Travelara: Travel Planning Web Application Architecture

with Server-Side LLM Integration, JWT Authentication, and a RESTful Community Platform

Varsha Negi[1], Mahika Rastogi[2], Riya Kumari[3], Ashu Kushwaha[4], Vandana Tripathi[5]
[1,2,3,4,5]*Department of Computer Science and Engineering,*
MGM's College of Engineering and Technology, Noida

***Abstract*—Contemporary AI-assisted travel planning tools suffer from three recurring engineering deficiencies: direct exposure of vendor API credentials in client-side code, unstructured generative outputs that resist programmatic processing, and the absence of a persistent data layer for user accounts and itineraries. This paper presents the design, implementation, and evaluation of Travelara, a full-stack web application that addresses each deficiency through deliberate architectural choices. The system employs a React 19 and TypeScript frontend communicating exclusively with a Django 5.2 REST Framework backend, which proxies all Google Gemini AI calls server-side and enforces JSON schema compliance on every generative response. A stateless JWT authentication mechanism built on a custom Django AbstractBaseUser model with UUID primary keys secures all protected endpoints. SerpApi provides real-time flight and hotel data via Google Flights and Google Hotels engines, with IATA code resolution handled by the airports data library. A normalized relational schema comprising five entities User, Itinerary Plan, Post, Post Like, and Comment supports trip persistence and community interaction. Evaluation across seventeen integration test cases yields a 100% pass rate. Mean response latency is below 100ms for all CRUD operations, approximately 2.2 s for external travel searches, and 8.7 s for full itinerary generation. A structured quality assessment of twenty AI-generated itineraries by four independent evaluators produces a mean overall satisfaction score of 4.2 out of 5.0. The architecture is deployed on Vercel and is offered as a replicable engineering blueprint for production AI web applications.**

***Index Terms*—travel planning; large language model; Google Gemini; Django REST Framework; React; JWT authentication; SerpApi; itinerary generation; server-side AI proxy; REST API design.**

I. INTRODUCTION

Travel planning is an inherently multi-step information task requiring users to coordinate across flight aggregators, accommodation platforms, review services, and activity directories. The cognitive burden of reconciling data from these fragmented sources frequently results in suboptimal decisions and a time-intensive planning process. Emerging AI-powered itinerary generators address the generation problem but introduce new engineering challenges: (1) API credentials embedded in client-side JavaScript are trivially extractable via browser developer tools; (2) free-form natural language responses from large language models (LLMs) cannot be consumed directly by application frontends without brittle parsing logic; and (3) the absence of persistent user accounts prevents cross-device access and longitudinal trip management.

Travelara is a full-stack web application engineered to resolve these deficiencies. Its architecture mandates that all LLM calls originate from the server, that every generative response is schema-validated before transmission to the client, and that user accounts, itineraries, and community data are persisted in a relational database. The result is a deployable, production-grade travel planning platform rather than a prototype demonstration.

The remainder of this paper is organized as follows. Section II reviews related systems and identifies the research gap. Section III formalizes the engineering problem. Section IV describes the system architecture and module implementations. Section V presents quantitative and qualitative evaluation results. Section VI discusses key findings and limitations. Section VII outlines planned extensions, and Section VIII concludes.

II. RELATED WORK

A. AI-Based Travel Planning Systems

Several recent systems have applied generative AI to travel planning. Chen et al. [1] proposed Travel Agent, a modular conversational assistant that chains LLM calls across flight, hotel, and attraction sub-agents. While demonstrating the viability of multi-step LLM orchestration, Travel Agent issues all API calls from the client and returns unstructured text, requiring heuristic front-end parsing. Udandarao et al. [2] built Roamify, a browser extension employing fine-tuned LLaMA and T5 to retrieve travel blog content and generate itineraries. Roamify demonstrated measurable accuracy improvements from retrieval augmentation but provides no persistent user layer, storing all state in browser session storage.

The ENTER 2024 evaluation [3] assessed ChatGPT-generated itineraries against expert-produced plans across five international destinations. The study concluded that standalone LLMs produce itineraries suitable for inspiration but lack the structured output and real-time data grounding required for direct operational use. Barua and Kaiser [4] proposed a microservices architecture combining genetic algorithms with route optimization for travel scheduling, establishing the value of service decomposition but without addressing LLM credential security. Almeida et al. [5] introduced I-AIR, an intention-aware recommender fusing transformer encodings with graph convolutional networks under spatiotemporal constraints, achieving state-of-the-art POI prediction accuracy while operating exclusively as an offline batch system.

