Airspace Copilot: Real-Time Multi-Agent Flight Monitoring System

Technical Report

Course: Agentic AI - Assignment 3
Date: November 2025

1. Introduction & Problem Statement

1.1 Background

The aviation industry generates massive amounts of real-time data through systems like the OpenSky Network, which tracks thousands of flights globally. However, this data remains largely inaccessible to both operational staff and individual travelers in an actionable, intelligent format. Operations personnel need quick situational awareness of airspace anomalies, while travelers want simple, natural-language updates about their specific flights.

1.2 Problem Statement

Current flight tracking solutions face several limitations:

1. **Data Overload**: Raw flight data is overwhelming and requires domain expertise to interpret
2. **Lack of Intelligence**: Most systems display data without analysis or anomaly detection
3. **Poor Accessibility**: Complex interfaces unsuitable for non-technical users
4. **No Personalization**: Generic displays don't cater to individual flight tracking needs
5. **Limited Proactivity**: Systems react to queries rather than providing intelligent insights

1.3 Gap Analysis

The gap exists between **data availability** and **actionable intelligence**. While real-time flight data is abundant, there's no system that:

- Intelligently analyzes airspace for operational anomalies
- Provides natural language explanations suitable for travelers
- Offers both operational and personal perspectives
- Uses AI agents to provide context-aware insights

1.4 Proposed Solution

The **Airspace Copilot** addresses these gaps through a multi-agent AI system with two specialized interfaces:

1. **Airspace Operations Copilot**: For operations staff to monitor regional airspace, detect anomalies, and receive AI-generated summaries
2. **Personal Flight Watchdog**: For travelers to track specific flights and interact via natural language chatbot

1.5 Why This System Matters

For Operations Teams:

- Quick identification of anomalous flights requiring attention
- AI-generated summaries reduce cognitive load
- Proactive anomaly detection prevents issues from escalating

For Travelers:

- Plain-language flight updates without technical jargon
- Conversational interface for asking questions
- Peace of mind through real-time, intelligent monitoring

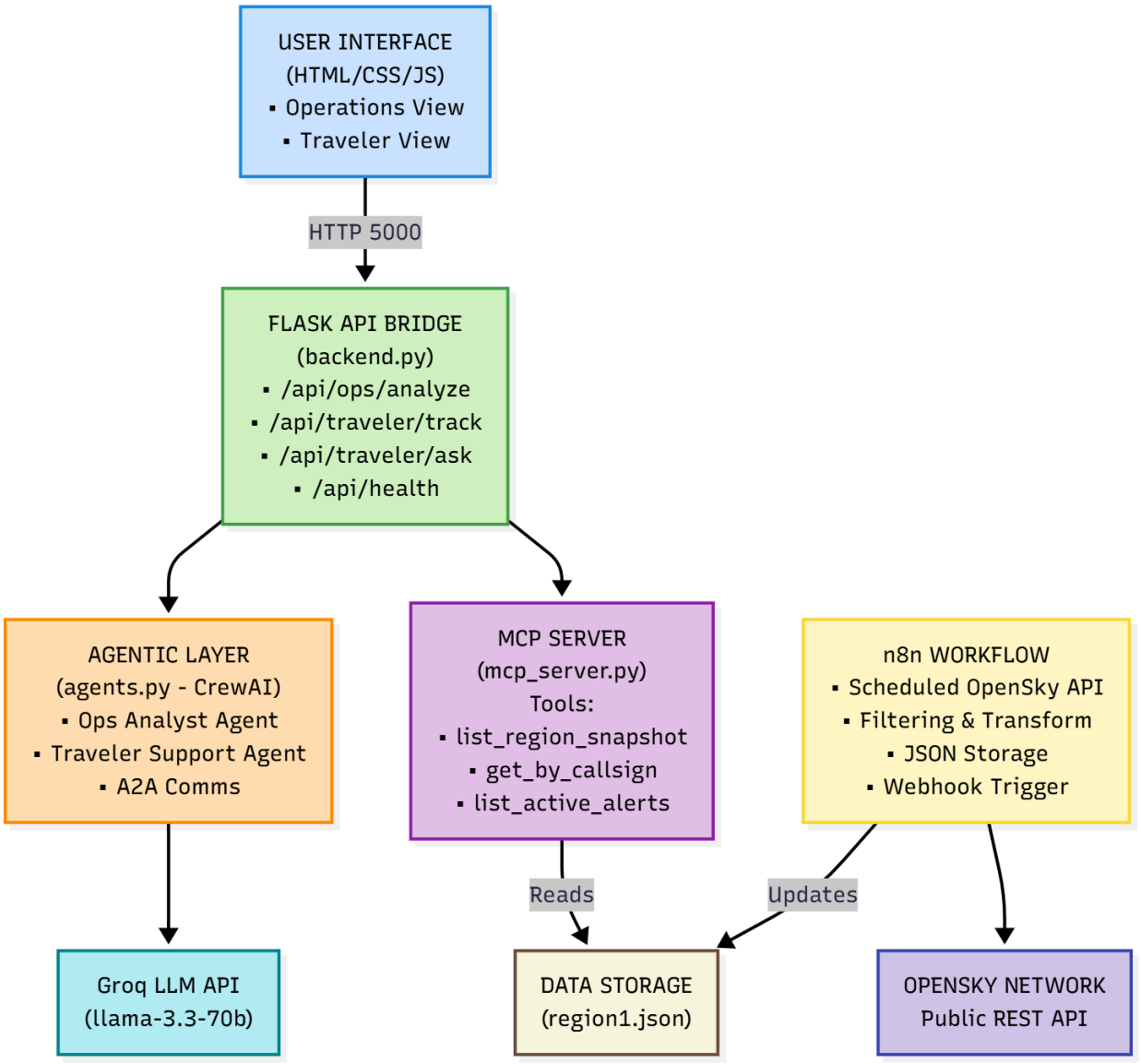
For the Industry:

