1. Product Requirements Document (PRD)

Project Title: Cloud-Native Weather & Air Quality Analytics Platform

1.1. Vision:

To create a near real-time and historical analytics platform that ingests, processes, and visualizes weather and air quality data for selected regions, enabling data-driven environmental insights.

1.2. Goals & Objectives:

- Data Ingestion: Automatically collect weather and AQI data from public APIs for multiple, configurable regions every hour.
- Data Processing: Cleanse, transform, and enrich the raw data to make it analysis-ready.
- Data Storage: Store raw data cost-effectively and processed data in an optimized format for analytical querying.
- Orchestration: Build a robust, automated, and scheduled pipeline.
- Data Analysis: Enable SQL-based analysis on historical data.
- Visualization: Provide an interactive dashboard for end-users to explore trends in temperature, humidity, AQI, and pollutant levels (PM2.5, PM10, O3, etc.).
- Cloud Focus: Utilize core AWS serverless technologies to demonstrate modern data engineering practices.

1.3. Scope:

- In-Scope:
  - Data from free-tier weather/air quality APIs (e.g., OpenWeatherMap, AirVisual API).
  - Configuring 5-10 major cities (e.g., London, New York, Tokyo, Delhi, Sydney).
  - Hourly data ingestion.
  - Historical data load for the past 30 days (if API allows).
  - Dashboard with basic time-series charts and metrics.
- Out-of-Scope:
  - Predictive modeling or ML.

- Real-time (sub-minute) alerting.
- User authentication and multi-tenancy for the dashboard.
- Paid data sources or APIs.

1.4. Functional Requirements:

1. The system shall fetch current weather and AQI data for all configured cities every hour.
2. The system shall store a raw, immutable copy of each API response.
3. The system shall parse, flatten, and transform the nested JSON API responses into a structured tabular format.
4. The system shall load the transformed data into a data warehouse for analysis.
5. The system shall provide a dashboard with filters for date and city to visualize:
    - Time series of Temperature, Humidity, Pressure.
    - Time series of AQI and key pollutants (PM2.5, PM10).
    - Average AQI by city over a selected period.

1.5. Non-Functional Requirements:

- Reliability: The pipeline should have error handling and retry mechanisms.
- Cost-Efficiency: Use serverless and pay-per-use services to minimize cost, especially for a demo project.
- Maintainability: Code should be well-documented and infrastructure should be defined as code (IaC).
- Scalability: The design should be able to scale to ingest data for 100s of cities without a major re-architecture.

---

2. High-Level Design (HLD)

Architecture Diagram:

```
[External Weather API] -> [AWS Lambda (Ingestion)] -> [Amazon S3 (Raw Layer)]
      ^
      |
[EventBridge Scheduler]

[Amazon S3 (Raw Layer)] -> [AWS Glue (Spark ETL Job)]
      |                        |
      |                        v
```

```
|                [Amazon S3 (Processed Layer - Parquet)]
|                        |
|                        v
|                [AWS Glue Data Catalog] <- (One-time setup for processed data)
|                        |
|                        v
+----------------->[Amazon Athena]
                 |
                 v
           [Amazon QuickSight]
```

Data Flow:

1. Orchestration: An Amazon EventBridge Scheduler triggers an AWS Lambda function every hour.
2. Ingestion (Lambda): The Lambda function reads the list of target cities from a config file (e.g., in S3). It then iterates through each city, calls the relevant APIs, and writes each JSON response as a separate file to an S3 bucket (`raw-layer/date=YYYY-MM-DD/hour=HH/api_name/city_id.json`).
3. Transformation (ETL): A scheduled AWS Glue Spark job:
   - Reads the raw data from the Glue Catalog tables.
   - Flattens the nested JSON, extracts relevant fields, handles data type conversions, and joins weather and AQI data where possible (using city, timestamp).
   - Writes the processed data into another S3 bucket in Apache Parquet format, partitioned by date (`processed-layer/date=YYYY-MM-DD/data.parquet`).
4. Cataloging (Processed): A Glue Crawler scans the `processed-layer` bucket and creates a table like `processed_weather_aqi`.
5. Querying: Amazon Athena is used to run SQL queries on the `processed_weather_aqi` table via the Glue Data Catalog.
6. Visualization: Amazon QuickSight connects to Athena as a data source and builds the analytical dashboard.

