TravelPlanner: A Scalable and Elastic Trip Planning System on AWS

Danxu Chen, Wayne Wang[1], Yuwei Sun[1], Yirong Wang[1]
Department of Computer Science, New York University
New York, United States
{dc4806, yw5954, ys4680, yw5490}@nyu.edu

***Abstract*—This paper presents the design and implementation of an AWS-powered intelligent travel planning assistant that automates and enhances the trip planning process. Leveraging scalable AWS services and large language model (LLM) capabilities, the system delivers personalized itinerary generation, smart route visualization, real-time weather forecasting, and proactive ticket alerts. The architecture integrates cloud-native components such as Lambda, API Gateway, RDS, ElasticCache and Simple Storage Service(S3) to enable scalable, event-driven workflows. By reducing manual overhead and incorporating AI-driven insights, the solution significantly improves planning accuracy, responsiveness, and user experience. This work demonstrates the feasibility and advantages of combining cloud elasticity with intelligent automation for next-generation travel applications.**

***Index Terms*—cloud computing, routing algorithms, Amazon Web Service(AWS), large language models, real-time data, intelligent systems**

I. Problem Statement

Despite the abundance of online tools for travel planning, users are still faced with fragmented workflows that need to be manually coordinated across multiple platforms, including booking websites, mapping services, weather apps and personal calendars. These disconnected systems lack personalization, real-time responsiveness and interconnectivity. At the same time, users had to jump between multiple services, resulting in a time-consuming and error-prone planning process.

Existing open-source travel planning solutions do not fully leverage the capabilities of cloud computing and artificial intelligence to automate and optimize itinerary generation. Popular platforms such as [1] and [2] typically rely on monolithic backend architectures and locally hosted SQL databases, which limit their scalability, elasticity, and real-time responsiveness. These designs are unsuitable for dynamic workloads and high user concurrency, making them ill-suited for large-scale deployment.

Moreover, current systems often incorporate large language models (LLMs) such as [3] use Gemini through superficial chatbot interfaces, without deep backend integration. This limited usage prevents full utilization of LLM capabilities, including contextual knowledge access, multimodal reasoning, and image understanding. In contrast, emerging industry trends demonstrate the effectiveness of tightly coupling LLMs with backend services to enhance system intelligence and automation.

Authors contributed equally and are listed in alphabetical order.

These challenges highlight the need to examine the problem from multiple perspectives and motivate targeted solutions to address each critical aspect, as discussed in the following section.

II. Motivation

As demand for personalized travel and just-in-time travel grows, traditional travel planning methods are no longer able to meet user expectations for intelligence, automation, and real-time responsiveness. Open-source solutions also fall short in scalability and personalized recommendation metrics. Users often face problems in manually coordinating trips across multiple platforms, lack of real-time information, and difficulty obtaining highly customized recommendations. This project aims to address the following core challenges:

A. Cloud Native Elastic Architecture:

Most existing travel platforms are based on a local deployment or a centralized backend, making it difficult to cope with highly concurrent user access and sudden traffic changes. By introducing AWS's cloud-native architecture (such as Lambda, API Gateway, and Aurora), this project realizes elastic expansion of services and on-demand billing, and improves the availability and economy of the system.

B. Real-time data integration capability:

Travel planning requires the integration of a variety of dynamic data, such as airfare information, travel map routes, and weather forecasts. Traditional programs often rely on static content or low-frequency updates, resulting in late advice and poor reliability. The system supports real-time access to external data sources to ensure that users receive up-to-date itinerary advice and reminders of airline tickets, thus reducing travel risks.

C. Deep Integration of Large Language Models:

Currently, some platforms use LLMs (e.g. OpenAI ChatGPT, Google Gemini) as chat assistants, but the lack of deep integration with back-end services limits its contextual understanding and multimodal capabilities. In this project, LLM is integrated into the core modules of itinerary generation, information recognition, and personalized recommendation, so that it can not only have dialogues, but also participate in the decision-making process, which comprehensively improves the intelligence level of the system.

