Section 1: Project Goal

Investigating Seasonal and Daily Trends in PM2.5 and PM10 Pollution

1.1 Scope

- **Cities:** Canberra (Australia), Amsterdam (Netherlands), Lima (Peru)
- **Timeframe:** 2022
- **Pollutants:** PM2.5 (fine particulate matter), PM10 (coarse particulate matter)

1.2 Objectives

2. **Seasonal Analysis:**
    - Uncover trends in PM2.5 and PM10 concentrations throughout the year in Canberra, Amsterdam, and Lima.
    - Identify periods with the highest and lowest pollution levels within each city.
3. **Daily Analysis:**
    - Examine day-to-day fluctuations in PM2.5 and PM10 levels within each city.
    - Determine weekdays vs. weekends pollution patterns.
    - Investigate if there are specific days with consistently high pollution.

1.3 Significance

This project aims to provide valuable insights into air quality patterns across diverse geographical locations. Understanding seasonal and daily pollution trends is crucial for:

- **Public Health:** Identifying periods of elevated risk and informing air quality advisories.
- **Environmental Policy:** Developing targeted strategies to mitigate pollution sources and improve air quality.
- **Urban Planning:** Guiding decisions on infrastructure and development in a way that minimizes pollution exposure. **Section 2: Background and Context**

Section 2: Background and Context

2.1 Air Pollutants: PM10 and PM2.5

- **PM10 (Particulate Matter 10 micrometers or less):** Coarse particles such as dust, pollen, and mold spores. Sources include construction sites, unpaved roads, and natural windblown dust.
- **PM2.5 (Particulate Matter 2.5 micrometers or less):** Fine particles generated by combustion processes (vehicles, power plants, wildfires), and chemical reactions in the atmosphere. PM2.5 can penetrate deep into the lungs and pose health risks.

2.2 City Profiles

City	Population (approx.)	Geographical Location	UTC Offset	Key Indus
Canberra	450,000	Southeastern Australia	UTC +11	Governmer
Amsterdam	900,000	Western Netherlands	UTC +1	Finance,
Lima	10 million	Coastal desert region of Peru	UTC -5	Mining, M

Section 3: Data Acquisition

3.1 OpenAQ Platform

- The OpenAQ Platform (https://openaq.org/) provides global real-time and historical air quality data.
- To access data, I've obtained an API key as per the OpenAQ API documentation (https://docs.openaq.org/).

3.2 API Familiarization

- Thorough review of the OpenAQ API documentation, specifically the "Getting Started" section (https://docs.openaq.org/docs/getting-started).
- Understanding of fundamental API endpoints and available parameters for data retrieval. **3.3 Exploratory API Calls**
- Initial test calls conducted to examine the structure of OpenAQ API responses, including:
  - Available cities, measurements, locations, and parameters.

3.4 Challenges and Adaptations

- **Challenge:** Limited availability of city-wide pollution data in the "city" parameter.
- **Solution:** Strategic decision to target specific sensor locations near major roads to ensure consistent data coverage.
  - Lima: SANTA ANITA
  - Canberra: Civic
  - Amsterdam: Amsterdam-Einsteindwag

3.5 Data Consistency Observations

- **Finding:** PM2.5 and PM10 measurements were consistently available across all targeted sensor locations.
- **Observation:** Other pollutants demonstrated sporadic availability.

Section 4: Data Extraction and Handling

4.1 Extraction Strategy

- Focus on hourly air quality measurements of PM2.5 and PM10 for 2022 from locations:

  - SANTA ANITA (Lima)
  - Civic (Canberra)
  - Amsterdam-Einsteindwag (Amsterdam)
- Employed targeted extraction from sensor locations near major roads for maximum data consistency.

4.2 Code Implementation (`extract.py`)

`[Include your Python code here]`

4.3 Code Functionality

- **API Interactions:** Leverages the OpenAQ API (v2) to retrieve air quality data based on location, date ranges, and desired parameters.
- **CSV Storage:** Efficiently stores extracted measurements into separate CSV files per parameter for organized data management.
- **Rate Limiting Handling:** Implements basic rate-limiting checks and sleep functions to mitigate exceeding API usage limits (HTTP 429 errors).
- **Error Handling and Logging:** Incorporates robust error handling (retries) and detailed logging for debugging and monitoring the extraction process.
- **Command-Line Interface:** Provides flexibility with command-line arguments for specifying location, dates, and parameters

4.4 Overcoming Challenges

- **Rate Limiting:** Addressed by introducing strategic pauses (sleep) when approaching API request limits.
- **Connection Interruptions:** Mitigated with a retry system and logging to ensure continuity and maintain data extraction integrity.

