i210277 Muhammad Talha Syed_i212710_Umar_Iftikhar

Airspace Copilot: An Agentic Flight-Monitoring System using n8n, MCP, and Groq

1. Introduction and Problem Statement

Commercial flight data is publicly available through APIs such as OpenSky, but it is exposed in a low-level, rapidly changing format that is not accessible to typical users. Operational staff and travelers have very different needs:

- **Passengers** want natural-language answers to questions such as “Where is my flight right now?” or “Is everything normal?”
- **Operations staff** need a high-level picture of a region: traffic load, potential anomalies, and which flights are most concerning at the moment.

Traditional dashboards either directly query live APIs (making them fragile and rate-limited) or require custom backend services that are tightly coupled to a single UI. On the LLM side, “agents” typically call external APIs directly, which makes it hard to introduce caching, rate-limit protection, or consistent tools shared between different agents and UIs.

This project addresses that gap by building an **agentic flight-monitoring system** with a clean separation of concerns:

1. **n8n** periodically pulls raw flight states from the OpenSky API, transforms them into region-level “snapshots,” and caches them in an internal Data Table.
2. A **Model Context Protocol (MCP) server** exposes those cached snapshots as typed tools (`flights_list_region_snapshot`, `flights_get_by_callsign`, `alerts_list_active`).
3. **Groq-hosted LLM agents** (Traveler Agent, Ops Agent, and a coordinating “Traveler+Ops” flow) call these tools instead of hitting OpenSky directly.
4. A **Streamlit UI** provides two views—Traveler and Ops—on top of the same tool layer.

The result is a modular system where:

- The **data plane** (n8n + Data Table) handles rate-limited ingestion and caching.
- The **tool plane** (MCP server) presents clean, reusable abstractions over that data.
- The **reasoning plane** (Groq agents) focuses purely on decision-making and explanations.
- The **UI plane** can be changed or extended without touching ingestion or tools.

This report describes the overall architecture, the concrete implementation of each layer, and how multi-agent (A2A) communication improves over a single-agent baseline.