D. Skill Development:

This project not only has engineering implementation value, but also provides valuable educational practice opportunities for team members. By building a cloud-native application, members systematically mastered the key services in the AWS platform. In the collaborative development process, team members improved their version control, module collaboration, and communication and collaboration capabilities.

While these motivations justify the need for a new system design, it is equally important to examine how existing travel planning solutions approach these challenges and where they fall short.

III. Existing Solutions

Numerous open-source solution have been proposed to assist users in travel planning. While many of these tools focus on specific functionalities such as routing or itinerary generation, they often lack end-to-end integration, intelligent automation, and cloud-native scalability. Below, we analyze several representative systems:

- **OpenTripPlanner [1]**: This project bring a open source travel planning engine that supports route planning. However, the tool lacks AI-driven recommendation features, limiting its application in personalized travel planning.
- **AI Trip Planner [3]**: The web-based application uses Google's generative AI to achieve personalized hotel and itinerary recommendations, and obtains location information with the help of Google Places API. However, its overall architecture lacks modular design and scalability, making it difficult to adapt to large-scale deployment scenarios.
- **TripIt [4]**: TripIt is highly focused on trip management, providing automatic ticket parsing and flight change notifications. However, TripIt's core functionality focuses on aggregating and formatting static information rather than active decision-making or intelligent recommendations.
- **GuideGeek [5]**: GuideGeek implements an AI-powered travel assistant that excels in front-end interaction, but as a chatbot, it lacks deep integration with back-end systems. This shortcoming prevents GuideGeek from planning user routes in real-time, which limits the application of GuideGeek to complex itinerary management and system scalability.

In contrast to prior systems that lack deep AI integration and runtime elasticity, our platform is designed to address these gaps through a modular, cloud-native architecture. We now present the system architecture and the rationale behind key design choices.

IV. System Architecture and Design Decisions

To address the limitations identified in existing travel planning systems, we designed the architecture of our intelligent travel planning platform to achieve high scalability, real-time responsiveness, and intelligent automation. The system is built based on cloud-native infrastructure and follows the modularization design principle, integrating key capabilities such as real-time data processing, large language model (LLM) call scheduling, and elastic back-end services. The overall architecture supports automatic itinerary generation, dynamic information updating and personalized recommendations.

A. Overview of System Architecture

The system focuses on optimizing the coordination of low-latency response, asynchronous processing and decoupled services to ensure high availability and fault tolerance in real-time concurrent access scenarios. At the same time, the architecture has good scalability, which facilitates the subsequent integration of new data sources or model reasoning capabilities without affecting the core logic of the system. Figure 1 shows the overall architecture of the system and its main components. At the system level we use AWS CloudWatch to track the status of each service. The system consists of the following 5 levels:

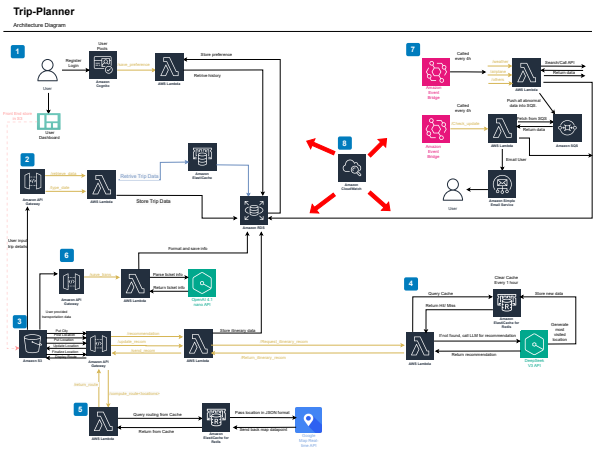

Fig. 1. System Architecture Diagram

B. Client Layer

The Client Layer is the core layer for the interaction between the system and the user. It is responsible for implementing the front-end interface for key functions such as user registration, login, itinerary viewing and planning.

