ChatGPT

UAV Traffic Management (UTM) System MVP – Technical Architecture

Introduction

Unmanned Aerial Vehicle (UAV) Traffic Management (UTM) systems enable the oversight and coordination of drone flights, ensuring safety and regulatory compliance in shared airspace. In this hackathon MVP, we design a **full-stack UTM application** that lets multiple users (pilots and administrators) register drones, submit flight plans, and monitor flights in real-time. The solution leverages a **React** (TypeScript) frontend, a **Python** backend (using FastAPI or Django), and is deployed on **Google Cloud Platform (GCP)** for scalability. Core features include user authentication, drone/pilot registration, flight request handling with no-fly zone checks, and live drone telemetry on a Google Maps interface. We emphasize clean architecture, robustness, and scalability to lay a foundation for future expansion.

Requirements and Scope

Key requirements for the MVP include:
- **Frontend:** Implemented in React + TypeScript, providing a responsive single-page application (SPA) UI. Integrates Google Maps API for geospatial visualization.
- **Backend:** Implemented in Python (FastAPI or Django). Exposes RESTful APIs and WebSocket endpoints. Handles business logic, data validation, and integration with external services.
- **Cloud Deployment:** Use GCP services for hosting. The backend will run on a scalable service (e.g. Cloud Run), and the database on a managed service (e.g. Cloud SQL for PostgreSQL). The frontend will be deployed as a static app (e.g. on Cloud Storage + CDN or via Cloud Run).
- **Database:** A suitable database for persistent storage (recommendation: PostgreSQL for relational data). Must support multi-user data and potential multi-tenancy.
- **Multi-User & Auth:** Support multiple pilot users and admin users (multi-tenant capable). Secure authentication (pilots and admins must log in). Role-based access control (pilots manage their flights; admins oversee system-wide operations).
- **Core MVP Features:**
1. **Pilot & Drone Registration:** Web forms for pilots to register an account and to register their drones. Data stored persistently.
2. **Flight Plan Submission:** Ability for a pilot to request a flight by specifying waypoints (latitude/longitude), altitude, time schedule. The system checks for restricted/no-fly zones before accepting the plan.
3. **Real-Time Flight Monitoring:** A live map view displaying drone positions as flights progress. The system provides live telemetry (drone coordinates updates) and raises alerts for violations (e.g. leaving approved airspace or communication loss).
- **Additional (Optional) Features:** If time permits, plan for integrating a live video feed from drones (RTSP stream), overlaying weather data on the map, and simulating Remote ID broadcasts (drone identity signals).
- **Integration & Instructions:** Clearly define how all components connect (frontend ↔ backend ↔ services). Provide API structure, database schema, deployment steps, and developer setup instructions. Emphasize a clean separation of concerns and scalability best practices throughout.

System Overview (High-Level Architecture)

The UTM MVP is structured as a cloud-based web application with distinct components for frontend, backend, data storage, and auxiliary services. **Figure 1** below illustrates the overall architecture and data flow:

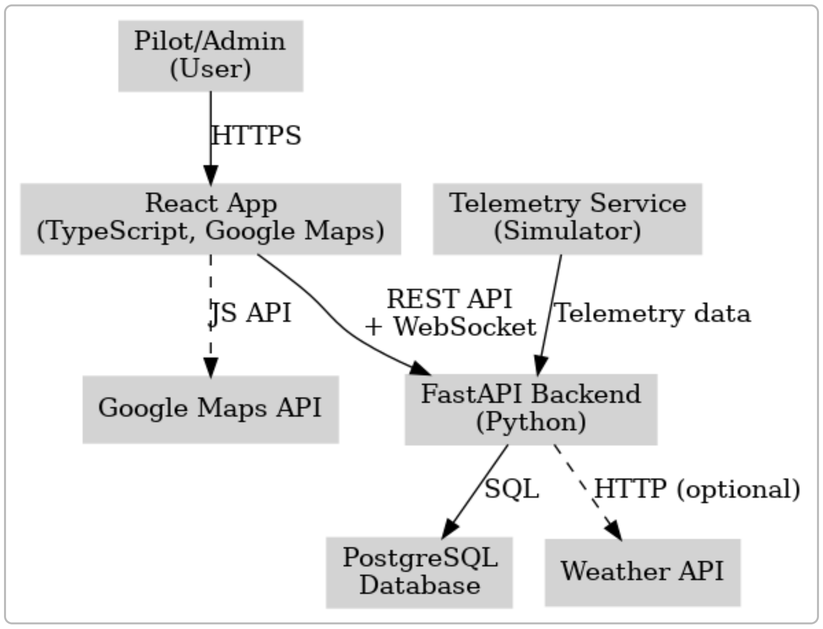

Figure 1: High-level architecture of the UTM system MVP. The React frontend communicates with the FastAPI (Python) backend via HTTPS REST calls and WebSockets for real-time updates. The backend persists data in a PostgreSQL database and integrates with external services (Google Maps API on the frontend, optional Weather API on backend). A telemetry simulation service generates live drone data and feeds it to the backend, which broadcasts to clients. All components are deployed on Google Cloud Platform.

Component Breakdown:
- **React Frontend:** Provides the user interface (web app) for pilots and admins. It uses Google Maps JavaScript API for map rendering and draws flight paths and drone markers. The frontend communicates with backend APIs for data and opens a WebSocket to receive live telemetry updates.
- **FastAPI Backend:** A Python server that exposes RESTful endpoints for all core functions (registration, flight plan, etc.) and a WebSocket endpoint for real-time telemetry. It contains the application logic: validating requests, enforcing permissions, checking no-fly zones, etc. The backend interacts with the database for persistent data and can call external APIs (e.g. weather data).
- **Database (PostgreSQL):** Stores persistent data – user accounts, drone info, flight plans, waypoints, telemetry logs, etc. We use a relational schema to capture the relationships between pilots, drones, and flights. (PostgreSQL on GCP's Cloud SQL is recommended for scalability, security, and managed maintenance [1] .)
- **Telemetry Simulation Service:** A module or microservice that mimics drone GPS telemetry. This could be a background task or separate process that periodically sends each active drone's location to the backend. The backend then relays these updates to clients via WebSocket. This simulates real drone tracking in the absence of actual hardware.
- **External Services:** - *Google Maps API* – used on the frontend to display maps. The React app loads Google

Maps SDK to embed maps, markers, and possibly drawing tools for waypoints. (The React Google Maps library provides an `APIProvider` to conveniently load the Maps JS API into React [2].)
- *Weather API* (optional) – the backend can fetch weather info or map tiles (e.g. from OpenWeatherMap) to overlay conditions (rain, wind) on the map.
- *RTSP Video Stream* (optional) – if integrated, a streaming server or service would convert RTSP from the drone to a browser-friendly format (WebRTC/HLS) since browsers can't play RTSP directly [3]. The frontend would then show the live video feed alongside telemetry.
- **Google Cloud Platform Deployment:** All components reside in GCP. We containerize the backend (and possibly the frontend) and deploy on **Cloud Run** for a serverless, autoscaling execution. The database is hosted on **Cloud SQL (Postgres)**. The React app can be served as static files (via Cloud Storage + Cloud CDN or via an nginx container on Cloud Run). GCP's infrastructure ensures the solution can scale and remain robust without heavy ops burden [4].

Frontend Architecture (React & TypeScript)

The frontend is a single-page application built with **React** and **TypeScript**. This choice ensures a modular, maintainable codebase with type safety, which is valuable for catching errors early in a hackathon setting. Key aspects of the frontend design include:

- **Structure & Routing:** Use React Router (if multiple views are needed) to handle pages like Login, Registration, Flight Dashboard, etc. The app will be bootstrapped (e.g. with Create React App or Vite) for fast iteration. We will organize code into reusable components (e.g. forms, map components, lists) and use state management (React Context or lightweight state libraries) to share user info (like auth state) across components.