---

2. System Architecture

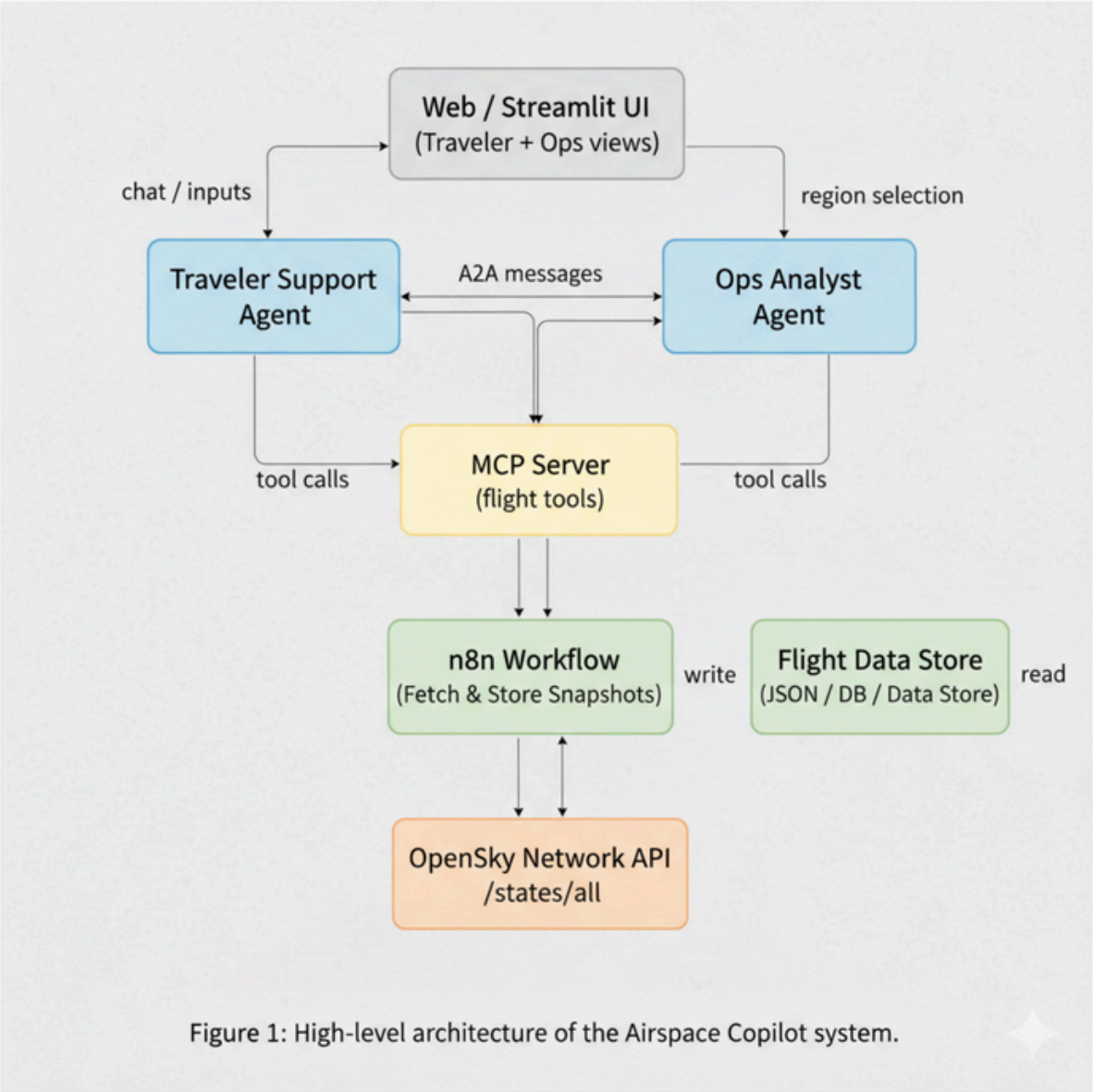

Figure 1: High-level architecture of the Airspace Copilot system.

Figure 1 summarizes the architecture as a four-layer pipeline:

1. **Data Ingestion & Caching (n8n)**
    - A scheduled workflow periodically calls the OpenSky REST endpoint for a geographic bounding box (Region 1).
    - The raw `states` array is transformed into a structured `snapshot` object and upserted into an n8n Data Table keyed by `region`.

2. **Snapshot API via Webhook (n8n)**
    - A second workflow branch exposes an HTTP webhook (`/webhook/latest-region1`).
    - On each call, it fetches the latest row for `region1` from the Data Table, parses `snapshot_json`, and returns it as JSON.
3. **MCP Server & Tools**
    - A FastMCP server wraps the n8n webhook, turning it into three typed tools:
        - `flights_list_region_snapshot(region)`
        - `flights_get_by_callsign(callsign, region)`
        - `alerts_list_active(region)`
    - The server runs as a Streamable HTTP MCP endpoint on `http://localhost:8000/mcp`, using the `StreamableHTTPSessionManager` to manage sessions and transport.
4. **Agent & UI Layer**
    - Python agents use the same tool implementations directly together with the Groq `chat.completions` API and tool-calling.
    - A Streamlit UI provides Traveler and Ops views, each wired to different agents and tools but sharing the same underlying snapshot data.

This layered design allows each component to be developed and debugged independently (for example, by inspecting MCP tools in MCP Inspector, or testing the webhook with `curl`) while still working together as a coherent agentic application.

---

3. n8n Workflows and Snapshot Storage

The **Region 1** workflow in n8n has two independent branches (top for ingestion, bottom for serving data).

3.1 Ingestion Branch (Scheduled Snapshot Fetch)

Nodes:

1. **Schedule Trigger**
    - Runs periodically (e.g., every minute) using a cron schedule.
    - This avoids hitting OpenSky on every UI interaction and gives a stable “latest snapshot.”
2. **HTTP Request**
    - Method: `GET`
    - URL: OpenSky `/states/all` with the bounding box for Region 1.

