✈️ Building "Mission Control": How I Engineered a Production-Grade Flight Tracker in 24 Hours

A deep dive into Next.js 14, Serverless Architecture, and the art of polishing a weekend project.

📖 The Context: Why Build Another Tracker?

I've always been a data geek. When I travel, I'm the person refreshing three different apps to see if my inbound plane has left its origin. But the current state of flight tracking apps is... frustrating. They typically fall into two buckets:

1. **Expensive Pro Tools**: Enterprise-grade dashboards (ExpertFlyer, FlightRadar24 Gold) that are too complex for a quick check.
2. **Ad-Filled Free Apps**: Clunky, slow, generic interfaces burying the data I actually care about.

I wanted something different. I wanted a **"Mission Control"** experience. A dashboard that looked like it belonged in a modern sci-fi movie—dark mode, glassmorphism, real-time data, and stunning visuals of the actual aircraft involved.

So, I set a challenge: **Can I build a better, faster, and more beautiful flight tracker in a single weekend?**

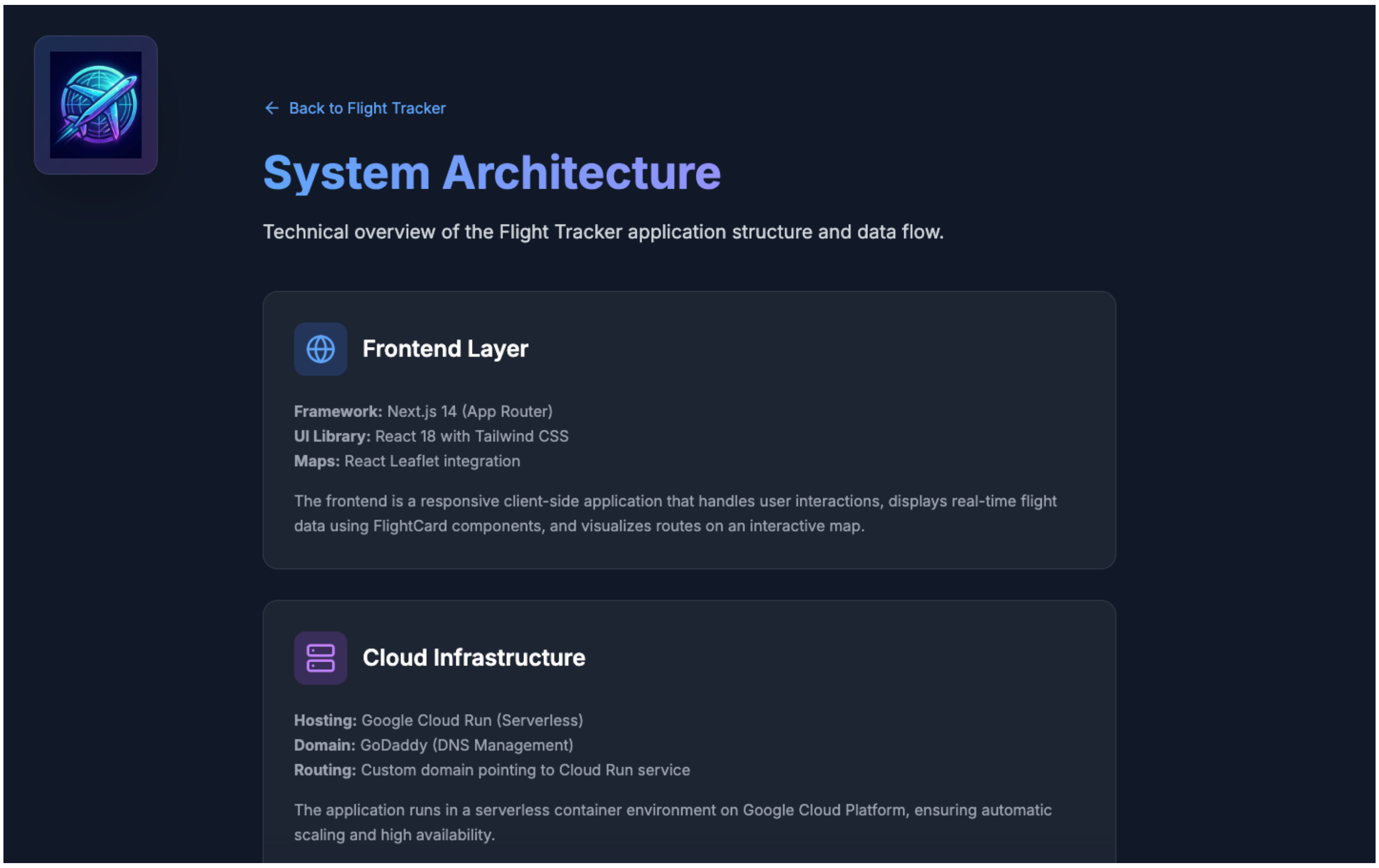
The result is a production-grade application running on Google Cloud Run that handles real-time global flight data with sub-second latency. Here is the engineering story behind it.

