed for reflection and high-level progress tracking.

The dedicated map page, accessible via the primary sidebar, provides the most expansive exploration environment. This page is optimized for searching and managing the user's entire travel catalog. It serves as a centralized geospatial library where travelers can jump to any saved itinerary worldwide. This mode prioritizes a global scope, allowing the user to interact with their data beyond the constraints of a single trip or daily schedule.

Finally, the map sidebar provides a specialized planning utility that is most active during trip creation and organization. This sidebar is implemented exclusively for the desktop application to support a sophisticated multi-panel workspace. When the traveler enters a specific trip workspace, the engine automatically pivots its data source to focus on the active itinerary. In this planning context, the navigational focus shifts from a global perspective to the specific temporal segments of the journey, allowing the traveler to focus on the logistical details of their upcoming plans.

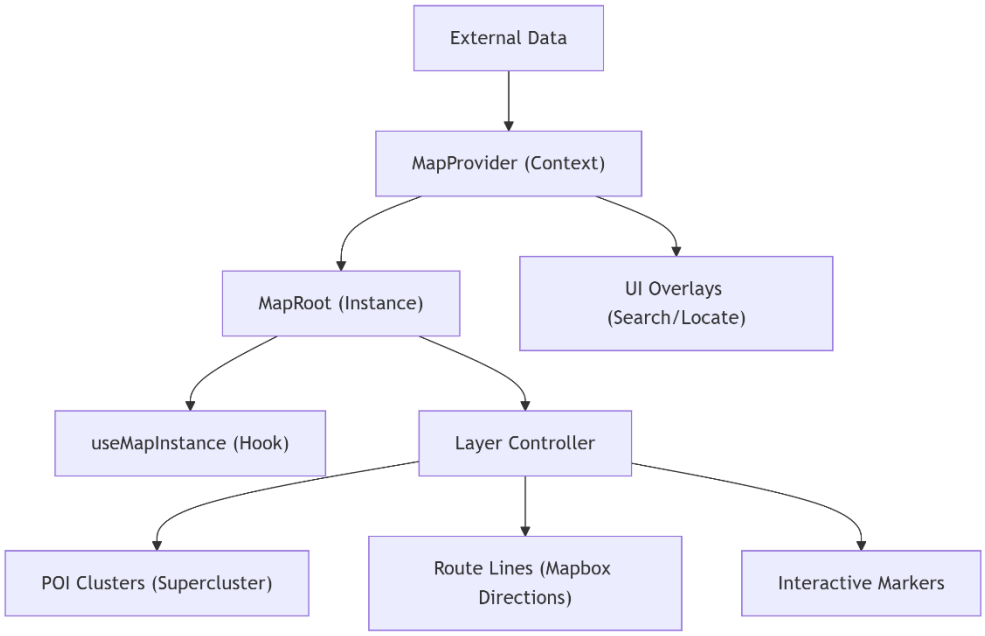

Diagram 15. Geospatial Component Hierarchy

8.2 Hierarchical Architecture and State Management

To maintain a responsive interface while handling heavy geospatial computations, the module uses a strictly hierarchical component structure. The MapSidebar serves as the global layout layer and a persistent geospatial companion. Given the complexity of split-view interactions and the precision required for itinerary organization, this sidebar is restricted to desktop viewports. Mobile users access the full range of geospatial functionality through the dedicated map page in the primary navigation sidebar, ensuring that the mobile experience remains functionally equivalent to the desktop version without the structural overhead of the integrated sidebar.

At the core of the module is the MapWrapper, which serves as the primary state management orchestrator. This component parses the navigational context, including trip, day, and POI identifiers, to

determine the canvas's visual state. It maintains several reactive states that govern the interface, including coloringMode, which determines whether markers are grouped by category or itinerary day. The MapWrapper also synchronizes external location searches with internal POIs, ensuring that selecting a new geocoded location automatically clears previously selected locations to maintain focused user intent. Below this layer, POIMarkers and LayerManager handle the final rendering of markers, lines, and clusters by subscribing to these shared state values.

8.3 Hybrid Search and Advanced Filtering Patterns

The module implements a hybrid search architecture that is consistently available across both the dedicated map page and the planning sidebar. This system allows travelers to query external location data via the Mapbox Geocoding API while simultaneously searching for existing points of interest in the internal database. When a traveler searches for a new location, the engine returns high-accuracy global results [44] and automatically extracts metadata to pre-populate creation forms, handling complex data such as precise coordinates and addresses automatically.

To ensure high usability and data discoverability, the MapWrapper coordinates a suite of interactive filters. Both the Global and Trip modes use shared filtering components for category and status, allowing users to isolate specific location types, such as attractions or accommodations. However, the Trip mode introduces a specialized MapColorFilter that lets travelers toggle the visual representation of the itinerary. In "Category" mode, markers use a standardized color palette based on activity type. In "Day" mode, the system uses a dynamic dayLegend state to assign unique colors to each day of the trip. This enables users to see at a glance how their journey is distributed across different days and locations. These filters use responsive positioning logic that adjusts their vertical alignment based on whether a specific location or POI is currently selected, ensuring the controls never obstruct the relevant content cards.

8.4 Context-Aware Zoom and Interaction Logic

The primary utility of the map lies in its ability to clarify the traveler's current planning focus through context-aware transitions. The system uses a specialized algorithm to manage the viewport and the visual prominence of markers based on the specific page the user is visiting. When a traveler navigates to a general trip ID page, the map engine performs a data isolation process. First, it identifies all POIs associated with that trip and automatically calculates a bounding box, zooming the camera until all relevant markers are visible. Simultaneously, the engine filters out markers from all other trips to prevent visual clutter and cognitive overload. This ensures the traveler is not overwhelmed or confused by unrelated data points, resulting in a focused environment that simplifies the planning process.

This behavior becomes more granular as the user navigates deeper into the itinerary. If the user visits a specific day ID page, the map recalculates its focus to zoom exclusively on the POIs scheduled for that date. To provide a "Focus + Context" experience, the system applies a grayscale effect to all other trip markers, ensuring the active day's activities are prominent while keeping the rest of the trip's geography visible as a reference. The same logic applies when a user navigates to a specific POI ID page. In this case, the map zooms directly into the individual location and grayscales all other markers in the trip, providing an isolated view of the selected activity within its broader environmental context.

8.5 Rendering Strategy and Performance Optimization

Managing the performance of geospatial data is a critical requirement for maintaining a fluid user experience. The application uses a hybrid rendering strategy that combines high-performance WebGL for the map background with React-based overlays for interactive elements [71]. This approach enables the system to use standard React features, such as tooltips and popovers, while leveraging the Mapbox canvas for the primary visual layer.

To handle large datasets without degrading performance, the engine uses native GeoJSON clustering. When markers are close together, they merge into a single cluster circle that shows the number of points in that area. This significantly reduces the number of elements in the Document Object Model and maintains a smooth sixty frames-per-second interaction during rapid panning and zooming. Furthermore, the map uses a local cache to minimize network overhead, ensuring that markers are re-rendered only when the underlying database record changes.

9. Functionality & User Workflow

This chapter explores the practical application of the system by detailing the core user workflows and functional journeys. By examining the step-by-step interactions across authentication, trip management, and collaboration, we can observe how the underlying technical architecture facilitates a seamless travel planning experience.

9.1 Authentication and Onboarding

The user journey begins with a secure onboarding process that establishes a verified identity within the application. A guest traveler starts this flow by visiting the landing page and accessing the sign-up interface. During registration, the system collects essential metadata, including the traveler's name and email address. Upon submission, the application redirects the user to a dedicated success page that instructs them to verify their identity by clicking a confirmation link sent to their inbox.

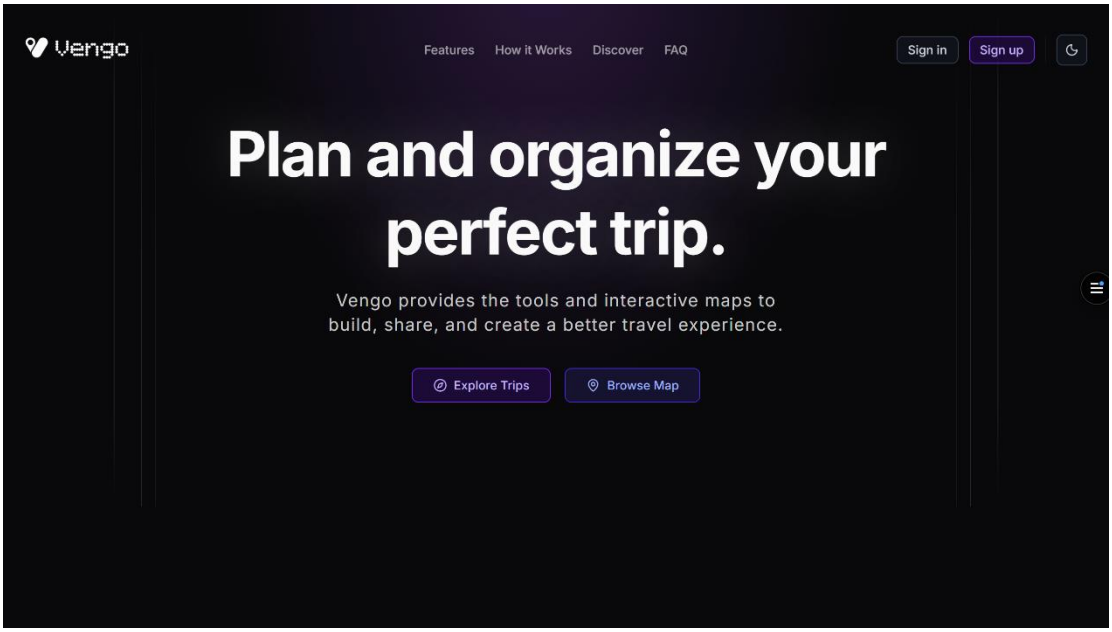

Figure 7. Landing page hero section for guest users.

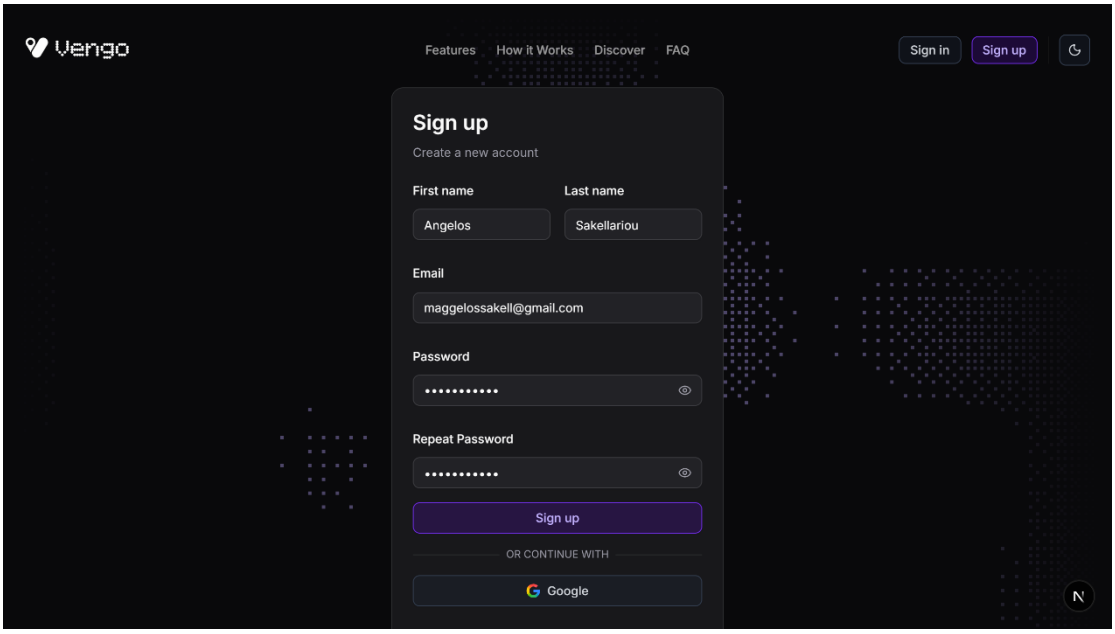

Figure 8. Registration form for collecting user metadata.

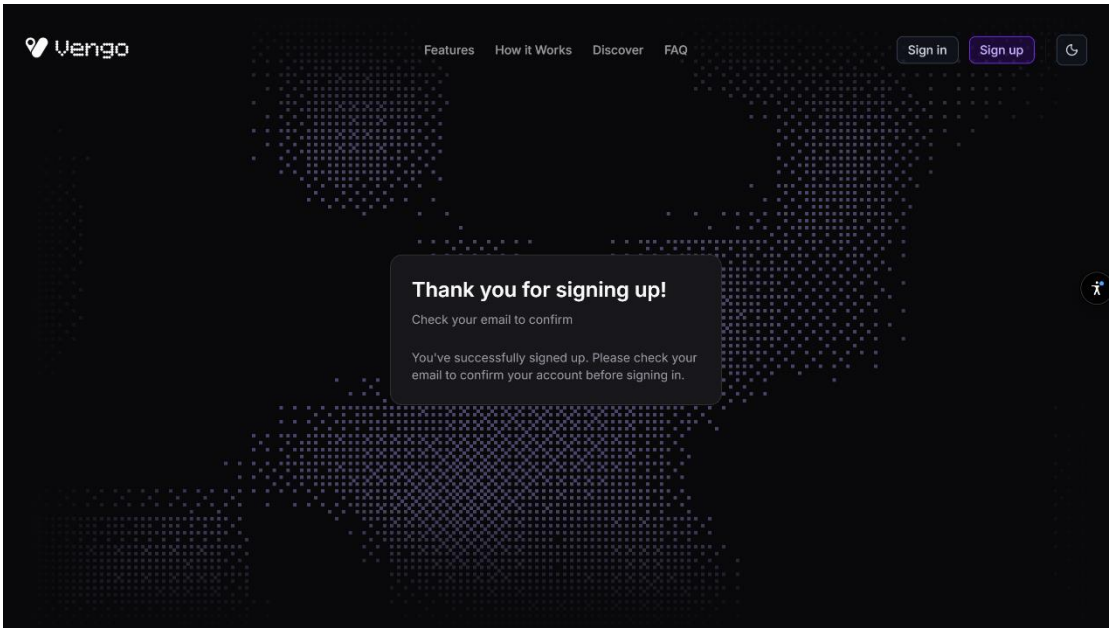

Figure 9. Post-registration success and verification prompt.

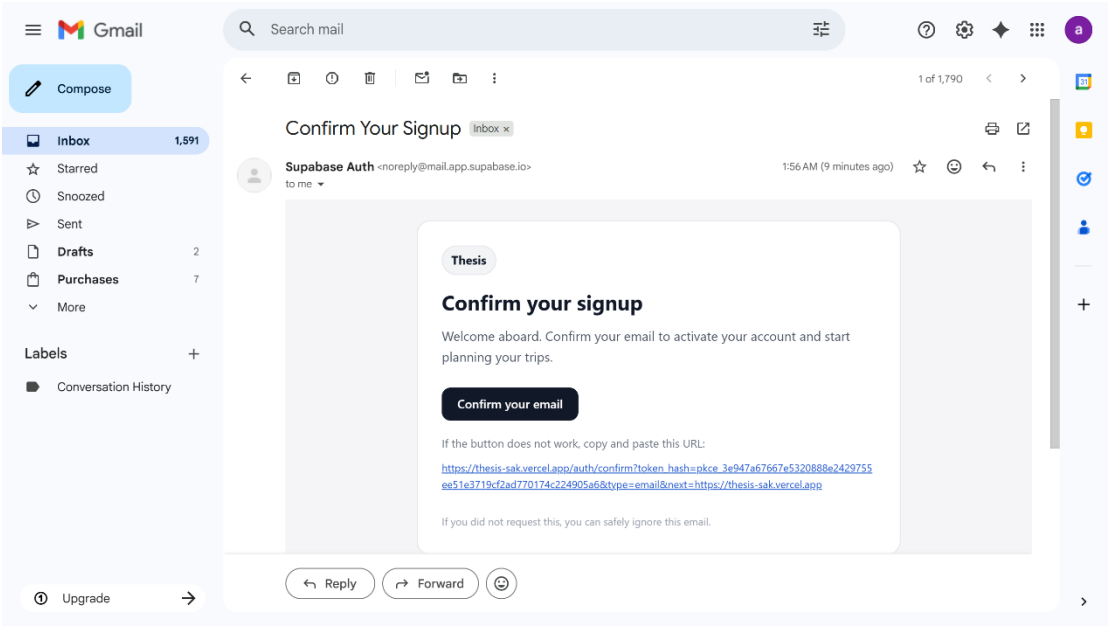

Figure 10. Custom-designed email for identity verification.