Output is a JSON object of the form:

```
{
 "time": 1764521587,
 "states": [
  [ "801648", "IGO562P ", "India", 1764521550, 1764521553, 76.6698, 25.1405, 9448.8, false, 225.33, 162.19, 0, null, 9906, null, false, 0 ],
  ...
 ]
}
```

    -
3. **Code in JavaScript** (Transform to snapshot)
    - Reads `items[0].json` from the HTTP node.
    - Normalizes each `states` row into a `Flight` object with named fields (`icao24`, `callsign`, `origin_country`, `longitude`, `latitude`, altitude, speed, etc.).

Builds a `snapshot` object:

```
const snapshot = {
 region: "region1",
 snapshot_time: snapshotTime,
 fetched_at_iso: new Date().toISOString(),
```

```
  flight_count: flights.length,
  flights,
};
```

-

Returns a single item:

```
 return [
  {
    json: {
      region: snapshot.region,
      snapshot_json: JSON.stringify(snapshot),
    },
  },
];
```

-
- If `states` is empty (e.g., rate limiting or temporary errors), the node returns `[]` so that no database write occurs. This implicitly implements **“use last good snapshot when the API is down”**.

4. **Data Table – Upsert row(s)**
    - Table: `flight_snapshots`.
    - Columns: `region` (text, primary key-like) and `snapshot_json` (text).
    - Operation: **Upsert** with match column `region`.
    - Mapping uses expressions `{{$json["region"]}}` and `{{$json["snapshot_json"]}}`, so each run overwrites the previous snapshot for that region instead of creating multiple rows.

The net effect is that n8n maintains a **single, always-up-to-date snapshot row per region**, with rate limits absorbed by caching.

3.2 Webhook Branch (Snapshot Reader API)

Nodes:

1. **Webhook**

- Method: `GET`
- Path: `latest-region1`
- Response mode: “Using Respond to Webhook node.”
- Production URL: `http://localhost:5678/webhook/latest-region1`.

2. **Data Table – Get row(s)**
   - Table: `flight_snapshots`.
   - Filter: `region = region1`, `limit = 1`.
3. **Code in JavaScript** (unwrap snapshot)

If no row is returned, it emits a JSON error like:

{ status: "error", message: "No snapshot found for region1" }

-
- Otherwise it parses `row.snapshot_json` with `JSON.parse` and returns the `snapshot` object as a single item.

4. **Respond to Webhook**
   - Response body: `{{$json}}`
   - Status: `200`, content type: JSON.

Testing with `curl` confirms that the webhook returns the full snapshot, for example:

curl http://localhost:5678/webhook/latest-region1

returns a JSON object with `region`, `snapshot_time`, `fetched_at_iso`, `flight_count`, and a `flights` array of structured flight objects.

---

4. MCP Server and Tools

The MCP server is implemented using **FastMCP** and exposes the snapshot data as typed tools.

4.1 Type Definitions

`server.py` defines explicit `TypedDict` types for:

- `Flight` – fields like `icao24`, `callsign`, `origin_country`, altitudes, velocity, vertical rate, squawk, etc.
- `Snapshot` – `region`, `snapshot_time`, ISO timestamp, `flight_count`, `flights: List[Flight]`.
- `Alert` and `AlertResult` – used by the anomaly detector.

These types make the tools self-documenting and are surfaced to MCP clients (e.g., the Inspector) to improve tool-calling reliability.

4.2 Snapshot Fetch Helper

The private `_fetch_snapshot(region)` function wraps the n8n webhook:

- Looks up the URL for the region in `REGION_WEBHOOKS` (configurable via environment variables).
- Issues an HTTP GET with a short timeout.
- Enforces defaults for `region` and `flight_count` if missing.

If region name is unknown or the HTTP call fails, it raises a Python exception, which surfaces as an error in MCP Inspector.

4.3 Exposed Tools

1. `flights_list_region_snapshot(region="region1") -> Snapshot`
    - Returns the entire latest snapshot for the region.
    - Used by the Ops agent and the UI's region table.