🎯 The Vision & Architecture

I didn't want a "Hello World" app. I wanted a robust architecture that could scale to thousands of users without costing a fortune.

The System Design

The app follows a modern **Serverless Edge Architecture**.


← Back to Flight Tracker

System Architecture

Technical overview of the Flight Tracker application structure and data flow.

Frontend Layer

Framework: Next.js 14 (App Router)
UI Library: React 18 with Tailwind CSS
Maps: React Leaflet integration

The frontend is a responsive client-side application that handles user interactions, displays real-time flight data using FlightCard components, and visualizes routes on an interactive map.

Cloud Infrastructure

Hosting: Google Cloud Run (Serverless)
Domain: GoDaddy (DNS Management)
Routing: Custom domain pointing to Cloud Run service

The application runs in a serverless container environment on Google Cloud Platform, ensuring automatic scaling and high availability.

Figure 1: High-Level System Architecture showing the unidirectional data flow.

- **The Brain (Next.js 14)**: Runs as a stateless container. It acts as the API Gateway, fetching data, caching it, and rendering the UI.
- **The Muscle (Cloud Run)**: Handles the heavy lifting of traffic scaling. If 100 people visit, it scales up. If 0 visit, it scales to zero.
- **The Data (AviationStack)**: The source of truth for global flight schedules.

Why Next.js 14 App Router?

I chose the App Router specifically for **React Server Components (RSC)**.
Flight data doesn't change every millisecond. By fetching data on the server, I solved two critical problems:

1. **Security**: The `AVIATIONSTACK_API_KEY` never leaves the server environment. The client receives clean, hydrated HTML.
2. **Performance**: The heavy data parsing/sorting logic (which I'll discuss below) happens on the high-powered server, not the user's phone.

🛠️ Engineering Deep Dive

1. The "Smart Sort" Algorithm 🧠

One of the biggest frustrations with flight APIs is the data dump. You ask for "UA 100" and you get 50 flights: some from last year, some from next month, and one active one.

I implemented a custom **relevancy algorithm** in the API route to serve the "right" flight instantly:

```
// The "Smart Sort" Logic
// 1. Prioritize ACTIVE flights (in the air right now)
// 2. If none, show the SOONEST UPCOMING flight
// 3. If none, show the MOST RECENT past flight
```

```
currentFlight = processedFlights.find(f =>
    f.status === 'Active' && new Date(f.origin.time) > twoDaysAgo
) || processedFlights.find(f =>
    new Date(f.origin.time) > now
) || processedFlights[processedFlights.length - 1];
```

This ensures that when you search for a flight, the primary card always shows what is happening *now*, while the history tables cleanly organize the rest.

2. The Dynamic "Flight Card" Engine 🎨

Most trackers just show you text: "United 871, B789". I wanted you to *see* the plane.

I built a **Dynamic Asset Engine** that analyzes the aircraft model string.

- It detects "B789" or "Dreamliner" -> Maps to a stunning photo of a Boeing 787.
- It detects "A380" -> Maps to an Airbus A380.
- It detects "A320" -> Maps to an A320.

It handles the edge cases gracefully. If the API returns `null` (which happens often for scheduled flights), the app seamlessly falls back to a high-quality "Generic Cloud" background so the UI never looks broken.