Once the user clicks the confirmation link, the Supabase authentication engine verifies the session and redirects the traveler to the primary dashboard. From there, the user can access the Account page to manage their profile. This interface allows the user to modify personal details and customize their avatar. To ensure full data sovereignty and to address the complex challenges of forgetting personal data and revoking consent under modern regulations (such as GDPR), the Account page also provides a definitive option for account deletion, which signs the user out and removes all associated personal data from the system [6].

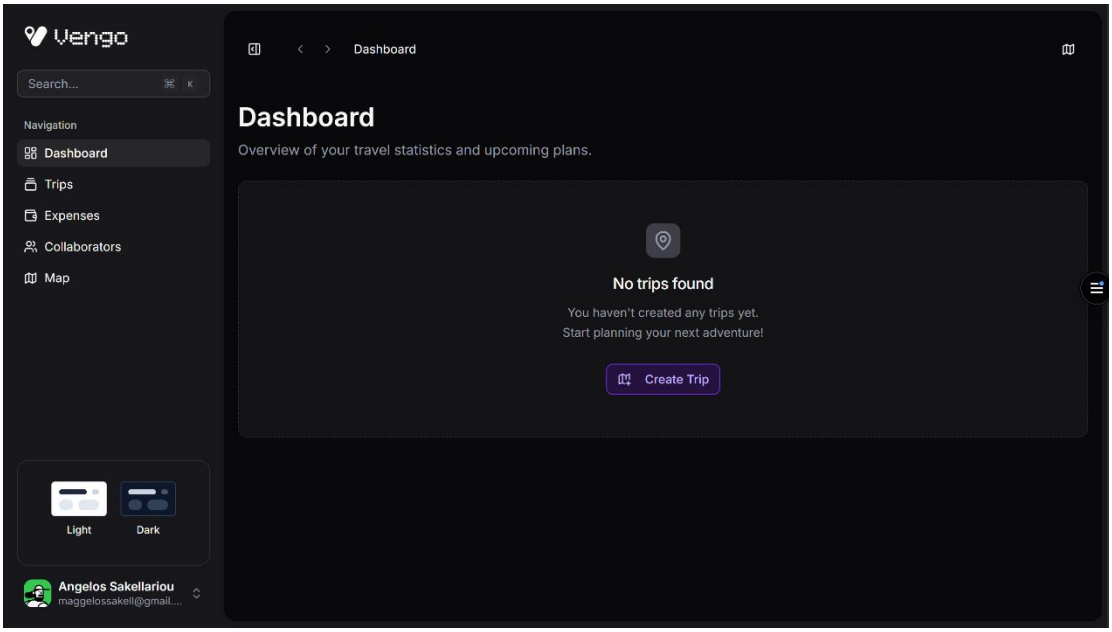

Figure 11. Initial empty dashboard state after authentication.

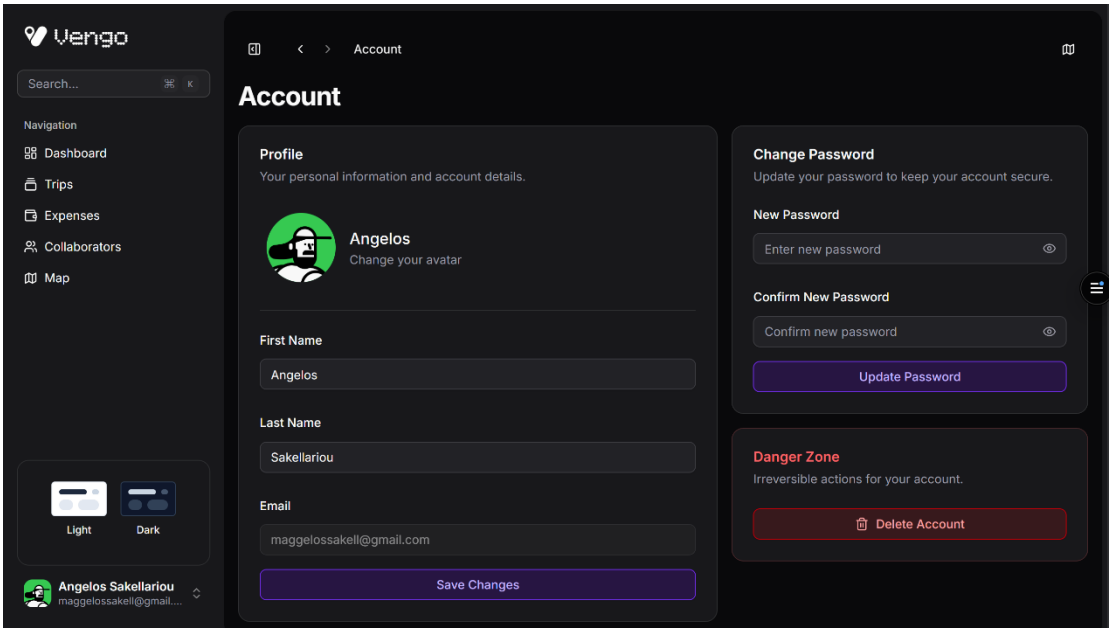

Figure 12. Account settings for profile and data management.

9.2 Trip Lifecycle Management

The primary workflow for a trip owner encompasses the entire lifecycle of a travel project, from its initial conception to its eventual removal. This process begins in the Trips module, where the user provides a title, a destination, and a specific date range. Upon creation, the system immediately initializes the trip workspace and redirects the owner to the overview. The owner can then personalize the project by uploading a custom cover image, which is stored in a private storage bucket and displayed as the primary visual anchor for the trip.

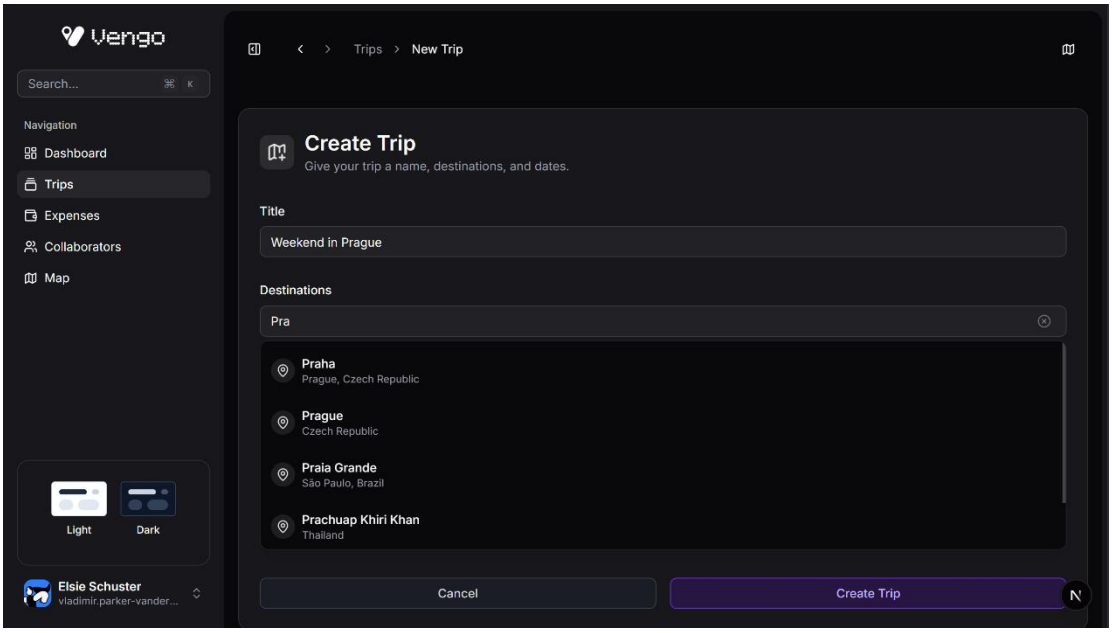

Figure 13. Inputting trip title and destination through Mapbox search.

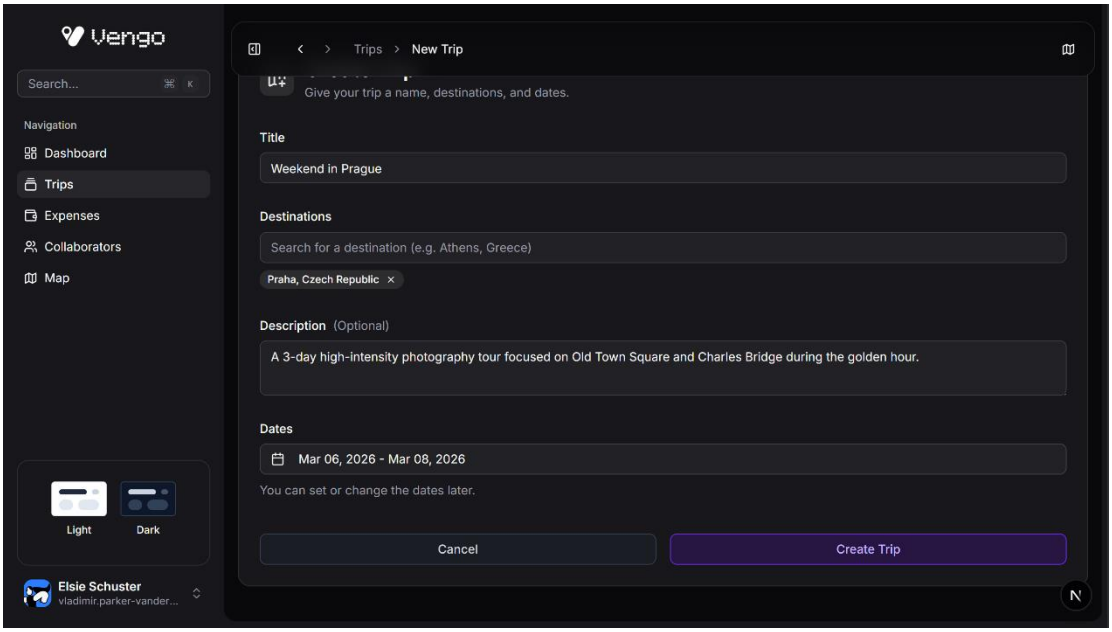

Figure 14. Setting trip descriptions and date ranges.

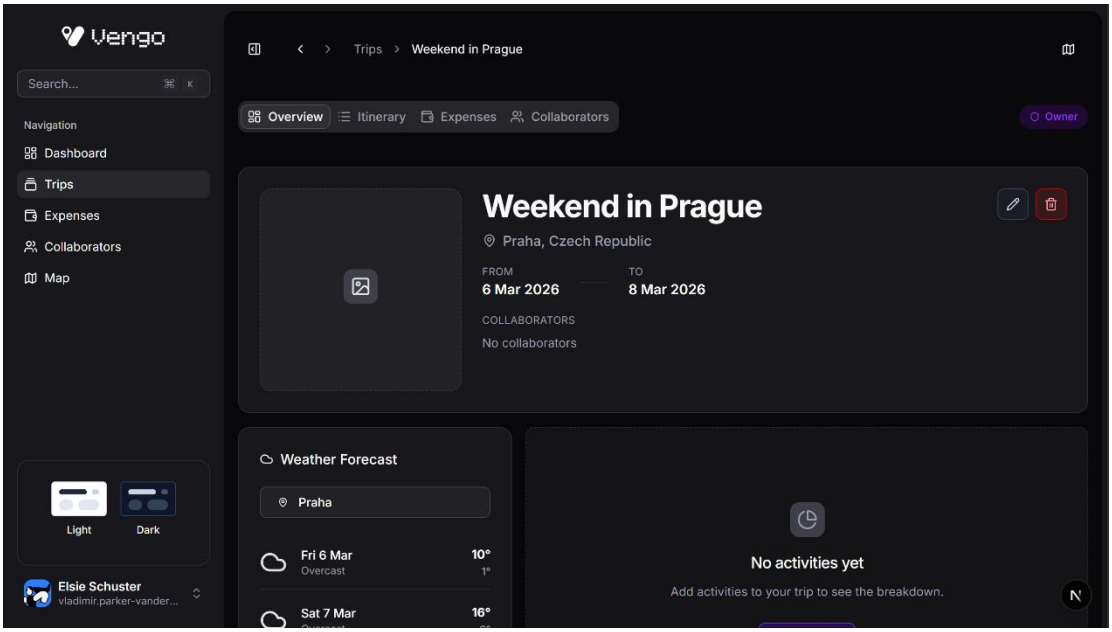

Figure 15. Newly initialized trip workspace overview after successful creation.

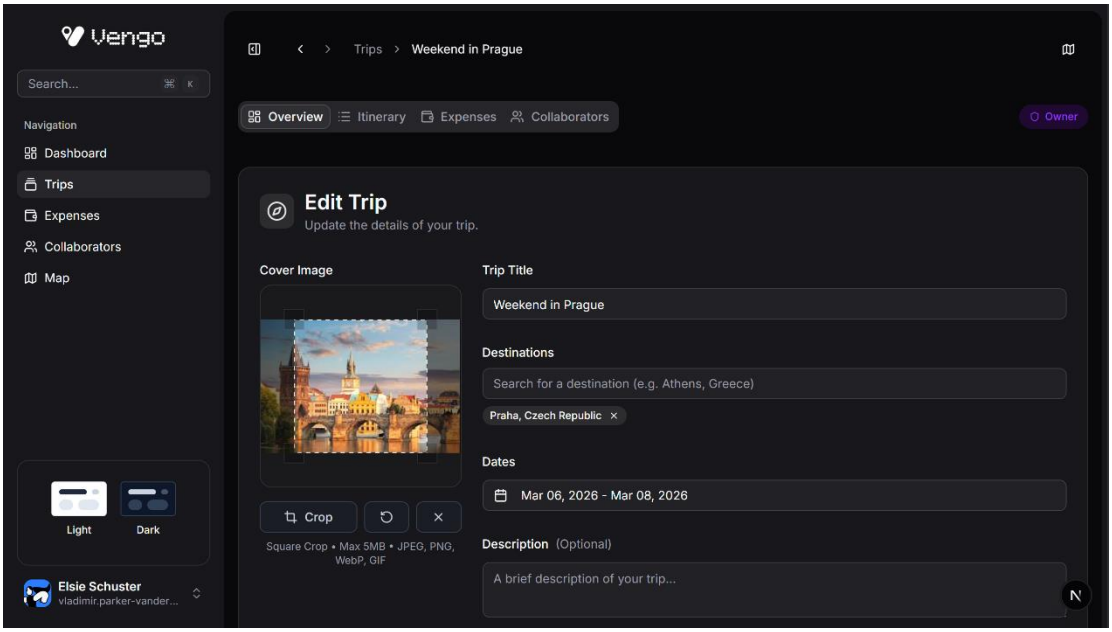

Figure 16. Uploading and cropping a custom cover photo.

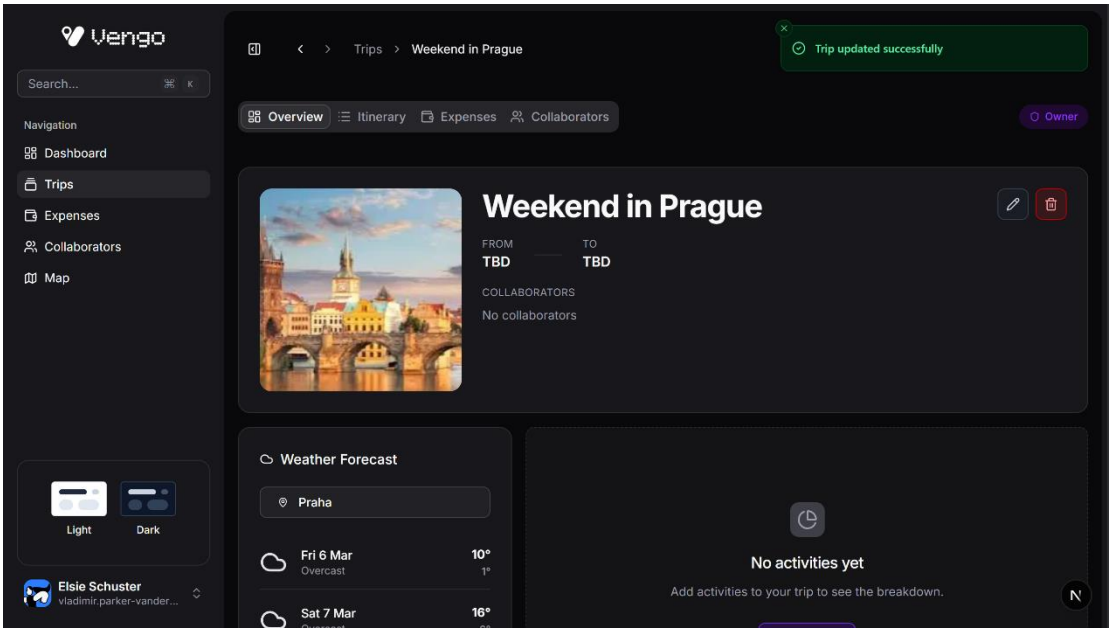

Figure 17. Successful insertion of the new image.