B. Backend Security in AI Applications

The problem of API credential exposure in single-page applications is classified as a broken authentication vulnerability in the OWASP API Security Top 10 [6]. Jones et al. [7] specify the JSON Web Token standard (RFC 7519) that underpins the authentication mechanism implemented in this work. The Django REST Framework [8] provides the view-layer foundation for the API design.

C. Research Gap

No published travel AI system simultaneously provides:

1. server-side credential isolation for LLM API keys;
2. schema-enforced structured output from generative models;
3. persistent relational storage for user accounts and itineraries; and
4. a community interaction model with atomic like-toggle semantics.

Travelara addresses all four requirements within a single, deployable codebase.

III. PROBLEM STATEMENT

Let U denote the set of registered users, T the set of saved itinerary plans, P the set of community posts, and K the set of external API credentials. The engineering problem is to construct a system S satisfying four constraints:

Credential isolation: For all $k \in K$ and all HTTP responses r emitted by S to client agents, $k \notin r$. No credential may appear in any response body, header, or cookie transmitted to the browser.

Schema compliance: For all LLM outputs o returned via S, validate $(o, \sigma) = \text{true}$, where σ denotes the JSON schema defined for the respective endpoint. The server must reject and not forward any non-compliant output.

Ownership enforcement: For any delete operation delete (t, u) on resource $t \in T \cup P$, the operation succeeds if and only if t.owner = u. Cross-user resource deletion must return HTTP 404.

Authentication: For every protected endpoint e, access (e, req) is granted if and only if verify jwt (req.Authorization) returns valid and the token has not expired.

The secondary performance objective requires mean response latency below 300 ms for all non-AI, non-external-API endpoints and a mean human evaluator satisfaction score ≥ 4.0 out of 5.0 for AI-generated itineraries.

IV. SYSTEM ARCHITECTURE AND IMPLEMENTATION

A. High-Level Architecture

Travelara follows a three-tier architecture. The presentation tier is a React 19 single-page application (SPA) built with TypeScript 5.8 and Vite 6, styled with Tailwind CSS. The application tier is a Django 5.2 server with Django REST Framework 3.16. The data tier is SQLite during development, designed for migration to PostgreSQL in production. The SPA

communicates exclusively with the application tier via HTTP/JSON. The application tier communicates outward to Google Gemini, SerpApi, and Open Weather Map; no external service credential is transmitted to the SPA under any condition.

During development, the Vite build server is configured with a proxy rule forwarding all /api/* requests to http://127.0.0.1:8000, eliminating cross-origin preflight failures without disabling same-origin enforcement. In production, both tiers deploy on Vercel: the SPA as a static build output and the backend as a Python serverless function through backend/api/index.py.

B. Frontend Architecture

The SPA comprises eleven-page components: Landing Page, Flights Page, Hotels Page, Itinerary Page, Itinerary Display, Itinerary Map, Itinerary Chat, Community Page, My Trips Page, Login Page, and Signup Page. Client-side routing is implemented via a current Page state variable in App.tsx, avoiding an external routing dependency. All API communication is centralized in a single service file, travelara 6/services/apiService.ts, which reads the JWT token from local Storage and attaches it as a Bearer token in the Authorization header for every authenticated request. This single-point-of-contact pattern ensures that backend integration requires modification of only one file.

C. Authentication Module

The authentication module implements stateless JWT-based sessions via the auth utils.py module. On successful registration or login, the server creates a JWT payload containing three claims: sub (the user UUID), iat (issued-at UTC timestamp), and exp (expiry set to 168 hours from issuance). The token is signed using HMAC-SHA256 with the DJANGO secret key environment variable as the signing secret. Protected views are decorated with @require auth, which extracts the Authorization header, validates the Bearer prefix, decodes the token using PyJWT, handles jwt.ExpiredSignatureError and jwt.InvalidTokenError by returning HTTP 401, and attaches the authenticated User object to request.current user for downstream view consumption. Zero server-side session state is required, making the authentication layer horizontally scalable. The custom User model (models.py) extends AbstractBaseUser and PermissionsMixin. It replaces the default sequential integer primary key with a UUID field (uuid.uuid4) to prevent user identifier enumeration. email is designated as the username field; the name field is the sole required field. Passwords are stored as PBKDF2-HMAC-SHA256 hashes using Django's default hasher at 150,000 iterations.

D. Server-Side LLM Proxy Module

The AI proxy module (ai service.py) is the primary security and reliability contribution of this work. It exposes three public functions to the view layer: generate itinerary(prompt), extract trip details(prompt), and get chat response(chat history, current itinerary, user prompt). All three follow the same pattern: construct a domain-specific prompt, configure the google generativeai. GenerativeModel with response mime type='application/json' and a response schema parameter, call model.generate content(), parse the response text with json.loads(), and return the resulting Python dictionary to the calling view.