- Demonstrates practical application of agentic AI in aviation
- Reduces manual monitoring workload
- Improves response times to potential issues

2. System Architecture

2.1 High-Level Architecture

The system follows a layered architecture with clear separation of concerns:



2.2 Component Interactions

Data Flow (Operations Mode):

1. User clicks "Analyze Airspace" so Frontend sends POST to Flask **`/api/ops/analyze`**
2. Flask calls **`run_ops_mode()`** in agents.py

3. Ops Analyst Agent calls MCP **`list_region_snapshot()`** tool
4. MCP reads region1.json and returns flight data with anomaly labels
5. Agent sends data + context to Groq LLM for analysis
6. LLM generates natural language summary
7. Response flows back through Flask to Frontend
8. UI displays statistics, AI summary, and flight table

Data Flow (Traveler Mode with A2A):

1. User tracks flight so Frontend sends POST to Flask **`/api/traveler/track`**
2. Flask calls `run_traveler_mode(flight_id)` in agents.py
3. Traveler Support Agent calls MCP **`get_by_callsign()`** tool
4. Agent sends flight data to Groq LLM for user-friendly summary
5. User asks: "Are there issues with other flights?"
6. **A2A Trigger**: Traveler Agent detects need for regional context
7. Traveler Agent calls **`OpsAnalystAgent.get_alerts_summary()`**
8. Ops Agent calls MCP **`list_active_alerts()`** tool
9. Both agents' responses combined in final answer
10. Response flows back to Frontend chat interface

2.3 Technology Stack

Layer	Technology	Purpose
Data Ingestion	n8n (Docker)	Scheduled API calls, data transformation
Data Access	FastAPI	MCP server exposing tools
Intelligence	CrewAI and Groq	Multi-agent system with LLM reasoning
Application	Flask	REST API bridge
Presentation	HTML/CSS/JS	User interface
Storage	JSON file	Flight data snapshots

2.4 Ports & Endpoints

- **5678**: n8n Web UI
- **8000**: MCP Server (FastAPI)
- **5000**: Flask Backend API
- **File**: index.html (opened in browser)

2.5 Component Responsibilities

The system comprises seven major components with specific responsibilities:

Frontend Interface: Provides user interaction through a single-page web application built with HTML, CSS, and JavaScript. Uses Tailwind CSS for responsive design and Lucide icons for visual elements. Communicates with the backend via REST API calls.

Flask API Bridge: Acts as the application gateway, routing HTTP requests to appropriate agent functions. Runs on port 5000 and provides endpoints for operations analysis, flight tracking, and question answering. Implements CORS support for browser-based access.

Agent Layer: Contains two AI agents implemented using CrewAI framework principles. The Operations Analyst Agent monitors regional airspace and detects anomalies. The Traveler Support Agent tracks specific flights and answers questions. Both agents integrate with Groq's llama-3.3-70b-versatile model for natural language processing.

MCP Server: Implements the Model Context Protocol specification using FastAPI. Exposes three tools for listing regional flights, retrieving specific flight data, and listing active alerts. Runs on port 8000 and includes built-in anomaly detection logic.

Data Storage: Maintains flight snapshots in JSON format at a known file location. Updated every sixty seconds by the n8n workflow. Contains timestamp metadata and arrays of flight state vectors.

n8n Workflow: Orchestrates data ingestion through a series of nodes including schedule triggers, HTTP requests to OpenSky, data transformation functions, and file writing operations. Runs in a Docker container on port 5678.

OpenSky Network API: External data source providing real-time flight state vectors for aircraft worldwide. Accessed anonymously with rate limiting constraints.

3. n8n Workflows and Data Storage

3.1 Workflow Architecture

The n8n workflow implements a data ingestion pipeline that fetches, processes, and stores flight data from the OpenSky Network. The workflow consists of five interconnected nodes that execute in sequence every sixty seconds.

3.2 Workflow Nodes

Schedule Trigger Node: This node initiates the workflow automatically using a cron schedule configured for sixty-second intervals. It ensures continuous data refresh without manual

intervention. The trigger can be temporarily disabled for maintenance or during API rate limit periods.

HTTP Request Node: Configured to perform GET requests to the OpenSky Network API endpoint. The node includes optional query parameters for geographic bounding boxes to filter flights by region. A ten-second timeout is configured to prevent hanging requests. Error handling is enabled with "Continue On Fail" to ensure workflow resilience during API downtime.

Function Node: This JavaScript function node processes the raw API response. It extracts the states array from the response and maps each state vector to a structured object. The transformation includes trimming whitespace from callsigns, normalizing null values, and selecting relevant fields such as ICAO24 identifier, callsign, country of origin, position timestamps, coordinates, altitude measurements, ground status, velocity, heading, and vertical rate. The processed data is wrapped in a timestamped snapshot object.

File Write Node: Writes the processed snapshot to the shared volume at the path accessible to other components. The write mode is set to overwrite, ensuring only the latest snapshot is maintained. JSON formatting is enabled with pretty-printing for human readability during debugging.

Webhook Node: Provides an optional manual trigger endpoint for testing and demonstrations. Accessible via HTTP GET request, this node allows operators to force an immediate data fetch outside the scheduled intervals.

3.3 Data Storage Structure

Flight snapshots are stored as JSON files in the n8n shared directory. Each file contains two top-level keys: timestamp and snapshot. The timestamp field records the Unix epoch time in milliseconds when the data was fetched. The snapshot field contains an array of flight objects.

Each flight object includes the ICAO24 transponder code which serves as a unique identifier, the callsign which may be null for flights without filed identifiers, the origin country, position and contact timestamps, longitude and latitude coordinates, barometric and geometric altitude measurements in meters, a ground status boolean, velocity in meters per second, true track heading in degrees, and vertical rate in meters per second.

3.4 Snapshot Management

The system maintains only the most recent snapshot rather than historical data. This design decision optimizes storage and simplifies agent queries while still providing real-time monitoring capabilities. The timestamp field enables the UI to display data freshness and detect stale snapshots during API outages.

When the OpenSky API fails due to rate limiting or connectivity issues, the n8n workflow continues operation but does not overwrite the existing file. This ensures the MCP server

always has access to the last known good data. The UI displays a timestamp indicating when data was last successfully updated.

3.5 Error Handling Strategy

The workflow implements multiple layers of error handling. The HTTP request node has "Continue On Fail" enabled, allowing the workflow to complete even when API calls timeout. The function node includes null checks and safe navigation to handle missing fields in API responses. The file write node has retry logic to handle temporary filesystem issues.

If a workflow execution fails completely, the previous snapshot remains available in storage. This graceful degradation ensures the system remains operational during external API issues, a critical requirement given OpenSky's rate limiting constraints on anonymous access.

4. MCP Server and Tools

4.1 Model Context Protocol Implementation

The MCP server implements a standardized protocol for exposing flight data as tools that AI agents can invoke. Built with FastAPI and served via Uvicorn, the server runs on port 8000 and provides a RESTful interface following MCP specifications. The server acts as an abstraction layer between raw data storage and intelligent agents, ensuring agents query data through well-defined tools rather than direct file access.

4.2 Tool Design Philosophy

The three exposed tools follow a hierarchical information model. The regional snapshot tool provides broad situational awareness of all flights in an area. The callsign lookup tool enables focused queries on specific flights. The alerts tool filters for anomalous situations requiring attention. This design allows agents to efficiently navigate from general overview to specific details based on user queries.

4.3 Tool One: List Region Snapshot

This tool returns all flights currently detected in a specified region along with computed anomaly labels. The tool accepts a region name parameter, reads the latest snapshot file, applies anomaly detection logic to each flight, and returns a structured response containing the timestamp and array of enriched flight objects.

The response includes all fields from the original snapshot plus an additional anomaly label field. Normal flights have null labels while anomalous flights display descriptive messages such as "Anomaly: Low speed at high altitude" or "Anomaly: Rapid vertical change". This enrichment happens at query time, ensuring labels reflect current rule sets without reprocessing stored data.

4.4 Tool Two: Get By Callsign

This tool retrieves detailed information for a specific flight identified by callsign or ICAO24 code. The tool performs case-insensitive matching on both the callsign field and ICAO24 identifier to maximize match success. If no flight matches the provided identifier, the tool returns an HTTP 404 error with a descriptive message.

The response includes all available fields for the matched flight plus the computed anomaly label. This single-flight focus enables agents to provide detailed answers to traveler questions without processing entire regional datasets.

4.5 Tool Three: List Active Alerts

This tool filters the regional snapshot to return only flights currently flagged with anomalies. The tool applies the same anomaly detection logic as the regional snapshot tool but excludes normal flights from the response. Each alert object includes the flight identifier, position coordinates, anomaly label, and a details string summarizing key metrics.

This tool enables the Operations Analyst Agent to quickly identify priority situations without manually filtering the full flight list. It also supports the Traveler Support Agent when answering questions about nearby flight issues through agent-to-agent communication.

4.6 Anomaly Detection Rules

The MCP server implements three rule-based anomaly detectors that identify potentially unusual flight situations:

Low Speed at High Altitude: Triggered when a flight maintains velocity below forty meters per second while above three thousand meters altitude. This pattern may indicate engine problems, stall conditions, or flights in holding patterns that warrant attention.

Rapid Vertical Change: Activated when absolute vertical rate exceeds fifteen meters per second in either direction. Such rates suggest emergency descents, aborted takeoffs, or go-arounds which operations teams should monitor closely.

Stationary at Low Altitude: Flagged when a flight shows velocity below five meters per second while below three hundred meters altitude. This pattern identifies aircraft that may be grounded, experiencing runway issues, or in emergency situations.

The rules use safe numeric conversion functions to handle null or invalid data gracefully, ensuring the server remains stable even with incomplete API responses.

4.7 Error Handling and Resilience

The MCP server implements comprehensive error handling at multiple levels. File reading operations include existence checks and exception catching, returning empty datasets rather than crashing when files are unavailable. JSON parsing includes validation of expected structure, with fallback to safe defaults if the format differs from expectations.

Individual flight field access uses safe conversion functions that return zero for null or invalid values, preventing computation errors. HTTP responses follow standard status codes with 200 for success, 404 for not found resources, and 500 for server errors. Each error response includes descriptive messages to aid debugging.

5. Agentic Layer: Agents, Prompts, and A2A Communication

5.1 Multi-Agent Design

The system implements two specialized agents following CrewAI framework principles. Each agent has a distinct role, goal, and set of capabilities tailored to its use case. Rather than a single general-purpose agent, this specialization enables more focused prompts and better performance on domain-specific tasks.

5.2 Operations Analyst Agent

Role and Responsibilities: The Operations Analyst Agent serves as an expert in airspace operations and safety analysis. Its primary goal is monitoring regional airspace, detecting anomalies, and providing actionable summaries for operations personnel.

Data Access: This agent calls the MCP list region snapshot tool to retrieve all flights in a specified region. It receives enriched data including anomaly labels already computed by the MCP server, allowing it to focus on analysis rather than detection logic.

Analysis Process: The agent first tallies statistics including total flights and count of anomalous flights. It then identifies the most critical anomaly based on severity rankings, with stationary situations at low altitude considered most urgent. The agent constructs a detailed prompt for the language model containing the flight count, anomaly count, and details of flagged flights.

Prompt Structure: The prompt instructs the language model to act as an aviation operations expert analyzing airspace status. It provides the raw data and requests a concise summary including overall status, critical anomalies requiring attention, and recommended actions. The response is constrained to under two hundred words to ensure readability.

Output Format: The agent returns a structured response containing the region name, timestamp, statistics, and natural language analysis generated by Groq. The raw data is also included for frontend visualization in tabular format.