One of the most reactive elements in this lifecycle is the automated weather synchronization. When an owner modifies the trip dates, the system triggers a background update to fetch the relevant meteorological data for the new period. The trip overview and itinerary headers update dynamically to reflect these changes, providing the traveler with an accurate forecast or historical average. The lifecycle concludes with the deletion of the trip, a privilege reserved exclusively for the owner. This action triggers a cascading deletion of all associated days, POIs, activities, votes, and images to ensure the database remains clean.

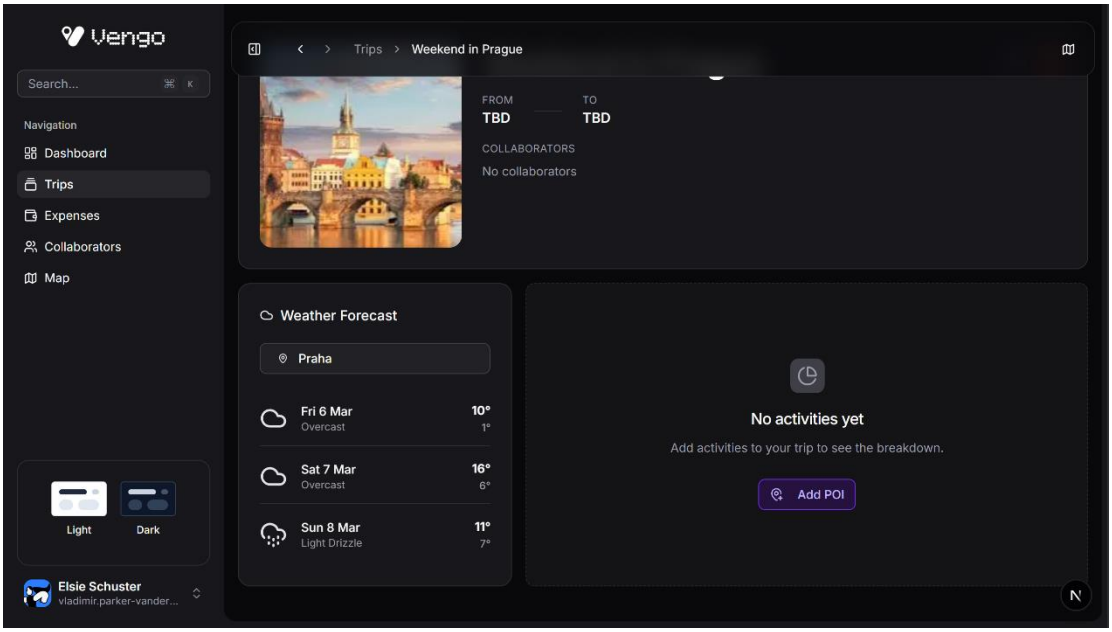

Figure 18. Automated weather predictions for trip dates.

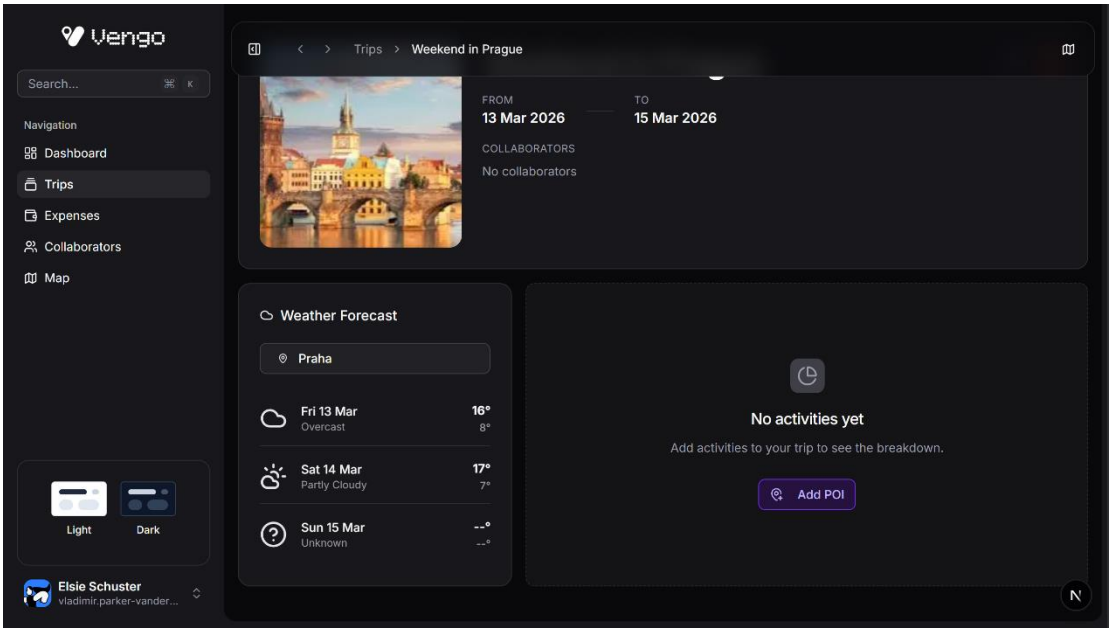

Figure 19. Weather synchronization after date modifications.

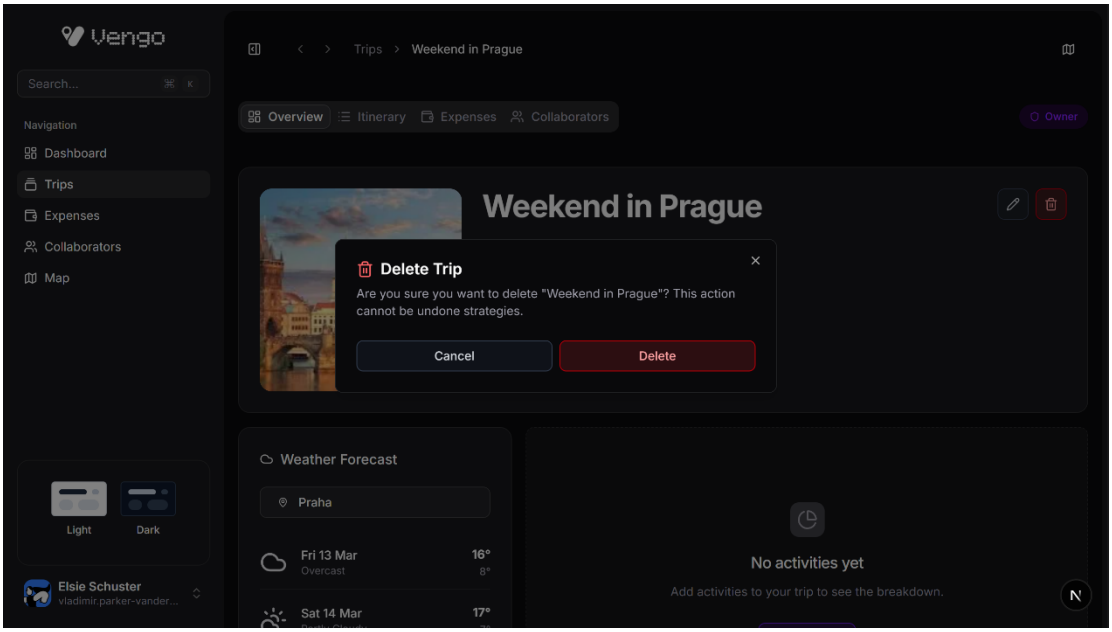

Figure 20. Cascading deletion confirmation for trip owners.

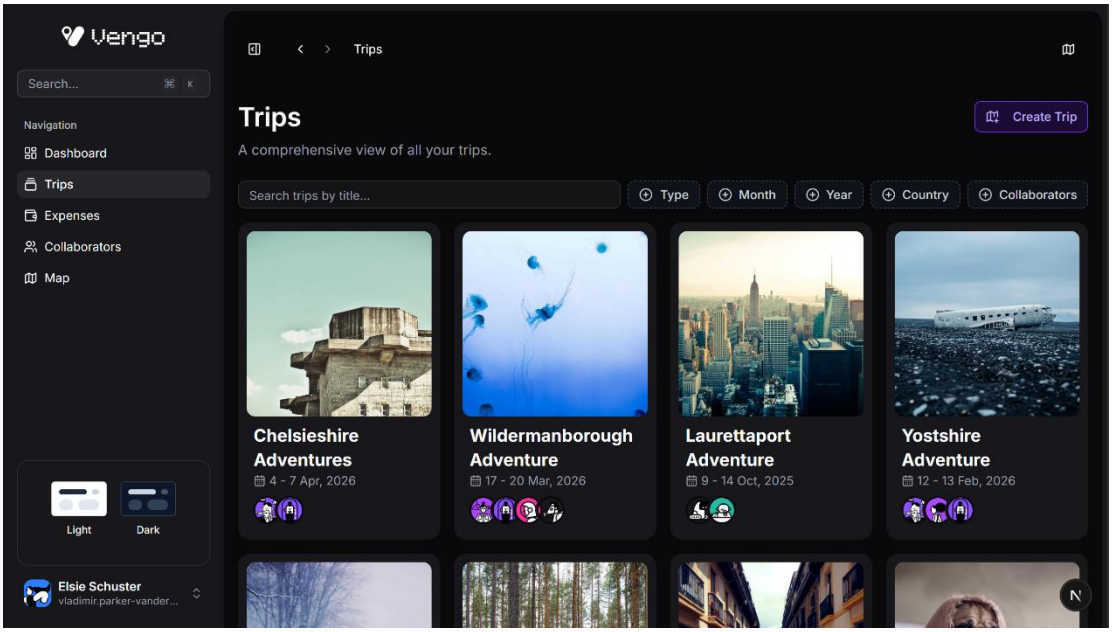

Figure 21. Trip management grid following project removal.

9.3 Itinerary and Geospatial Planning

The itinerary workspace is the functional heart of the application, where users organize their daily activities. Travelers can toggle between standard views and a specialized row layout to optimize the interface for dense schedules. Planning typically involves creating POIs using the integrated search form within the add POI form. As an alternative, a more user-friendly workflow, travelers can also search for locations directly through the map sidebar. By clicking the add to trip button in the location popup, the system automatically populates the creation form with the selected point's geospatial data. Once a location is identified and saved to the trip library, it can be assigned as an activity to a specific day.

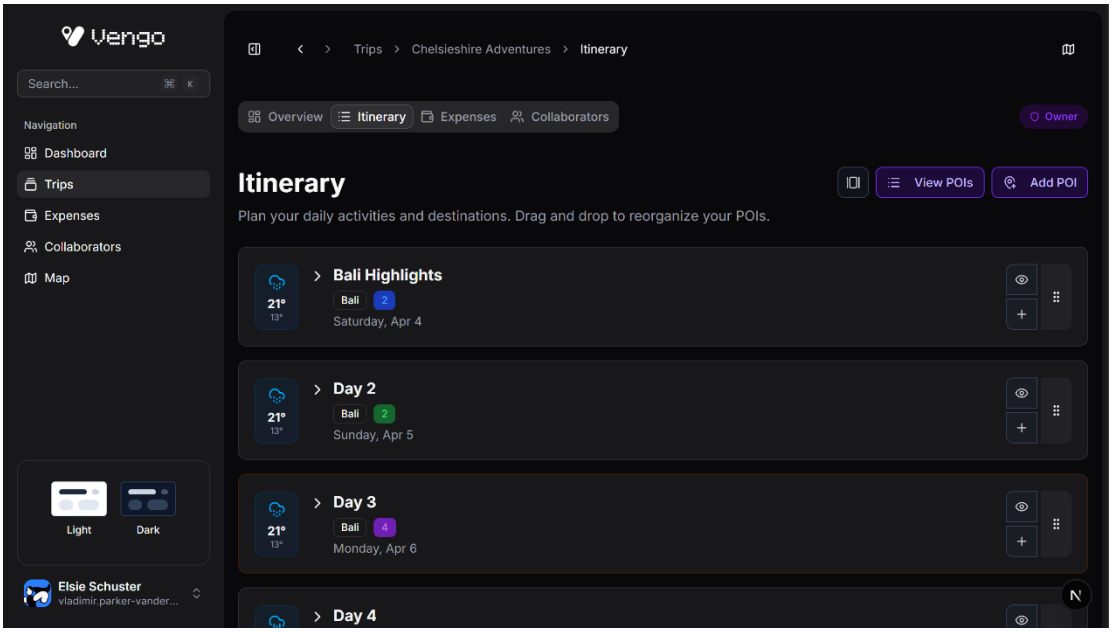

Figure 22. Itinerary page in column view.

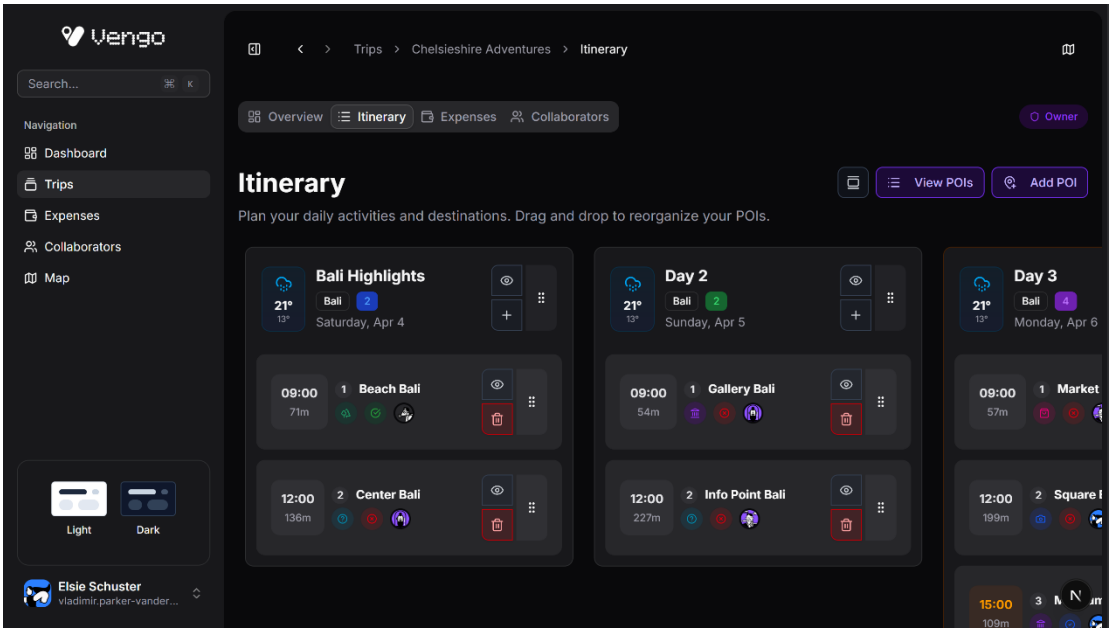

Figure 23. Itinerary page in row view.

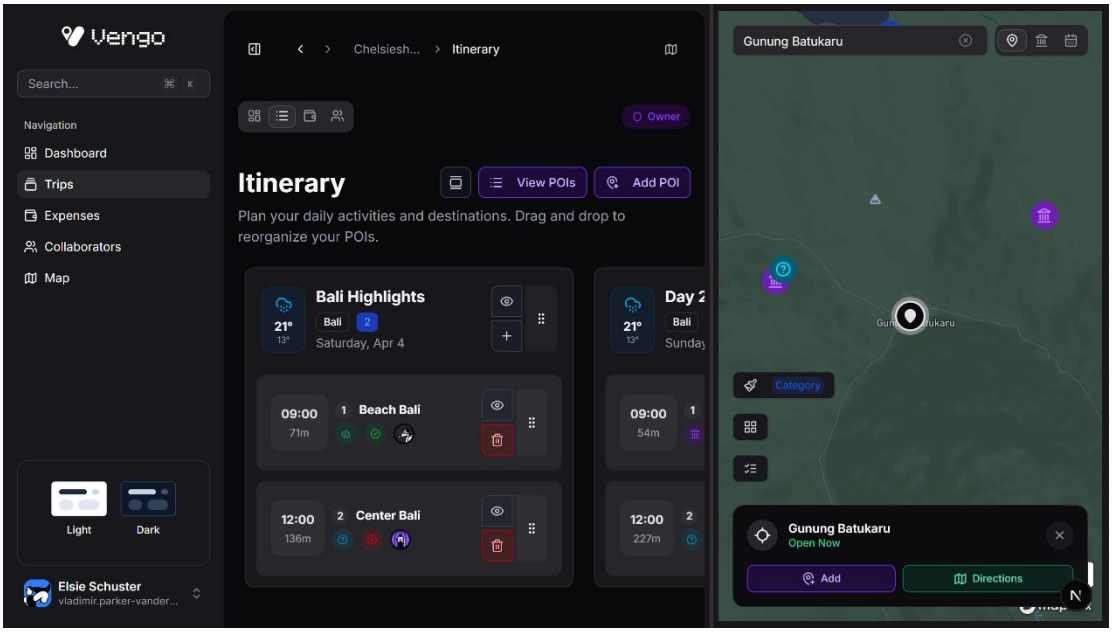

Figure 24. Identifying global locations via the map sidebar.