- **UI Components:**

- *Registration Forms:* A pilot registration form (collect name, email, password, etc.) and a drone registration form (drone model, identifier, etc.). Use controlled components with validation (e.g. using a library like Formik or React Hook Form for quick form handling). These forms send data to backend endpoints via HTTP POST.
- *Flight Plan Form:* A specialized interface where a pilot can plan a flight. This will include a **map widget** and input fields. The pilot can select or draw a route on the map – for example, dropping waypoints on the Google Map. We can integrate the Google Maps Drawing tools or simply allow clicking the map to add waypoints. Waypoint coordinates (and perhaps altitude per waypoint) are gathered, along with desired start time and duration. The form sends a flight request to the backend for validation and storage.
- *Live Flight Dashboard:* The main view for monitoring. It contains a Google Map centered on the operating area, with markers (or moving icons) for each active drone. The map may also show flight plan paths (polylines connecting waypoints) and delineate restricted zones (polygons or circles) for visual reference. Pilots see their own drones; admins can toggle to see all active flights. A sidebar or overlay can list active flights and their status (e.g. "On Schedule", "Violation: Exited Area", "Signal Lost").

- *Login & Authentication:* A login page for all users. Upon success, the client stores an auth token (e.g. JWT) and uses it for subsequent API calls. Use HTTPS for all communication and possibly secure HttpOnly cookies or localStorage for token storage (taking care with XSS/CSRF as needed).

- **Google Maps Integration:** We embed Google Maps by using the Maps JavaScript API in React. A recommended approach is to use the official @react-google-maps library (now part of `vis.gl/react-google-maps`), which provides React components and hooks for Google Maps. For example, we wrap our app in an `APIProvider` with our Maps API key to load the library [5], then use the `<Map>` component to render the map and `<Marker>` or custom overlays for drones. The Maps API will also allow reverse-geocoding or distance calculations if needed (not core for MVP, but available). Ensure the Google Maps API key is secured (restricted to our domain) and stored in a config file or environment variable for the frontend.

- **Real-Time Updates via WebSocket:** The frontend opens a WebSocket connection to the backend after user login (e.g. `ws://backend-url/ws` or a specific endpoint). This is used to receive **push notifications of telemetry**. For instance, when a drone's coordinates update, the backend broadcasts the new position to all relevant clients. In the React app, we handle these messages (likely JSON containing flight ID, new lat/long, altitude, maybe a status) and update the map marker position accordingly. This avoids constant polling and provides live feedback. We'll use the browser WebSocket API (or a library for convenience) to connect when the dashboard mounts.

- **User Experience & Feedback:** To guide the user, incorporate alerts/modals for important events (e.g. "Flight plan intersects a restricted zone!" or "Drone XYZ has left its approved area!"). Use a consistent design (maybe a UI toolkit like Material-UI or Ant Design to accelerate styling). Keep the interface intuitive: for example, color-code drones (green = normal, red = violation, grey = lost signal), and use icons (a drone icon for markers, etc.).

- **Front-End Multi-Tenancy & Roles:** If multi-tenancy means separate organizations, the UI could potentially show different branding or segregate data. For MVP, we assume a simpler approach: the data shown is inherently filtered by the logged-in user's permissions (pilots see only their info, admins see all). Role-based rendering can hide admin-only views from pilots. (In a more advanced scenario, we could load tenant-specific configuration or styling after login, but this is beyond MVP needs.)

- **Error Handling & Validation:** The frontend will validate user input (e.g. form fields not empty, coordinates in proper format) before sending to backend. However, it will also handle backend validation errors gracefully: e.g., if the backend returns "Flight path enters restricted zone X," the UI should present this message clearly so the pilot can adjust their plan.

Backend Architecture (Python API Server)

On the backend, we propose using **FastAPI** (a modern high-performance Python web framework) for its succinct syntax, asynchronous capabilities, and built-in support for docs and WebSockets. (Django could be an alternative if an ORM and admin interface are prioritized, but FastAPI offers speed and flexibility ideal for a hackathon MVP.) The backend is responsible for all core application logic and integrates the various parts of the system. Key elements of the backend design include:

- **API Endpoints (REST):** We organize the API into logical routes, likely under a common prefix (e.g. `/api`). Each feature corresponds to one or more endpoints:

- **Auth**: `POST /api/auth/register` (for pilot signup), `POST /api/auth/login` (returns JWT or session), `POST /api/auth/logout`. If using JWT, we'll implement token issuance and a simple OAuth2 password flow (FastAPI's `OAuth2PasswordBearer` can help).
- **Pilots & Drones**: `GET /api/pilots/me` (pilot's profile), `GET /api/drones` (list pilot's drones), `POST /api/drones` (register a new drone), possibly admin-only endpoints like `GET /api/pilots` (all pilots) or `DELETE /api/drones/{id}`.
- **Flights**: `POST /api/flights` (submit a new flight request with plan data), `GET /api/flights?status=active` (list active flights for user or all for admin), `GET /api/flights/{id}` (details of a specific flight plan, including waypoints), `PUT /api/flights/{id}/start` (pilot or system triggers start of flight), `PUT /api/flights/{id}/stop` (end a flight early or mark completed). If an approval workflow is needed, e.g. `PUT /api/flights/{id}/approve` (admin approves a pending flight).
- **Telemetry**: While real-time data goes via WebSocket, we might also provide `GET /api/flights/{id}/telemetry` to fetch recent positions (for a map replay or if WebSocket not available). Also, a `POST /api/telemetry` could be used by the telemetry simulator service to send data into the system (this would be an internal-auth or microservice endpoint, not public).

- **No-Fly Zones**: (if applicable) `GET /api/restricted-zones` to fetch known restricted areas (so front-end can display them). Alternatively, include these in flight plan validation logic only. Admins could have `POST /api/restricted-zones` to add a new zone (out of scope for MVP unless needed for demo).

- **WebSocket Endpoint:** FastAPI allows easy addition of WebSocket routes. We will have an endpoint like `/ws` (or `/ws/telemetry`) that clients connect to after authentication. We will authenticate the WebSocket connection (e.g. by requiring a token query param or performing a token check on connect). The server will keep track of active connections and which user or flight they are interested in. For simplicity, the MVP can broadcast all telemetry to all connected clients (and let clients filter if it's not their drone). If scaling up, we'd refine this to rooms or groups per flight or per user. FastAPI's in-memory broadcast solution is straightforward for a single-process demo, but note that in-memory broadcasts only work on one instance and won't share state across multiple server instances [6]. In production, if we scale out to multiple replicas, we'd use a messaging layer (Redis pub/sub, etc.) to sync messages across instances for consistent real-time updates.

- **Business Logic & Services:** We adhere to clean architecture principles by separating concerns: the API layer (FastAPI route functions) will delegate to service classes or functions that implement the core logic. For example, a `FlightService.submit_flight(plan)` method will handle saving the flight and performing the no-fly zone checks, while a `TelemetryService.broadcast(position)` handles distributing new telemetry to WebSocket clients. This separation makes it easier to test logic independently and improves maintainability.

- **Data Validation & Pydantic Models:** FastAPI heavily uses Pydantic models for defining request and response schemas. We will define models for entities like `PilotCreate`, `DroneCreate`, `FlightPlanCreate`, etc., specifying required fields and types. This ensures the backend only processes valid data, and automatically returns useful error messages for missing/invalid fields. It also makes it easy to produce API docs (OpenAPI docs) for our endpoints without extra work.

- **Authentication & Authorization:** We implement authentication at the API layer. When a user logs in, if using JWT, the backend generates a token (signed with a secret) that encodes the user ID and role (and possibly tenant info). Subsequent requests must include this token (e.g. in the `Authorization` header). FastAPI can use dependency injections to auto-check tokens and reject unauthorized calls with 401. We will protect routes so that, for example, `POST /api/flights` requires an authenticated pilot, and admin-only routes check the user's role. Passwords will be stored hashed (using a library like Passlib bcrypt). If time allows, implement token refresh or expiration as needed (for MVP, tokens could be long-lived or simply re-login if expired). Multi-tenancy is handled by associating users (and their data) with a tenant ID and scoping queries accordingly. This is the simplest approach (single database, tenant field filter) [7] for the MVP, ensuring data separation by software logic. (Other strategies include separate schemas or databases per tenant [7], but those add complexity not needed initially.)

- **No-Fly Zone Checking:** A critical part of flight submission logic is ensuring the flight path is in allowed airspace. We will maintain a list of restricted zones. For MVP simplicity, this can be a static dataset (e.g. a JSON of coordinates/radius for no-fly zones like airports) loaded on startup or stored in a table. When a flight plan is submitted, the backend will check each waypoint (or line segments between waypoints) against these zones. If any waypoint falls inside a restricted area (e.g. within a radius of a restricted point or inside a polygon), the backend responds with an error describing the violation. We might use a basic point-in-circle calculation for circular no-fly zones or point-in-polygon for more complex shapes. (If using Postgres, one could leverage PostGIS for geospatial queries in the future, but a Python library or simple math is sufficient for MVP.) The response could list which waypoint is invalid or simply that the route is not allowed. The pilot can then adjust and resubmit. Admin users could override or approve flights that trigger minor violations (but that process can be kept simple or manual for now).

- **Telemetry Handling:** When a flight is started, the backend begins accepting or generating telemetry for that flight. If using an external telemetry simulator service, the backend provides an endpoint for it to send data (authenticated with a special token or API key known to the service). Alternatively, the backend itself can spawn a background task (using `asyncio.create_task` or FastAPI's BackgroundTasks) to simulate the drone movement. For example, once a flight is marked "active" (either immediately at scheduled time or via a pilot's start command), a background loop could interpolate positions between waypoints over time and call the broadcast function with new positions every few seconds. Each telemetry update will include the drone/flight ID, timestamp, current lat/long (perhaps altitude), and a status (e.g. NORMAL, or flags like LOST_SIGNAL if we simulate that). The backend then **broadcasts** this message to WebSocket connections. FastAPI can manage a list of WebSocket clients; we can broadcast to all, or filter by flight/user as needed. (In a scaled environment, we'd use a tool like `encode/broadcaster` with Redis or a message broker to coordinate websockets across instances [6].) For MVP, a single instance can handle a moderate number of clients easily.

- **Alerting & Rule Enforcement:** The backend should also enforce rules during flight. For example, if a drone's telemetry shows it *left* its approved corridor or exceeds altitude limits, we change its status to "violation" and broadcast an alert. This could be done by comparing current coords to the plan (if the drone strays more than X meters from any path segment or beyond last waypoint radius) or entering a known restricted zone. Similarly, to simulate **signal loss**, our telemetry service could stop sending updates for a period or send an explicit "signal lost" status. The frontend, upon not receiving updates for, say, >10 seconds, could mark the drone as lost comms. The backend could also detect this if telemetry is time-stamped and expected at intervals. All these checks provide realism and can be tuned. For MVP, even simple rules (like a boolean flag if out-of-bounds) are fine.

- **External Integrations:** The backend may call external APIs as needed. For example, if implementing the weather overlay, the backend can call a weather service (with appropriate API key) to get current weather data (wind speed, precipitation) for the flight area and send that to the frontend (or directly have the frontend call a weather API – but doing it via backend avoids exposing the weather API key and allows caching). These calls should be done asynchronously to avoid blocking main threads (FastAPI's async support helps here). Another integration is if we emulate **Remote ID**: potentially, the backend could generate a "Remote ID" JSON for each active flight (which includes drone ID, pilot registration info, position, heading, etc.) and provide it via an endpoint, as if an external system queried the Remote ID of drones. This would demonstrate compliance with Remote ID requirements in a simulated way.

- **Scalability & Performance:** Python FastAPI, running on Uvicorn, can handle many concurrent requests, especially I/O-bound (like network calls, DB queries) due to its async nature. By deploying on Cloud Run, the backend can automatically scale out to multiple instances under high load. The stateless design of REST APIs means each instance can serve requests independently. WebSockets are stateful connections, so Cloud Run will by default stick a client to one instance (session affinity) if using HTTP/1.1 WebSockets. We need to ensure one instance can handle the expected number of concurrent websockets (which is typically in the thousands per CPU). For hackathon demo purposes, one instance is likely sufficient. If we needed more, we'd consider using a dedicated service for websockets or ensure our broadcasting mechanism works across instances (as noted earlier). Storing and querying data is offloaded to Cloud SQL which can be scaled (vertical scaling, read replicas) if necessary. By using GCP managed services, we inherit a lot of scalability and reliability features (e.g., Cloud Run restarts instances on failure, Cloud SQL manages backups, etc.), letting us focus on the application logic.

Data Model and Persistence (Database Schema)

We choose a **relational database (PostgreSQL)** for storing persistent data. This fits the highly relational nature of the data (users, drones, flights, waypoints) and ensures **ACID** compliance for critical transactions (e.g., ensuring a flight plan and its waypoints are all saved together). PostgreSQL is also well-supported on GCP (Cloud SQL) and can easily handle geospatial extensions (PostGIS) in the future if needed.

Proposed Schema: We outline the primary entities and their relationships below. (Primary keys are denoted by PK, foreign keys by FK.)

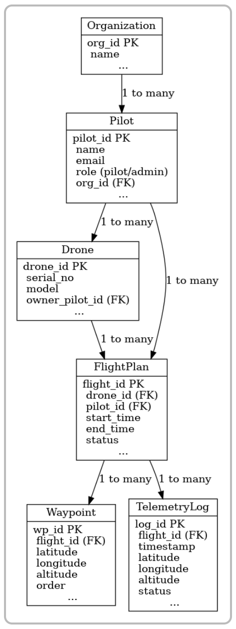

Figure 2: Entity-Relationship diagram for the UTM system database. Key tables include Organization (for multi-tenancy), Pilot (user accounts), Drone, FlightPlan, Waypoint, and TelemetryLog. Arrows indicate one-to-many relations (e.g. one Pilot can have many Drones; one FlightPlan can have many Waypoints and Telemetry records).

Key tables and fields:

- **Organization** – (Optional, for multi-tenancy) Holds info about an organization or tenant. Fields: `org_id (PK)`, `name`, etc. Each Pilot is linked to an organization. (If multi-tenancy isn't needed, this table can be omitted and a default org assumed.)

- **Pilot** – Represents a user (pilot or admin). Fields: `pilot_id (PK)`, `name`, `email` (unique login), `password_hash`, `role` (e.g. "pilot" or "admin"), `org_id (FK)` if using orgs. Additional fields could include pilot license number, contact, etc., if needed.

- **Drone** – Represents a drone registered in the system. Fields: `drone_id (PK)`, `serial_no` or registration ID, `model`/type, perhaps `hardware_id`, etc., and `owner_pilot_id (FK)` linking to the Pilot who owns it. Possibly an `org_id` if the drone is tied to an organization asset pool.