5.3 Traveler Support Agent

Role and Responsibilities: The Traveler Support Agent specializes in assisting travelers with flight tracking and questions. Its goal is providing friendly, clear updates in plain language suitable for non-technical users.

Data Access: This agent primarily uses the MCP get by callsign tool to retrieve specific flight information. When questions involve nearby flights or regional context, it can also invoke the Operations Analyst Agent through agent-to-agent communication.

Query Processing: When tracking a flight, the agent retrieves current data and generates a natural language summary describing location, altitude, speed, and status. When answering questions, it analyzes the query to determine required information and constructs an appropriate prompt.

Prompt Structure: Prompts instruct the language model to adopt a friendly, helpful tone appropriate for travelers. The prompt includes the full flight data and the user's specific question. The model is asked to provide clear answers based on available data, using plain language and avoiding technical jargon. Responses are constrained to under one hundred fifty words.

Status Checking: The agent includes a dedicated function to check for flight issues by examining the anomaly label. If an anomaly exists, it provides a cautionary message. If the flight appears normal, it provides reassurance.

5.4 Agent-to-Agent Communication

The system implements A2A communication enabling agents to collaborate on complex queries. This is demonstrated when a traveler asks about nearby flights experiencing issues.

Triggering Logic: The Traveler Support Agent analyzes incoming questions for keywords such as "other flights", "nearby", "around", or "near". When detected, it recognizes the query requires broader regional context beyond single-flight data.

Communication Flow: The Traveler Support Agent invokes the Operations Analyst Agent's get alerts summary method, which retrieves active alerts from the MCP server and generates a natural language summary. This summary is then appended to the Traveler Support Agent's prompt as additional context.

Prompt Enhancement: The enhanced prompt informs the language model that regional context has been provided by another agent. The model can then answer questions about nearby issues using both the specific flight data and the broader operational picture.

Example Scenario: When a traveler asks "Are there any other flights nearby having issues?", the Traveler Support Agent first retrieves data for the tracked flight, then calls the Operations Analyst Agent for regional alerts, and finally combines both information sources to generate a comprehensive answer mentioning both the traveler's flight status and any nearby anomalies.

5.5 Groq LLM Integration

Both agents integrate with Groq's API using the llama-3.3-70b-versatile model. This model was selected for its strong performance on analytical and conversational tasks combined with fast inference times suitable for real-time applications.

API Configuration: Requests use a temperature setting of 0.3 to balance consistency with appropriate variation. Maximum token limits are set to 400-500 for most queries to ensure concise responses. System messages establish the agent's expertise domain and persona.

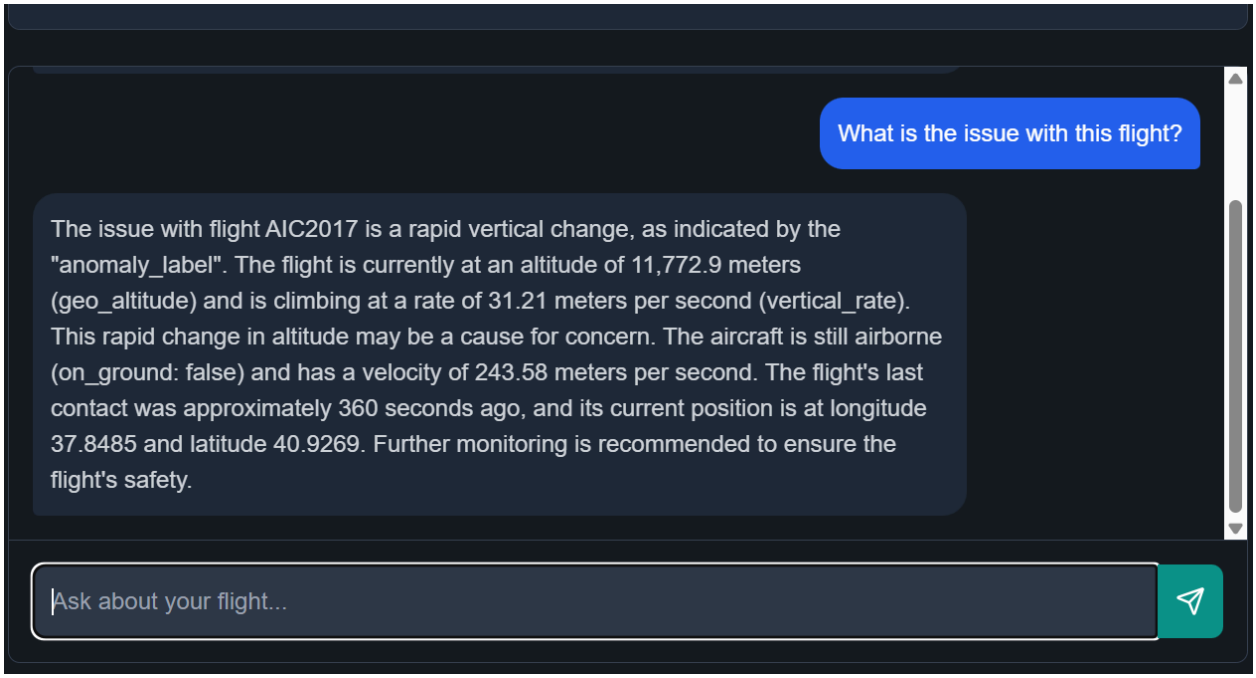
Prompt Engineering: Prompts follow a consistent structure with system context establishing the agent role, user context providing relevant data, and clear instructions for response format. Prompts explicitly request plain language, specify word limits, and ask for specific information categories.

