Agentic AI Assignment 3
Real-Time Airspace Copilot with Agentic Multi-Agent System

Mode: Individual or Pairs (max 2 students)

1. High-Level Idea

You will build a small but realistic **Agentic AI system** that monitors live flight traffic using the **OpenSky Network** public API and provides an intelligent **Airspace Copilot** consisting of:

- **Airspace Ops Copilot (Operations View)**: monitors flights in a region, highlights anomalies (e.g., weird speed/altitude, long stationary time), and summarises the situation for an "operations" user.
- **Personal Flight Watchdog (Traveler View)**: lets a traveler enter their flight (callsign or ICAO24 ID) and get updates plus natural-language explanations via a chatbot.

The system must be **agentic** (multi-agent), use **n8n** for workflow orchestration, **Groq LLM API** for reasoning, and **MCP** (Model Context Protocol) to expose your data/tools to at least one LLM agent. **No external servers or paid services** are required: everything runs on your laptop (Docker + local scripts).

2. Technologies You *Must* Use

a) **n8n (local via Docker)** for:
   - Scheduling/triggering data fetches from the OpenSky API.
   - Pre-processing and storing flight snapshots (e.g., in a file, n8n data store, or simple DB).
   - Exposing a simple HTTP endpoint (webhook) for your front-end / MCP to read the latest data.

b) **Groq LLM API (free tier)** to:
   - Analyse raw flight data and anomalies.
   - Power the chatbot that answers user questions.

c) **Agentic Layer:** either **CrewAI** or **LangGraph** (your choice, or both) to:

- Define at least **two agents** (e.g., "Ops Analyst Agent" and "Traveler Support Agent").
- Implement simple **A2A communication** (one agent can call another when needed).

d) **MCP (Model Context Protocol)**:

- You must implement or configure **at least one MCP server** that exposes your flight data as tools, e.g.:
  - `flights.list_region_snapshot`
  - `flights.get_by_callsign`
  - `alerts.list_active`
- Your CrewAI/LangGraph agents should access data **through MCP tools**, not by reading raw files directly.

e) **Simple UI (Frontend)**:

- Can be a basic HTML/JS page, React app, Streamlit, or any simple frontend.
- Must allow user to enter input and shows nicely formatted results from your backend/agents.

3. Data Source: OpenSky Network API (Anonymous Access)

You will use the **OpenSky Network** public REST API. For this assignment you **do not** need login credentials; use anonymous endpoints such as:

```
https://opensky-network.org/api/states/all
```

Key points:

- The endpoint returns a list of current flights ("states") with fields such as: ICAO24 ID, callsign, country, longitude, latitude, altitude, velocity, heading, etc.
- **Bounding boxes (no maps):** You *may* use the API query parameters for geographic filtering, such as:

  ```
  ?lamin=<min_lat>&lomin=<min_lon>&lamax=<max_lat>&lomax=<max_lon>
  ```

  In this assignment, a "bounding box" simply means four numbers: min/max latitude and min/max longitude. **You do not need any map library or visual map.** You only use these numbers to filter which flights to keep.
- You can hard-code 1–3 regions in a config (e.g., "Region A", "Region B") by specifying their lat/lon ranges.

4. System Behaviour: Inputs and Outputs

4.1. Inputs (What Can the User Provide?)

Your system must at minimum support the following:

a) **Traveler Mode Input (Personal Flight Watchdog):**

- A simple form in your UI where a user can type:
  - **Flight identifier:** either callsign (e.g., `THY4KZ`) or ICAO24 (e.g., `4baa1a`).
  - **Optional:** A short label like “My flight” or “Friend’s flight” (for display only).
- A text box (chat area) where the user can ask questions like:
  - “Where is my flight now?”
  - “Is my flight climbing or descending?”
  - “Has my flight been circling for too long?”

b) **Operations Mode Input (Airspace Ops Copilot):**

- A drop-down or radio buttons to choose a predefined region, e.g.:
  - Region 1: `[min_lat1, max_lat1, min_lon1, max_lon1]`
  - Region 2: `[min_lat2, max_lat2, min_lon2, max_lon2]`
- A button like “**Fetch Latest Snapshot**” to manually trigger the workflow (for demo).
- Or you may refresh automatically every 60 seconds and show the last refreshed time.

4.2. Outputs (What Must the System Show?)

At minimum, your system must produce:

a) **Traveler View Output:**

- A panel showing details for the tracked flight:
  - Last known latitude/longitude, altitude, speed, heading.
  - Plain-language summary (e.g., “Your flight is cruising at 10,500 m, heading east.”), generated by the LLM.
- A chat-style area showing:
  - User questions and LLM answers, based only on your latest flight snapshots.
  - Answers must be **grounded in real data** (you should pass structured flight data into the LLM prompt).

b) **Operations View Output:**

- A table or list of flights in the selected region with columns such as:
  - callsign, ICAO24, altitude, speed, whether it is considered “normal” or “anomalous”.
- A computed **simple anomaly score or label**. For example:
  - “Low speed at high altitude”
  - “Stationary for more than N seconds”
  - “Sudden altitude drop more than X meters between snapshots”

  You can hard-code reasonable thresholds.
- A natural-language summary for the region, e.g.:

  > “Region 1 currently has 25 flights. 3 flights are flagged as anomalous due to unusual speed or altitude. The most critical case is THY4KZ, which is descending rapidly near latitude 41.38.”

c) **Logs / Internal Output:**

- Maintain a simple JSON file, CSV, or n8n data store containing recent snapshots and any generated alerts.
- This store should be what your MCP tools read from.

5. Required Functionalities

Your solution must include at least the following:

5.1. Data Fetch & Preprocessing Workflow (n8n)

- An n8n workflow that:
  i) Calls the OpenSky API every X seconds or minutes (you can choose the interval for demo).
  ii) Optionally uses bounding box parameters to limit to a region.
  iii) Extracts only the fields you care about (e.g., time, ICAO24, callsign, lat, lon, altitude, velocity, heading).
  iv) Stores the latest snapshot in:
     - a local JSON file; or
     - an n8n data store; or
     - a simple SQLite/CSV file.
  v) Exposes a simple HTTP endpoint (n8n webhook) that returns the latest snapshot in JSON when called, e.g.:

     `GET http://localhost:5678/webhook/latest-region1`

5.2. Agentic Layer (CrewAI or LangGraph)

- Design at least two agents:
  - **Ops Analyst Agent:** uses MCP tools to query region snapshots and compute anomalies, then summarises the situation.
  - **Traveler Support Agent:** answers user questions about a specific flight using MCP tools and LLM reasoning.
- At least one example of **A2A interaction**, e.g.:
  - Traveler Support Agent calls Ops Analyst Agent when user asks something like "Are there any other flights near mine that are having issues?"

5.3. MCP Server

- Implement an MCP server (FastAPI or any framework you prefer) that:
  - Reads from your flight store (JSON/DB) used by n8n.
  - Exposes tools such as:
    * `flights.list_region_snapshot(region_name)` – returns the most recent snapshot for that region.
    * `flights.get_by_callsign(callsign)` – finds the latest record for a given flight.
    * `alerts.list_active()` – returns currently flagged anomalies.
  - Registers itself in your MCP registry so that your agents can call these tools.

5.4. UI / Frontend

- A minimal but clean UI with:
  - Tabs or sections for:
    * Traveler Mode
    * Ops Mode
  - Input fields described in Section 4.1.
  - A chat area showing user messages and agent responses.
  - A table/list for region flights and their anomaly labels.
- The UI may:
  - Call your agent backend (CrewAI/LangGraph app) via HTTP; or
  - Be integrated directly (e.g., Streamlit calling the agents in Python).

6. Suggested Implementation Steps (Guideline)

These are guidelines to help you; you may adapt the order.