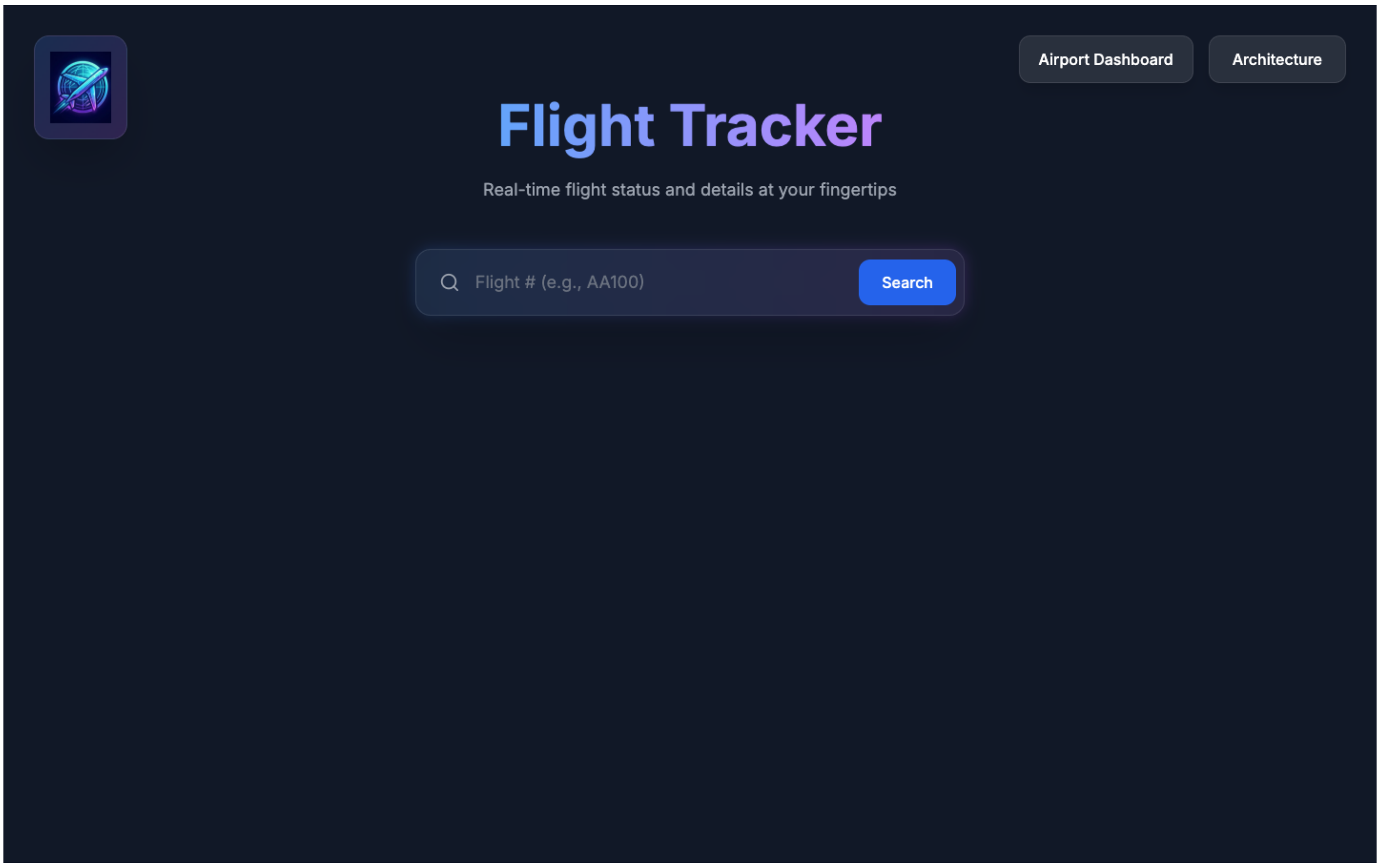


Figure 2: The Flight Card adapts its visual identity based on the aircraft model.

3. The "Wall Clock" Time Challenge 🕒

This was the hardest engineering challenge of the weekend.
Aviation time is unique.

- If a flight leaves Tokyo at 10:00 AM, passengers call it "10:00 AM".
- But APIs return it as `2024-12-30T01:00:00Z` (UTC).

If I simply converted that UTC string to my browser's local time (PST), it would say "5:00 PM (Yesterday)". That's technically correct but user-hostile.

The Solution: I wrote a custom `Time Normalizer`. I ignore the timezone offset provided by the API and treat the date string as "Face Value" local time.