- **Simple Storage Service (S3):** As a storage infrastructure for front-end code and components, we provide users with stable and reliable web access by enabling static web hosting in S3. Besided the static resource hosting platform for front-end pages and ticket information, S3 also supports tens of thousands of GET requests per second and scales automatically without pre-configured capacity. This capability ensures high availability and low-latency response of the front-end service during the initial rollout of the system or when there is a sudden increase in user access.
- **AWS Cognito:** To ensure access security, this layer is deeply integrated with Amazon Cognito to handle user authentication and permission management. By improving the degree of coupling with Cognito, the system hosts key security logic such as user credentials and

session control on the AWS cloud platform, which significantly reduces potential security risks and simplifies the authentication process. The service natively supports deep integration with API Gateway and Lambda, and has the ability to handle 50 authentication requests per second(QPS) [6] and scale on demand.

- **API Gateway:** Building on top of the AWS Cognito, the front-end JavaScript code is connected to the API Gateway in a secure way to achieve token exchange and request signing, further reducing the possibility of information leakage and unauthorized access. At the same time, as a unified entry point for front-end and back-end communication, Amazon API Gateway excels in high concurrency processing capability. It supports 10,000 requests per second by default [7], and has automatic horizontal scaling.

In summary, these three AWS services form the key foundation of the system client layer in terms of elastic expansion, concurrency support, and maintainability, significantly improving the reliability and sustainability of the overall architecture.

C. API Layer

The API Layer of this system serves as a communication bridge between the front-end and the back-end, and is responsible for receiving, parsing, and forwarding business requests from clients. It is based on a serverless architecture, has good elastic scalability and high concurrent processing capabilities, and supports the core logic distribution mechanism of the entire system.

Through the design of this layer, we have achieved a high degree of modularization of system functions. The front-end accesses system capabilities through the RESTful API without having to understand the details of the back-end logic; and the back-end services can also be independently deployed, tested, and updated, effectively improving the maintainability and scalability of the system. AWS services integrated into the system are listed below:

- **Amazon API Gateway:** As same as in the Client Layer, the API Gateway serves as the unified API access layer, responsible for routing, permission checking, request flow restriction, and calling AWS Lambda functions.
- **AWS Lambda:** Lambda functions serve as the core compute units in this layer, responsible for handling specific business logic such as `/save_preference`, `/get_trip_itinerary`, and `/itinerary_recommendation`. Each function is deployed independently, maintaining modularity and stateless execution. By default, AWS Lambda supports 1,000 concurrent executions per account [8], with the ability to scale horizontally in response to incoming request volume.

API Layer plays a key ”mediation” role in this system, providing standardized, safe and reliable interface access capabilities for the front-end, and achieving flexible execution of back-end logic through integration.

D. Data Layer

The data layer is responsible for the persistence, association and access of all structured information in the system, including user registration information, travel preferences, itinerary arrangements, location points, daily routes, ticket data and reminder status. This layer is based on the relational database as shown in Figure 2, establishes clear data dependencies, and optimizes read and write performance with the cache mechanism.

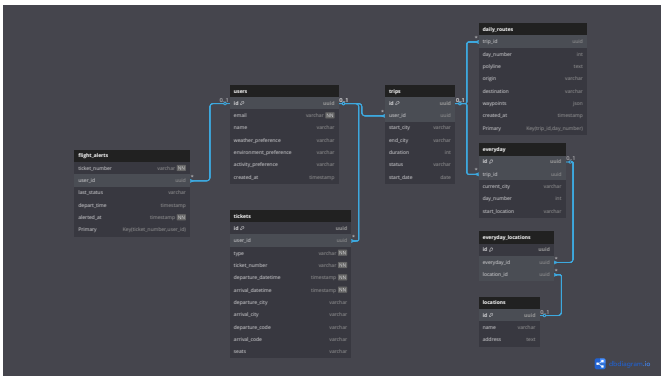

Fig. 2. Relational database schema used in the system.