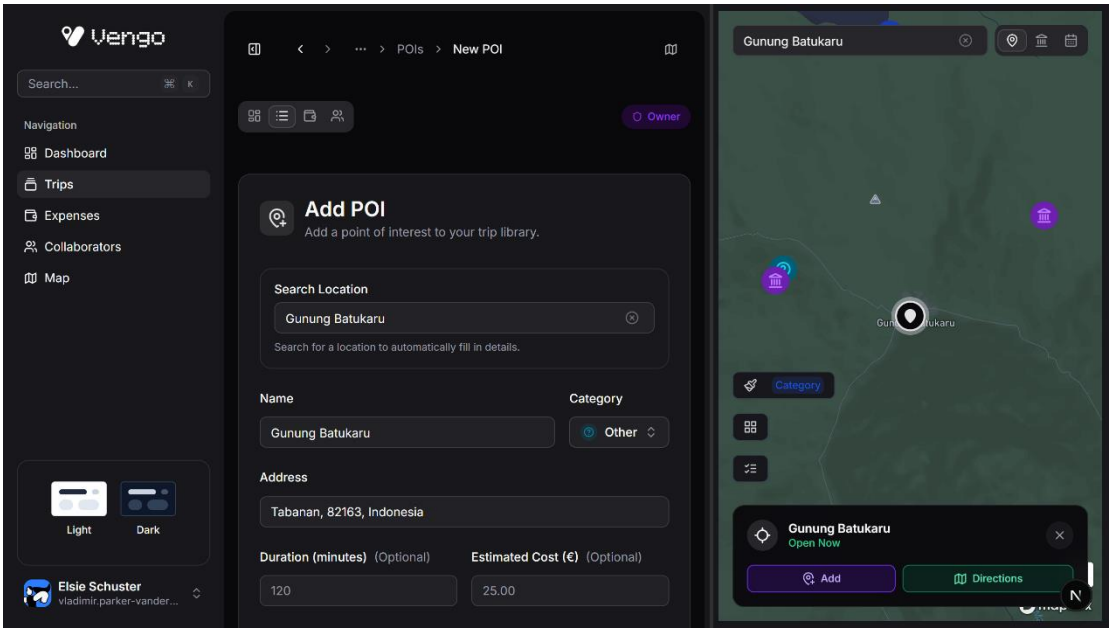

Figure 25. Pre-populating POI forms with geospatial data.

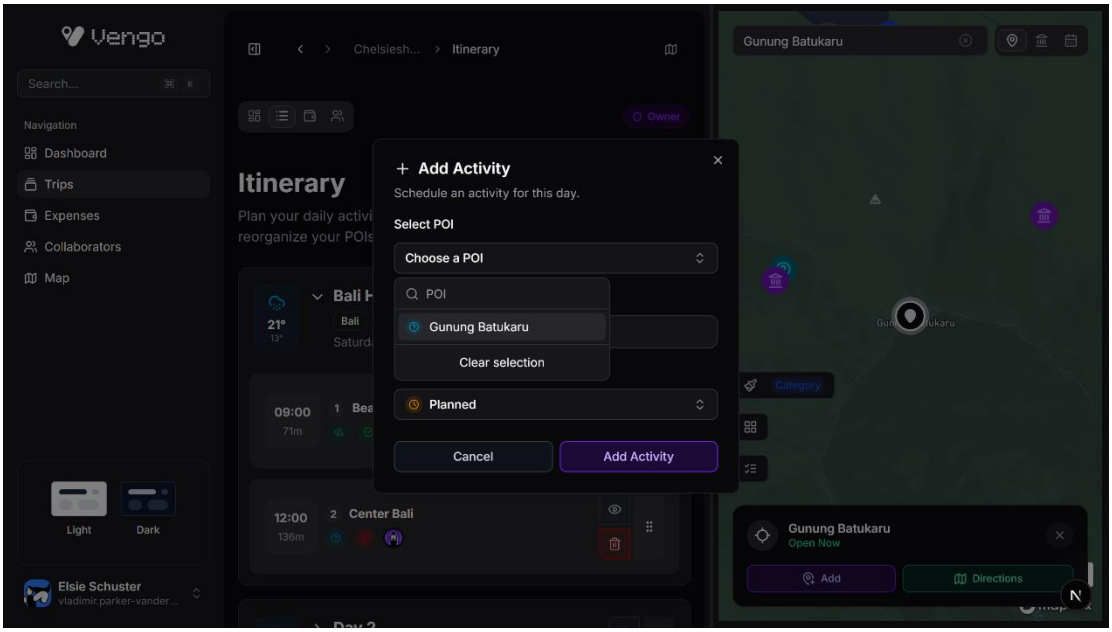

Figure 26. Creation of new activity with the newly created POI.

The system enables high-precision scheduling by letting users set or modify the scheduled time for each activity. When a POI is added as an itinerary item, its duration is automatically retrieved and saved from the POI metadata to ensure the timeline reflects realistic time requirements for each stop, recognizing the critical role of time-related factors in itinerary planning systems [14]. To help maintain schedule integrity, the application provides immediate visual warnings for potential timing discrepancies. Specifically, an activity card is highlighted with a yellow border if its duration overlaps with another scheduled item, and with a red border if the activities are out of chronological order. To accommodate the fluid nature of travel planning, the itinerary supports a comprehensive drag-and-drop interface. This allows users to reorder activities within a single day, move them between days, or even reorder the itinerary days themselves to shift the entire schedule. Furthermore, users can delete activities from the itinerary at any time to prune their plans or correct mistakes. This interaction uses the Optimistic UI pattern to provide instant visual feedback while the server persists the changes in the background. If the user is a Viewer, the system automatically hides these modification controls, preserving the integrity of the project.

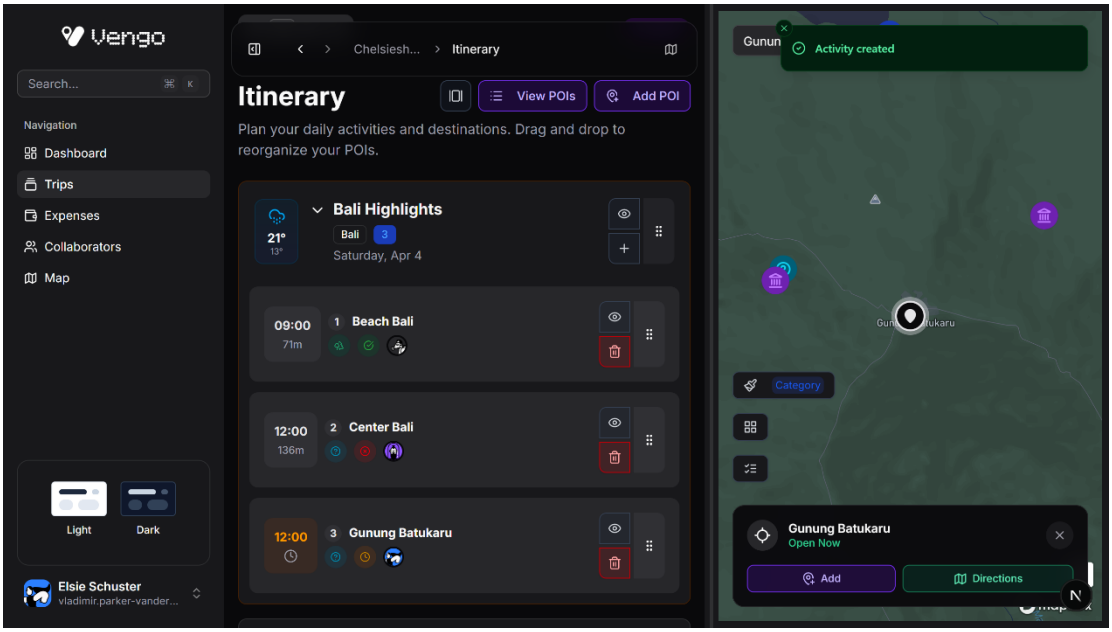

Figure 27. Successfully scheduled activity with visual alert for scheduled time.

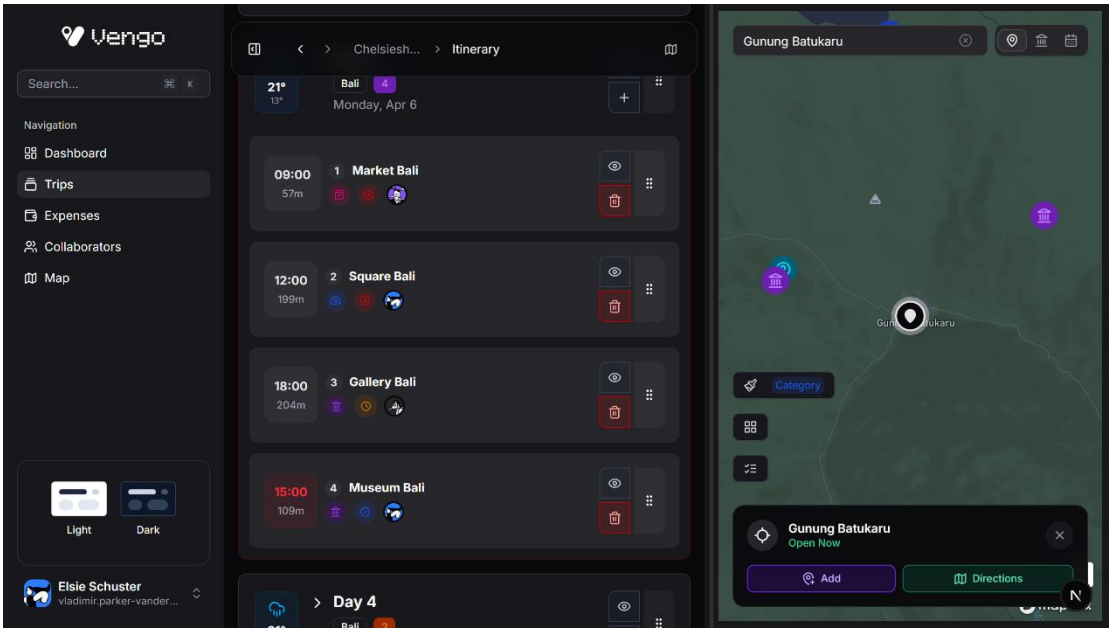

Figure 28. Visual alerts for activity timing conflicts.

9.4 Collaboration and Role-Based Access

The collaborative journey is managed through a token-based invitation system that ensures secure onboarding for both new and existing team members. An owner or editor initiates this flow via the Collaborators tab by generating an invitation assigned to a permanent role, such as Editor or Viewer, that determines the invitee's future privileges. When the invitee accesses the unique link, the system's Global Invite Listener immediately validates the token and the user's current relationship with the trip. If the invitee is already a member, a Sonner notification confirms their status and automatically redirects them to the specific trip dashboard. This automated check prevents redundant steps and ensures that active collaborators are returned to the workspace without unnecessary friction.

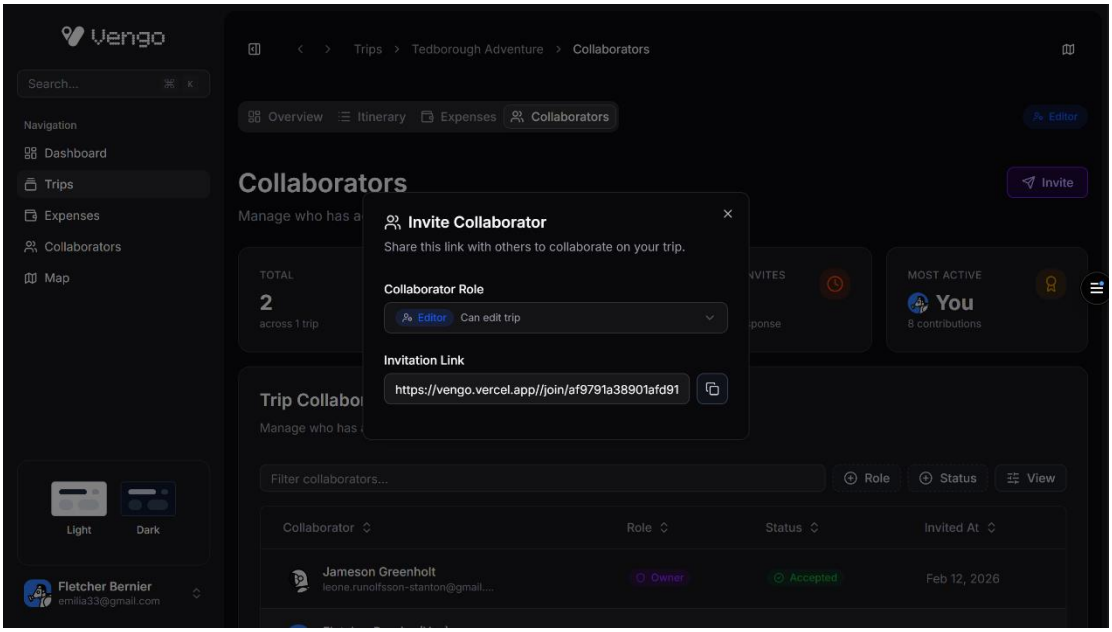

Figure 29. Generating role-based invitation tokens.

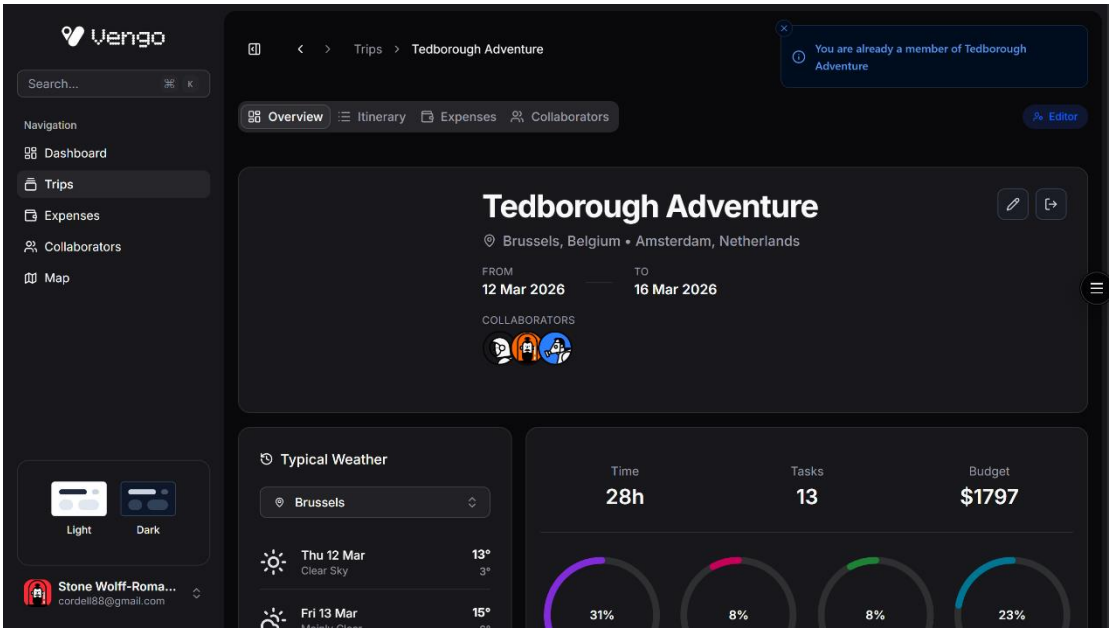

Figure 30. Alert verifying existing collaborator status.

For those not yet part of the trip, the interface intelligently adapts to the user's authentication state, using a responsive Dialog on desktop or a Drawer on mobile devices. Unauthenticated guests are presented with a streamlined workflow to sign in or create an account while preserving their invitation context, while authenticated users receive a clear prompt to accept or decline the join request. Once the invitation is accepted, the trip workspace dynamically adjusts its capabilities based on the assigned role: Editors gain full rights to contribute to the itinerary and manage expenses, whereas Viewers are restricted to a read-only perspective. This ensures a secure, tailored environment where every participant can follow the journey without the risk of accidental data modification.