Section 5: Data Cleaning and Preprocessing

5.1 Data Quality Assessment

- Initial examination of extracted data revealed the following issues:
  - **Missing Values (NaN):** Presence of missing values at specific hourly intervals.
  - **Invalid Measurements (-999):** Existence of sensor readings marked as `-999`, indicating potential errors or sensor malfunctions.

5.2 Data Cleaning Strategies

- **Missing Value Handling:**
  - Employed median imputation with forward and backward fill (`df.fillna()`) to address missing hourly data points, allowing for continuous time series representations.
- **Invalid Measurement Handling:**

– Replaced invalid measurements (`-999`) with `NaN` to ensure compatibility with imputation and analysis methods.

5.3 Outlier Detection

- **Implementation:** Implemented outlier detection using a Z-score based approach in the provided `check_missing_times_and_outliers()` function.
- **Rationale:** Identifying unusual or extreme data points that may be caused by measurement errors or exceptional events, improving overall data quality for analysis.

5.4 Code Implementation (`clean.py`)

[Include your Python code here]

5.5 Code Functionality

- **Data Loading and Conversion:** Loads CSV data, formats the 'datetime' column as proper timestamps, and sets `datetime` as the index.
- **Missing Time Detection:** Identifies missing hourly timestamps by resampling to hourly frequency and comparing indexes.
- **Outlier Detection:** Calculates Z-scores for measurements and flags data points exceeding a specified threshold (default: 3 standard deviations from the mean).
- **Output:** Reports missing times and detected outliers.

Section 6: Database Integration

6.1 Database Selection:

- **Choice:** PostgreSQL selected as the local database solution due to its reliability, robust SQL support, and compatibility with data analysis tools.

6.2 Table Creation and Data Loading (`load.py`)

[Include your Python code here]

6.3 Code Functionality

- **Database Connection:** Establishes a connection to the target PostgreSQL database using user-provided credentials.
- **Table Management:** Creates a PostgreSQL table (if it doesn't exist) with `datetime` as the primary key and a `value` column (real data type). Handles potential naming conflicts from location identifiers by replacing dashes (`-`) with underscores (`_`).
- **Data Insertion:** Efficiently reads CSV data (assuming `datetime` and `value` columns) and inserts it into the database using prepared statements (preventing SQL injection) via `executemany()`.

6.4 Challenges and Solutions

- **Naming Conflicts:** Table creation errors caused by dashes in location names were resolved by converting dashes to underscores programmatically.
- **Edge-Case Missing Values:** Remaining missing values at the beginning and end of datasets were addressed by incorporating forward fill (`ffill`) and backward fill (`bfill`) into the `clean.py` script.

Section 7: Configurable and Dynamic ETL

7.1 Configuration Management (`config.yaml`)

- Employs a YAML configuration file to define and centralize essential ETL parameters:
  - `extract`:
    - `locations`: List of target sensor locations.
    - `parameters`: List of air quality parameters (pollutants) to extract.
    - `start_date`/`end_date`: Defines the desired time range for data extraction.
  - `load`:
    - `host`: PostgreSQL database host.
    - `dbname`: PostgreSQL database name.
    - `user`: PostgreSQL user with access.

7.2 ETL Orchestration (`etl.py`)

[Include your Python code]

7.3 Code Functionality

- **Parses Configuration:** Reads and loads parameters from `config.yaml`.
- **Extract Phase:**
  - Iterates through configured locations and parameters.
  - Calls `scripts/extract.py` using `subprocess.run()` for each location/parameter combination, handling date formatting.
- **Clean Phase:**
  - Dynamically identifies generated CSV files with the `measurements_*.csv` pattern.
  - Executes `scripts/clean.py` via `subprocess.run()` to apply cleaning to all extracted data files.
- **Load Phase:**
  - