**Quantaco Data Engineer - Case Study**

**Walking Through the Solution and Thought Process**

This template is designed to help you guide the panel members through your solution in a simple, clear, and concise way.

**Walkthrough: Weather Data Pipeline Solution**

**1. Problem Statement (From Case Study)**

The goal was to build a pipeline to fetch weather data from an external API, process it, and insert it into a cloud-hosted database. The API should allow users to input:

* **Venue ID**
* **Start Date**
* **End Date**  
  It should then fetch **hourly weather data** and return a success or error message based on the data processing and storage.

**2. Solution Architecture Overview**

**Technologies Used:**

* **API Design**: OpenAPI 3.0 Specification
* **Business Logic**: Python (Flask-based API)
* **Cloud Platform**: AWS
* **Database**: MySQL on AWS RDS
* **API Gateway, Runtime Environment, and CI/CD Pipeline**
* **Source Code Repository**: [GitHub Repository Link](https://github.com/Praveengoud25/weather_app)

**3. Solution Components and Flow**

1. **User Input**: API accepts venue\_id, start\_date, and end\_date via POST request.
2. **Database Tables**:
   * Venue Table: Stores venue ID, name, latitude, and longitude.
   * Weather Table: Stores hourly weather data (temperature, humidity, precipitation, etc.).
3. **External Weather API Integration**:
   * Calls the Open-Meteo API with latitude, longitude, and date range to fetch hourly weather data.
4. **Data Processing**:
   * Data validation checks ensure that all required weather parameters are present.
   * Data is processed and formatted to align with the weather table schema.
5. **Database Insertion**:
   * Processed data is inserted into the MySQL database hosted on AWS RDS.
   * SQL joins allow validation and verification of the data for specific dates and venues.
6. **Error Handling**:
   * Meaningful error responses are returned in case of any API failure or data processing issues.
7. **API Response**:
   * On successful insertion, the API returns a success message. Otherwise, it returns an appropriate error message.

**4. CI/CD Pipeline Automation**

* **Source Repository**: The code is stored in a GitHub repository.
* **Pipeline Setup**: Automated CI/CD pipeline using tools like GitHub Actions or AWS CodePipeline to:
  + Run automated tests on each push.
  + Deploy the Flask-based API to AWS EC2 or AWS Lambda.
  + Deploy database migrations if applicable.

**5. SQL QA Checks for Data Consistency**

To ensure the output is consistent for frontend applications:

* **Validation of Weather Data**: SQL checks ensure no missing or duplicate weather data for specific venues and date ranges.
* **Joins Between Tables**: SQL joins between Venue and Weather tables validate hourly data.
* **Anomaly Detection**: SQL queries help detect outliers or unrealistic values (e.g., negative precipitation).

**6. Testing with Postman**

A **README.md** file is included in the repository, providing instructions on:

* Setting up the Flask API locally.
* Running the API with Postman or Curl.
* Testing API responses for both success and error scenarios.

**7. Challenges and Solutions**

* **Handling API Rate Limits**: Solution involved optimizing API calls and batching data requests to reduce API load.
* **Data Volume**: Apache Parquet format and indexing strategies were considered for future scalability.
* **Pipeline Failures**: Implemented retry logic and logging to CloudWatch for better

**QA Checks**

To apply QA checks in SQL on key dimensions/metrics and ensure that the output of your table remains consistent for frontend applications, you can follow these steps:

**1. Define Key Dimensions and Metrics**

* **Dimensions**: Attributes that define the context of the data (e.g., venue\_id, timestamp, latitude, longitude).
* **Metrics**: Numerical values that provide insights (e.g., temperature, humidity, precipitation).

**2. Identify QA Checks Based on Common Data Quality Issues**

You can apply SQL-based QA checks on key dimensions and metrics to ensure data consistency and integrity. Below are some common checks:

**a) Check for Null or Missing Values**

To ensure there are no missing data points for critical columns:

Sql:

SELECT \*

FROM weather

WHERE temperature IS NULL OR humidity IS NULL OR timestamp IS NULL;

If any rows are returned, it means there are missing values that need to be addressed.

**b) Check for Duplicate Records**

Ensure that there are no duplicate weather records for the same venue\_id, timestamp:

Sql:

SELECT venue\_id, timestamp, COUNT(\*)

FROM weather

GROUP BY venue\_id, timestamp

HAVING COUNT(\*) > 1;

This identifies duplicate rows, which may lead to inconsistent data on the frontend.

**c) Check for Outliers in Numerical Data**

Detect anomalies like unrealistically high or low values:

Sql:

SELECT \*

FROM weather

WHERE temperature < -100 OR temperature > 60; -- Example range for temperatures

Adjust the range based on realistic expectations for your dataset.

**d) Verify Data Completeness for Time Ranges**

Ensure that hourly data is available for each venue within a given date range:

Sql:

SELECT venue\_id, DATE(timestamp) AS date, COUNT(\*)

FROM weather

GROUP BY venue\_id, DATE(timestamp)

HAVING COUNT(\*) < 24; -- Check if fewer than 24 hourly records exist per day

If fewer than 24 records exist for any date, it indicates incomplete data.

**e) Ensure Referential Integrity Between Tables**

Make sure that every venue\_id in the weather table exists in the venue table:

Sql:

SELECT w.venue\_id

FROM weather w

LEFT JOIN venue v ON w.venue\_id = v.id

WHERE v.id IS NULL;

If rows are returned, it means there are orphaned records in the weather table.

**3. Automate SQL QA Checks**

Once you've identified the key QA checks, automate them using scheduled jobs or scripts.

* **Tools**: You can implement QA checks using:
  + **Airflow DAGs** to run SQL checks at regular intervals.
  + **Stored Procedures** to run a series of QA checks and log issues.
  + **Alerts** via email or Slack notifications if checks fail.

**4. Generate QA Reports**

Create summary reports showing:

* Number of null values, duplicates, or outliers detected.
* Percentage of data completeness (e.g., % of hours covered per venue/day).
* Anomalies or violations detected in key metrics (like temperature, humidity).

Example query for reporting:

Sql:

SELECT

COUNT(\*) AS total\_records,

SUM(CASE WHEN temperature IS NULL THEN 1 ELSE 0 END) AS null\_temperature\_count,

SUM(CASE WHEN humidity IS NULL THEN 1 ELSE 0 END) AS null\_humidity\_count,

SUM(CASE WHEN temperature < -100 OR temperature > 60 THEN 1 ELSE 0 END) AS outlier\_temperature\_count

FROM weather;

**5. Present QA Results to the Team**

After implementing the QA checks:

* Document the checks in your repository (e.g., in the README.md file).
* Share QA results and any anomalies detected with the engineering and data teams.
* Ensure feedback loops are in place to fix recurring data quality issues.