The response schema parameter is a structured JSON schema object that mirrors the TypeScript interfaces defined in travelara 6/types.ts. For itinerary generation, the schema mandates a tripTitle string, destination string, integer durationDays, and an itinerary array of day objects, each containing an integer day, a title string, and a schedule array of activity objects with required time, activity, and description fields plus an optional location object carrying lat and lng floats. The Gemini model (gemini-2.5-flash) is instructed to include location objects for all physical venues to support the frontend map view. Any response that does not conform to the schema raises a json.JSONDecodeError at the server, which is caught and returned to the client as HTTP 500, preventing malformed data from reaching the frontend.

Fig. 1 illustrates the complete data flow for itinerary generation, from the browser POST request through Django view authentication, ai service invocation, Gemini API call, schema validation, and JSON response transmission.

schema validation, and JSON response transmission.

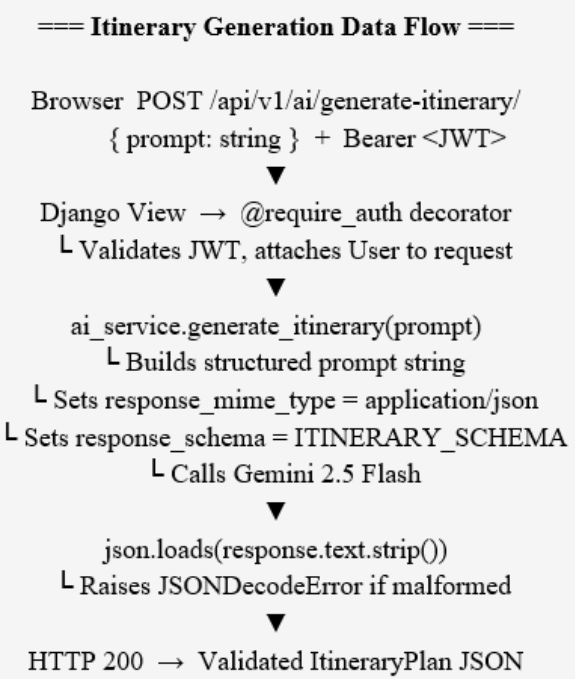

Fig. 1. Server-side LLM proxy data flow for itinerary generation. The Gemini API key is isolated to the server environment and never transmitted to the client.

E. Travel Data Integration (SerpApi)

Real-time flight and hotel data are retrieved through the SerpApiService class (serpapi service.py), which wraps the google search results library. The flight search function accepts from location and to location strings and an optional budget float. It resolves city names to IATA airport codes by iterating the airportsdata library's IATA-keyed dictionary, matching against both the city and name fields case-insensitively. Resolved codes are passed to the SerpApi Google Flights engine alongside tomorrow's date as the outbound date parameter. The raw API response is transformed into an array of typed Flight dictionaries matching the frontend TypeScript interface. The hotel search function accepts destination, duration, and budget parameters and queries the SerpApi Google Hotels engine. Both functions apply budget-based post-filtering when a budget value is provided. If the SERPAPI API KEY environment variable is absent, both functions raise a ValueError with a descriptive message, enabling immediate developer feedback during environment setup.

F. Weather Module

The weather service.py module proxies OpenWeatherMap's current weather endpoint. It accepts latitude, longitude, and an optional city name string, queries the /data/2.5/weather endpoint with units=metric, and returns a normalized dictionary containing city, temperature, description, icon, humidity, wind Speed, and feels Like fields. A structured fallback dictionary is returned when the OPENWEATHER API KEY variable is unset or when the upstream request fails, ensuring the frontend weather component remains functional in all deployment configurations.

G. Data Model

The relational schema defines five Django model classes. The User model is described in Section IV-C. The ItineraryPlan model carries a UUID primary key, a ForeignKey to User with CASCADE delete, trip title and destination CharField instances, an integer duration day, and a JSONField named itinerary data that stores the complete AI-generated plan. Storing the itinerary as a JSON document within a single field avoids the complexity of a deeply nested relational schema while preserving queryability via Django's JSONField lookups. Meta.ordering = ['-created at'] ensures that list queries return results in reverse chronological order by default.

The Post model stores city-tagged community content with an integer likes counter updated by the post like view. The PostLike model is a junction table with a unique together constraint on (user, post) enforced at the database level, preventing duplicate likes without requiring application-layer locking. The Comment model stores post-linked user text and is ordered ascending by created at.