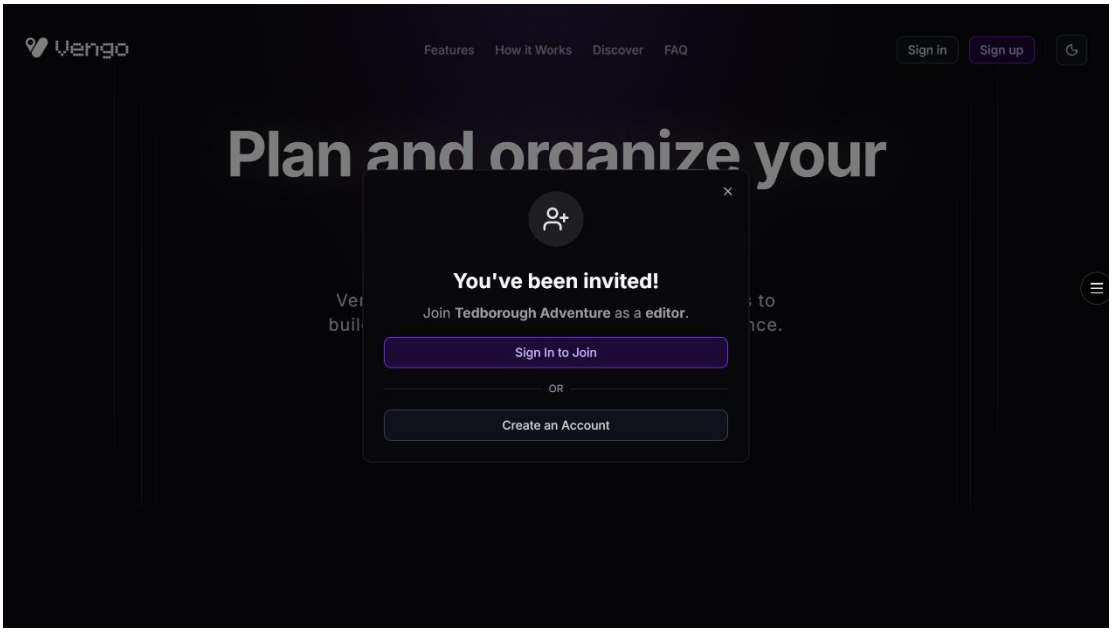

Figure 31. Invitation workflow for unauthenticated users.

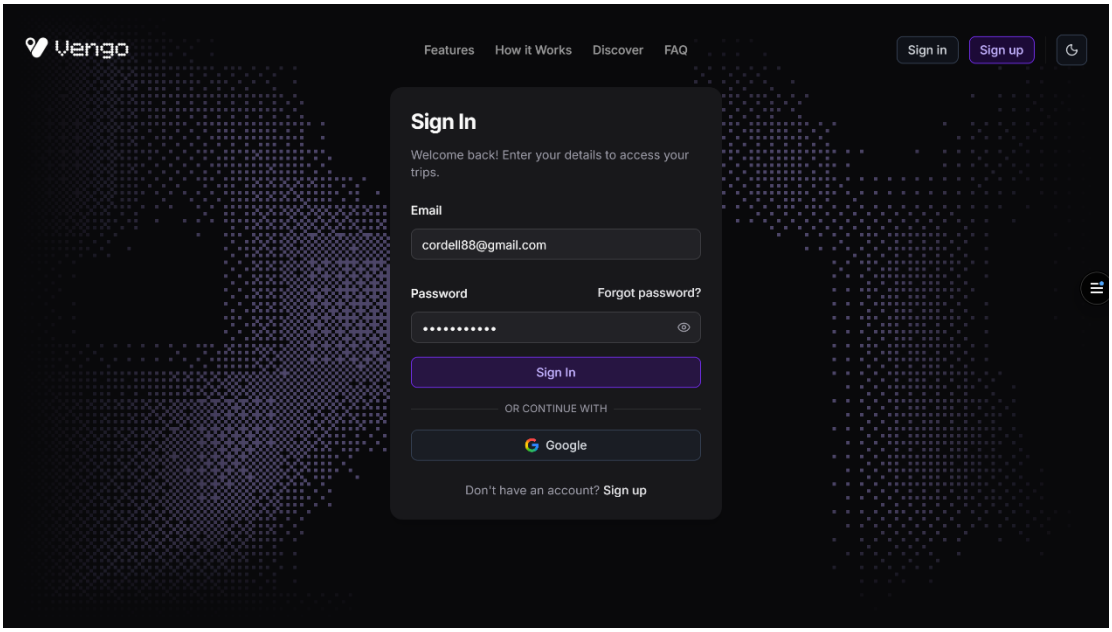

Figure 32. Sign-in page for invitees to be able to join the trip.

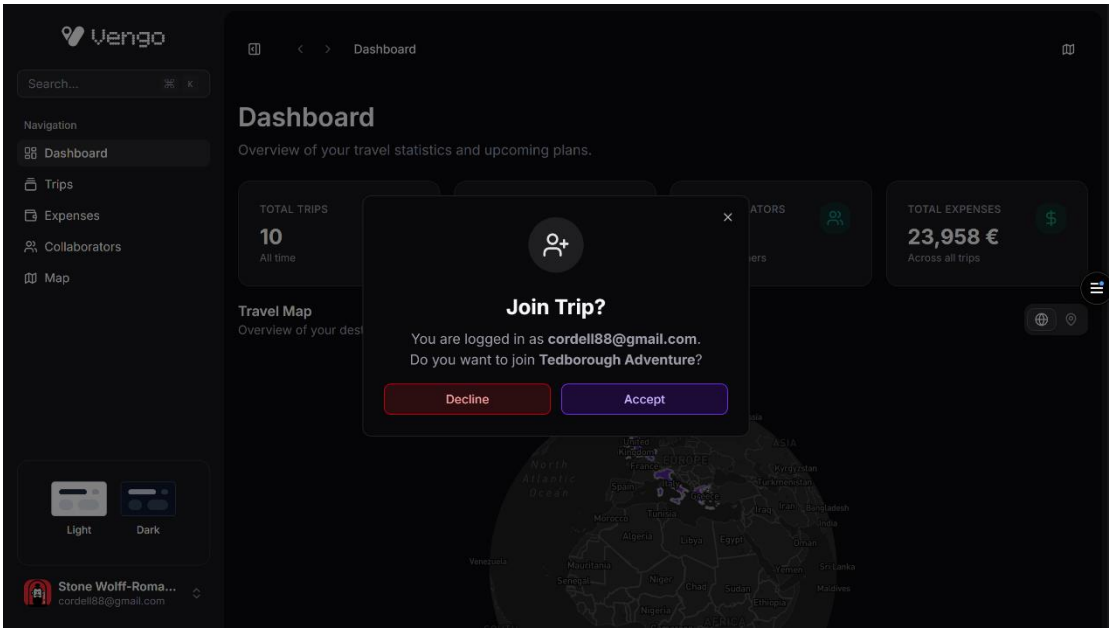

Figure 33. Invitation acceptance and join request portal.

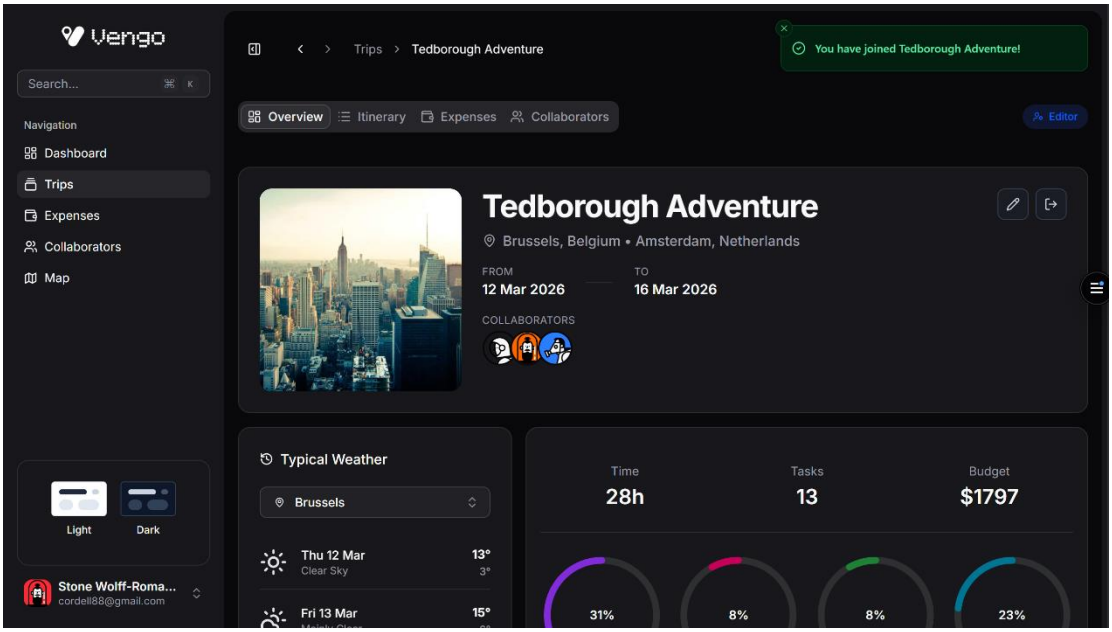

Figure 34. Trip overview page access after joining.

The automated logic for this invitation and onboarding sequence, including authentication checks and role assignment, is modeled in the UML Activity Diagram shown in Diagram 16.

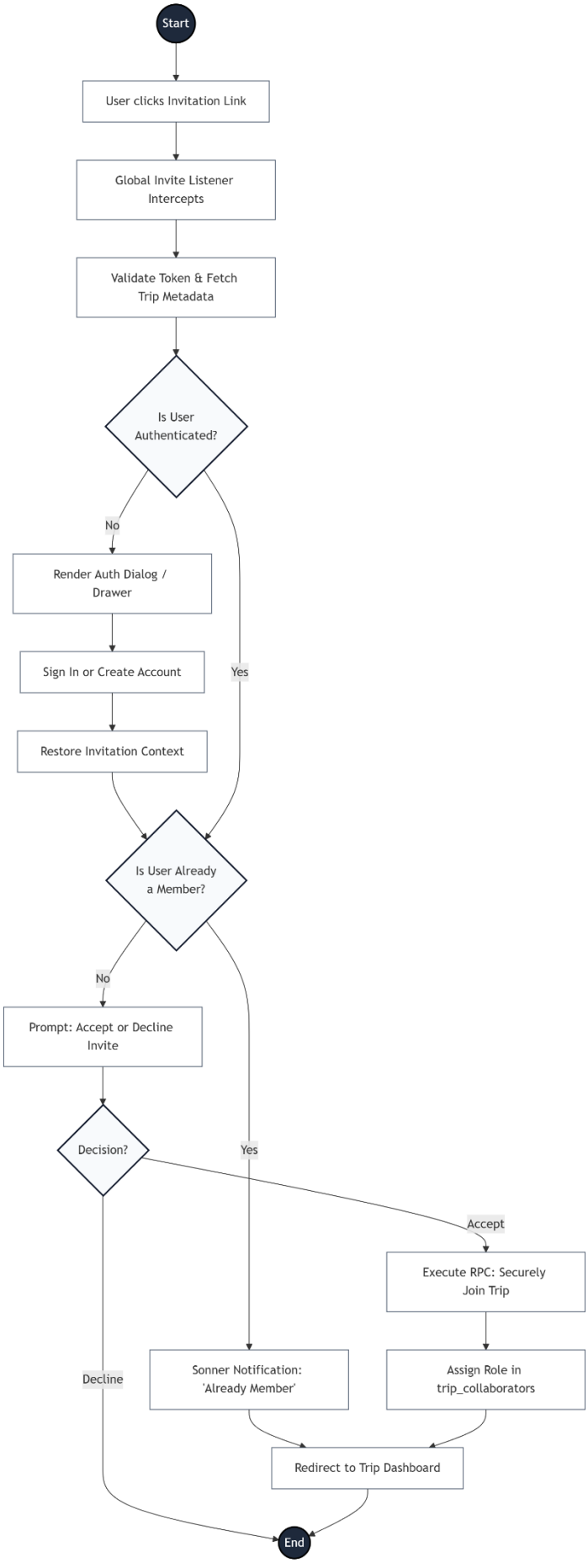
Start
User clicks Invitation Link
Global Invite Listener Intercepts
Validate Token & Fetch Trip Metadata
Is User Authenticated?
No
Yes
Render Auth Dialog / Drawer
Sign In or Create Account
Restore Invitation Context
Is User Already a Member?
No
Yes
Prompt: Accept or Decline Invite
Decision?
Accept
Decline
Execute RPC: Securely Join Trip
Sonner Notification: 'Already Member'
Assign Role in trip_collaborators
Redirect to Trip Dashboard
End

Diagram 16. Activity Diagram of the Role-Based Onboarding and Invitation Workflow.

9.5 Real-Time Synchronous Planning

Once users have successfully joined a trip workspace with Editor privileges, the application environment shifts to support synchronous multi-user collaboration. This feature transforms the platform from a static, asynchronous repository into a live collaborative workspace.

The functionality of this real-time engine is entirely transparent to the user. Travelers no longer need to coordinate via external chat applications or manually refresh their browsers to see the latest itinerary updates. For instance, when one editor reorders the schedule using the drag-and-drop interface or adds a new Point of Interest from the map sidebar, those changes appear instantly on the screens of all other connected collaborators.

To maintain contextual awareness during these live planning sessions, the interface uses non-blocking toast notifications. When the itinerary updates in the background, a subtle alert informs active users that another collaborator has made changes. This design keeps travelers up to date without interrupting their immediate workflow or forcing them to reset their current view. Ultimately, this live synchronization drastically reduces the cognitive burden associated with group coordination, fulfilling the application's core objective of serving as a dynamic, unified "Single Source of Truth".

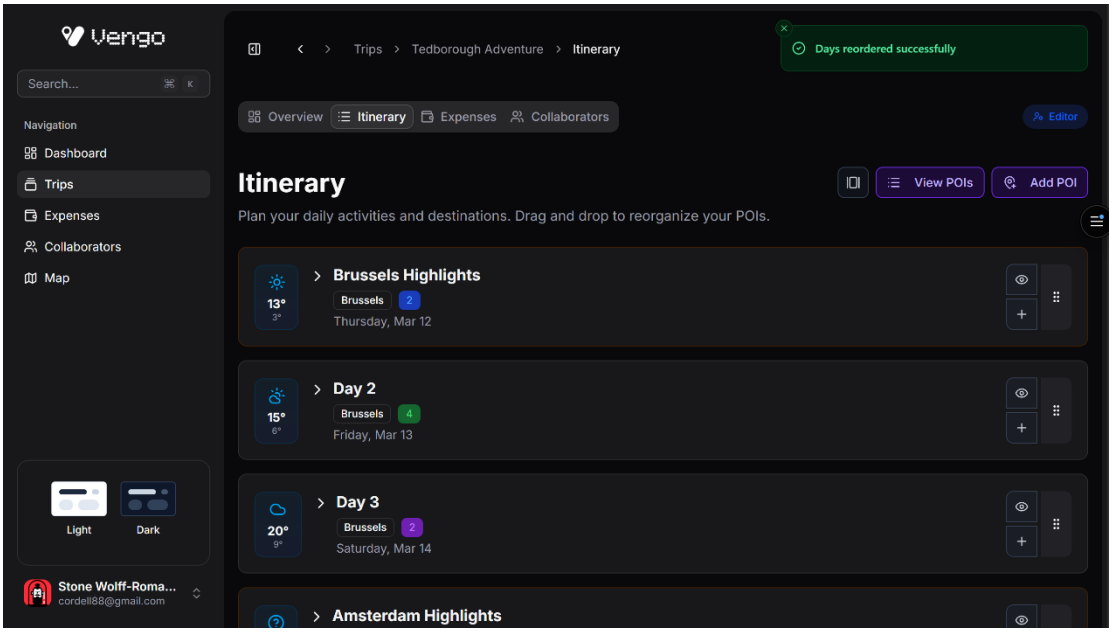

Figure 35. Instant local feedback during itinerary mutations.

