AI Trip Advisor: Cloud-Native Travel Planning Platform

Sultan Fahim, James McKim, Karan Khademi, Will Huff, Clinton Uguwanyi
Department of Computer Science, West Chester University of Pennsylvania

Abstract / Overview

The AI Trip Advisor platform represents a novel demonstration of cloud-native engineering concepts applied to the domain of intelligent travel planning. By leveraging artificial intelligence, distributed systems, and container-orchestrated microservices, the platform transforms raw user preferences into complete, personalized itineraries within seconds. Built on Kubernetes and supported by scalable backend services, the system automates itinerary generation, integrates live geocoding data, and continuously adapts to changing user needs.

From a marketing and real-world value standpoint, AI Trip Advisor showcases how cloud-native intelligence enables consumers to make faster, more informed travel decisions. Users can enter high-level travel goals, preferences, or constraints, and the platform's backend orchestrates data retrieval, itinerary synthesis, and optimization using an AI Agent Service. This architecture demonstrates how modern organizations are shifting from monolithic travel-recommendation systems to AI-driven microservices capable of real-time personalization, failure resilience, and horizontal scalability.

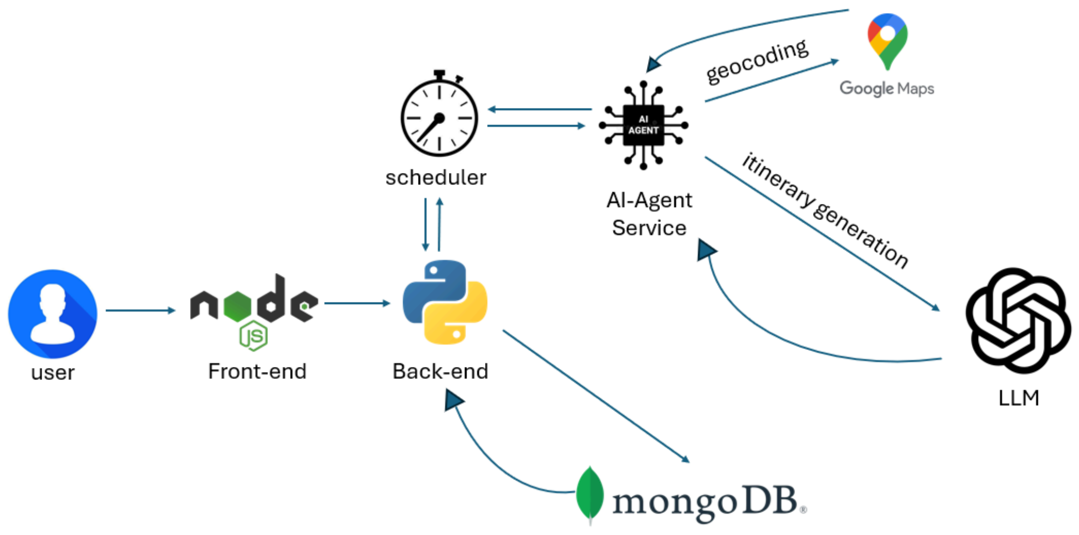
This report summarizes the system architecture, data-flow decisions, pod structure, routing logic, persistence design, and observed Kubernetes behaviors under scaling and self-healing conditions. It concludes with known limitations and future opportunities to enhance system automation, performance, and portability.

1. Architecture Recap

The system follows a microservice architecture deployed on Kubernetes, consisting of:

- **Front-end Pod (Node.js)** – Provides UI, handles user input, makes API calls to the backend, and renders itinerary results.
- **Backend Pod (Python/Flask)** – Orchestrates workflow, stores preferences, calls the AI Agent Service, and interacts with MongoDB.
- **AI Agent Service Pod** – Executes itinerary generation using LLM processing, geocoding APIs, and summarization logic.
- **MongoDB Stateful Pod** – Stores user profiles, trip requests, and generated itineraries.
- **Scheduler / Worker Logic** – (If implemented) handles asynchronous travel-plan generation.
- **Ingress / Service Mesh** – Routes traffic between pods and exposes the front-end to external users.

1.1 Updated Architecture Diagram



1.2 Architectural Choices

- **Microservices over Monolith:** Enables independent scaling and faster deployment cycles.

- **Containerization via Docker:** Ensures reliable runtime environment consistency.
- **Kubernetes Orchestration:** Provides automatic rescheduling, health checks, load balancing, and external service access.
- **Loose Coupling:** Each component communicates via well-defined REST endpoints.

