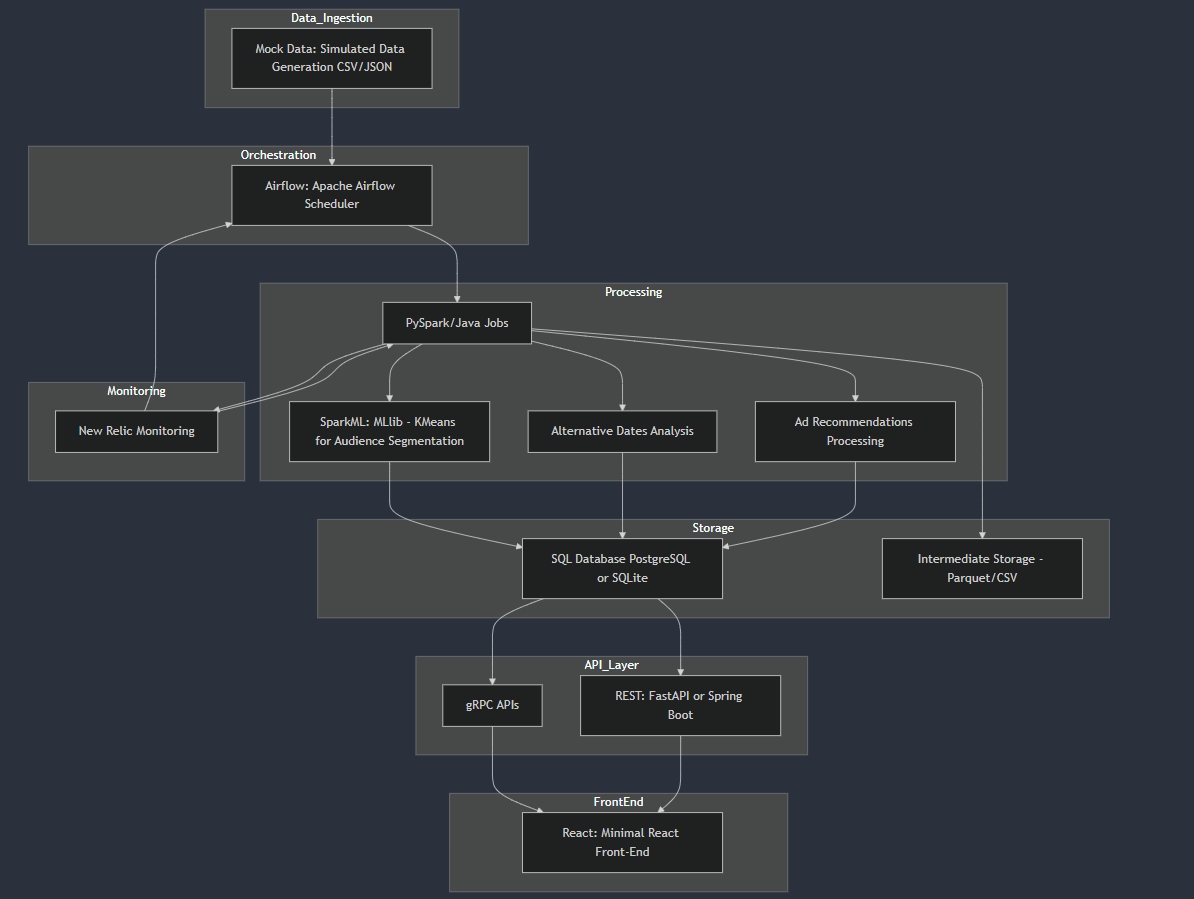
## ****FlightFlex System Design Overview****

****

### ****High-Level Workflow****

1. **Data Ingestion**: Mock flight logs, user behavior, and advertisement data ingestion.
2. **Orchestration**: Airflow schedules and manages Spark jobs for data processing.
3. **Data Processing Layer**:
   * Spark (PySpark/Java) processes Audience Segmentation, Alternative Dates, and Ad Recommendations.
4. **Monitoring**: New Relic tracks performance of Spark jobs and API health.
5. **Data Storage**: Results are stored in SQL (PostgreSQL) databases.
6. **API Layer**:
   * gRPC services expose processed insights.
   * FastAPI or Spring Boot provides endpoints to serve the processed data.
7. **Front-End UI**: Minimal React (TypeScript) front-end consumes APIs for visualization.