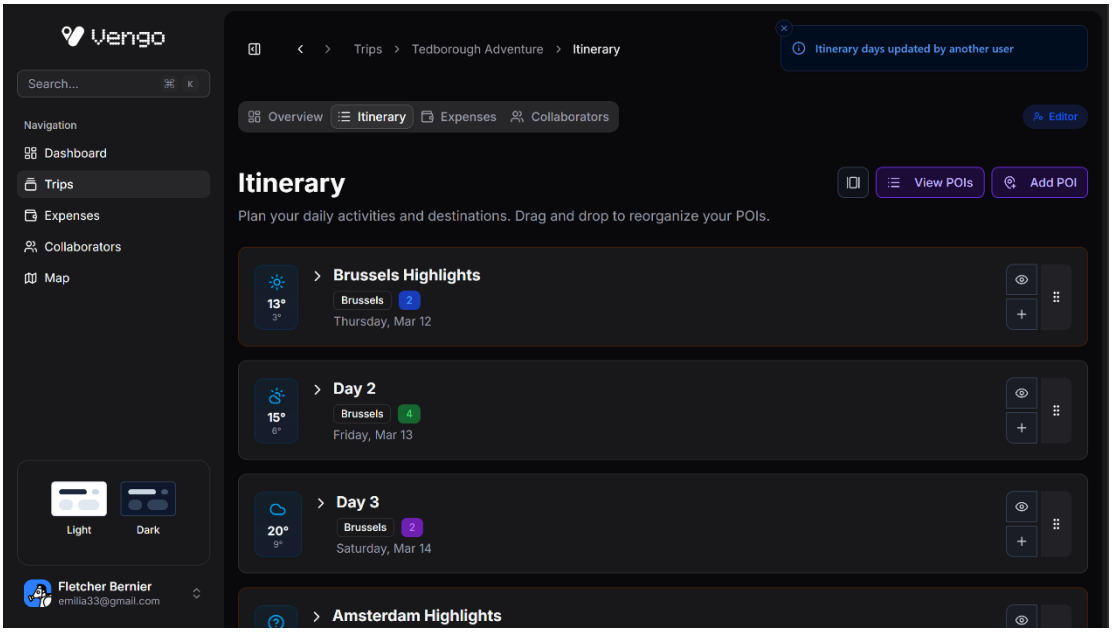

Figure 36. Live alert for external collaborator updates.

10. Testing Strategy and Quality Assurance

To maintain high reliability in a complex application with server actions, data validation, and intricate UI states, this project employs a dual-layer testing strategy. This approach separates concerns between fast internal logic verification and realistic full-stack user-journey validation. The primary objective is to balance development velocity with cross-layer confidence.

The testing architecture uses Jest to provide a rapid safety net for internal correctness, focusing on unit and integration-like scenarios isolated from the browser [42]. Complementing this, Playwright is used for End-to-End (E2E) testing to validate the application against a running Next.js environment, ensuring that routing, authentication, and database side effects function correctly in a real browser engine (Chromium) [39].

10.1 Unit and Integration Testing (Jest)

10.1.1 Purpose and Scope

The Jest testing suite, located in the __tests__/ directory, is designed to validate logic independently of browser timing. Given the project's reliance on the Next.js App Router, server actions, and Zod schemas, it is critical to verify business logic and data contracts before they interact with the UI. These tests serve several specific purposes:

1. **Logic Verification**: They safeguard business logic in actions, services, and utilities, ensuring deterministic outputs for given inputs.
2. **Data Integrity:** They validate data mappers and edge cases, ensuring strict contract adherence.
3. **Component Isolation:** They assess component behavior under mocked runtime dependencies, preventing regressions in rendering states via snapshot-based coverage.

10.1.2 Operational Architecture

The testing environment is bootstrapped with next/jest, which sets up the required Next.js context [66]. The system uses a "setup file" strategy to simulate the runtime environment. Specifically, jest.setup.js loads environment variables (.env.local) and installs mocks for backend dependencies such as next/cache and headers. Simultaneously, jest.setup.component.js handles frontend mocks, including next/navigation, next/link, and toast notifications.

This architecture keeps tests fast and deterministic by mocking heavy framework dependencies. By default, the configuration prioritizes speed, but it supports opt-in integration safety checks, such as database connectivity tests, controlled via the RUN_DB_CONNECTIVITY_CHECK environment variable.

To quantify the effectiveness of the testing suite, the infrastructure supports code coverage generation via the V8 provider. Running pnpm test --coverage produces detailed metrics on statement and branch execution.

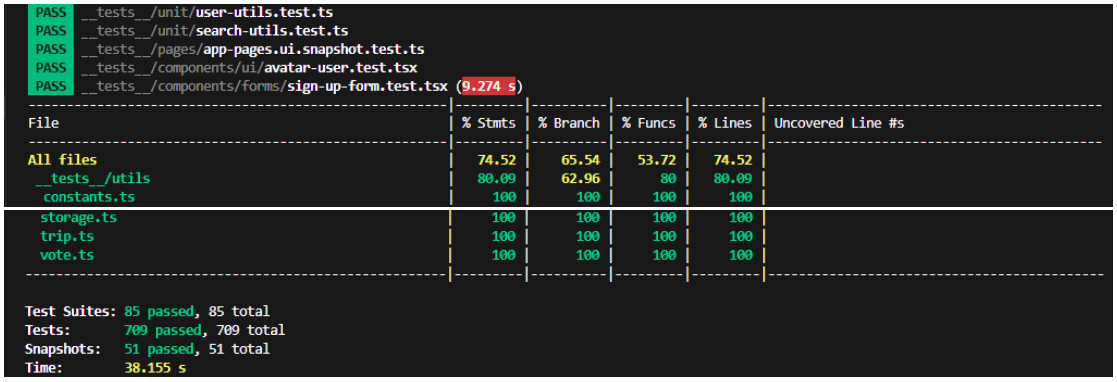

Figure 37. Jest Code Coverage Report for statement and branch metrics.

This reporting mechanism identifies gaps in logic verification, ensuring that critical business rules, particularly in the utils and actions directories, are not left untested.

10.1.3 Implementation and Best Practices

The codebase enforces strict file-level isolation to prevent state leakage between tests. A "focused assertion" approach is preferred over timing-sensitive expectations, as it promotes stability. The directory structure mirrors the application's architecture, with dedicated suites for actions, components, hooks, and utilities, as shown in the table.

Table 3. Jest Testing Suite Directory Structure and Coverage Scope

Directory(__tests__/)	Scope & Coverage	Example Targets
actions/	**Server Mutations:** Validates authentication guards, permission checks, and database write logic for Server Actions.	activity.test.ts, auth.test.ts, trip.test.ts
components/	**UI Interactions:** Tests rendering states, event handlers, and prop validation in React components.	data-tables/, forms/, map/
context/	**Global State:** Verifies the logic in React Context providers.	trip-context.test.tsx

data/	**Read Operations:** Ensures data access layers execute correctly and return expected DTOs.	activities.test.ts, trips.test.ts, weather.test.ts
hooks/	**Client Logic:** Tests custom hooks for side effects and state management.	use-database-action.test.tsx, use-debounce.test.tsx
pages/	**Route Rendering:** Snapshot testing of top-level pages to detect visual regressions.	__snapshots__/, app-pages.snapshot.test.ts
rls/	**Security Policies:** Integration tests for Row-Level Security logic.	profile-weather-rls.test.ts, trip-rls.test.ts
unit/	**Pure Logic:** Tests for mappers, validations, and utility functions with no external dependencies.	mappers/, validations/, utils/

10.2 End-to-End Testing (Playwright)

10.2.1 Rationale for E2E Testing

While Jest validates internal logic, it cannot account for cross-layer behaviors inherent in modern web applications. Playwright is used to address risks that unit tests miss, such as proxy redirects (a network-level boundary previously known as middleware in older Next.js versions), authenticated layouts, and client-side routing synchronization. These tests are essential for validating critical user journeys, including trip creation, RBAC, and complex asynchronous flows involving map interactions and database side effects.

10.2.2 Execution Environment

Playwright initializes the full application server (pnpm dev) using its webServer configuration [52], [67]. Tests run against this instance at http://localhost:3000. Unlike Jest's mocked environment, Playwright interacts with the application through a real browser engine, verifying that hydration, routing, and server-client integration work seamlessly.

To ensure stability during resource-intensive operations, the configuration is set to fullyParallel=false, and the worker count is limited to 1. This prevents race conditions in complex database transactions or state updates.

10.2.3 Authentication and State Management

A significant challenge in E2E testing is efficiently handling authentication. This project uses a "Global Setup" pattern in which a dedicated setup project performs the initial authentication and saves the session state to .auth/user.json. Subsequent test suites reuse this storage state, avoiding repeated logins.

For scenarios that require a logged-out state, such as sign-up flows or public invites, tests explicitly create a fresh browser context with an empty storage state. This approach ensures that tests do not inadvertently rely on shared session tokens, preventing false positives in route protection.

10.2.4 Critical Test Scenarios and Architecture

To maintain a clean, scalable test suite, the project uses the Page Object Model (POM) design pattern. Files like trips.po.ts encapsulate the dashboard and trip settings logic, while itinerary.po.ts handles the planning board and map's complex interactions. This abstraction lets the test specifications focus on business logic rather than DOM implementation details. The E2E suite uses these objects to validate four critical areas of the application

Authentication and Middleware Security

The auth.spec suite validates the application's security perimeter by testing the entry points of the user journey. It verifies the correct rendering of sign-in, sign-up, and forgot-password interfaces while ensuring that form validation logic correctly identifies invalid credentials. Beyond the UI, this suite performs critical checks on the application's middleware, asserting that unauthenticated access attempts to restricted routes like /trips or /dashboard are successfully intercepted and redirected to the login flow.

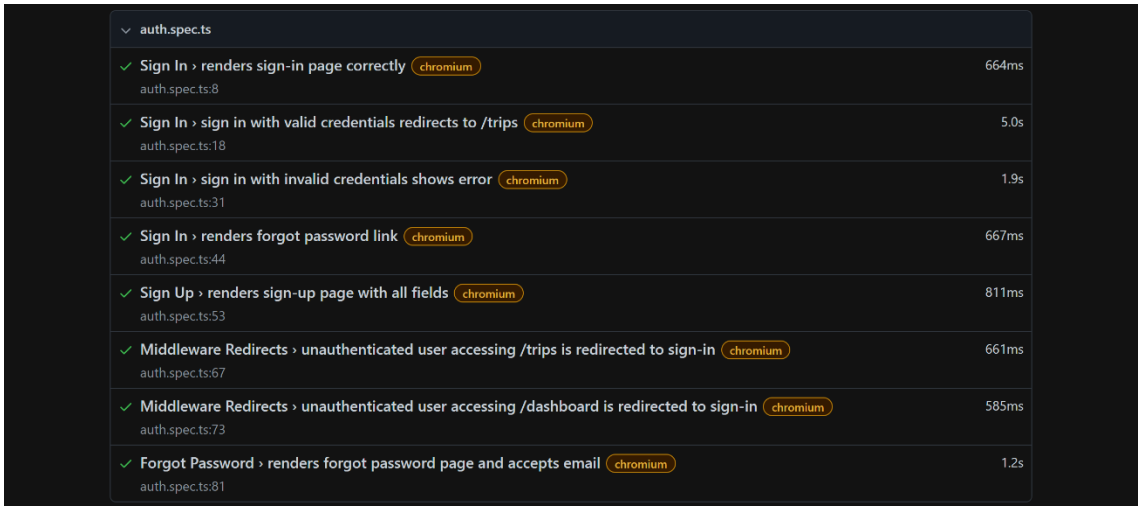

Figure 38. E2E validation of authentication flows and middleware redirects.

Account Life Cycle and Personalization

The account-management.spec file focuses on the integrity of user-specific data and state persistence. This suite automates updates to profile attributes, such as names and avatars, ensuring that changes persist across session reloads and propagate to the dashboard UI. Additionally, it handles the sensitive "Account Deletion" workflow. By using dynamically generated disposable credentials, the test creates a temporary user, performs a full deletion, and verifies that the system revokes all access and purges the associated records from the backend.

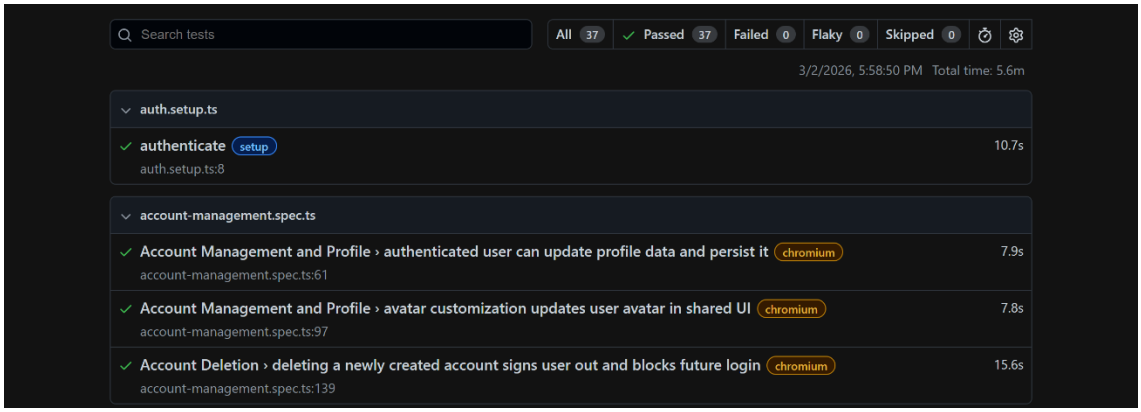

Figure 39. Profile personalization and account deletion lifecycle validation.

Trip Lifecycle and Data Integrity

The trip.spec.ts suite validates the persistence and integrity of the trip entity by combining UI actions with direct Supabase client assertions. The tests automate the full flow of creating a trip, uploading and cropping a cover image, and adding multiple destinations. A key part of the suite is verifying that when a second destination is added, the weather service correctly generates forecast rows for both locations. The suite concludes with a critical safety test: deleting a parent trip that contains child entities (POIs, activities) to verify that the cascading deletion logic functions correctly in the database, ensuring no orphaned data remains.

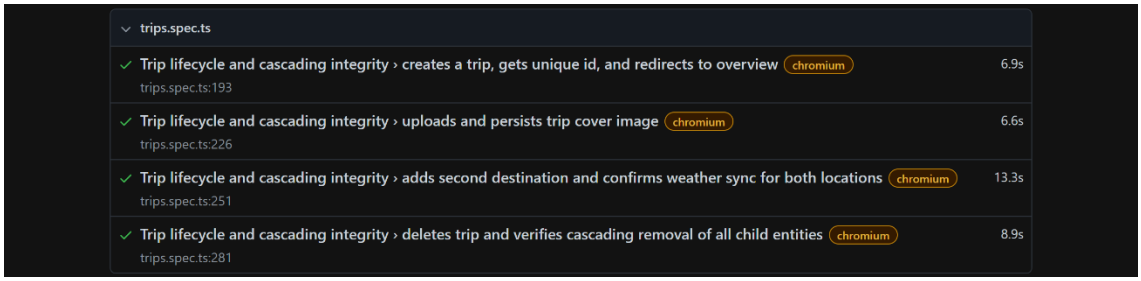

Figure 40. E2E validation of the trip lifecycle.

Itinerary Workflow and Map Integration

The itinerary.spec.ts suite focuses on the planning board's interactive elements. It mocks Mapbox API responses to test the sidebar popup's functionality, verifying that users can search for locations and add them directly to the trip via the map interface. The suite also validates the Kanban board's drag-and-drop logic, ensuring that moving an activity updates the UI state correctly. Specifically, it checks for visual feedback cues, such as red borders on the day column, that indicate "out-of-order" timing conflicts when an activity is placed in a slot that contradicts its scheduled time.

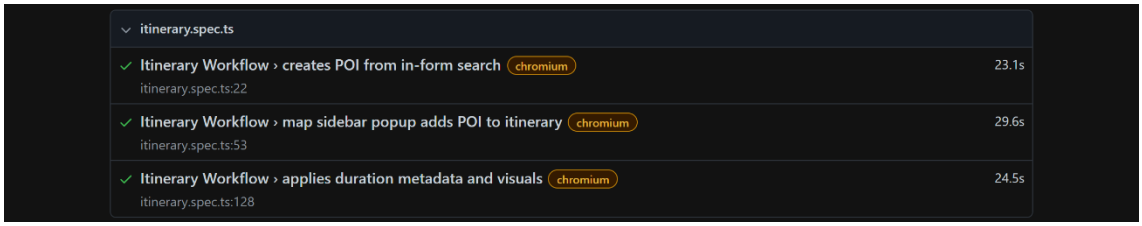

Figure 41. E2E validation of the itinerary workflow.

Collaborative Invitation Logic

The invite.spec suite validates the application's most complex multi-user workflow through a serial, multi-context execution pattern. It orchestrates a flow in which a trip owner generates a secure invitation link, which an invitee then accesses in a separate, isolated browser context. The test validates the entire "Join" lifecycle, from the unauthenticated invitation listener to the post-login acceptance dialog. Finally, the suite bridges the UI and the database layer by using the Supabase admin client to verify that the invitee was correctly inserted into the trip_collaborators table with the expected role.