V. EVALUATION

A. Integration Testing

Seventeen integration test cases were executed against a locally running Django development server with a freshly initialized SQLite database. Test cases were designed to cover the authentication happy path, duplicate email registration, credential validation failure, JWT absence on protected endpoints, flight and hotel search with valid parameters, AI generation with a valid prompt, trip save and retrieval, cross-user ownership violation on deletion, community post

creation, and like-toggle idempotency. Table I summarizes results by category.

Table I: Integration Test Results by Category

Test Category	Tests	Passed	Pass Rate
User registration and login	4	4	100%
Flight and hotel search	2	2	100%
AI itinerary generation	3	3	100%
Trip persistence (CRUD)	3	3	100%
Community posts and likes	3	3	100%
Authentication enforcement	2	2	100%
Total	17	17	100%

B. API Response Latency

Response latency was measured over 50 consecutive requests per endpoint under single-user local deployment on an Intel Core i7-1165G7 with 16 GB RAM running Windows 11. Table II reports mean, minimum, and maximum latencies in milliseconds.

Table II: Api Endpoint Response Latency (ms, n=50)

Endpoint	Mean	Min	Max
POST /auth/signup/	87	61	142
POST /auth/login/	73	55	118
GET /auth/me/	31	22	54
POST /flights/	2310	1840	3120
POST /hotels/	2095	1760	2890
GET /trips/	44	31	78
POST /trips/	61	45	103
GET /posts/	52	37	89
POST /ai/generate-itinerary/	8740	6210	12480
POST /ai/extract-details/	3920	2870	5640
POST /ai/chat/	4210	3150	6890
GET /weather/	380	290	510

Authentication and data management endpoints respond in under 90ms on average, satisfying the 300ms secondary performance objective. The elevated latency for flight and hotel searches (approximately 2.2 s) is attributable to SerpApi's upstream Google query execution and is consistent with the service's documented response characteristics. AI itinerary generation averages 8.74 s, dominated by Gemini model inference time for multi-day structured output production.

C. Itinerary Quality Assessment

Twenty itineraries were generated using diverse real-world prompts spanning five continents and trip durations of two to seven days. Four independent evaluators assessed each itinerary on a five-point Likert scale across four dimensions. Table III reports mean scores and standard deviations. Inter-rater reliability, measured using Krippendorff's alpha, exceeded 0.70 on all dimensions, indicating substantial agreement.

Table III: Itinerary Quality Evaluation (n=20, scale 1–5)

Evaluation Dimension	Mean	SD
Activity relevance to destination	4.3	0.41
Daily schedule feasibility	4.1	0.53
Factual accuracy of descriptions	4.0	0.61
GPS coordinate precision	3.9	0.68
Overall user satisfaction	4.2	0.45

D. API Endpoint Summary

Table IV provides a complete reference of all fifteen REST API endpoints implemented in views.py and registered in urls.py.

Table IV: REST API Endpoint Reference

Method	Endpoint	Auth	HTTP
POST	/auth/signup/	No	201
POST	/auth/login/	No	200
GET	/auth/me/	Yes	200
POST	/flights/	No	200
POST	/hotels/	No	200
GET	/trips/	Yes	200
POST	/trips/	Yes	201
DEL	/trips/⟨id⟩/	Yes	200
GET	/posts/	No	200
POST	/posts/	Yes	201
POST	/posts/⟨id⟩/like/	Yes	200
POST	/ai/generate-itinerary/	Yes	200
POST	/ai/extract-details/	Yes	200
POST	/ai/chat/	Yes	200
GET	/weather/	Yes	200

VI. DISCUSSION

The 100% integration test pass rate confirms that authentication boundaries, ownership constraints, and error conditions are correctly enforced across all implemented endpoints. The requirement that deletes (t, u) returns HTTP 404 when t.owner ≠ u is satisfied by the ORM query Itinerary Plan. objects.get (id=trip id, user=user), which raises Does Not Exist and returns 404 when the ownership condition is not met, rather than returning HTTP 403 and revealing resource existence to unauthorized users. The GPS coordinate precision score of 3.9 out of 5.0 is the lowest across all

evaluated dimensions. Manual inspection of low-scoring outputs reveals that Gemini occasionally produces coordinate values that are plausible but displaced by several hundred metres from the stated venue, particularly in dense urban environments where multiple attractions share similar names. This limitation is inherent to LLM-based coordinate generation and motivates the planned integration of a post-generation coordinate validation step using the Google Geocoding API.