- **FlightPlan** – A flight plan or request. Fields: `flight_id (PK)`, `pilot_id (FK)` (who submitted it), `drone_id (FK)` (the drone to be used), `start_time`, `end_time` (or duration/eta), `status` (e.g. PENDING, APPROVED, ACTIVE, COMPLETED, REJECTED), and possibly a JSON or text field for misc info (like a flight name or purpose). If we choose not to have a separate Waypoint table, we could store the waypoints as an array or geoJSON path in this table, but a separate table is cleaner.

- **Waypoint** – Stores the waypoints for each flight plan (if we need detailed path). Fields: `wp_id (PK)`, `flight_id (FK)` referencing FlightPlan, `latitude`, `longitude`, `altitude`, and maybe an `order` or sequence number to sort the waypoints. This table allows a variable number of waypoints per flight. (Alternatively, we might not need this if we store route in FlightPlan directly for MVP simplicity, but using a table allows easy extension to editing and analysis of flight routes.)

- **TelemetryLog** – Records telemetry points for flights (primarily for history or debug, since real-time display doesn't require persistent logging unless we want to support replay). Fields: `log_id (PK)`, `flight_id (FK)`, `timestamp`, `latitude`, `longitude`, `altitude`, `status` (NORMAL/VIOLATION/LOST, etc.). We can log important events or periodic positions here. This can be a large table if logging at high frequency, so for MVP we might log infrequently or only store significant events (e.g., entry into no-fly zone, flight end, etc.).

- **RestrictedZone** (optional) – If we want to store no-fly zones in the DB, a table for them: e.g., `zone_id`, `name`, `shape` (could be polygon coordinates or center & radius), `altitude_limit` (if zone has a vertical component), etc. For MVP, we can hardcode a couple of zones instead.

- **Other Tables:** If using Django, there might be auth tables out-of-the-box. If using FastAPI with SQLAlchemy, we'll create these tables via ORM models. We will also likely have a join table if needed (not obvious one here aside from what's described). For Remote ID simulation, we might not need a table, as it's more of a real-time broadcast, unless we store broadcast history.

Multi-Tenancy Consideration: We include `org_id` on Pilot (and by extension on related data via the pilot). This way, if the platform is used by multiple organizations, data can be partitioned logically. In a simple single-schema approach, every query will filter by org_id to ensure isolation [8]. We must be diligent in code to always apply this filter for multi-tenant safety (or use an ORM feature or middleware to add the filter globally). For MVP and demo purposes, we might not actively demonstrate multi-org usage, but the schema allows it. The alternative approaches (separate DB per tenant, or separate schema per tenant) are more complex to implement quickly [7], so we stick to a single schema with tenant ID strategy as it's the "cheapest way" and fine as long as our code enforces separation [9].

Database Access: In the backend, we'll use an ORM for convenience and quicker development. FastAPI pairs well with **SQLAlchemy** or the newer **SQLModel** (which is Pydantic + SQLAlchemy under the hood). This allows defining the tables as Python classes and performing CRUD easily. Alternatively, if we chose Django, we'd use Django's ORM with models for each table. In either case, we'll keep transaction scope short – each request that modifies data will open a DB session, write, commit, and close. For heavy read endpoints (like listing all flights), we may add indexing to key columns (e.g. flight status, pilot_id) to maintain query performance. Cloud SQL will handle scaling; in a hackathon scenario, the data volume is small anyway.

Core MVP Use Cases and Workflows

Let's detail how the main features are executed within the system, showing integration of the components:

1. User & Drone Registration

Workflow: A new pilot navigates to the app and creates an account, then registers a drone.
- **Pilot Registration:** The user fills the signup form on the React app. The form data (name, email, password, etc.) is sent via `POST /api/auth/register`. The backend receives it, hashes the password, creates a Pilot record in the database (assigning default role "pilot" and maybe default org if applicable), and returns

a success response (or perhaps directly returns a JWT token upon registration to keep the user logged in). If the email is already taken, the backend returns 400 and the frontend shows an error.
- **Pilot Login:** After registering, or for returning users, the pilot uses the login form. On `POST /api/auth/login` with credentials, the backend verifies the password (e.g. using bcrypt check). If valid, it generates a JWT token containing the user's ID and role (and org). The token is returned to the frontend. The frontend stores it (e.g. in `localStorage` or a cookie) and from now on attaches it in the `Authorization: Bearer <token>` header for future API calls. This token will be used to authenticate the WebSocket connection as well (for example, by including it as a query param or header during the WebSocket handshake). If using Django instead, we might use its session auth with cookies, but JWT is simpler for SPA compatibility.
- **Drone Registration:** Once logged in, the pilot can register a drone. In the UI, they fill a form with drone details (perhaps drone nickname, serial number, type). On submit, a `POST /api/drones` is called with these details. The backend, having authenticated the user (token), creates a Drone record linked to that pilot. The response could include the new drone's ID. The pilot can register multiple drones if needed (e.g. a fleet). In the database, we ensure `owner_pilot_id` is set to the requesting user's ID (the auth middleware provides that). If an admin user registers a drone, possibly they can choose which pilot it belongs to (but this is beyond MVP scope unless an admin is managing drones for pilots).

Data Integration: The Pilot and Drone tables now contain these records. The relationships allow queries like "get all drones for this pilot" (used when rendering the pilot's dashboard or flight submission form so they can choose which drone is flying).

Multi-Tenancy: If multi-tenancy is used, the `org_id` for the pilot (and thus drone via pilot) is set. All queries will automatically filter by that org if using an ORM query with the user's org context. This ensures, for example, Pilot A from Org1 cannot see or register drones for Org2. Admin users could either be global (not tied to an org, in which case they have a special role that bypasses filters) or they could be admin per org (with org_id set and only manage that org's pilots). We can decide that based on requirements; a likely approach is to have a global admin role for regulators who see everything, and maybe organization admins with role "pilot" but having oversight via separate features. For MVP simplicity, we might treat "admin" as global.

Developer Notes: For development, we can create some seed users. For example, create an admin account in the database directly (since no UI to register admin likely). Ensure password hashing is correctly done. Since FastAPI doesn't provide an auth system out of the box (unlike Django), we implement these bits ourselves or use a plug-in library if pressed for time. However, writing a quick JWT auth in FastAPI is feasible within a hackathon timeframe due to its excellent docs and examples (e.g. FastAPI docs have a JWT auth example we can follow).

2. Flight Plan Submission and No-Fly Zone Validation

Workflow: A pilot creates a new flight plan for a selected drone, specifying waypoints and time, which the system validates and records.
1. **Plan Design (Frontend):** The pilot opens the "Plan Flight" view. The React app could present a form with fields (date/time, altitude, etc.) and a map where the user can click to add waypoints. Suppose the pilot selects their drone *Drone A*, chooses a start time (say 15 minutes from now, or immediate), a duration or end time, and plots a path by clicking on the map to create 3 waypoints (forming a route). The frontend might show latitude/longitude of each point in a list as well.