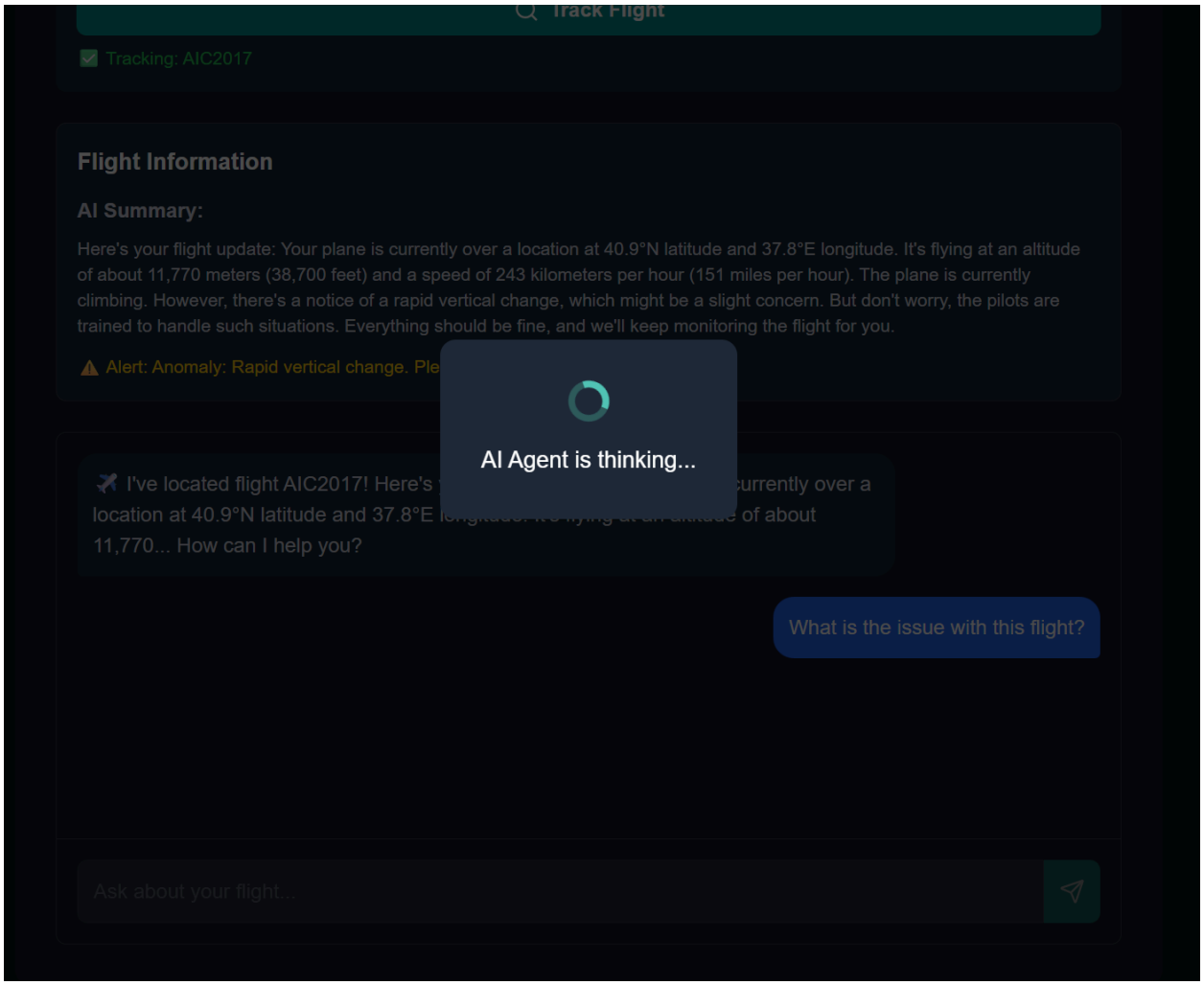
Error Handling: The integration includes try-catch blocks to handle API timeouts, rate limits, and network errors. When the Groq API fails, agents return fallback messages rather than crashing, ensuring system resilience.