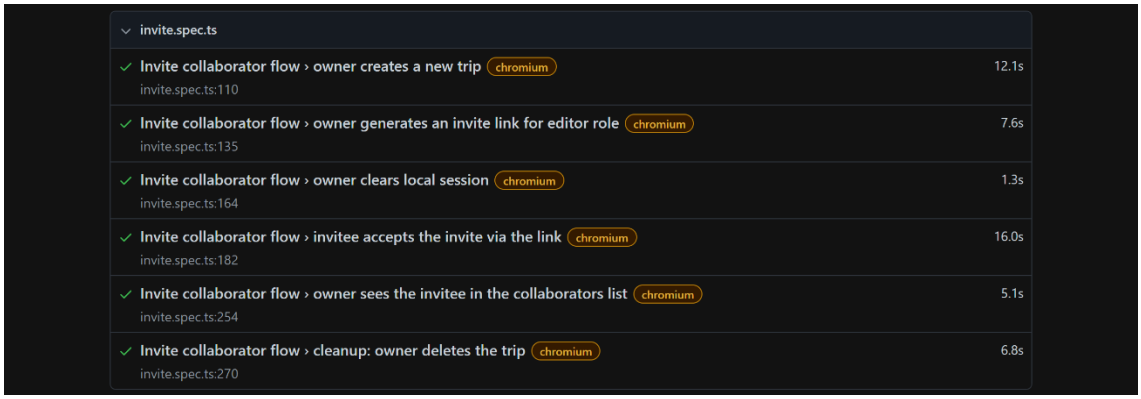

Figure 42. E2E validation for the collaborative invitation workflow.

Role-Based Access Control (RBAC)

The permissions.spec.ts suite enforces security by iterating through Owner, Editor, and Viewer roles and asserting strict UI visibility rules. For the Owner, the test verifies full access, including the ability to delete the trip and manage collaborators. For the Editor, it confirms that while they can modify the itinerary and invite users, they are explicitly blocked from deleting the trip or removing other collaborators. Finally, for the Viewer, the test asserts a read-only state in which all mutation controls, such as edit buttons, drag handles, and delete options, are hidden from the DOM.

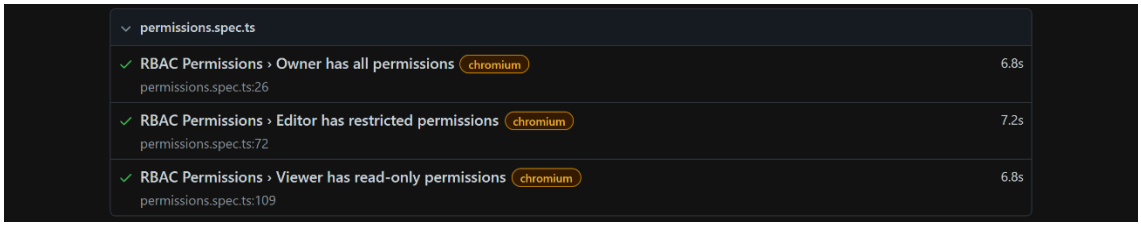

Figure 43. E2E validation of role-based access control and UI visibility rules.

Navigation, Routing, and Error Resilience

The navigation.spec.ts suite ensures that client-side routing keeps the browser URL and active UI tabs synchronized, with deep-linking tests verifying that direct navigation to specific itinerary days or tab URLs loads the correct context without losing state. This is complemented by the route-status.spec suite, which enforces routing resilience through "negative testing." By intentionally navigating to non-existent top-level paths or valid routes with missing database entities, such as non-existent Trip, Day, or POI IDs, the suite confirms that the application returns 404 status codes and renders designated "Not Found" error boundaries. This prevents the frontend from entering an inconsistent state and provides a robust fallback for the user during invalid navigation.

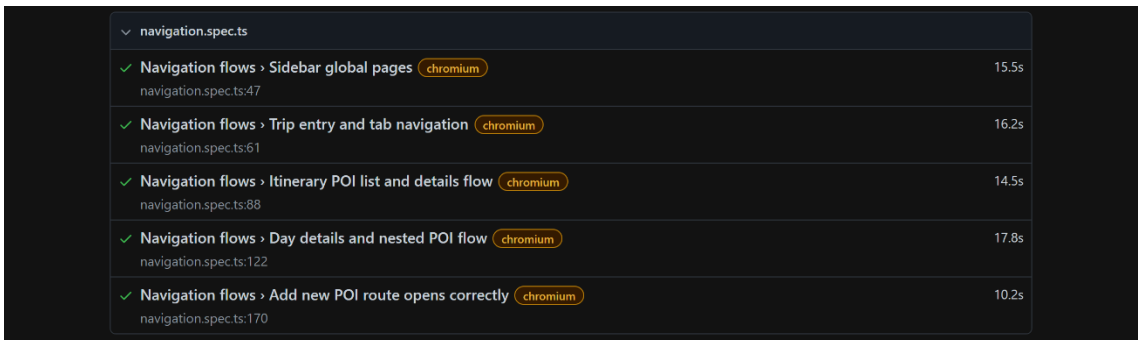

Figure 44. E2E validation for navigation synchronization and route error resilience.

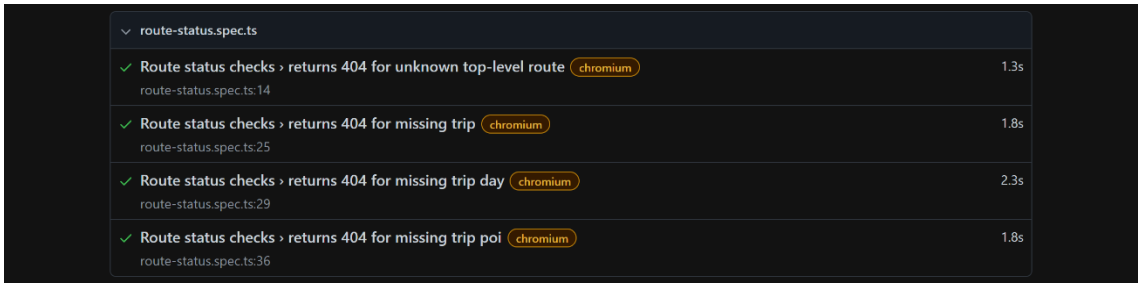

Figure 45. Routing integrity: deep-link synchronization and 404 error handling.

10.3 Testing Best Practices and Stability

To maintain a stable and reliable continuous integration pipeline, the project adheres to strict testing guidelines that apply to both the unit and end-to-end layers.

In the End-to-End suite, resilient locators take priority. Tests rely on data-testid attributes and ARIA roles rather than brittle text or CSS selectors, ensuring validation remains robust even if the application's visual styling changes significantly. Each test is also designed to be deterministic. Playwright handles rigorous cleanup of any created entities, such as test trips and user permissions, to prevent state leakage between suites.

On the unit testing side, framework dependencies are consistently mocked in Jest to prevent flaky tests caused by internal Next.js implementation details. By isolating the logic from the framework's internal routing and caching mechanisms, Jest focuses solely on the correctness of the code. Full integration checks, which require the actual runtime environment, are reserved exclusively for the Playwright suite.

10.4 Comparative Methodology and Conclusion

The decision to use Jest or Playwright is guided by the specific layer of the application under test. Jest is the tool of choice for logic-level correctness, handling scenarios where functions transform data, validate schemas, or render components based on props, enabling rapid iteration and immediate development feedback. In contrast, Playwright is reserved for full user journeys that involve navigating between pages, persisting data to Supabase for retrieval on different screens, or verifying authorization gates. While these tests are slower to execute, they provide the highest level of confidence in the system's integrity from the end-user's perspective.

Table 4. Comparative Analysis of Testing Methodologies and Execution Contexts

Testing Layer	Tool	Primary Focus	Execution Context
Unit/Component	Jest	Logic correctness, Rendering states, Utility functions	Node.js / JSDOM
Integration (Server)	Jest	Data mapping, Server Action validation, Zod schemas	Node.js (Mocked DB)
End-to-End	Playwright	Full User Journeys, RBAC, Routing, Proxy logic	Chromium Browser

Maintaining this clear dichotomy between unit and end-to-end testing ensures a comprehensive and robust testing strategy. Jest ensures the application's building blocks are internally sound, while Playwright confirms that these blocks assemble into a cohesive, functional user experience.

11. Conclusions and Future Work

11.1 Summary of Work

The primary objective of this thesis was to address the widespread fragmentation and cognitive burden of modern group travel planning. By developing a comprehensive, web-based Collaborative Travel Planning Application, this project created a unified "Single Source of Truth" for travelers. The application bridged temporal scheduling and geospatial visualization by integrating a dynamic, drag-and-drop itinerary board with a context-aware mapping engine powered by Mapbox.

By implementing a modern, serverless architecture with Next.js and the Supabase BaaS ecosystem, the system achieved high performance and robust data integrity. End-to-end type safety was enforced with TypeScript and Zod, supporting a strict defensive programming strategy. Furthermore, the application modeled complex group dynamics by establishing a granular RBAC system (Owner, Editor, Viewer), enabling real-time multi-user synchronization and collaborative decision-making.

11.2 Critical Assessment

Reflecting on the development lifecycle, the project achieved several significant successes despite notable technical challenges.

Successes:

- **Robust Security Architecture:** The implementation of PostgreSQL RLS proved to be a highly effective Zero Trust paradigm. By shifting authorization logic directly to the database engine, the application ensured that data isolation and user privacy were strictly maintained, independent of the client-side implementation.
- **Seamless Geospatial Integration:** The context-aware map implementation successfully transitioned the map from a static viewing tool to a dynamic planning workspace. The hybrid rendering strategy and native GeoJSON clustering enabled efficient handling of large datasets without sacrificing the 60 frames-per-second interaction required for a premium user experience.
- **Perceived Performance:** The strategic use of Optimistic UI patterns effectively masked network latency. By delivering instant visual feedback and handling server synchronizations in the background, the application achieved a native-like, fluid feel.

Challenges:

- **Complexity of Server Actions and State Management:** Transitioning to the Next.js App Router introduced steep learning curves for orchestrating React Server Components and Server Actions. Ensuring that the server cache was correctly invalidated and revalidated without causing hydration errors required meticulous data flow management.
- **Geospatial State Synchronization:** Managing state between the React component tree and the external WebGL context of the Mapbox canvas was technically demanding. Ensuring that map bounds, zoom levels, and active markers accurately reflected the routing context without triggering infinite re-render loops required complex hierarchical state coordination.

11.3 Future Work

While the current application provides a robust foundation for collaborative travel planning, several avenues for future research and technical development have been identified to enhance the platform's capabilities:

11.3.1 Enhanced User Experience and Collaboration

Future iterations of the platform should prioritize reducing the initial learning curve and expanding real-time collaboration. This could be achieved by introducing an interactive onboarding flow paired with an AI-driven chatbot that helps users navigate the interface and provides contextual sightseeing recommendations. Furthermore, while the current system supports asynchronous POI voting, incorporating a real-time in-trip chat module, an event-driven notification engine, and integrated document management for packing checklists would centralize all logistical planning within the workspace. Finally, transitioning from a strictly private utility to a platform that supports social discovery via opt-in public profiles could foster community engagement and drive organic user growth.

11.3.2 Advanced Geospatial and Technical Capabilities

To elevate the logistical planning experience within the broader context of smart tourism [2], the geospatial engine can be expanded beyond static points of interest. By drawing on established research in the design and development of web-based transit information systems [12]. Integrating routing APIs would enable the system to calculate optimal paths, provide precise time estimates, and recommend appropriate transit methods between scheduled activities. Furthermore, the platform could automatically generate highly personalized, day-by-day schedules by incorporating advanced user modeling techniques [4] alongside expert systems designed for bi-objective, multi-period itinerary generation [10]. This capability would be further enhanced by integrating Large Language Models (LLMs) with recent advances in cooperative coevolution algorithms for large-scale planning [11]. Additionally, given the reality of travel, where network connectivity is often unreliable, migrating the system to a Progressive Web App (PWA) with a local-first database architecture is necessary. Exploring a migration to a React Native mobile application would also grant deeper access to device-level APIs, such as native GPS for real-time navigation.

11.3.3 Infrastructure and Scalability

As the platform scales, the underlying infrastructure must evolve to meet enterprise standards. Upgrading the deployment lifecycle with rigorous Continuous Integration and Continuous Deployment (CI/CD) pipelines, including automated end-to-end testing, quality gates, and staging environments, will significantly improve software reliability and developer velocity. Simultaneously, expanding user acquisition will require architecting dynamic, server-rendered public landing pages optimized for search engines (SEO), ensuring the platform remains highly visible and competitive in the smart tourism market.

Glossary

A

- **Adaptive Design**: A system that ensures the UI is responsive and performant across diverse device sizes, such as transforming desktop modals into mobile drawers.
- **Agile Development:** An iterative software development methodology focused on continuous refinement through cycles of design, implementation, and testing.
- **App Router:** The Next.js router introduced in version 13, built on top of React Server Components. It uses file-system based routing and supports layouts, nested routing, loading states, error handling, and more.
- **Asynchronous:** A non-blocking execution pattern where the system initiates multiple requests in parallel without halting the initial page render.

B

- **Backend-as-a-Service (BaaS):** A cloud computing model in which developers offload infrastructure-heavy tasks, such as authentication and database hosting, to third-party providers like Supabase.

C

- **Client Component:** A React component that runs in the browser. In Next.js, Client Components can also be rendered on the server during initial page generation. They can use state, effects, event handlers, and browser APIs, and are marked with the "use client" directive at the top of a file.

D

- **Deno:** A secure runtime for JavaScript and TypeScript that executes serverless Edge Functions.

E

- **Edge Computing:** A paradigm that executes code at the network's "edge" node closest to the user to minimize latency and improve performance.

F

- **Framer Motion:** A hardware accelerated React library for managing fluid UI animations and interactive gestures.
- **Full Stack:**

G

- **Geospatial Analysis:** The process of collecting and visualizing location-based data to support travel logistics.
- **GitHub:** A cloud-based Git repository hosting service for version control and source code management. In this project, it is integrated with Vercel to fully automate the continuous build and deployment pipelines.

H

- **Hydration:** React’s process of attaching event handlers to the DOM to make server-rendered static HTML interactive. During hydration, React reconciles the server-rendered markup with the client-side JavaScript.

J

- **JWT-based Authentication:** A stateless security mechanism that uses JSON Web Tokens to verify a user's identity. In this application, Supabase manages these tokens and passes them to the PostgreSQL database to enforce RLS policies.

L

- **Layout:** UI that is shared between multiple pages. Layouts preserve state, remain interactive, and do not re-render on navigation. Defined by exporting a React component from a layout.js file.
- **Last-Write-Wins (LWW):** A conflict resolution strategy for distributed systems that resolves simultaneous updates by accepting the update with the most recent timestamp. It was chosen to ensure a fluid user experience by minimizing the processing overhead of complex merge algorithms.
- **Lightweight CQRS:** An architectural pattern that strictly separates data mutation (Writes) from data fetching (Reads). In this project, it is implemented by routing read operations through React Server Components and write operations through Next.js Server Actions.

M

- **Mapbox GL JS:** A high-performance library for rendering interactive vector maps and managing geospatial data.
- **Node.js:** An open-source, cross-platform JavaScript runtime environment that executes code outside of a web browser. In this project, it is used to execute Server Actions and server-side rendering.

N

- **Next.js:** A full-stack React framework that supports server-side rendering and the App Router architecture.
- **NoSQL:** A class of database management systems that do not use the traditional relational tabular model.

O

- **Open Source:** Software with source code that is made freely available and may be redistributed and modified, such as the Supabase platform and the Open-Meteo API used in this project.
- **Optimistic UI:** A development pattern in which the interface updates instantly after a user action, assuming success while the network request completes in the background.

P

- **Page:** UI that is unique to a route. Defined by exporting a React component from a page.js file within the app directory.
- **PostgreSQL:** An open-source relational database management system that serves as Supabase's core data engine.

R

- **React Server Components (RSC):** A technology that enables server-side data fetching without sending additional JavaScript to the client, improving initial load times.
- **Row-Level Security (RLS):** A database-level security policy that restricts which rows a user can access or modify based on the user's identity.

S

- **Schema-First Development:** A methodology in which data contracts and validation schemas (e.g., Zod) are defined before writing any user interface code.
- **Server Component:** The default component type in the App Router. Server Components render on the server, can fetch data directly, and don't add to the client JavaScript bundle. They cannot use state or browser APIs.
- **Server Action:** A Server Function that is passed to a Client Component as a prop or bound to a form action. Server Actions are commonly used for form submissions and data mutations.
- **Supabase:** An open-source platform offering relational database, authentication, real-time, and storage services.

- **Supabase CLI:** A command-line tool for developing Supabase projects locally, managing database migrations, and deploying infrastructure changes via code.
- **Suspense boundary:** A React <Suspense> component that wraps async content and displays fallback UI while it loads. In Next.js, Suspense boundaries define where the static shell ends and streaming begins, enabling Partial Prerendering.
- **Synchronous:** An execution pattern that ensures the user interface and database state remain in immediate alignment during real-time interactions.