- **Amazon Aurora (MySQL):** The system uses Amazon Aurora hosted MySQL instance for data storing. The system supports complex JOIN query and transaction operations. Hierarchical mapping between users and itineraries, itineraries and daily schedules, and daily schedules and locations is realized by using foreign key associations, and complex JOIN queries and transaction operations are supported.Aurora supports automatic backup, multi-zone availability deployment and performance monitoring, and has good elasticity scaling capability, which is suitable for OLTP-type typical application loads. In terms of concurrency processing, Aurora MySQL supports hundreds of thousands of queries per second (QPS) [9], much higher than traditional self-built MySQL instances.
- **Browser Cache:** In the client layer, some user preference information (e.g., weather, activity type) and static location data (e.g., location ID, address) are read quickly through the browser's local cache, which reduces repetitive requests to the server, improves the page response speed, and reduces the pressure on the backend.
- **Amazon ElastiCache:** Beside intelligent layer calls LLM to generate recommendations, it caches the recommendation results in cache to prevent the same user from repeatedly triggering the inference process in a short period of time. The cache is cleaned daily and the data is kept fresh by setting TTL (Time-To-Live), further improving the system response speed and optimizing costs.

The data layer uses Amazon Aurora to build a highly available and highly consistent relational data storage structure, combined with ElasticCache and front-end local cache

mechanism, to form the core support for the stable operation and efficient response of the system.

E. Trip Routing and Visualization Layer

The routing and visualization module is responsible for transforming user-submitted trip itineraries into interactive, map-based visualizations. Rather than relying on backend-generated static maps, our system uses polyline-based routing to reduce backend workload and maximize frontend interactivity.

When a user initiates a route generation request, the frontend sends a list of location coordinates to the backend via API Gateway. An AWS Lambda function then queries the Aurora database to check whether a polyline already exists. If not, the Lambda uses Google Directions API to compute the optimal route and stores the resulting polyline string into the database for caching and reuse.

The frontend then decodes and renders the polyline using Google Maps JavaScript API, while retrieving detailed POI data—such as photos, ratings, opening hours—from Google Places API. This approach enables a dynamic, immersive navigation experience with real-time information overlay.

To further reduce Google API calls and improve image loading speed, the system uses Amazon S3 as a caching layer for frequently accessed POI and city images. Images are fetched from Google APIs upon first access, uploaded to S3, and then served from S3 for subsequent requests. This optimization also enables CDN acceleration and cost control.

The architecture supports high concurrency by design. In a simulated 100-user concurrent test, Lambda's warm invocations responded in under 120ms on average, while polyline caching ensured that redundant computations were avoided. Overall, the routing module exemplifies scalable, event-driven architecture aligned with modern serverless design principles.

F. Intelligence Layer (LLM and Recommendation Engine)

The intelligent layer is the module responsible for core intelligent reasoning and decision-making in the system. It mainly undertakes two tasks: first, generating personalized travel recommendations based on user preferences and destinations; second, extracting structured itinerary information from images or ticket documents uploaded by users to achieve automatic itinerary identification and restoration. The following are the core functions and services used in this layer:

- **DeepSeek V3 API:** Used to generate personalized attraction recommendations based on the travel preferences and destinations entered by the user. DeepSeek is a high-performance, low-cost ($0.07 per Million Tokens) large language model inference interface suitable for high-frequency recommendation tasks [10]. The system generates complete recommendation results including attraction names, descriptions, and sorting suggestions by dynamically constructing prompts.
- **OpenAI GPT-4.1 Nano (multimodal model):** Used to process travel photos or PDF tickets uploaded by users. GPT-4.1 Nano supports mixed image and text input, has strong image understanding capabilities, can identify key information such as departure time, location, and mode of transportation, and convert it into a structured JSON format to provide input for subsequent itinerary generation modules.
- **AWS Lambda & API Gateway:** The serverless architecture achieves logical decoupling and elastic scheduling, ensuring the availability and maintainability of data get from LLM under high concurrent access conditions.

With the cost-effective model reasoning capabilities of DeepSeek, the image understanding capabilities of OpenAI's multimodal model, and the caching and serverless capabilities provided by AWS, this layer provides stable, efficient, and intelligent recommendation support for the entire system.

G. Automation and Monitoring Layer

This layer delivers timely updates on weather and flight status through an event-driven, serverless pipeline. All infrastructure is defined in a CloudFormation template to ensure reproducibility and automated deployment.