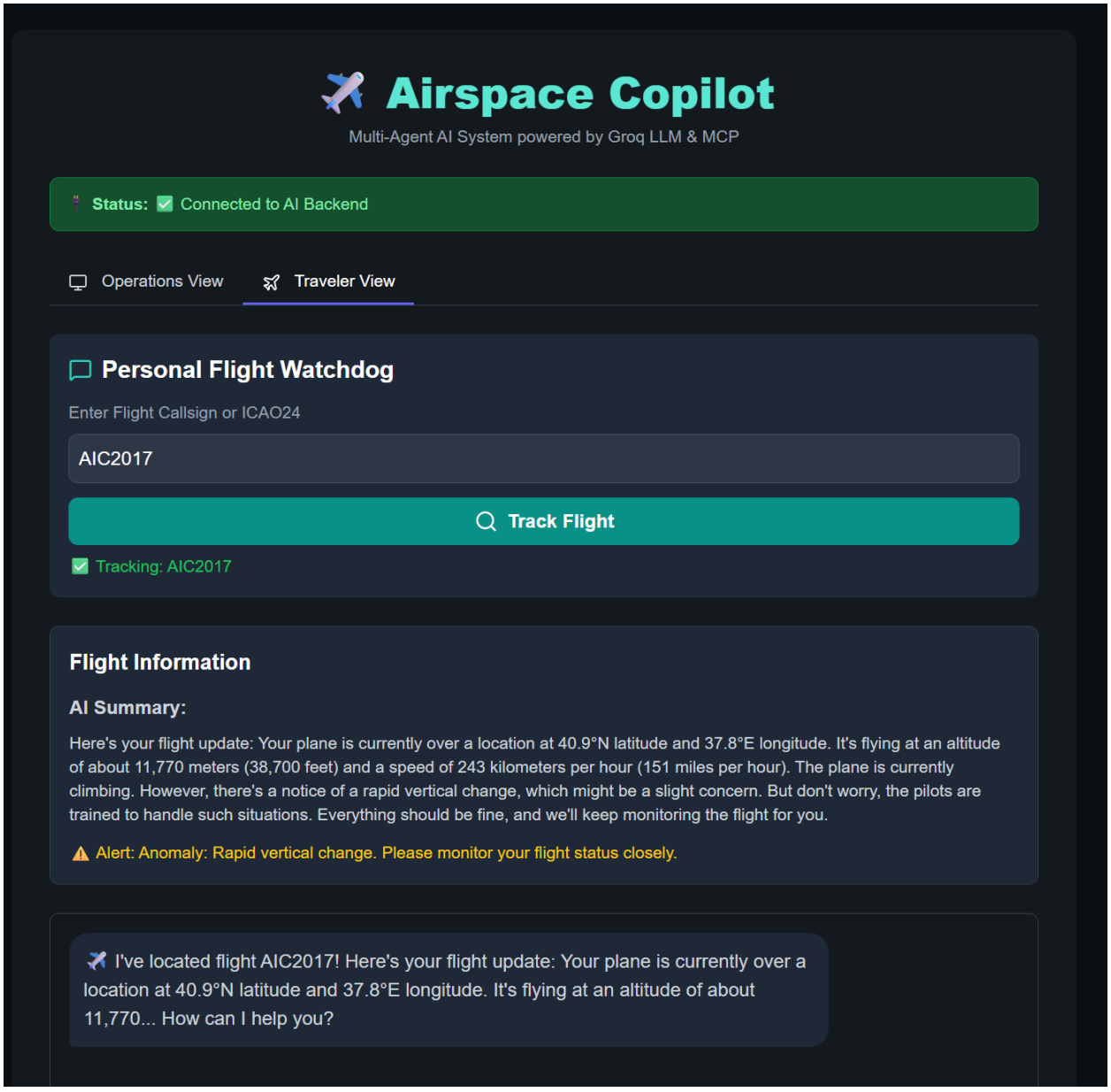
6. UI Design and User Journey

6.1 Interface Overview

The user interface is a single-page web application featuring a dark theme optimized for operational environments. The design uses a card-based layout with clear visual hierarchy and smooth transitions between states. The interface provides two primary modes accessible via tab navigation: Operations View and Traveler View.


What is the issue with this flight?
The issue with flight AIC2017 is a rapid vertical change, as indicated by the "anomaly_label". The flight is currently at an altitude of 11,772.9 meters (geo_altitude) and is climbing at a rate of 31.21 meters per second (vertical_rate). This rapid change in altitude may be a cause for concern. The aircraft is still airborne (on_ground: false) and has a velocity of 243.58 meters per second. The flight's last contact was approximately 360 seconds ago, and its current position is at longitude 37.8485 and latitude 40.9269. Further monitoring is recommended to ensure the flight's safety.
Ask about your flight...