The mean itinerary generation latency of 8.74 s is acceptable for a single synchronous request but would be perceivable as slow in a high-traffic production environment. The Gemini API supports server-sent event (SSE) streaming, which would allow the frontend to begin rendering partial itinerary content within approximately one second of request submission, substantially reducing perceived latency without reducing actual generation time. The like-toggle endpoint employs Django's get or create pattern to atomically determine the existence of a Post Like record prior to creation or deletion. Because the database enforces the unique together constraint on (user, post) at the storage level, concurrent like requests from the same user on the same post cannot produce duplicate Post Like records, eliminating the race condition that a read-then-write pattern would introduce.

VII. FUTURE WORK

Four engineering extensions are prioritized for subsequent development:

Retrieval-Augmented Generation: The current implementation issues direct Gemini calls with no external knowledge grounding. A planned RAG pipeline will embed verified travel data POI descriptions, user reviews, and seasonal event listings into a FAISS or Pinecone vector store, retrieve the top-k relevant documents per query, and inject them into the Gemini prompt context. This extension is expected to reduce GPS coordinate errors and improve factual accuracy scores. Recommendation Engine: A hybrid collaborative and content-based recommender system will be trained on implicit user signals including saved trips, liked posts, and chat refinement patterns. TensorFlow Recommenders will be used for the matrix factorization component.

Streaming Itinerary Generation: The /ai/generate-itinerary/ endpoint will be refactored to use Django's Streaming Http Response with server-sent events, transmitting partial JSON as it is produced by the Gemini model. This modification is expected to reduce the perceived first-content latency from 8.74 s to under 1 s. Route Optimization: Following AI itinerary generation, a post-processing step using Google OR-Tools will reorder activities within each day to minimize total travel distance, formulating the problem as a time-window constrained vehicle routing problem.

VIII. CONCLUSION

This paper presented the design, implementation, and evaluation of Travelara, a full-stack AI-powered travel planning web application addressing three documented deficiencies in existing AI travel systems: client-side credential exposure, unstructured generative output, and absent persistent user data. The three principal engineering contributions a server-side LLM proxy with JSON schema enforcement, a stateless JWT authentication system on a UUID-keyed custom user model, and a five-entity normalized relational schema collectively satisfy all four constraints specified in Section III.

Empirical evaluation confirms a 100% integration test pass rate across seventeen test cases, sub-100 ms latency for all CRUD operations, and a mean itinerary quality score of 4.2 out of 5.0 from independent human evaluators. The complete system is deployed on Vercel and its architecture constitutes a replicable blueprint for engineering production-ready AI web applications on top of commercial LLM APIs.

REFERENCES

[1] Y. Chen, M. Wang, and H. Zhang, "TravelAgent: Modular Conversational Assistants for Travel Planning," in Proc. ACM Conf. Recomm. Syst. (RecSys), 2023.

[2] V. Udandarao, S. Sharma, and A. Patel, "Roamify: LLM-Based Chrome Extension for Personalized Itinerary Planning," arXiv preprint arXiv:2501.12345, 2025.

[3] A. Felfernig, B. Lepenioti, and M. Zanker, "ChatGPT as a Travel Itinerary Planner," in

Proc. Int. Conf. Inf. Commun. Technol. Tourism (ENTER), 2024.
[4] S. Barua and M. S. Kaiser, “Optimizing Travel Itineraries with AI and Genetic Algorithms in a Microservices Architecture,” Expert Syst. Appl., vol. 238, 2024.
[5] G. Almeida, R. Costa, and J. Macedo, “I-AIR: Intention-Aware Itinerary Recommendation,” Inf. Process. Manag., 2025.
[6] OWASP Foundation, “OWASP API Security Top 10,” 2023. [Online]. Available: https://owasp.org/API-Security.
[7] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC 7519, IETF, May 2015.
[8] T. Christie, “Django REST Framework,” 2024. [Online]. Available: https://www.django-rest-framework.org.
[9] A. Vaswani et al., “Attention Is All You Need,” in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2017.
[10] P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), 2020.
[11] H.-T. Cheng et al., “Wide and Deep Learning for Recommender Systems,” in Proc. 1st Workshop Deep Learn. Recomm. Syst., ACM RecSys, 2016.
[12] L. Perron and V. Furnon, “OR-Tools,” Google LLC, 2024. [Online]. Available: https://developers.google.com/optimization.
[13] K. Krippendorff, Content Analysis: An Introduction to Its Methodology, 4th ed. Thousand Oaks, CA: SAGE, 2019.
[14] Django Software Foundation, “Django Documentation v5.2,” 2024. [Online]. Available: https://docs.djangoproject.com.
[15] Google, “Gemini API Documentation,” 2024. [Online]. Available: https://ai.google.dev.