- **SQS queue:** Configured with visibility timeouts and dead-letter support, the SQS queue buffers notification tasks. Lambda consumers process messages in batches and render them into Jinja2-based HTML templates. This design smooths traffic spikes and improves system stability.
- **AWS SES:** SES sends concise, link-optimized email notifications with external asset references to reduce email size, enhance client rendering, and improve cache performance.
- **AWS Lambda & EventBridge:** Stateless Lambda functions, triggered by EventBridge, scale automatically to handle load surges. Reserved concurrency protects critical paths like email dispatch from resource contention.

Together, these components form a scalable, fault-tolerant notification system that ensures low-latency alerts with minimal operational overhead.

V. Design Decisions

To achieve an elastic, user-friendly architecture, we made targeted design choices across the client, API, LLM, and data layers.

1) Design Decisions in Client Layer: **Static Hosting vs SSR:** We use Amazon S3 for static front-end hosting instead of SSR frameworks like Next.js. Since our UI mainly interacts with APIs asynchronously and doesn't require server-side session management, static hosting offers lower latency and better availability.

Amazon Cognito vs Custom Auth: We selected Amazon Cognito over a self-built authentication system to reduce complexity and enhance scalability. Cognito supports OAuth 2.0, MFA, and native API Gateway integration, simplifying secure identity management with minimal overhead.

2) Design Decisions in API Layer: **RESTful API vs GraphQL:** REST was chosen for its structural clarity, ease of caching, and seamless integration with API Gateway. While GraphQL offers flexible field queries, it introduces higher implementation costs and complexity, which aren't justified for our relatively stable data schema.

AWS Lambda vs Kubernetes: We opted for AWS Lambda over containerized services like EKS/Fargate. Lambda's event-driven, pay-per-use model better suits our stateless workflows and reduces resource overhead while enabling fast, scalable deployment.

3) Design Decisions in Data Layer: **Aurora vs DynamoDB:** Amazon Aurora (MySQL) was chosen over DynamoDB for its strong relational model, transaction support, and JOIN capabilities. While DynamoDB excels in scalability, it lacks the relational consistency required for our multi-table dependencies (users, trips, locations).

4) Design Decisions in LLM Layer: **Amazon Q vs external LLM interface(Deepseek & OpenAI):** To achieve intelligent recommendation and bill parsing, the system compared Amazon Q with external large language model interfaces such as DeepSeek and OpenAI. Although Amazon Q has advantages in security and AWS native integration, its reasoning capabilities, Cost and multimodal support are still limited. Therefore, the system uses:

- **DeepSeek V3**: for high-frequency text recommendations, fast response and low reasoning cost;
- **OpenAI GPT-4.1 Nano**: for image and text bill parsing, supporting advanced image understanding and semantic reasoning.

Multimodal LLM vs Amazon Text Extract: In the ticket image parsing task, the system did not use traditional OCR (such as Amazon Textract), but chose a multimodal LLM that supports joint reasoning of images and text. Although OCR is suitable for simple text extraction, it has obvious limitations in complex layout, semantic understanding and structured output. The multimodal model can process visual layout and language context at the same time, improving the accuracy and robustness of information extraction, especially for document scenarios such as air tickets and train tickets with inconsistent formats.

5) Design Decisions in Routing Layer: **Polyline vs Static Maps:** We use polyline-encoded routes instead of backend-generated static images to reduce storage costs and enable dynamic, interactive map rendering on the frontend.

Google Maps vs AWS Location Service/OSM: Google Maps offers superior styling, frontend integration, and ecosystem support. AWS Location Service lacks frontend tools, while OSM's ecosystem is fragmented and less customizable.

Route Caching in Aurora: Each route is computed once via Google Directions API and stored in Aurora. Future requests retrieve the cached polyline to reduce latency and API usage.

S3 Image Caching: To avoid exceeding Google Places API quotas, POI images are fetched once, cached in S3, and served from there on future access—improving load speed and reducing cost.

Serverless Logic: AWS Lambda handles route generation on demand, ensuring scalability and cost-efficiency. This, combined with API Gateway and Aurora, supports high concurrency and clear module boundaries.