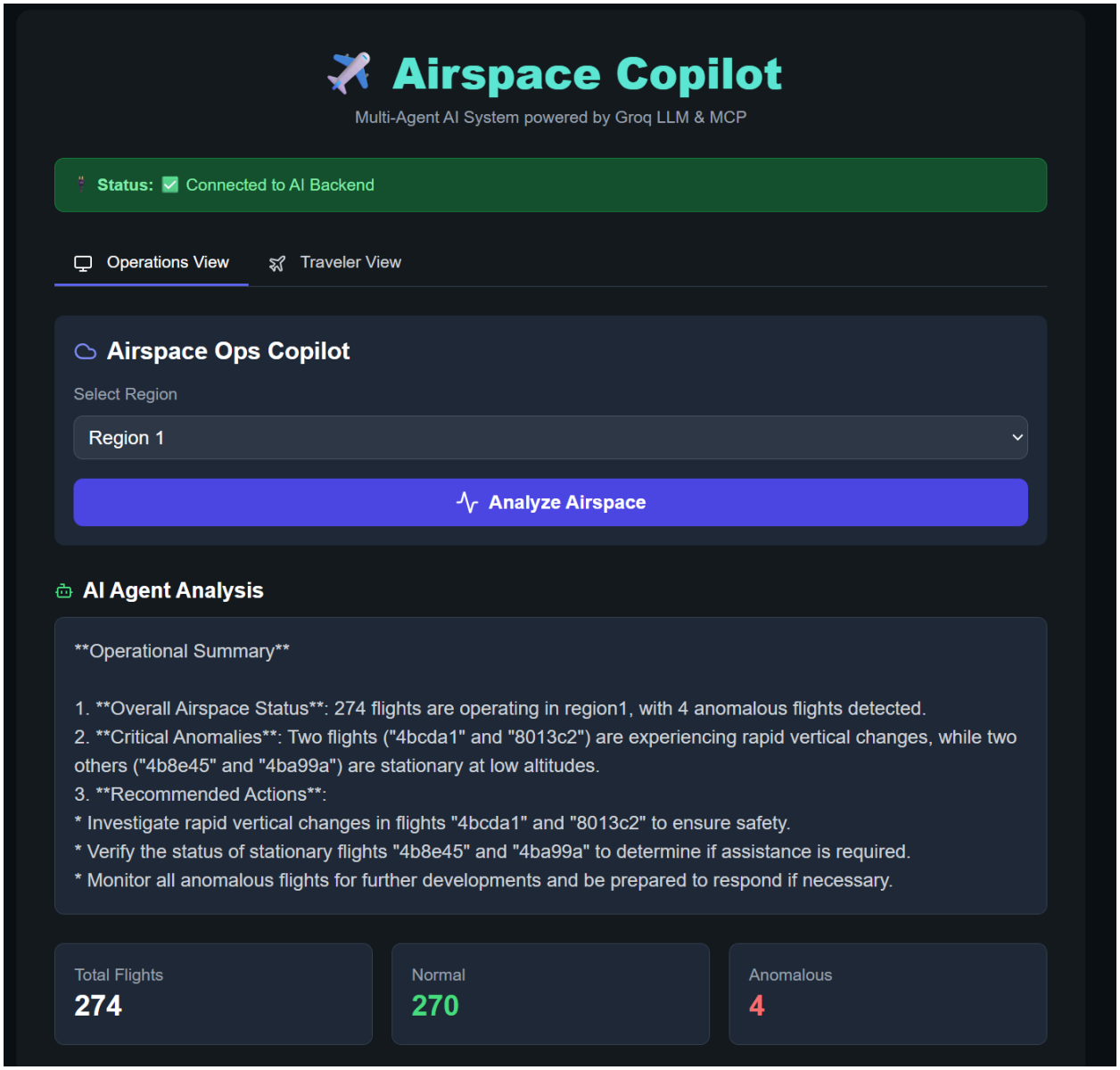

Track Flight
Tracking: AIC2017
Flight Information
AI Summary:
Here's your flight update: Your plane is currently over a location at 40.9°N latitude and 37.8°E longitude. It's flying at an altitude of about 11,770 meters (38,700 feet) and a speed of 243 kilometers per hour (151 miles per hour). The plane is currently climbing. However, there's a notice of a rapid vertical change, which might be a slight concern. But don't worry, the pilots are trained to handle such situations. Everything should be fine, and we'll keep monitoring the flight for you.
Alert: Anomaly: Rapid vertical change. Ple
AI Agent is thinking...
I've located flight AIC2017! Here's
currently over a
location at 40.9°N latitude and 37.8°E
of about
11,770... How can I help you?
What is the issue with this flight?
Ask about your flight...



ISR558	501c08	12131.04	264.98	Normal
SXS39M	4bcda1	8930.64	246.16	Anomaly: Rapid vertical change
HVN50	8880e0	11460.48	265.31	Normal
MEA212	748051	12390.12	250.45	Normal
FDB1450	8965b1	12702.54	249.12	Normal
SXS4BK	4bce0a	11109.96	221.38	Normal
AFL2141	15202b	10119.36	235.32	Normal

Active Flights

CALLSIGN	ICAO24	ALTITUDE (M)	SPEED (M/S)	STATUS
MBU6238	51116f	12062.46	223.87	Normal
PGT450J	4bc8c8	10721.34	229.58	Normal
PGT822A	4bc8c9	10187.94	237.49	Normal
PGT3PW	4bc8c3	861.06	92.72	Normal
CFG433	511171	11201.4	222.76	Normal
PGT1924	4bc8d5	10866.12	226.02	Normal
PGT533G	4bc8d3	5692.14	191.14	Normal
PGT7XE	4bc8d2	10728.96	218.7	Normal
BAW660	408134	10888.98	240.11	Normal
PGT3DQ	4bc8cb	10759.44	258.85	Normal
AEE52R	46b825	8138.16	210.86	Normal
TBJ25	a15d58	14470.38	231.08	Normal
WMT7UG	471f62	11170.92	227.95	Normal
WMT153	471f71	205.74	64.31	Normal
DLH714	3c6714	12031.98	264.22	Normal
WMT516	471f03	198.12	67.78	Normal
DLH713	3c670c	12306.3	242.62	Normal
DLH716	3c4b30	9563.1	254.8	Normal
DLH630	3c656c	12123.42	265.4	Normal



6.2 Design Principles

Visual Hierarchy: The interface uses size, color, and spacing to establish importance. Critical alerts use red highlighting, normal status uses green, and informational elements use neutral grays. Headings and icons provide clear section delineation.

Responsive Design: Built with Tailwind CSS utility classes, the interface adapts to different screen sizes. Card layouts stack vertically on mobile devices while maintaining horizontal arrangements on larger screens.

Real-Time Feedback: The interface provides immediate visual feedback for all user actions. Button clicks show loading states, API calls display animated spinners with descriptive messages, and results appear with smooth fade-in animations.

Status Awareness: A persistent banner at the top displays connection status to the backend, changing color from blue (checking) to green (connected) to red (offline). This ensures users always know if the system is operational.

6.3 Operations View User Journey

Initial State: Users see the Operations View by default upon opening the application. The view displays a region selector dropdown with three predefined options, an analyze button, an empty summary panel with placeholder text, statistics cards showing dashes, and an empty flight table.

Triggering Analysis: Users select a region from the dropdown and click the "Analyze Airspace" button. The interface immediately displays a loading overlay with an animated spinner and message "AI Ops Agent is analyzing airspace". The analyze button becomes disabled to prevent duplicate requests.

Receiving Results: After three to five seconds, results populate the interface. The summary panel displays the AI-generated analysis with natural formatting. Statistics cards update to show total flights, normal flights, and anomalous flights with appropriate color coding. The flight table populates with rows for each detected aircraft.

Flight Table Details: Each row displays callsign, ICAO24 identifier, altitude in meters, speed in meters per second, and status. Normal flights show green "Normal" text while anomalous flights display red bold text with specific anomaly descriptions. Hovering over rows provides slight highlighting for improved readability.

Interpreting Analysis: Users read the natural language summary which explains overall airspace status, highlights critical anomalies by callsign, and provides recommended actions. The summary uses plain language appropriate for operations personnel while maintaining technical accuracy.

6.4 Traveler View User Journey

Initial State: Users navigate to Traveler View via the tab button. The interface displays an input field for flight identifier, a track flight button, an empty chat area with an initial greeting from the AI watchdog, and a disabled message input field.

Entering Flight ID: Users obtain a flight identifier from the Operations View table or external sources. They enter the callsign or ICAO24 code into the input field and click "Track Flight". The interface shows a loading overlay with message "AI Traveler Agent is tracking your flight".

Flight Found: If the flight exists, the interface displays several updates. A green status message confirms tracking with the flight identifier. A new flight information panel appears showing the AI-generated summary, current metrics, and status indicator. The chat area updates with a personalized greeting mentioning the flight. The message input field becomes enabled for questions.

Flight Not Found: If the flight identifier doesn't match any current flights, the interface displays a red error status, shows an error message suggesting the user check the identifier, and keeps the chat input disabled. Users can modify the identifier and retry.

Asking Questions: With a flight tracked, users type natural language questions into the message input. Questions can ask about altitude, speed, location, status, or potential issues. Pressing Enter or clicking the send button submits the question.