---

2. Data-Flow Diagram & System Workflow

The system's operational workflow:

1. **User Input:** User submits destination, interests, travel dates, budget, and constraints through the Node.js front-end.
2. **Backend Validation:** Backend pod receives input, logs the request, and writes the initial entry to MongoDB.
3. **AI Agent Invocation:** Backend forwards structured preferences to the AI Agent Service.
4. **Geocoding + LLM Planning:**
    - The AI Agent retrieves location metadata via Google Maps APIs.
    - The LLM generates structured itineraries.
    - Additional services refine and segment daily activities.
5. **Response Assembly:** The AI service returns a structured itinerary object to the backend.
6. **Persistence Step:** Backend stores the finalized itinerary in MongoDB.
7. **Front-end Rendering:** User receives a formatted, human-readable itinerary.

Key principle:
All components remain independently replaceable without affecting upstream/downstream services.

---

3. Pod Grouping & Sidecar Decisions

3.1 Pod Grouping

Pods were organized based on functional responsibility:

Pod	Primary Function	Rationale
Frontend Pod	UI rendering, user authentication, request forwarding	High-traffic component; benefits from horizontal scaling
Backend Pod	Orchestration, API gateway, service routing	Acts as system “brain”
AI Agent Pod	LLM calls, geocoding, itinerary generation	CPU-intensive logic isolated to avoid blocking other services
MongoDB Pod	Persistent data storage	Requires StatefulSet behavior

3.2 Sidecar Decisions

No sidecars were used in the final version, but we considered two:

- **Logging Sidecar (Fluentd):** Offloads log aggregation
- **Caching Sidecar (Redis):** For repeated API responses

We opted not to implement them to preserve minimal complexity and ensure deterministic pod behavior for class demonstration.

---

4. Services, Ingress, and Routing

4.1 Internal Services

- **ClusterIP** for backend ↔ AI Agent communication
- **ClusterIP** for backend ↔ MongoDB

- **NodePort / Ingress** to expose the front-end externally

4.2 Routing Logic

Traffic flow:

1. External request → Ingress
2. Ingress → NodePort Front-end service
3. Front-end → Backend ClusterIP service
4. Backend → AI Agent ClusterIP service
5. Backend ↔ MongoDB Stateful service

4.3 Resilience Features

- Kubernetes restarts unhealthy pods via **liveness/readiness probes**
- DNS-based pod discovery ensures stable service access despite pod turnover
- Load-balancing across replica sets ensures stable request flow under load

---

5. Scaling and Self-Healing Behavior

5.1 Horizontal Scaling Tests

We increased load using concurrent front-end requests. Observations:

- **Frontend scaled efficiently** with multiple replicas; CPU usage flattened across pods.
- **Backend scaling helped latency**, especially under heavy itinerary generation.
- **AI Agent was a bottleneck** under large LLM requests; adding replicas significantly improved throughput.

5.2 Self-Healing Observations

Tests included:

- Manually deleting a running pod
- Artificially failing liveness probes
- Flooding pods with excess traffic

Results:

- Kubernetes automatically restarted terminated pods within seconds
- DNS routing seamlessly redirected requests to healthy pods
- Stateful MongoDB recovered without data loss due to persistent volume claims

These behaviors demonstrate Kubernetes' built-in reliability for production workloads.

---

6. Persistence Design & Verification

6.1 MongoDB Schema Design

Core collections:

- **users** – id, name, preferences
- **requests** – raw travel preferences
- **itineraries** – final AI-generated itineraries

6.2 Verification Steps

To ensure data integrity:

- Inserted test records using the backend API

- Verified per-request round-trip through MongoDB
- Ensured itinerary retrieval remained functional after pod restarts
- Simulated network partition by deleting the MongoDB pod

6.3 Results

- PersistentVolumeClaim successfully preserved all state
- Backend reconnection logic worked as intended
- Database remained accessible across pod rescheduling events

---

7. Known Limitations

1. **LLM latency** may slow itinerary generation under high load.
2. **No rate-limiting** for external APIs.
3. **No Redis caching layer** means repeated geocoding increases latency and cost.
4. **Minimal authentication**—future deployments require secure user accounts.
5. **Frontend styling** is functional but not production-grade.
6. **Scheduler not fully implemented** (optional component).

---

8. Future Work

- Add a Redis or Memcached caching sidecar
- Implement OAuth authentication for multi-user access

- Add continuous deployment pipelines (GitHub Actions → ArgoCD)
- Expand itinerary features (transportation, hotels, pricing estimates)
- Integrate tracing (Jaeger, OpenTelemetry)
- Deploy autoscaling rules based on custom CPU/memory thresholds
- Improve UI/UX design for mobile-friendly travel planning