---

3. Low-Level Design (LLD)

3.1. Data Sources & Ingestion (Lambda Function):

- API: OpenWeatherMap (Current Weather Data) & AirVisual (Air Quality). You might need to create free accounts.
- Lambda Runtime: Python 3.10

3.2. Data Storage:

- Amazon S3 Buckets:
  - `weather-project-raw`: Storage for raw JSON API responses.
    - Partitioning: `date=YYYY-MM-DD/hour=HH/api_name/`
  - `weather-project-processed`: Storage for processed data in Parquet format.
    - Partitioning: `date=YYYY-MM-DD/`
- Data Format: Apache Parquet for the processed layer. It's columnar, compressed, and efficient for Athena queries.

3.3. Data Transformation (AWS Glue Job - Spark):

- Job Type: Spark ETL Job (Python)
- Script Logic:
  1. Create DynamicFrames from the Glue Catalog tables for `raw_weather` and `raw_aqi`.
  2. Use `Relationalize` or `Spark SQL` functions to flatten the nested structures.
  3. Select and rename relevant columns.
     - *From Weather:* `dt (timestamp)`, `name (city name)`, `main.temp`, `main.humidity`, `main.pressure`, `coord.lat`, `coord.lon`
  4. Join the two datasets on city name and timestamp (might need tolerance for slightly different timestamps).
  5. Write the final DataFrame to S3 in Parquet format, partitioned by `date`.

3.4. Data Model (Processed Layer Schema):

Table Name: weather_analytics.`processed`

```
city_name STRING,
region STRING,
country STRING,
latitude DOUBLE,
longitude DOUBLE,
timezone STRING,
```

```
forecast_date STRING,
timestamp_epoch BIGINT,
observation_time STRING,
temperature_c DOUBLE,
temperature_f DOUBLE,
humidity INT,
pressure_mb DOUBLE,
wind_speed_kph DOUBLE,
precipitation_mm DOUBLE,
cloud_cover INT,
visibility_km DOUBLE,
uv_index DOUBLE,
processing_time TIMESTAMP
```

4. Tools & Technologies

- Ingestion & Compute: AWS Lambda (Serverless), Python (`requests`, `boto3`)
- Orchestration: AWS EventBridge Scheduler (Simple, serverless cron)
- Storage: Amazon S3 (Durable, cheap object storage)
- Data Catalog: AWS Glue Data Catalog (Central metadata repository)
- ETL: AWS Glue (Serverless Spark)
- Data Warehouse & Querying: Amazon Athena (Serverless SQL querying on S3)
- Visualization: Amazon QuickSight (Cloud-native BI tool)
- Infrastructure as Code (IaC): AWS CDK (Python/TypeScript) or Terraform (Highly Recommended for the project). This allows you to script the creation of all resources (S3 buckets, Lambda, Glue Jobs, etc.).

---

5. Implementation Plan & Milestones

1. Milestone 1: Setup & Configuration
    - Create AWS accounts (use Free Tier).
    - Sign up for API keys from OpenWeatherMap and AirVisual.
    - Write IaC script to create S3 buckets, IAM roles, and policies.
2. Milestone 2: Ingestion Pipeline
    - Write and deploy the Lambda function.
    - Configure EventBridge Scheduler to trigger it hourly.
    - Validate that raw JSON files are landing correctly in S3.

3. Milestone 3: Data Cataloging
    - Configure Glue Crawlers for the raw S3 data.
    - Verify that tables (`raw_weather`, `raw_aqi`) are created in the Glue Data Catalog.
4. Milestone 4: ETL Pipeline
    - Develop the AWS Glue ETL script (in Python).
    - Schedule the job to run after the hourly ingestion.
    - Verify that processed Parquet files are created in the `processed-layer` bucket.
    - Run a Glue Crawler on the processed data to create the `processed_weather_aqi` table.
5. Milestone 5: Analytics & Visualization
    - Run test SQL queries in Athena (e.g., `SELECT * FROM processed_weather_aqi WHERE date = '2023-10-27' LIMIT 10;`).
    - Connect QuickSight to Athena.
    - Build a dashboard with time series charts for temperature, AQI, and pollutant levels.
6. Milestone 6: Documentation & Cleanup
    - Document the project in a README (include architecture diagram, how to run, etc.).
    - Write a project summary/blog post.
    - (Optional) Create a GitHub repository with IaC code and Lambda/Glue scripts.
    - Important: Schedule a cleanup script to delete resources to avoid ongoing costs.