Receiving Answers: User questions appear as blue bubbles on the right side of the chat area. After two to four seconds, the AI agent's response appears as a gray bubble on the left. Answers provide specific information based on current flight data in friendly, conversational language.

Agent-to-Agent Communication: When users ask about nearby flights with issues, the response takes slightly longer as the system performs A2A communication. The answer includes both information about the user's specific flight and context about regional anomalies, demonstrating coordination between agents.

6.5 Visual States and Feedback

Loading States: All asynchronous operations display a full-screen semi-transparent overlay with a spinning icon and descriptive message. This prevents users from triggering duplicate requests and clearly indicates the system is processing.

Success States: Successful operations show green text and checkmark icons. Connected status displays in the banner, tracked flights show confirmation messages, and normal flight status uses green highlighting.

Error States: Failures display red text and X icons. Connection issues show in the banner, flight not found errors appear in status text, and API failures present user-friendly error messages rather than technical details.

Empty States: Before data loads, empty states use gray placeholder text to guide users. The operations summary says "Click Analyze Airspace to start", flight tables show "No data yet", and chat areas display welcoming messages.

6.6 Accessibility Considerations

The interface uses semantic HTML elements for screen reader compatibility. Color is never the sole indicator of status - text labels accompany all color coding. Font sizes meet minimum readability standards and contrast ratios exceed WCAG guidelines for dark themes. Interactive elements have sufficient size for touch targets on mobile devices.

7. Limitations and Future Improvements

7.1 Current Limitations

OpenSky API Rate Limits: The anonymous API access is limited to approximately ten calls per minute. This constrains refresh rates and can cause temporary unavailability during high-demand periods. The system mitigates this through caching but cannot provide true real-time updates faster than sixty-second intervals.

Single Region Monitoring: The current implementation monitors one region at a time through the n8n workflow. Operations teams managing multiple airspaces must manually switch between regions or run multiple workflow instances. There is no unified view across regions.

Rule-Based Anomaly Detection: The three hardcoded anomaly rules are simplistic compared to machine learning approaches. They may produce false positives on unusual but normal operations such as search and rescue helicopters or aerobatic flights. They may miss subtle patterns that indicate developing issues.

No Historical Analysis: The system maintains only the current snapshot without historical data storage. This prevents trend analysis, pattern recognition over time, or answering questions about flight history such as "Has this flight been delayed today?"

Limited Geographic Filtering: While the n8n workflow supports bounding box parameters, the current implementation does not provide user-friendly region definition. Regions are hardcoded rather than configurable through the UI.

Stateless Sessions: The web interface does not maintain user sessions or preferences. Tracked flights and chat history reset when users refresh the page or switch tabs. There is no authentication or multi-user support.

Language Model Dependency: The system requires internet connectivity for Groq API access. All intelligent analysis depends on external LLM availability. Local model deployment is not implemented, creating a single point of failure for AI capabilities.

No Push Notifications: Users must manually refresh or check the system for updates. There are no proactive alerts for critical anomalies or changes to tracked flights. The system operates in pull mode rather than push mode.

7.2 Immediate Improvements

Multiple Region Support: Extend the n8n workflow to fetch data for multiple predefined regions in parallel. Store separate JSON files for each region. Update the UI to display a multi-region dashboard showing simultaneous status across all monitored airspaces.

Authenticated API Access: Register for OpenSky Network credentials to access higher rate limits and extended historical data. This would enable more frequent updates and historical queries without rate limit failures.

Enhanced Error Recovery: Implement exponential backoff retry logic for API failures. Add health monitoring that automatically switches to backup data sources or cached snapshots during extended outages.

User Session Management: Implement browser localStorage to persist tracked flights and preferences across page reloads. Add the ability to track multiple flights simultaneously with tabbed or listed interfaces.

7.3 Advanced Enhancements

Machine Learning Anomaly Detection: Replace rule-based detection with trained models that learn normal flight patterns. Implement unsupervised learning approaches such as isolation forests or autoencoders to identify truly unusual behavior without false positives.

Historical Data Storage: Integrate a time-series database such as InfluxDB or TimescaleDB to store all snapshots. Enable trend analysis, pattern recognition, and historical queries. Provide visualizations showing flight paths over time.

Interactive Map Visualization: Add a geographic map display using libraries such as Leaflet or Mapbox. Show real-time flight positions, paths, and anomaly locations. Enable clicking flights on the map to drill into details.

Advanced Agent Capabilities: Expand agent autonomy to include proactive monitoring and alerting. Implement agents that continuously monitor for conditions requiring human attention and push notifications when found. Add agents specialized in specific scenarios such as weather impacts or airport congestion.

Predictive Analysis: Develop models that predict flight delays, arrival times, or potential issues based on current trajectory and historical patterns. Provide proactive recommendations such as "Your connecting flight may be tight based on current arrival trajectory".

Multi-Language Support: Implement internationalization to support travelers and operations personnel worldwide. Provide language model responses in user-preferred languages while maintaining technical accuracy.

Mobile Application: Develop native iOS and Android applications with push notification capabilities. Enable travelers to receive proactive updates about their flights without manually checking the system.

Regulatory Compliance: Add features supporting aviation safety reporting requirements. Implement audit logging for anomaly detections, structured data export for regulatory submissions, and compliance reporting dashboards.

7.4 Scalability Considerations

Distributed Architecture: The current single-server implementation would not scale to enterprise use. Future versions should implement microservices architecture with separate containers for n8n, MCP server, agent processing, and API gateway. Deploy behind load balancers with horizontal scaling capabilities.

Database Migration: Move from JSON file storage to a robust database system. PostgreSQL with PostGIS extensions would support geographic queries efficiently. Implement read replicas for query scaling and write sharding for update distribution.

Caching Layer: Add Redis or Memcached between agents and the MCP server. Cache frequently accessed flight data, anomaly computations, and LLM responses to reduce redundant processing and API calls.

Rate Limiting and Throttling: Implement user-level rate limiting on the