**Extra Credit - Deployment:**

To deploy the API, database, and a scheduled version of the data ingestion code on AWS, I would use the following tools and AWS services:

* **Amazon EC2 (Elastic Compute Cloud)**: For deploying the Flask application. EC2 provides resizable compute capacity in the cloud and allows us to run applications on virtual servers.
* **Amazon RDS (Relational Database Service)**: For hosting the PostgreSQL database. RDS manages database administration tasks like backups, software patching, and scaling, allowing us to focus on application development.
* **AWS Lambda**: For the scheduled version of the data ingestion code. Lambda allows running code without provisioning or managing servers. We can trigger Lambda functions using AWS CloudWatch Events, which can be scheduled to run at specific intervals.
* **Amazon S3 (Simple Storage Service)**: For storing static files, such as uploaded CSV files or any other resources required by the application.
* **Amazon VPC (Virtual Private Cloud)**: For configuring a virtual network environment in which the EC2 instances and RDS database can operate securely.
* **AWS CloudFormation**: For automating the deployment process by defining infrastructure as code. CloudFormation templates can be used to define the resources and configurations required for the deployment.
* **AWS IAM (Identity and Access Management)**: For managing access to AWS services securely. IAM roles and policies can be configured to grant permissions to resources and actions.

**Deployment Approach:**

1. **Setup Amazon RDS**: Create a PostgreSQL database instance using Amazon RDS. Configure security groups to control access to the database.
2. **Deploy Flask Application on EC2**:
   * Launch an EC2 instance and install the necessary dependencies (Python, Flask, etc.).
   * Deploy the Flask application on the EC2 instance. Configure security groups to allow inbound traffic on port 5000 (or any other port used by the Flask application).
   * Optionally, use Elastic Load Balancing (ELB) for distributing incoming application traffic across multiple EC2 instances.
3. **Scheduled Data Ingestion with AWS Lambda**:
   * Write the data ingestion code as a Lambda function.
   * Create a CloudWatch Events rule to trigger the Lambda function at the desired schedule (e.g., daily, hourly).
   * Configure the Lambda function to access the necessary AWS resources (e.g., S3 for storing files, RDS for database access).
4. **Set Up Amazon S3** (if required): Create an S3 bucket to store static files or any resources needed by the application.
5. **Configure Networking with Amazon VPC**:
   * Set up a VPC to isolate the deployed resources and control network access.
   * Configure subnets, route tables, and security groups to define network settings.
6. **Automate Deployment with AWS CloudFormation** (optional):
   * Define a CloudFormation template to provision and configure the required AWS resources.
   * Use the template to deploy the infrastructure consistently across different environments.
7. **Manage Access with AWS IAM**:
   * Create IAM roles and policies to grant the necessary permissions to AWS resources.
   * Ensure the principle of least privilege is followed to restrict access only to required resources and actions.
8. **Monitoring and Logging**:
   * Set up CloudWatch Alarms to monitor EC2 instances, RDS database, and Lambda functions for performance and health metrics.
   * Configure CloudWatch Logs to centralize logging and monitor application logs for errors and debugging.
9. **Scale and Maintain**:
   * Monitor the application's performance and scale resources as needed to handle changes in traffic or workload.
   * Regularly update and patch the application, database, and underlying infrastructure to ensure security and performance.

By leveraging these AWS services and tools, we can deploy the API, database, and scheduled data ingestion code reliably and securely in the cloud, ensuring scalability, availability, and ease of management.

**Problem 1:**

For this problem, since we have weather data from multiple stations and spanning several years, I'll design a relational database schema. Given that the data includes information about temperature and precipitation, I'll create tables to store this information.

I'll use PostgreSQL for this exercise. Here's a basic outline of the schema:

**Table:**

**weather\_data:**

* + **Purpose:** Stores individual weather data records.
  + **Columns:**
    - id: Unique identifier for each weather data record.
    - date: Date of the weather observation.
    - station\_id: Identifier for the weather station where the data was recorded.
    - max\_temp: Maximum temperature recorded on the given date.
    - min\_temp: Minimum temperature recorded on the given date.
    - precipitation: Amount of precipitation recorded on the given date.
  + **Constraints:**
    - id is the primary key.
    - date cannot be null.
    - station\_id cannot be null.
    - Unique constraint on (station\_id, date) to ensure no duplicate records for the same station and date.

**Problem 2:**

I'll write Python code to read the raw text files containing weather data, handle missing values, parse them, and insert the records into the PostgreSQL database. I'll use the ORM provided by SQLAlchemy to interact with the database. The code will handle duplicate entries and log the start and end times, as well as the number of records ingested.

While ingestion, the following log output is generated:

A screen shot of a computer

Description automatically generated

If code is re-run, and duplicates are detected but not ingested, thee following log output is generated:

A screen shot of a computer

Description automatically generated

**Problem 3:**

For this problem, I'll calculate the required statistics (average max temperature, average min temperature, total precipitation) for every year and every weather station. Missing data will be ignored while calculation. Then, I'll design a new data model to store these calculated results. I'll use a similar relational schema with additional tables to store the statistics:

**Table:**

**weather\_statistics**:

* + **Purpose**: Stores aggregated weather statistics data.
  + **Fields**:
    - **id**: Unique identifier for each weather statistics record.
    - **year**: Year for which the statistics are calculated.
    - **station\_id**: Identifier for the weather station these statistics belong to.
    - **avg\_max\_temp**: Average maximum temperature for the year.
    - **avg\_min\_temp**: Average minimum temperature for the year.
    - **total\_precipitation**: Total precipitation for the year.
  + **Constraints**:
    - **id** is the primary key.
    - **year** cannot be null.
    - **station\_id** cannot be null.

**Problem 4:**

I'll choose Flask as the web framework to create a REST API. I'll implement two GET endpoints **/api/weather** and **/api/weather/stats** to retrieve weather data and statistics, respectively. These endpoints will accept query parameters for filtering by date and station ID, and the responses will be JSON-formatted and paginated. Additionally, I'll include Swagger documentation for the API.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A black rectangular object with white border

Description automatically generated

A screenshot of a computer

Description automatically generated

A black and white lines

Description automatically generated with medium confidence