2. **Submission (Frontend->Backend):** When ready, the pilot hits "Submit Flight Request". The frontend packages the flight plan data – likely as JSON: `{ droneId: X, startTime: "...", endTime: "...", waypoints: [ {lat:..., lon:..., alt:...}, {...} ] }` – and sends it via `POST /api/flights` with the auth token.
3. **Processing (Backend):** The backend authenticates the token (ensuring this pilot can create a flight). It then proceeds to validate the request:
- Check that the drone ID belongs to the requesting pilot (preventing a pilot from using someone else's drone). If not, return 403 Forbidden.
- Validate the time window (start < end, start is in the future or near current, etc.).
- **No-Fly Zone Check:** For each waypoint (or possibly each segment between them), check against restricted zones. For each restricted zone in our dataset: if any waypoint lies within a zone's boundaries (or a certain buffer), flag it. If any segment between waypoints crosses into a zone (this is more complex to compute; MVP can simplify by just checking the waypoints themselves and perhaps intermediate points by interpolation if we want to be thorough). Let's say a waypoint is within 1 km of an airport zone – the backend then rejects the plan, responding with a 400 status and message like "Waypoint 2 is inside a restricted area (Airport XYZ). Adjust your route." If the path is clear, proceed.
- Optionally, check altitude against any altitude restrictions (e.g. if local law says max 120m AGL, ensure requested altitude <= 120m). If a waypoint altitude is too high, also reject.
- If all checks pass, the backend creates a new FlightPlan record in the DB with status "PENDING" or "APPROVED" depending on whether manual approval is required. For MVP, we might auto-approve if no violations (status = APPROVED/ACTIVE). If we include an admin review step, put status PENDING and notify admins (though notifying could be via an admin dashboard list rather than push).
- Save the waypoints: either as separate Waypoint records (linking to FlightPlan) or embed in a JSON field. A normalized approach is better design; we'll assume we create Waypoint entries in the Waypoint table for clarity.
- The backend returns a success response. This could include the flight plan details and its status. For example: `201 Created { flightId: 123, status: "APPROVED", message: "Flight plan submitted successfully." }`. If the plan was auto-approved, the frontend might also be informed that the flight can be started at the scheduled time. If it's pending, perhaps the pilot waits for admin approval (which could be out-of-scope for MVP to implement fully, unless we manually update DB or have a quick admin action in UI).

1. **User Feedback (Frontend):** Upon response, if it was rejected, the frontend shows the error message and perhaps highlights the offending waypoint on the map (we can do this if the message indicates which one). If accepted, the pilot sees the flight listed in their "My Flights" list, possibly with status. If the start time is immediate or very soon, we might allow them to press "Start" to initiate the flight (or it could auto-start at the given time in our simulation).

2. **Admin Approval (Optional):** In a scenario where admin oversight is needed, an admin user would log into their dashboard, see a list of pending flights, and have the option to approve or deny them. Approving would simply flip the status and maybe notify the pilot (for MVP, perhaps just change status which the pilot's UI could poll or refresh to see). Denying would set status "REJECTED" and possibly attach a reason. This feature can be included if time permits; otherwise, assume auto-approval.

Integration Considerations:
- The **Google Maps API** on the frontend can be used not only for picking waypoints but also to display an

overlay of restricted zones for user awareness. We could load known no-fly zones as GeoJSON and show them on the map in a distinct style (e.g. red shaded areas). This requires either hardcoding them in frontend or fetching from backend. A quick approach: have a pre-prepared list of zones and use the Maps API to draw circles/polygons. This way pilots can visually adjust their plan before submission.
- The **backend logic for geofence** could use simple math or a library (like Shapely in Python for polygon contains, if available). But given hackathon time, coding a point-in-circle check (distance formula) is trivial and covers most needs if we represent zones as center+radius. We'll ensure to convert lat/long to an appropriate projection or use haversine formula for distance calculation if needed (since working in degrees). For small distances and if all in one locale, a planar approximation is fine.
- The database should enforce that when a FlightPlan is created, the pilot and drone IDs are valid (foreign keys ensure referential integrity). We also might consider a uniqueness constraint like a drone can't have overlapping flights times. We won't delve deep on scheduling conflicts in MVP, but ideally if a drone is already scheduled at time X, another flight request for the same drone at overlapping time should be prevented. That check can be implemented by a query (find any flight of that drone where times overlap) on submission.

3. Real-Time Flight Monitoring

Workflow: Once a flight is active, the system provides real-time tracking on the map, and monitors for any violations or events, alerting users as needed.

- **Flight Start:** At the scheduled start time (or when a pilot/admin triggers it), we mark the flight as **ACTIVE**. For simulation, we can have the pilot click "Start Flight" on the UI (which calls `PUT /api/flights/{id}/start`). The backend will verify the flight is approved and not already started, then update its status to ACTIVE and record a `start_time_actual`. This action also kicks off telemetry streaming. If we implement auto-start at scheduled time, we could simply check in the telemetry loop and start when current time >= flight.start_time. But manual start gives more control during a demo.

- **Telemetry Simulation:** When a flight is active, the **Telemetry Service** comes into play. This could be a thread in the backend or an external process. For clarity, let's assume an external service (which could just be a Python script or even part of the backend application launched via threading for simplicity). The telemetry service knows the flight plan waypoints (it can fetch from the backend via an API call like `GET /api/flights/{id}` to get the route). It then generates a stream of positions: e.g., one could interpolate a straight line between Waypoint1 and Waypoint2 and move the drone along that path. For example, if the distance is D and we want updates every second for N seconds to reach the next waypoint, we increment lat/long gradually. A very simple approach is to linearly interpolate between successive waypoints over a fixed number of steps or based on time difference. The service then sends these coordinates to the backend at a set frequency (say, every 1 second). There are multiple ways to send:

- **Direct DB update:** not ideal for real-time, but one could update the drone's current position in the DB and have the backend query it. (This is heavy and slow; not recommended for real-time UI)
- **API POST:** e.g. `POST /api/telemetry` with a JSON {flightId, lat, lon, alt, status}. The backend receives it and immediately uses its WebSocket connection manager to broadcast to clients listening for that flight. This is straightforward: the telemetry service acts like a client sending data to our server, and the server redistributes.

- **Internal function call:** if the telemetry generator is a part of the backend process (like a background task), it could call the broadcast function directly without HTTP overhead. This is efficient but coupling is okay for MVP.

For hackathon simplicity, we might actually implement it as a background task within FastAPI: when flight starts, spawn a coroutine that sends updates. However, be mindful that if the process restarts, you lose that thread. A more robust approach is a separate service. But we can keep it simple.

- **WebSocket Broadcast:** FastAPI's WebSocket support allows broadcasting messages to multiple clients. We maintain a mapping of active WebSocket connections, possibly grouped by flight or user. For MVP, perhaps we broadcast all telemetry to all connected and let the frontend filter by flight ID (this is simplest). If there are many flights, this is a bit wasteful, but with few users in a demo it's fine. The telemetry message can be a small JSON like:

  ```
  {
    "flightId": 123,
    "droneId": "DR-001",
    "lat": 43.770,
    "lon": -79.410,
    "alt": 100,
    "status": "NORMAL"
  }
  ```

  The backend sends this out via the WebSocket. On the frontend, our open socket receives the message, we parse it and update the corresponding marker on the map. If the message's flightId doesn't belong to the current user (and user is not admin), the frontend can ignore it. Alternatively, to reduce noise, we could have separate channels like `/ws/{pilotId}` to only get that pilot's drones. But implementing separate channels might complicate the broadcast logic slightly, so broadcasting everything is acceptable initially.

- **Live Map Update (Frontend):** When the React app gets a telemetry message, it finds the map marker for that drone/flight and moves it (e.g. by updating the state that holds the marker's coordinates, causing the marker component to re-render at the new position). The map view can remain centered if the user chooses (maybe an option "follow this drone"). If multiple drones are active, all their markers update independently as data arrives. We might throttle the UI updates if messages are very frequent, but at 1 Hz or so it should be fine. The UI can also update other info – e.g., display the drone's current speed (which could be computed from position deltas if needed), or simply the coordinates.