2. `flights_get_by_callsign(callsign, region="region1") -> FlightLookupResult`
    - Normalizes and uppercases callsigns.
    - Scans `snapshot.flights` for a matching `callsign`.
    - Returns either a `flight` object and success message, or `flight: null` and a user-friendly error string.
3. `alerts_list_active(region="region1") -> AlertResult`
    - Implements a simple rule-based anomaly detector:
        - **Low speed at high altitude:** altitude > 8000 m and velocity < 100 m/s.
        - **High vertical rate:** `vertical_rate < -20` (fast descent) or `> 20` (fast climb).
    - For each flagged flight, produces an `Alert` with `reason` and `severity`.
    - Returns the list plus metadata (`alert_count`, `snapshot_time`, etc.).

The server is started with:

```
if __name__ == "__main__":
    mcp.run(transport="streamable-http")
```

which uses the `StreamableHTTPSessionManager` to handle HTTP sessions and streaming payloads.

Using MCP Inspector, the three tools can be inspected and invoked interactively, which was used extensively for debugging.

---

5. Agents, Prompts, and A2A Communication

The **agent layer** is implemented in `agents.py` using Groq's `chat.completions` API with OpenAI-style tools.

5.1 Tool Registry and Execution

`TOOLS` defines the JSON schema for each MCP tool so that Groq can call them via function-calling:

- Name, description, and parameter schema for: `flights_list_region_snapshot`, `flights_get_by_callsign`, `alerts_list_active`.

`execute_tool(name, arguments)` dispatches to the corresponding Python function imported from `server.py`.

5.2 Generic Tool-Calling Loop

`run_tool_calling_chat(system_prompt, user_query, allowed_tools)` implements a standard pattern:

1. Build a `messages` list with system + user.
2. Call Groq with `tools=TOOLS` (filtered by `allowed_tools`) and `tool_choice="auto"`.
3. If the model returns **no** `tool_calls`, treat the first response as the final answer.
4. If there are `tool_calls`:
    - Execute each tool locally.
    - Append a `tool` role message with JSON-encoded results.
5. Make a second Groq call with the enriched message history and return the final answer.

This loop is reused for both Traveler and Ops agents.

5.3 Traveler and Ops Agents

1. **Traveler Agent**
    - System prompt: “You are a traveler support assistant… Help passengers understand where their flight is… Use the `flights_get_by_callsign` tool

when needed."

- Allowed tools: only `flights_get_by_callsign`.
- Typical question:

  "My flight PIA293 is going from Pakistan. Where is it roughly now and is everything normal?"

2. The agent uses the tool to retrieve the latest flight state and translates it into an explanation (location, altitude, speed, and whether anything looks unusual).
3. **Ops Agent**
   - System prompt: "You are an airline operations specialist monitoring airspace safety… Use `flights_list_region_snapshot` and `alerts_list_active` to analyze current traffic…"
   - Allowed tools: `flights_list_region_snapshot`, `alerts_list_active`.
   - Typical question:

     "Give me a concise situation report for region1. How many flights, any anomalies, and which one should I worry about first?"

4. The agent combines snapshot counts and anomaly results to produce a short SITREP summarizing traffic levels and highlighting the most critical flight.

5.4 Traveler+Ops A2A Orchestration

`traveler_with_ops(callsign, passenger_question)` implements a simple Agent-to-Agent (A2A) pattern:

1. Calls **Traveler Agent** with an enriched question mentioning the callsign and region.
2. Calls **Ops Agent** with a prompt focused on the same flight and its regional context.

3. Makes one final Groq call with a “coordinator” system prompt that sees both replies and synthesizes **a single answer for the passenger**.

This pattern demonstrates the key benefit of A2A:

- Traveler Agent alone can answer only about that flight, with limited awareness of what else is happening.
- Ops Agent alone can describe the region but not personalize the answer.
- The coordinated answer leverages both: personalized reassurance plus honest operational context (e.g., “your flight is normal; another flight nearby is descending rapidly but not related to yours”).