T

- **Tailwind CSS:** A utility-first CSS framework for building highly responsive and custom user interfaces.
- **TanStack Table:** A headless UI library for building highly customizable and state-driven data grids and tables in React applications.
- **Turbopack:** A fast, Rust-based bundler built for Next.js. Turbopack is the default bundler for next dev and available for next build. It provides significantly faster compilation times compared to Webpack.
- **TypeScript**: A strongly typed superset of JavaScript that provides static typing to prevent runtime errors.

V

- **Vercel Edge Network:** A global Content Delivery Network (CDN) and serverless execution environment. It optimizes application performance by caching static assets and rendering Server Components on the edge node closest to the user, thereby minimizing network latency.

Z

- **Zero Trust Architecture:** A security model that assumes every request is a potential threat and requires verification at every layer—specifically via RLS at the database level in this project.
- **Zod:** A TypeScript-first schema declaration and validation library for enforcing data integrity at application boundaries.

Abbreviations

- **API:** Application Programming Interface
- **BaaS:** Backend-as-a-Service
- **CLI:** Command Line Interface
- **CQRS:** Command Query Responsibility Segregation
- **CSCW:** Computer-Supported Cooperative Work
- **CTA:** Call-to-Action
- **CRUD:** Create, Read, Update, Delete
- **DIY:** Do It Yourself (referring to independent travel planning)
- **DOM:** Document Object Model
- **DTO:** Data Transfer Object
- **E2E:** End-to-End (Testing)
- **HTML:** HyperText Markup Language
- **HTTP:** HyperText Transfer Protocol
- **HTTPS:** Hypertext Transfer Protocol Secure
- **JSON:** JavaScript Object Notation
- **JWT:** JSON Web Token
- **MIME:** Multipurpose Internet Mail Extensions
- **OAuth:** Open Authorization

- **PKCE:** Proof Key for Code Exchange
- **POI:** Point of Interest
- **RBAC:** Role-Based Access Control
- **REST:** Representational State Transfer
- **RLS:** Row-Level Security
- **SEO:** Search Engine Optimization
- **SSL/TLS:** Secure Sockets Layer / Transport Layer Security
- **SSR:** Server-Side Rendering
- **UI:** User Interface

Bibliography

[1] A. Zigomitros, F. Casino, A. Solanas, and C. Patsakis, "A survey on privacy properties for data publishing of relational data," *Ieee Access*, vol. 8, pp. 51071–51099, 2020.

[2] A. Kontogianni, E. Alepis, M. Virvou, and C. Patsakis, "Conceptualizing Smart Tourism," in *Smart Tourism–The Impact of Artificial Intelligence and Blockchain*, A. Kontogianni, E. Alepis, M. Virvou, and C. Patsakis, Eds., Cham: Springer Nature Switzerland, 2024, pp. 7–31. doi: 10.1007/978-3-031-50883-7_2.

[3] J. Garofalakis, Y. Panagis, E. Sakkopoulos, and A. Tsakalidis, "Contemporary web service discovery mechanisms," *Journal of Web Engineering*, pp. 265–290, 2006.

[4] A. Kontogianni, K. Kabassi, and E. Alepis, "Designing a Smart Tourism Mobile Application: User Modelling Through Social Networks' User Implicit Data," in *Social Informatics*, vol. 11186, S. Staab, O. Koltsova, and D. I. Ignatov, Eds., in Lecture Notes in Computer Science, vol. 11186. , Cham: Springer International Publishing, 2018, pp. 148–158. doi: 10.1007/978-3-030-01159-8_14.

[5] S. Sioutas, E. Sakkopoulos, C. Makris, B. Vassiliadis, A. Tsakalidis, and P. Triantafillou, "Dynamic Web Service discovery architecture based on a novel peer based overlay network," *Journal of Systems and Software*, vol. 82, no. 5, pp. 809–824, 2009.

[6] E. Politou, E. Alepis, and C. Patsakis, "Forgetting personal data and revoking consent under the GDPR: Challenges and proposed solutions," *Journal of cybersecurity*, vol. 4, no. 1, p. tyy001, 2018.

[7] "Smart tourism: State of the art and literature review for the last six years," *Array*, vol. 6, p. 100020, Jul. 2020, doi: 10.1016/j.array.2020.100020.

[8] A. Kontogianni and E. Alepis, "Social Network Data Enabling Smart Tourism," in *2023 14th International Conference on Information, Intelligence, Systems & Applications (IISA)*, Jul. 2023, pp. 1–6. doi: 10.1109/IISA59645.2023.10345898.

[9] D. A. Herzog, "A User-Centered Approach to Solving the Tourist Trip Design Problem for Individuals and Groups".

[10] "An effective approach for bi-objective multi-period touristic itinerary planning," *Expert Systems with Applications*, vol. 240, p. 122437, Apr. 2024, doi: 10.1016/j.eswa.2023.122437.

[11] Z. Zhang, P. Xu, Y. Sun, Y. Shi, and W. Luo, "Decomposability-Guaranteed Cooperative Coevolution for Large-Scale Itinerary Planning," Jun. 13, 2025, *arXiv*: arXiv:2506.06121. doi: 10.48550/arXiv.2506.06121.

[12] "Design and development of interactive trip planning for web-based transit information systems," *Transportation Research Part C: Emerging Technologies*, vol. 8, no. 1–6, pp. 409–425, Feb. 2000, doi: 10.1016/S0968-090X(00)00016-4.

[13] M. Apperley, D. Fletcher, B. Rogers, and K. Thomson, "Interactive visualisation of a travel itinerary," in *Proceedings of the working conference on Advanced visual interfaces*, in AVI '00. New York, NY, USA: Association for Computing Machinery, May 2000, pp. 221–226. doi: 10.1145/345513.345321.

[14] J. Angskun, S. Korbua, and T. Angskun, “Time-related factors influencing on an itinerary planning system,” *Journal of Hospitality and Tourism Technology*, vol. 7, no. 1, pp. 16–36, Feb. 2016, doi: 10.1108/JHTT-10-2014-0056.
[15] “Using Geographic Information Systems and the World Wide Web for Interactive Transit-Trip Itinerary Planning,” *Journal of Public Transportation*, vol. 3, no. 2, pp. 37–50, Apr. 2000, doi: 10.5038/2375-0901.3.2.3.
[16] “What Is User Experience (and What Is It Not)?,” Nielsen Norman Group. Accessed: Feb. 20, 2026. [Online]. Available: https://www.nngroup.com/articles/what-is-user-experience/
[17] L. Ciolfi, M. Lewkowicz, and K. Schmidt, "Computer-Supported Cooperative Work," in *Handbook of Human Computer Interaction*, J. Vanderdonckt, P. Palanque, and M. Winckler, Eds. Cham: Springer International Publishing, 2023, pp. 1–26. doi: 10.1007/978-3-319-27648-9_30-1.
[18] “Agile Software Development - Software Engineering,” GeeksforGeeks. Accessed: Feb. 21, 2026. [Online]. Available: https://www.geeksforgeeks.org/software-engineering/software-engineering-agile-software-development/
[19] “Atomic Design Systems: Why the Labels Don’t Matter.” Accessed: Feb. 21, 2026. [Online]. Available: https://www.qt.io/blog/atomic-design-systems-why-the-labels-dont-matter
[20] R. Kadali, “Breaking the Monolith: Lessons from the Edge Computing Frontier,” *International Journal of Advances in Engineering and Management*, vol. 7, pp. 342–350, Feb. 2025, doi: 10.35629/5252-0702342350.
[21] S. Bhattacharya *et al.*, “Database-Level End User Authorization (DB-EUA),” *International Journal of Global Innovations and Solutions*, vol. 2, Dec. 2025, doi: 10.63412/j4rckr15.
[22] Y. Shkuro, B. Renard, and A. Singh, “Positional Paper: Schema-First Application Telemetry,” *ACM SIGOPS Operating Systems Review*, vol. 56, pp. 8–17, Jun. 2022, doi: 10.1145/3544497.3544500.
[23] “What is BaaS? | Backend-as-a-Service vs. serverless.” Accessed: Feb. 21, 2026. [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/
[24] “API Reference: Turbopack | Next.js.” Accessed: Feb. 20, 2026. [Online]. Available: https://nextjs.org/docs/app/api-reference/turbopack
[25] A. Singh, “Build Modern Maps in Next.js with Mapbox and shadcn/ui,” Medium. Accessed: Feb. 19, 2026. [Online]. Available: https://medium.com/@sainianmol16/build-modern-maps-in-next-js-with-mapbox-and-shadcn-ui-80c276a1e9bf
[26] ncdai, “Chánh Đại – Design Engineer,” Chánh Đại. Accessed: Feb. 19, 2026. [Online]. Available: https://chanhdai.com
[27] Supabase, “Cron | Supabase Docs.” Accessed: Feb. 19, 2026. [Online]. Available: https://supabase.com/docs/guides/cron
[28] Supabase, “Database | Supabase Docs.” Accessed: Feb. 19, 2026. [Online]. Available: https://supabase.com/docs/guides/database/overview
[29] sadmann7, “Dice UI,” Dice UI. Accessed: Feb. 19, 2026. [Online]. Available: https://diceui.com
[30] Supabase, “Edge Functions | Supabase Docs.” Accessed: Feb. 19, 2026. [Online]. Available: https://supabase.com/docs/guides/functions
[31] “First Contentful Paint (FCP) | Articles,” web.dev. Accessed: Feb. 19, 2026. [Online]. Available: https://web.dev/articles/fcp
[32] “Getting Started: Error Handling | Next.js.” Accessed: Feb. 19, 2026. [Online]. Available: https://nextjs.org/docs/app/getting-started/error-handling
[33] “Getting Started: Server and Client Components | Next.js.” Accessed: Feb. 19, 2026. [Online]. Available: https://nextjs.org/docs/app/getting-started/server-and-client-components
[34] “Getting Started: Updating Data | Next.js.” Accessed: Feb. 19, 2026. [Online]. Available: https://nextjs.org/docs/app/getting-started/updating-data
[35] “How to highlight countries with Mapbox | Blog.” Accessed: Feb. 19, 2026. [Online]. Available: https://www.nieknijland.nl/blog/how-to-highlight-countries-with-mapbox
[36] S. Markbåge, “How to Think About Security in Next.js.” Accessed: Feb. 19, 2026. [Online]. Available: https://nextjs.org/blog/security-nextjs-server-components-actions
[37] imskyleen, “Icons,” Animate UI. Accessed: Feb. 19, 2026. [Online]. Available: https://animate-ui.com

[38] "Implementing Optimistic UI in React.js/Next.js," DEV Community. Accessed: Feb. 19, 2026. [Online]. Available: https://dev.to/olaleyeblessing/implementing-optimistic-ui-in-reactjsnextjs-4nkk
[39] "Installation | Playwright." Accessed: Feb. 19, 2026. [Online]. Available: https://playwright.dev/docs/intro
[40] "Intro," Zod. Accessed: Feb. 19, 2026. [Online]. Available: https://zod.dev/
[41] "JavaScript With Syntax For Types." Accessed: Feb. 19, 2026. [Online]. Available: https://www.typescriptlang.org/
[42] "Jest." Accessed: Feb. 19, 2026. [Online]. Available: https://jestjs.io/
[43] "Kibo UI." Accessed: Feb. 19, 2026. [Online]. Available: https://www.kibo-ui.com/
[44] "Locate the user | Mapbox GL JS," Mapbox. Accessed: Feb. 19, 2026. [Online]. Available: https://docs.mapbox.com//mapbox-gl-js/example/locate-user
[45] Supabase, "Login with Google | Supabase Docs." Accessed: Feb. 19, 2026. [Online]. Available: https://supabase.com/docs/guides/auth/social-login/auth-google
[46] "Lucide Icons," Lucide. Accessed: Feb. 19, 2026. [Online]. Available: https://lucide.dev
[47] "Mapbox GL JS," Mapbox. Accessed: Feb. 19, 2026. [Online]. Available: https://docs.mapbox.com//mapbox-gl-js/guides/
[48] "Mobile first - Glossary | MDN," MDN Web Docs. Accessed: Feb. 21, 2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Mobile_First
[49] "Motion — JavaScript & React animation library." Accessed: Feb. 20, 2026. [Online]. Available: https://motion.dev
[50] "Next.js Docs | Next.js." Accessed: Feb. 19, 2026. [Online]. Available: https://nextjs.org/docs
[51] I. Farooq, "Next.js Software Design, Architecture & Best Practices," Medium. Accessed: Feb. 19, 2026. [Online]. Available: https://javascript.plainenglish.io/next-js-software-design-architecture-best-practices-855fc4ec806d
[52] "next.js/examples/with-playwright at canary · vercel/next.js," GitHub. Accessed: Feb. 19, 2026. [Online]. Available: https://github.com/vercel/next.js/tree/canary/examples/with-playwright
[53] Supabase, "OAuth 2.1 Flows | Supabase Docs." Accessed: Feb. 19, 2026. [Online]. Available: https://supabase.com/docs/guides/auth/oauth-server/oauth-flows
[54] "Free Open-Source Weather API | Open-Meteo.com." Accessed: Feb. 21, 2026. [Online]. Available: https://open-meteo.com/
[55] "Radix UI." Accessed: Feb. 19, 2026. [Online]. Available: https://radix-ui.com/
[56] "React." Accessed: Feb. 19, 2026. [Online]. Available: https://react.dev/
[57] shadcn, "React Hook Form." Accessed: Feb. 19, 2026. [Online]. Available: https://ui.shadcn.com/docs/forms/react-hook-form
[58] "Responsive design - Core concepts." Accessed: Feb. 21, 2026. [Online]. Available: https://tailwindcss.com/docs/responsive-design
[59] "Reusing Logic with Custom Hooks – React." Accessed: Feb. 19, 2026. [Online]. Available: https://react.dev/learn/reusing-logic-with-custom-hooks
[60] Supabase, "Row Level Security | Supabase Docs." Accessed: Feb. 21, 2026. [Online]. Available: https://supabase.com/docs/guides/database/postgres/row-level-security
[61] "Split expenses with friends.," Splitwise. Accessed: Feb. 20, 2026. [Online]. Available: https://www.splitwise.com/
[62] "Stippl: The All In One Adventure Planner." Accessed: Feb. 20, 2026. [Online]. Available: https://www.stippl.io/
[63] Supabase, "Storage | Supabase Docs." Accessed: Feb. 19, 2026. [Online]. Available: https://supabase.com/docs/guides/storage
[64] Supabase, "Supabase UI Library." Accessed: Feb. 19, 2026. [Online]. Available: https://supabase.com/ui
[65] "Tailwind CSS - Rapidly build modern websites without ever leaving your HTML." Accessed: Feb. 19, 2026. [Online]. Available: https://tailwindcss.com/
[66] "Testing: Jest | Next.js." Accessed: Feb. 19, 2026. [Online]. Available: https://nextjs.org/docs/app/guides/testing/jest
[67] "Testing: Playwright | Next.js." Accessed: Feb. 19, 2026. [Online]. Available: https://nextjs.org/docs/app/guides/testing/playwright

[68] shadcn, “The Foundation for your Design System - shadcn/ui.” Accessed: Feb. 19, 2026. [Online]. Available: https://ui.shadcn.com/
[69] “Travel with Google - Planning a Trip Is Simple with Google.” Accessed: Feb. 20, 2026. [Online]. Available: https://travel.google/
[70] “True Lies Of Optimistic User Interfaces,” Smashing Magazine. Accessed: Feb. 19, 2026. [Online]. Available: https://www.smashingmagazine.com/2016/11/true-lies-of-optimistic-user-interfaces/
[71] “Use Mapbox GL JS in a React app | Help,” Mapbox. Accessed: Feb. 19, 2026. [Online]. Available: https://docs.mapbox.com//help/tutorials/use-mapbox-gl-js-with-react
[72] “Vercel: Build and deploy the best web experiences with the AI Cloud,” Vercel. Accessed: Feb. 19, 2026. [Online]. Available: https://vercel.com/home
[73] *vercel/next-app-router-playground*. (Feb. 16, 2026). TypeScript. Vercel. Accessed: Feb. 19, 2026. [Online]. Available: https://github.com/vercel/next-app-router-playground
[74] “Wanderlog travel planner: free vacation planner and itinerary app.” Accessed: Feb. 20, 2026. [Online]. Available: https://wanderlog.com/
[75] “Your All-In-One Travel App,” Lambus. Accessed: Feb. 20, 2026. [Online]. Available: https://lambus.com/
[76] ReUI, “The Foundation for your Design System - ReUI.” Accessed: Feb. 22, 2026. [Online]. Available: https://reui.io/
[77] “A new, modern UI component library built on top of Base UI - coss ui.” Accessed: Feb. 26, 2026. [Online]. Available: https://coss.com/ui
[78] Supabase, “Local Development & CLI | Supabase Docs.” Accessed: Feb. 26, 2026. [Online]. Available: https://supabase.com/docs/guides/local-development