**Components and Flow Breakdown**

**1. Data Sources**

* **User Data**: Mock data of users and search logs.
  + Example schema:

user\_id, search\_date, origin, destination, price\_range, booking\_status

* **Flight Data**: Simulated flight schedule and price availability.
  + Example schema:

flight\_id, departure\_date, origin, destination, price, availability

* **Ad Inventory**: Mock ad data targeting specific user segments.
  + Example schema:

ad\_id, target\_segment, ad\_content, click\_through\_rate (CTR)

**2. Data Processing Layer**

* **Technologies**: PySpark (Spark Core, Spark SQL, MLlib), Databricks (optional for orchestration).
* **Spark Jobs**:
  + **Audience Segmentation**:
    - Use KMeans Clustering to group users based on flight search behavior.
    - Output table:

user\_id, segment

* + **Alternative Dates**:
    - Analyze price trends and availability for better dates.
    - Output table:

flight\_id, suggested\_date, price, availability

* + **Ad Recommendations**:
    - Join segmented user data with ad inventory to recommend ads.
    - Output table:

user\_id, recommended\_ads (list)

**3. Data Storage**

* **SQL Database**: PostgreSQL.
* **Tables**:
  + user\_segments: Stores user segmentation data.

user\_id, segment

* + alternative\_dates: Stores flight dates with alternative suggestions.

flight\_id, suggested\_date, price, availability

* + ad\_recommendations: Stores ads for specific users.

user\_id, ad\_content, CTR

**4. API Layer (Springboot)**

* APIs to serve processed data:
  1. **Audience Segmentation**:

GET /api/segments/{user\_id}

Response:

{

"user\_id": "123",

"segment": "Frequent Traveler",

"suggested\_ads": ["Business Class Deals", "Luxury Hotels"]

}

* 1. **Alternative Dates**:

GET /api/alternative-dates?origin=NYC&destination=LON&date=2024-01-15

Response:

[

{"date": "2024-01-16", "price": 450, "availability": "High"},

{"date": "2024-01-17", "price": 430, "availability": "Medium"}

]

* 1. **Ad Recommendations**:

GET /api/recommend-ads?user\_id=123

Response:

[

{"ad\_id": "456", "content": "Discount Flights to NYC", "CTR": 0.72}

]

**5. Front-End (Minimal React/TypeScript UI)**

* Use a simple React or Vue.js app to consume and display API outputs.
* Minimal components:
  + Table or JSON view for Audience Segmentation results.
  + Table for Alternative Dates suggestions.
  + List for Ad Recommendations.

**Where to Start: Step-by-Step Guide**

**Phase 1: Data Simulation**

1. **Create Mock Data**:
   * Use Python to generate user behavior logs, flight schedules, and ad inventory.
   * Save the mock data as CSV files.

Example Python Code:

import pandas as pd

from random import randint, choice

from datetime import datetime, timedelta

users = [{"user\_id": i, "search\_date": (datetime.now() - timedelta(days=randint(1, 30))).strftime("%Y-%m-%d"),

"origin": choice(["NYC", "SFO", "LAX"]), "destination": choice(["LON", "PAR", "BER"]),

"price\_range": randint(300, 1000), "booking\_status": choice(["Booked", "Not Booked"])}

for i in range(1, 100)]

pd.DataFrame(users).to\_csv("user\_logs.csv", index=False)

**Phase 2: Spark Processing**

1. **Setup Spark and PySpark**:
   * Ensure your local Spark is running (pyspark works without issues).
2. **Audience Segmentation**:
   * Write a PySpark job to perform clustering (KMeans from Spark MLlib).
   * Output the segmentation results into a CSV or SQL table.
3. **Alternative Dates**:
   * Process flight schedules to identify lower prices or better availability.
4. **Ad Recommendations**:
   * Join user segments with ad inventory.