- **Violation Detection & Alerts:** The telemetry service or backend will continuously check the drone's location against allowed parameters:

- If the drone goes beyond its planned path corridor (e.g., >100m away from any planned waypoint path), mark a violation.
- If the drone enters a restricted zone (perhaps not caught in planning if dynamic or edge case), immediately mark violation.

- If altitude exceeds plan or legal limit, violation.
- If no telemetry received for, say, >5 seconds (simulate signal loss), mark the drone as "Lost Signal".

How to implement: The telemetry generator can intentionally simulate an off-path excursion by deviating from the route randomly (to test violation handling), or the backend can detect based on comparing current coord to nearest waypoint. For MVP, a simple strategy: the flight plan defines a bounding box or radius that is "allowed" – say 200m radius around each waypoint or segment. If the drone's reported position falls outside that allowance, we flag it. When a violation is detected, the backend sends an alert message. This could be done via the same WebSocket (e.g. a message with `status: "VIOLATION", message: "Drone 123 left approved area"`). The frontend will catch that and perhaps show a notification (and could change the marker color to red). Similarly, for **signal loss**, if our telemetry service stops sending (we simulate by maybe pausing it for a duration), the frontend can notice the gap and mark it, or we have the backend send a special message: e.g. if no data for 5 sec, backend sends `status: "LOST"` for that drone.

- **Ending a Flight:** When the drone reaches the final waypoint (the telemetry simulation finishes the route), or if the pilot/admin clicks "End Flight", the flight is concluded. The telemetry service would send a final update or the backend marks the flight as **COMPLETED**. The backend can broadcast a message indicating completion (or simply the absence of further messages and a status change which front can query). We update the FlightPlan status in DB to COMPLETED for record. The map marker can be removed or changed to a "landed" icon. The flight is no longer active, so it might move from the "Active flights" list to a history list (if we implement one).

- **Concurrency & Load:** Real-time monitoring can generate a lot of messages, but our design keeps messages small and uses WebSockets (which are efficient for repeated sends). Even if we had, say, 50 drones sending 1 message per second, that's 50 msg/sec, which is trivial for the server and network. The main load is on the client rendering if many markers move, but that's manageable. Cloud Run can handle these small bursts easily, and we've noted the in-memory broadcast limitation (which is fine for single-instance initially). If we anticipated heavy use, we might incorporate a distributed pub/sub or a dedicated real-time channel solution.

Note: This real-time feedback loop is at the heart of demonstrating UTM functionality. Even for MVP, ensuring its reliability is important. We should test scenarios: normal flight, geofence breach, signal loss, and ensure the UI and system respond correctly.

Additional Features (Future Enhancements)

Should time permit during the hackathon, or in future development, the following features are desirable. They are not strictly required for the MVP, but we outline how they would fit into the architecture:

- **RTSP Video Stream Integration:** Many drones can broadcast a live video feed via RTSP. Natively, browsers cannot play RTSP streams [3], so we need a gateway to convert it to a web-friendly format. A common solution is to use a **WebRTC** or **HLS** converter. For example, an open-source tool like *RTSPtoWebRTC* can ingest the RTSP stream and serve it to the browser as a WebRTC stream [10]. Alternatively, FFmpeg can be used to transcode RTSP to HLS (HTTP Live Streaming) which the browser can play using an HTML5 video element or a library like hls.js. In our architecture, we could deploy an **RTSP media server** (perhaps as another microservice container on GCP) that the drone's feed is pushed to. The React frontend would then have a component (maybe on the flight monitoring page) that connects to this stream (for example, using a <video> element with a source URL to the HLS playlist or using a WebRTC player). We would also need to handle authentication for the video stream (only authorized users can view). This could be done by generating unique stream URLs per flight or by requiring a token to access the stream. Because of time constraints, an MVP might not implement the full video pipeline; instead, one could simulate by playing a prerecorded video or a placeholder. But architecturally, the design would simply add this component alongside telemetry. The key is decoupling: video data is heavy and should not go through our main backend; it should stream directly from the media server to the client. We just coordinate its start/stop.

- **Weather Data Overlay:** Knowing weather conditions (wind, rain, etc.) can be crucial for flight safety. We can integrate a weather API such as OpenWeatherMap. They provide both current weather APIs and weather map layers [11]. Two possible implementations:

- **Overlay Tiles:** OpenWeather (and others) offer tile layers (e.g., for precipitation or cloud coverage) that can be superimposed on a map by adding a tile layer URL. For example, Google Maps API allows custom tile overlays; we could use the Maps API to overlay a semi-transparent weather radar layer for the region. This requires only front-end work (fetching tiles via URL templates that include our API key).

- **Weather API Data:** Alternatively, the backend can call a weather API for specific coordinates (e.g., get wind speed at drone's location, or forecast along the route) and then feed that info to frontend. This could be done when planning a flight (to warn if bad weather expected) or in real-time (display current wind at drone's position). For MVP, a simpler approach is overlaying existing weather maps from a service.
  Implementation-wise, we'd add a component or layer on the frontend map. The user might toggle "Show Weather". The cost is minimal: just include the script or image layer. If using an API that requires server-side proxy (to hide key), our backend could fetch an image and serve it, but many map APIs allow client direct use with an API key.

- **Remote ID Emulation:** Remote ID is like an electronic license plate for drones, broadcasting identity and location. To emulate this, we can have our system output something akin to what a Remote ID receiver would see. For instance, the backend can provide a feed or endpoint that lists all active drones' public info (drone ID, location, pilot ID or anon ID). An admin could use this to see any drone in the airspace, even if not under their control (in our system, all drones are known, but imagine integrating unknown drones broadcasting remote ID). For MVP, a simple emulation: when a drone is active, our backend could periodically output a "Remote ID" message (which might just be the same telemetry but with some standardized format) that could be picked up by a hypothetical law enforcement app. We might not have a separate UI for this, but we can mention that our telemetry system could be extended to broadcast on a public channel. Since this is mostly conceptual, architecture doesn't change much – it's an additional format of output. If anything, ensure that pilot privacy is respected (in real Remote ID, the ID is typically a registration number, not personal info, so we would probably broadcast drone serial and location only, no personal data, which aligns with our data model where drone has an identifier).

All these features would be integrated in a modular way, not breaking existing structure. For example, adding RTSP streaming adds a Media Server component in the architecture that interacts with the frontend;

adding weather uses an external API to augment front-end; adding remote ID uses our existing data to produce an additional feed. The core architecture (React front, FastAPI back, DB) remains the backbone and is sufficient to handle these extensions.

Integration of Components and Communication

This section describes **how to wire up the frontend, backend, and services** together and the steps for development and deployment. Ensuring smooth integration involves configuring endpoints, managing environment variables (keys, URLs), and handling CORS and other cross-component issues:

- **API Contract between Frontend and Backend:** We will document the request/response format for each API route. During development, the team should agree on JSON schemas. For example, when the frontend POSTs a flight plan, it should know exactly what the backend expects (units, data types). Tools like Swagger UI (automatically available with FastAPI docs at `/docs`) can help the frontend developers see the API structure. We might also create a short README or wiki for developers listing all endpoints and example payloads.

- **CORS Configuration:** Because the frontend might be served from a different domain (especially in dev mode, e.g. localhost:3000 for React and localhost:8000 for API), we must enable CORS on the backend. FastAPI makes this easy with its CORSMiddleware configuration – we will allow the dev origin and in production, allow the domain where the frontend is hosted. This ensures the browser will permit the React app to call the API.