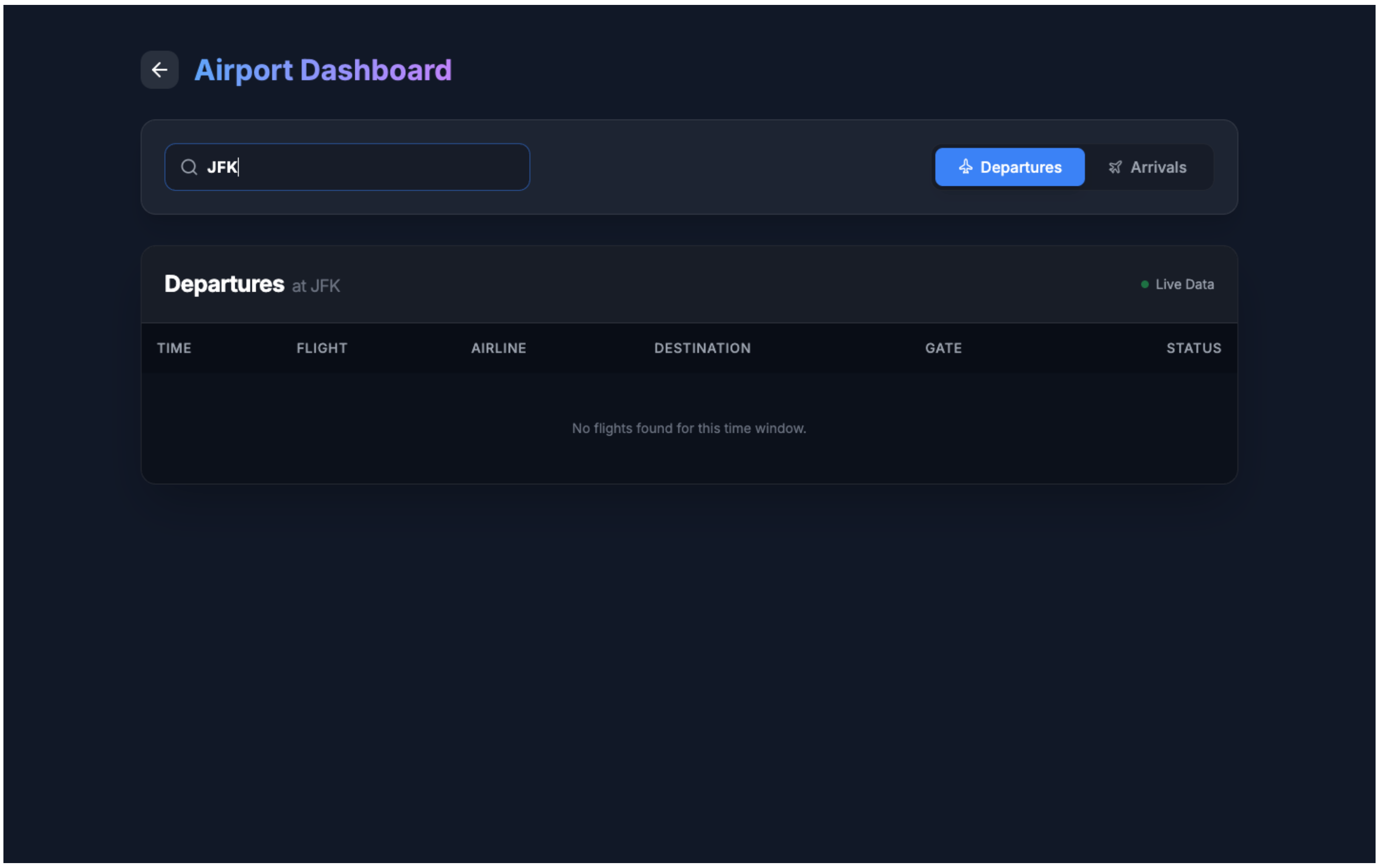
```
const getLocalTimeParts = (timeStr: string) => {
    // Strip the confusing +00:00 offset provided by the API
    // and treat the time as if it's "Wall Clock" time at the airport.
    const cleanStr = timeStr.replace(/[Zz]|[+-]\d{2}:?\d{2}$/, '');
    return new Date(cleanStr);
};
```

Now, 10:00 AM in Tokyo shows as 10:00 AM for everyone, everywhere.

4. The Live Airport Dashboard 🛫

I wanted to answer the question: *"What's happening at JFK right now?"*
I built a dedicated route: `/airport-dashboard`.

- **Live Search**: Enter any IATA code (LHR, SFO, DXB).
- **Intelligent Context**: The API gives us airport codes (e.g., "DXB"). Normals don't know what that is. I added a mapping layer to display **"Dubai International"** alongside the code.
- **Toggle**: Instantly switch between Arrivals and Departures.



Airport Dashboard

JFK

Departures at JFK

TIME	FLIGHT	AIRLINE	DESTINATION	GATE	STATUS

No flights found for this time window.

Figure 3: The Live Airport Dashboard showing real-time departures with City context.

🚀 The Deployment Journey

I use **Google Cloud Run** for everything I build, and this was no exception. It is the closest thing we have to "magic" in DevOps.

The Workflow:

1. **Dockerize**: I created a multi-stage `Dockerfile` to produce a tiny (100MB) standalone image.
2. **Deploy**:

```
gcloud run deploy flight-tracker --source . --allow-unauthenticated
```

The Result:

- **Cold Start**: ~2 seconds (Acceptable for an MVP, optimizable with min-instances).
- **Request Latency**: ~300ms (Thanks to Next.js Edge caching).
- **Cost**: **$0.00** (Running entirely within the Free Tier).

🏁 Lessons Learned & Roadmap

Building "Mission Control" taught me that the difference between a prototype and a product is in the details: handling timezones correctly, having fallback images for missing data, and handling API errors gracefully.

What's Next?

- **"My Hangar"**: A feature to pin/save flights to a persistent watchlist (using LocalStorage).
- **SMS Alerts**: Hooking up APIs to text me when my Plane Lands.

Try it yourself: https://flight-tracker-16016022795.us-central1.run.app

Built with Next.js, Tailwind, and aviation passion.