**Phase 3: Database Integration**

1. Set up a local PostgreSQL database.
2. Load processed data (user\_segments, alternative\_dates, ad\_recommendations) into database tables.

**Phase 4: Build APIs**

1. Use **Springboot** to serve processed data from your database.
   * Endpoint for /api/segments/{user\_id}.
   * Endpoint for /api/alternative-dates.
   * Endpoint for /api/recommend-ads.

**Phase 5: Minimal Front-End**

1. Use React (TypeScript) to display:
   * Audience Segmentation results.
   * Alternative Date suggestions.
   * Ad Recommendations.

**System Design Summary**

| **Component** | **Technology** | **Description** |
| --- | --- | --- |
| Data Simulation | Python, Pandas | Generates mock user, flight, and ad data. |
| Data Processing | PySpark, Spark MLlib | Processes data for segmentation and analysis. |
| Data Storage | PostgreSQL | Stores processed data in SQL tables. |
| API Layer | Springboot | Serves APIs for segments, dates, and ads. |
| Front-End (Optional) | React + TypeScript | Consumes APIs and displays results. |

## ****Where to Start****

### ****Step 1****: Data Simulation

* Create mock data with Python and save it to CSV/JSON.

### ****Step 2****: Spark Jobs

* Write PySpark jobs for Audience Segmentation and Alternative Dates.
* Test locally with spark-submit.

### ****Step 3****: Airflow Integration

* Schedule Spark jobs in Airflow DAGs.

### ****Step 4****: Database Setup

* PostgreSQL for data storage.

### ****Step 5****: API Development

* Build gRPC APIs and optional Springboot endpoints.

### ****Step 6****: Monitoring

* Add New Relic for Spark jobs and API monitoring.

### ****Step 7****: Minimal Front-End

* Use React to consume APIs and display results.

### ****Step-by-Step Execution Workflow****

1. **Mock Data Generation** Run:

python scripts/mock\_data\_generator.py

1. **Spark Job Execution** Submit Spark jobs locally or to a cluster:

spark-submit spark/audience\_segmentation/segmentation.py

1. **Database Integration**
   * Load Spark outputs into PostgreSQL tables using PySpark or Python scripts.
2. **Run Airflow** Initialize Airflow with PostgreSQL:

airflow db init

airflow scheduler

airflow webserver

1. **Spring Boot API**
   * Serve processed data via REST or gRPC APIs.
   * Start the Spring Boot server:

mvn spring-boot:run

### Prioritized Execution Plan

Given your current status, here's the logical flow:

1. **Mock Data Generation**
   * Complete Python scripts for user\_logs.csv, flights.csv, and ads.csv.
2. **Set Up Spark**
   * Ensure Spark is installed and working locally or via Docker.
   * Write a simple PySpark job to process one mock dataset (e.g., user segmentation).
3. **Airflow Integration**
   * Write Airflow DAGs to run PySpark jobs.
   * Verify workflows in the Airflow UI.
4. **PostgreSQL Integration**
   * Create schemas and tables.
   * Use Python scripts or PySpark to load results into PostgreSQL.
5. **API Development**
   * Build and test APIs with dummy responses.
   * Connect APIs to the database.
6. **Front-End Development**
   * Create a minimal React UI.
   * Connect the UI to APIs and display results.
7. **Monitoring and Optimization**
   * Set up monitoring tools.
   * Optimize Spark jobs and database queries.

### ****Overview of Steps****

1. **Set Up AWS EMR (Elastic MapReduce) for Spark**
   * EMR is AWS's managed service for big data processing.
   * Provides pre-configured Spark clusters.
2. **Prepare Your Environment**
   * Install AWS CLI and configure credentials.
   * Set up S3 for storage of input/output data.
3. **Launch an EMR Cluster**
   * Use the AWS Management Console or AWS CLI to launch the cluster.
   * Install Apache Spark on the EMR cluster.
4. **Submit Spark Jobs**
   * Package your Spark application.
   * Upload to the cluster and submit jobs.
5. **Monitor and Manage**
   * Use CloudWatch and EMR's monitoring tools to track job progress and cluster performance.