- **WebSocket Endpoint Usage:** The frontend must connect to the correct WebSocket URL. If using Cloud Run, the backend might be at an HTTPS URL (e.g. https://utm-api-xyz.run.app). For WebSockets, the URL would be `wss://utm-api-xyz.run.app/ws` (wss for secure). In development, if running locally, it's `ws://localhost:8000/ws`. We will ensure to use secure WebSockets (wss) in production by configuring Cloud Run with HTTPS (which it does by default) – the same domain can upgrade to WebSocket. The integration detail: once the user logs in and gets a token, the frontend will open the WebSocket connection including an authentication step. One approach: include the JWT as a query parameter (e.g. `wss://.../ws?token=XYZ`). The backend's websocket endpoint can then read that token and authenticate the connection (or if unauthorized, close it). Another approach is to use a cookie token and let it be sent automatically, but that's more complex. We'll likely go with the query param for simplicity in a hackathon.

- **Frontend ↔ Backend development workflow:** During development, to run the system:

- Start the database (e.g. run a local Postgres via Docker). Alternatively use SQLite for quick start (FastAPI and SQLAlchemy can use SQLite with minimal config, which avoids needing a separate DB while prototyping). We'll have to adjust connection strings accordingly.
- Start the backend server: `uvicorn main:app --reload`. This will serve at `http://localhost:8000` by default. Ensure any env vars (like DB URL, secret key, etc.) are set – maybe using a `.env` file that we load.
- Start the React dev server: `npm start` (which typically runs on `http://localhost:3000`). We should configure the React app's proxy or base URLs so that API calls go to the backend. For

example, in development, the React app can proxy `/api` calls to `localhost:8000` or we prefix the fetch URLs with the backend address. For WebSocket, use the proper local URL.
- Run the telemetry simulator: if it's part of backend (spawned thread on flight start), just triggering a flight will auto-start it. If it's external, we run a script like `python telemetry_sim.py` which perhaps connects to backend or calls an API repeatedly. We'd need to supply it with an API endpoint and maybe credentials. For MVP demonstration, it might be easier to manually trigger one flight at a time and have telemetry run internally to that backend process.

- With all pieces running, a developer can test the end-to-end flow: register user (via UI or adding directly in DB), login, create drone, plan flight, start flight, see map updates.

- **CICD and Tooling:** While not required, we could set up a simple GitHub Actions or Cloud Build pipeline to build and deploy the app to GCP. But given hackathon timing, manual deployment might be fine. Still, containerizing the app is important for Cloud Run. We will create a Dockerfile for the backend (the dev.to guide we cited provides a template [12]). For the React app, since it's static, we can either containerize it with an nginx serving the build output or use a GCP bucket.

- **Deployment Steps on GCP:**

- *Database:* Create a Cloud SQL for PostgreSQL instance. Set up credentials (username/password) and a database name. Enable the Cloud SQL Admin API (so Cloud Run can connect) [13] . Note the connection string or instance connection name.
- *Backend on Cloud Run:* Build a Docker image for the FastAPI app. Example Dockerfile: use Python 3.11 slim, copy code, install requirements, expose port, and use Uvicorn as entrypoint (as shown in the snippet [14]). Build and push this image to Container Registry or Artifact Registry. Then deploy to Cloud Run: specify the image, set environment variables for DB connection (e.g. `DB_HOST`, `DB_USER`, `DB_PASS`, etc.), and **attach the Cloud SQL instance** to the service (using the `--add-cloudsql-instances` flag or via console UI). This will allow the container to connect to the database via a Unix socket or TCP. In our code, we will use the host as a Unix socket path (`/cloudsql/<instance-connection-name>`) or a private IP if using that, as recommended by Google. Also, set an environment variable for our JWT secret key, and any other configs (like allowed CORS origins, etc.). Deploy with appropriate CPU/memory (default 256MB is fine to start) and allow at most 1 or 2 concurrent requests if using websockets (or use concurrency settings carefully). Cloud Run will provide a HTTPS URL for the service.

- *Frontend on Cloud:* Two approaches:

  1. **Static Hosting**: Build the React app (`npm run build`) which produces static files (HTML, JS, CSS). Upload these to a GCP Storage bucket (enable website hosting on it). Set up a Cloud CDN or use a Load Balancer to serve the static site globally. This method requires configuring the bucket to allow public read (or using CDN signed URLs). Alternatively, use **Firebase Hosting** (since it's part of GCP) which easily hosts SPAs and can also handle routing. Firebase Hosting would just need the build directory and a firebase.json config (with rewrite rules to index.html for SPA routes). It also provides a free SSL and domain.
  2. **Containerize and Cloud Run:** We can make a Docker image FROM nginx:alpine, copy the React build output to /usr/share/nginx/html, and deploy that to Cloud Run as well. This results in two Cloud Run services (one for API, one for front-end). We'd probably want to set a

custom domain or at least have them on the same domain for cookie sharing (if we used cookies). But with JWT in header, different subdomains are fine, just need CORS. This approach is slightly heavier but keeps everything in Cloud Run.

In either case, we'll end up with a URL for the frontend (e.g. https://utm-frontend.web.app if using Firebase, or Cloud Run URL). We must configure the frontend to know the API base URL (if on different domain). Usually, we set something like `REACT_APP_API_URL="https://<cloud-run-api-url>"` in the build.
- *Domain & HTTPS:* For a polished result, one might set up a custom domain (like utm-demo.com) and point it to the frontend (and perhaps proxy API under the same domain). This can be done with Cloud Run domain mapping or by hosting both behind a single reverse-proxy. But for MVP, using the provided *.run.app domain or a Firebase subdomain is acceptable. Those are all HTTPS by default.*
- Environment Variables: *Manage secrets like the JWT secret and database password using GCP Secret Manager or at least environment vars in Cloud Run (which are stored encrypted at rest). The Google Maps API key is needed on the frontend; we can inject it at build time or have the frontend fetch it from the backend (less ideal). Simpler: treat it as public but restrict its usage via Google Cloud's API restrictions to our domain.*
- Scaling Config: *On Cloud Run, we can set the max instances to a reasonable number (like 1-2 for demo, or higher if expecting many users). We should also enable Cloud SQL connections** efficiently (Cloud Run has connection pooling libraries or one can use PGbouncer, but for low load, direct connections are fine). The dev.to article suggests using the Cloud SQL proxy via flags [15] which is a common approach.

- **Logging and Monitoring:** Cloud Run and Cloud SQL provide logs. The backend should be configured to log important events (e.g., use Python's logging to log submissions, errors, etc.), which will appear in Cloud Logging. This helps in debugging if something goes wrong. If we had more time, setting up alerts or using Cloud Monitoring dashboards for our app could be done, but not necessary for MVP.

- **Developer Setup Documentation:** We will provide a README (in code or as part of this report) instructing how to run the project. It includes:

- Prerequisites: Node.js, Python, maybe Docker if using that, Google Cloud SDK if deploying.
- Steps to run backend: e.g., `pip install -r requirements.txt`, set env vars (DB connection, etc.), then `uvicorn main:app`. If using SQLite for local dev, instruct accordingly.
- Steps to run frontend: `npm install`, `npm start`. Where to configure the API URL (maybe a config file or use proxy settings in package.json).
- How to simulate telemetry: perhaps a script usage or note that starting a flight triggers it automatically.

- Any troubleshooting tips (common errors like CORS issues, DB connection errors).

- **Cleaning up & Future Work:** Emphasize that while the MVP is functional, further steps like writing automated tests, improving security (e.g., using HTTPS everywhere which we do in Cloud, but also considering rate limiting on API, etc.), and enhancing the UI/UX can be addressed after the hackathon.