1. **Set up n8n (Docker)** and test a basic HTTP request node to OpenSky.
2. **Build the Data Fetch Workflow:** fetch states, filter fields, store JSON, expose a webhook that returns the current snapshot.
3. **Design your MCP server:** read the stored JSON, implement the required tools, test them independently.
4. **Connect MCP to your Agentic Framework:** configure CrewAI or LangGraph agents to call MCP tools when reasoning.
5. **Design Prompting and Roles:** write clear system prompts for Ops Analyst and Traveler Support agents.
6. **Build the Frontend:** simple UI to send user questions and show agent answers; controls to pick region and flight ID.
7. **Add Simple Anomaly Logic:** even a basic rule-based system (e.g., thresholds) is acceptable, as long as it is clearly documented.

7. Architecture Diagram

You may adapt the following architecture, but the key components should remain present.

8. Deliverables

You must submit:

1. **Technical Report (4–6 pages)** in PDF, containing:
   - Introduction & problem statement: why this system is useful and what gap it fills.
   - System architecture (using Figure 1 or your modified version).
   - Description of n8n workflows and how snapshots are stored.
   - Description of MCP server and tools.
   - Description of agents, prompts, and A2A communication.
   - UI design and user journey (screenshots allowed).
   - Limitations and possible future improvements.