---

6. UI Design and User Journey

The UI is implemented as a **Streamlit** web app with two tabs: Traveler View and Ops View.

6.1 Traveler View

Main elements:

- **Inputs:**
  - Callsign (`PIA293`, `IGO562P`, etc.).
  - Free-text passenger question.
- **Actions:**
  - “Fetch current flight data (tools only)” – directly calls `flights_get_by_callsign` and shows the raw JSON tool result, useful for debugging.
  - “Ask Traveler Agent” – runs `traveler_agent` and displays the natural-language answer.
  - “Ask Traveler + Ops (coordinated)” – runs the `traveler_with_ops` A2A flow.

User Journey:

1. Passenger enters their flight callsign and question.
2. Optionally inspects the raw tool data to see the exact altitude, speed, and position.
3. Uses either the Traveler Agent or the coordinated Traveler+Ops answer to get a clear explanation in everyday language.

6.2 Ops View

Main elements:

- **Region selector:** currently only `region1` but designed to support more.
- **Buttons:**
    - "Fetch latest snapshot" – calls `flights_list_region_snapshot` and renders a table with callsign, country, altitude, speed, and on-ground status.
    - "Analyze anomalies" – calls `alerts_list_active` and shows a table of flights flagged by the simple rules.
    - "Ask Ops agent for SITREP" – prompts the Ops agent to summarize the situation.

User Journey:

1. Operations user opens Ops View and fetches the latest snapshot.
2. Scans the table of flights for an overall sense of traffic.
3. Runs the anomaly analysis to see if any flights require attention.
4. Uses the Ops agent SITREP as a concise, natural-language summary, which could be used in daily briefings or incident logs.

The two tabs demonstrate how **the same underlying tools** can be reused for very different user personas.

---

7. Limitations and Future Improvements

Despite working end-to-end, the current system has several limitations:

1. **Single Region, Single Snapshot**
    - Only one region (`region1`) is implemented, and only the latest snapshot is stored.
    - Future work: support multiple regions and a time series of snapshots, enabling trend analysis and “replay” of events.
2. **Simple Anomaly Rules**
    - The alert logic uses fixed thresholds on speed, altitude, and vertical rate.
    - These rules ignore aircraft type, route, or phase of flight (take-off, climb, cruise, descent).
    - Future work: learn anomaly thresholds from historical data or integrate aviation domain knowledge (e.g., typical climb/descent profiles).
3. **OpenSky Rate Limits and Data Quality**
    - The system depends on the OpenSky free API, which is rate-limited and sometimes sparse.
    - The caching design mitigates this but cannot compensate for missing or incorrect data.
    - Future work: multi-source data fusion (e.g., ADS-B aggregators, airline internal feeds).
4. **Security and Authentication**
    - The demo uses unauthenticated local webhooks and MCP endpoints.
    - In a production setting, HTTPS, API keys, and authentication between n8n, MCP, and the UI are mandatory.
5. **Agent Reliability and Hallucinations**

    - Although tools provide structured data, LLM agents can still misinterpret or over-generalize.

    - Future work: stricter response schemas, automated tool-result validation, and guardrails around safety-critical statements.

6. **UI Features**

    - The current UI is a single-user local app.

    - Future work: multi-user roles, persistent sessions, and richer visualizations (maps, timelines, altitude vs. time charts).

---

8. Conclusion

This project demonstrates how to build a **practical agentic system** for flight monitoring by combining:

- **n8n** for robust, rate-limit-aware ingestion and caching of OpenSky flight states.
- **MCP** for exposing clean, typed tools over those cached snapshots.
- **Groq-based LLM agents** (Traveler and Ops) that call these tools for reasoning rather than directly querying external APIs.
- A **Streamlit UI** that makes the system usable for both passengers and operations staff.

The architecture cleanly separates ingestion, tooling, reasoning, and presentation, making the system easier to extend and maintain. While the current anomaly logic and region coverage are intentionally simple, the same pattern can scale to more sophisticated safety analytics, multi-region monitoring, and richer multi-agent workflows.