Frontend-Centric Visualization: All routing visuals are rendered client-side, supporting smooth transitions, zooming, and POI info overlays—prioritizing interactivity over static visuals.

6) Design Decisions in Automation and Monitoring Layer: **Microservice vs Monolithic:** A microservice approach was chosen over a monolithic Lambda or containerized solution to maximize agility, fault isolation, and independent deployment cycles. Serverless components eliminate the need to provision or patch servers, reduce operational overhead, and incur cost only when code executes.

Notify by SQS vs Fetch from Database: Amazon SQS was selected for its exactly-once processing and dead-letter capabilities, superior to a polling database approach which would complicate retries and error handling.

VI. Future Work

While the current system supports modular design and intelligent planning, further enhancements are needed to optimize performance under high concurrency, reduce latency, and improve global scalability. Key future directions include:

- **Routing Optimization for Scalability:** The routing module will support collaborative multi-user editing via AWS AppSync or WebSocket, along with segmented route planning by time blocks and travel mode. Cached polylines and POI images will be managed with TTL-based lifecycle policies to reduce redundant API calls under burst workloads.
- **Offline Access and Multi-Map Backend Integration:** To support diverse user conditions, the system will enable preloading of map and image data from S3 for offline use, and conditionally switch between providers (Google Maps, Mapbox, OSM) based on user location and API cost-performance.
- **Global Frontend Acceleration:** CloudFront CDN will be deployed to serve static web assets globally, improving page load time and availability for users in different regions.
- **Multi-Device UI Adaptation:** The frontend will be optimized for tablets and high-resolution devices, ensuring responsive behavior across platforms and improving accessibility.
- **Security and Deployment Automation:** Future iterations will include fine-grained access control, encrypted storage, GDPR compliance mechanisms, and CI/CD pipelines using AWS CodePipeline or GitHub Actions for secure, automated deployment.
- **Intelligent Monitoring and Resilience:** AWS DevOps Guru and CloudWatch Synthetics will be used for

anomaly detection and automated remediation. Key components will adopt Step Functions for fault-tolerant orchestration under heavy load.

These improvements aim to evolve the system into a globally deployable, low-latency, high-concurrency platform capable of delivering robust, real-time travel planning experiences.

VII. Results

TripPlanner has been validated through extensive user functionality testing across subsystems such as trip routing and LLM-based recommendation. Below we highlight major user-facing features with corresponding visual demonstrations.

A. User Interface and Sign-In Workflow

The main page (Figure 3) presents an overview of the platform and provides entry points for login or registration.

Fig. 3. Main interface of TripPlanner

Users can log in or register via the login and sign-up interfaces (Figures 4 and 5). Preferences are collected at registration.

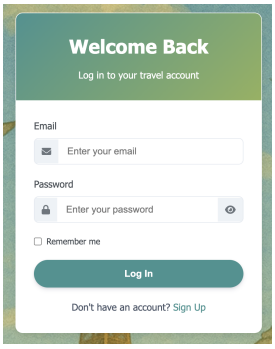

Fig. 4. User login page

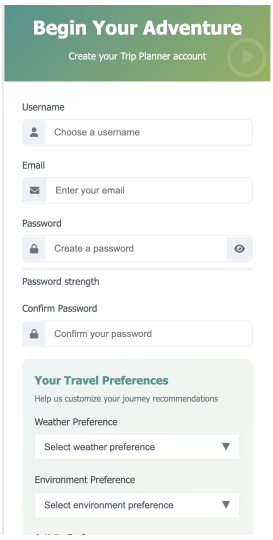

Fig. 5. User registration (sign-up) page

After login, users are directed to the dashboard to view personal info and access saved trips (Figure 6). Clicking "View" opens the trip card (Figure 7).