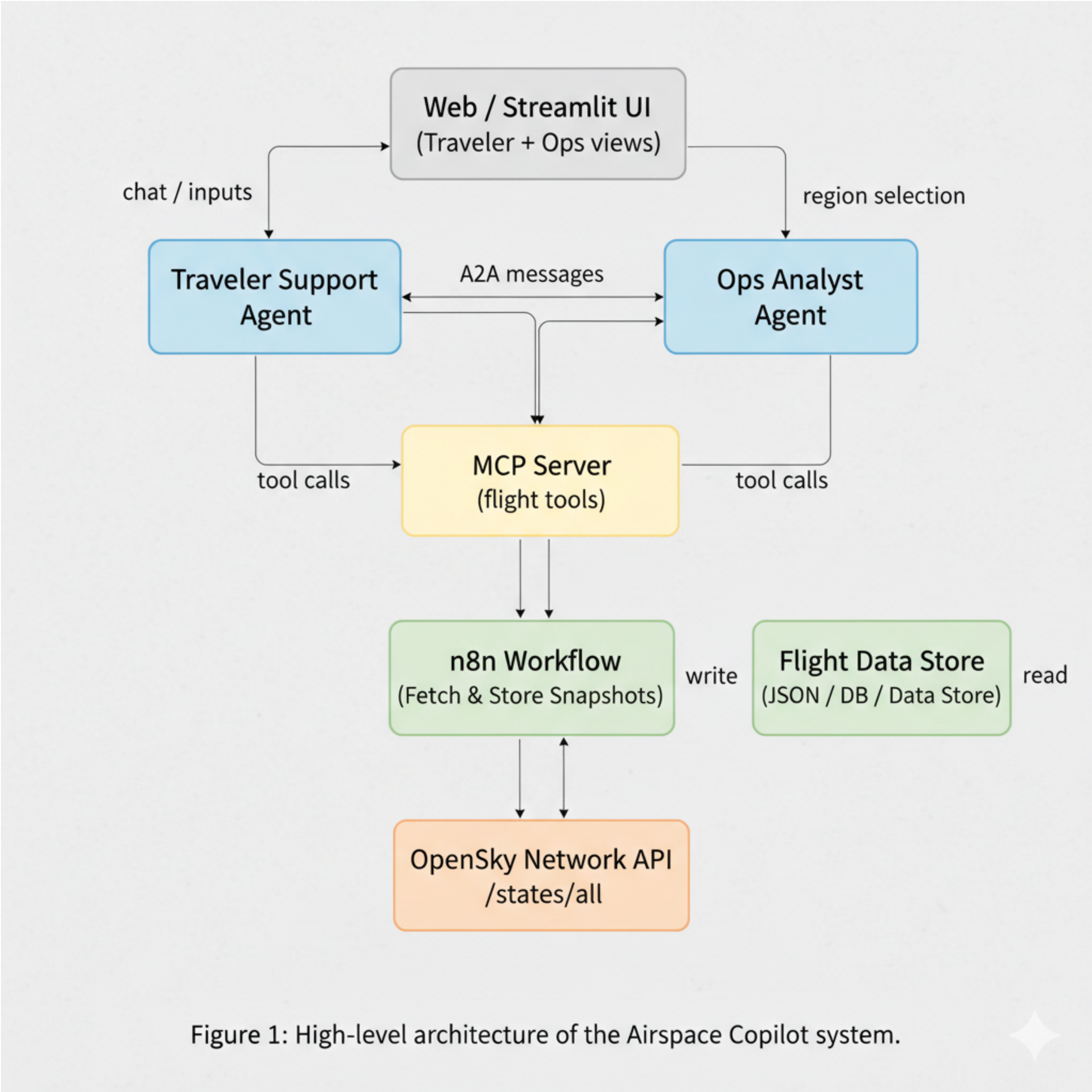

Figure 1: High-level architecture of the Airspace Copilot system.

2. **Source Code:**

    - n8n workflow exports.
    - MCP server code.
    - Agentic layer (CrewAI/LangGraph) code.
    - Frontend code.
    - A short `README.md` explaining how to run each component locally.

3. **Demo Video (3–5 minutes):**

    - Briefly show data fetching, traveler query, ops summary, and at least one anomaly example.

9. Marking Criteria (Indicative)

- **Correctness & Functionality (30%):** Required features implemented; input and output behaving as specified.
- **Agentic Design & MCP Integration (30%):** Clear use of multiple agents, A2A communication, and MCP tools (not just direct API calls).
- **Use of n8n & Data Handling (20%):** Clean workflows, proper snapshot storage, and well-defined endpoints.
- **UI/UX & User Journey (10%):** Simple, understandable interface for both traveler and ops views.
- **Report Quality & Reflection (10%):** Clear explanation, architecture diagram, limitations, and future work.

10. Constraints & Notes

- You must use only free tools: local Docker n8n, Groq free tier, and public OpenSky endpoints.
- You do **not** need any map visualisation; numeric latitude/longitude and your own text summaries are enough.
- You may simulate some scenarios by saving a fixed JSON response and replaying it (for stable demos).
- Work must be your own. Plagiarism or direct copy of internet projects is not allowed.

11. Handling OpenSky API Rate Limits (Important)

The OpenSky anonymous API enforces strict usage limits. In many cases, the endpoint `/states/all` may return:

- HTTP 429 (Too Many Requests),
- empty JSON,
- delayed responses, or
- temporarily unavailable data.

This behaviour is normal and expected when using the anonymous public API.

11.1 Requirement: Your System Must Continue Working Even When the API Fails

Your design *must not* depend on the API being online at every moment. If a request fails:

1. The system should detect the failure (e.g., status code $\neq 200$).
2. It should load the **most recent successful snapshot** stored locally (JSON file, database, or n8n data store).
3. It must still allow:
   - Traveler questions (chat mode),
   - Operations summaries,
   - Agent reasoning,

   using the previously stored data.

11.2 Recommendation

- Limit API calls (e.g., no faster than once every 10–15 seconds).
- Cache snapshots and always show the "Last updated at: `HH:MM:SS`" timestamp in the UI.
- Log rate-limit events and show a non-blocking message such as:

  > "OpenSky API temporarily unavailable. Displaying last known flight snapshot."

- For demonstrations, you may replay a previously saved JSON snapshot.

11.3 Marks Are Not Deducted for API Downtime

The system will be graded on:

- correct architecture,
- agentic behaviour,
- MCP integration,
- UI design,
- robustness and fallback handling.

Therefore, occasional API unavailability **will not affect your grade**.