Scalability, Robustness, and Clean Architecture

From the ground up, we designed this MVP with principles that ensure it can grow beyond a hackathon prototype:

- **Clean, Modular Architecture:** We separated the system into distinct layers: frontend vs backend, and within the backend we separate routing, business logic, and data access. This follows a basic clean architecture approach. For example, one could envision swapping the FastAPI REST interface with a GraphQL interface in the future without changing the underlying services, or moving from WebSocket to another protocol – because our core logic (in services) is decoupled from the transport details. Code will be organized into modules (e.g., `auth.py`, `flights.py` for routes, `models.py` for DB models, `schemas.py` for Pydantic models, `services/flight_service.py` for flight logic, etc.). This makes the codebase maintainable as features expand. The React frontend also uses reusable components and hooks to avoid monolithic scripts. This modularity aids scalability of development (multiple devs can work on different parts without stepping on each other's toes).

- **Scalability:** The choice of cloud-native deployment (Cloud Run, Cloud SQL) means the app can handle increasing load. Cloud Run will scale out additional instances if needed under high request volume. PostgreSQL can handle many concurrent transactions; if needed, we can scale it vertically or introduce read replicas for heavy read scenarios. The stateless nature of our API (aside from WebSocket connections) ensures scaling is straightforward. For WebSockets, as noted, we might employ a strategy to support multi-instance (e.g., a common Redis for pub/sub) if we foresee many users. Since the MVP might run on a single instance during the hackathon demo, we're safe, but it's good to know how to scale that later (e.g., using a library like Broadcaster to send messages across instances via Postgres NOTIFY or Redis pubsub [16]). In terms of *developer scalability*, using known frameworks (React, FastAPI) and standard protocols (REST/JSON, WebSocket) means future team members can quickly get up to speed.

- **Robustness & Fault Tolerance:** By leveraging managed services, a lot of heavy lifting is offloaded. Cloud Run will automatically restart containers if they crash. Cloud SQL manages backups and replication to avoid data loss. We should still code defensively: input validation prevents bad data from causing issues (e.g., ensure numeric fields are numbers, string lengths are reasonable to avoid injection issues). We will use try/except blocks around external calls (like if weather API fails, just log and continue without weather, rather than crash). The telemetry simulation should be written such that it can handle exceptions (for instance, if a client disconnects, FastAPI's WebSocket send might error – we should catch and handle that, removing that client from the list). We also consider network issues: WebSocket might disconnect – the frontend should be able to reconnect or at least alert the user. For MVP, a simple page refresh can restore it, but a more robust solution could be implemented with exponential backoff reconnects. Another aspect is **security**: we enforce auth on all sensitive operations. We use HTTPS in production so data (including JWTs) is encrypted in transit. Passwords are hashed so even if DB is leaked, they aren't in plain text. We also ensure that a pilot can only ever modify their own data – the backend checks the token's user ID against resource ownership (for example, if pilot tries to GET another pilot's flight, the service will deny it unless admin role). These checks prevent privilege escalation.

- **Maintaining Performance:** We design our data flows to be efficient. Using WebSockets for telemetry ensures low-latency updates without constant polling overhead. Database indices on foreign keys (like flights by pilot, waypoints by flight) will ensure queries remain fast as data grows. We avoid any heavy computation on the request path – e.g., no complex pathfinding or heavy analytics in the middle of a flight update. Those can be done offline if needed. The most computational thing might be no-fly zone polygon checks; with a small number of zones, that's negligible, but if it grew we could use spatial indexing or delegate to the DB with PostGIS. In short, the MVP is performant out-of-the-box for expected usage (a handful of concurrent flights/users). As usage scales, one can profile and identify bottlenecks, but our architecture has easy options for mitigation (scale out, caching, etc.).

- **Extensibility:** New features can be added without reworking the core. For example, if we want to integrate **drone command and control** (like sending commands to drones), we could add a microservice for commanding via MQTT, and connect it to our system (the backend could have an API for "return to home" which signals that service). Our architecture can accommodate that by adding new components; the central API can remain the orchestrator. Similarly, adding a new frontend view or an additional role (say, an "observer" role that can only view flights) would be straightforward. We'd just add new endpoints or reuse existing ones with different auth scopes, and update the UI. The database can also be evolved with migrations (using Alembic for SQLAlchemy or Django migrations), so future changes to schema are manageable.

- **Clean Code Practices:** We plan to use linters/formatters (like ESLint/Prettier for JS, black/flake8 for Python) to keep code quality high even in rapid development. This reduces errors and makes the codebase consistent. While testing might be minimal due to time, we can still test critical functions (like the no-fly zone check logic) in isolation. The separation of logic into functions means we could write simple unit tests for those if desired.

In summary, the architecture not only meets the immediate hackathon goals but also demonstrates a blueprint that is **scalable, secure, and maintainable** for real-world use. It combines proven technologies (React, FastAPI, PostgreSQL) in a cloud-native deployment to ensure that even as usage grows, the system can adapt and remain robust [1]. By adhering to clean architecture principles and thoughtful design (as evidenced by handling of multi-tenancy, real-time communication, etc.), this UTM MVP can evolve into a production-grade system with minimal refactoring.

Conclusion

This technical architecture outlines a comprehensive approach to building a UAV Traffic Management MVP that is feature-rich and future-proof. We addressed all core functionalities – from user authentication and multi-tenant data management to flight plan validation and live monitoring – using a cohesive stack suited for rapid development and easy scaling. The use of **React** and **FastAPI** accelerates development while ensuring high performance and clarity (their pairing is known to be effective for large-scale projects [17]). Deployment on **Google Cloud Platform** leverages managed services to achieve scalability, security, and minimal ops overhead [1].

By following this design, developers can implement the MVP in iterative steps (e.g., set up basic auth and data models first, then add flight submission, then real-time tracking, etc.), each step building on a solid foundation. The report also serves as documentation for setting up and integrating each component, which will help new team members or hackathon judges understand how to run and test the system.

In conclusion, the proposed architecture meets the hackathon requirements and provides a clear path for demonstrating a functional UTM system. It emphasizes clean separation of concerns, proper use of technology for each task (e.g., database for persistence, WebSocket for real-time, etc.), and readiness for enhancements like video and weather integration. With this plan, the team can confidently proceed to implementation, knowing that the result will be a robust MVP showcasing UAV traffic management in action.

Sources: (References to relevant materials and documentation have been preserved in the text above, indicated by 【†】 notation for clarity and further reading.)

---

[1] [4] [12] [13] [14] [15] Deploying a FastAPI + PostgreSQL App Using Google Cloud Run and Cloud SQL - DEV Community
https://dev.to/hexshift/deploying-a-fastapi-postgresql-app-using-google-cloud-run-and-cloud-sql-56c5

[2] [5] Add a Google map to a React app | Maps JavaScript API | Google for Developers
https://developers.google.com/codelabs/maps-platform/maps-platform-101-react-js

[3] [10] node.js - I need show "rtsp//:" live streaming on react - Stack Overflow
https://stackoverflow.com/questions/69159701/i-need-show-rtsp-live-streaming-on-react

[6] [16] WebSockets - FastAPI
https://fastapi.tiangolo.com/advanced/websockets/

[7] [8] [9] FastAPI Multi Tenancy | Class Based Solution | Medium
https://sayanc20002.medium.com/fastapi-multi-tenancy-bf7c387d07b0

[11] Weather maps 1.0 - OpenWeatherMap
https://openweathermap.org/api/weathermaps

[17] A Full-Stack Project With React and FastAPI — Part 1 - Medium
https://medium.com/this-code/a-full-stack-project-with-react-and-fastapi-part-1-82367f43911a