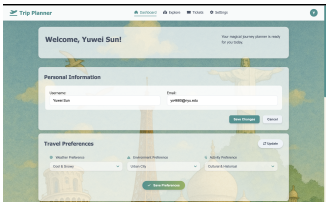

Fig. 6. User dashboard after login

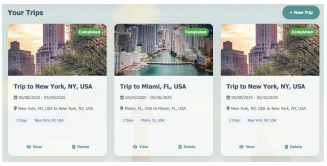

Fig. 7. Trip card preview on dashboard

B. Itinerary Planning Workflow

Users initiate trip planning by selecting a destination (Figure 8) and trip date/duration (Figure 9). The LLM engine recommends personalized locations.

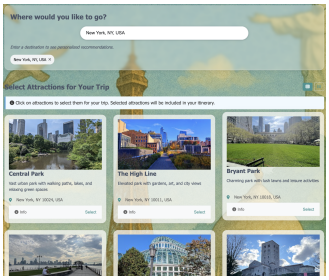

Fig. 8. Explore page with LLM-based recommendations

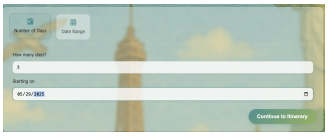

Fig. 9. Date selection page

Daily trip details can be customized using the drag-and-drop planning interface (Figure 10), and users can view final routing output (Figure 11) based on saved preferences.

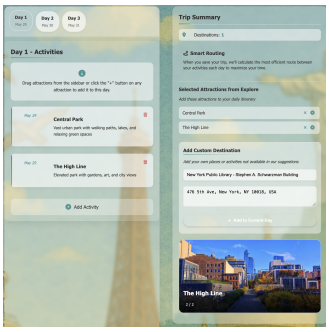

Fig. 10. Daily planning interface with drag-and-drop

Fig. 11. Generated routing view for saved trip

C. Multimodal Ticket Parsing and Notification

Users can upload travel tickets (Figure 12), which are parsed using a multimodal LLM. Notifications (Figure 13) are sent based on weather and flight data.

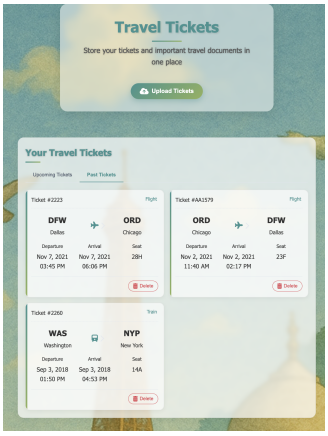

Fig. 12. Ticket upload and parsing interface

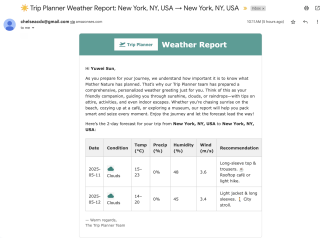

Fig. 13. Sample email notification with weather update

REFERENCES

[1] OpenTripPlanner, ”Open Source Multimodal Trip Planner,” [Online]. Available: https://www.opentripplanner.org/

[2] Trip Planner Website, ”A web application designed to assist users in planning their trips efficiently,” GitHub. [Online]. Available: https://github.com/hrshrayank/trip-planner-website

[3] AI Trip Planner, ”Full-Stack Trip Planner Web App,” GitHub. [Online]. Available: https://github.com/barika001/ai-trip-planner

[4] TripIt, “TripIt: Find your travel plans in one place,” [Online]. Available: https://www.tripit.com/web/

[5] GuideGeek, “GuideGeek is a new way to plan and book travel.,” [Online]. Available: https://guidegeek.com/

[6] Amazon Web Services, “Quotas in Amazon Cognito,” AWS Documentation, [Online]. Available: https://docs.aws.amazon.com/cognito/latest/developerguide/limits.html

[7] Amazon Web Services, “Amazon API Gateway quotas and important notes,” AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html

[8] Amazon Web Services, “AWS Lambda quotas,” AWS Documentation, 2025. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html

[9] Amazon Web Services, ”Amazon Aurora Performance,” AWS Whitepaper, 2023. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.BestPractices.html

[10] DeepSeek, “DeepSeek API Documentation and Pricing,” DeepSeek Official Docs, 2025. [Online]. Available: https://platform.deepseek.com/docs/overview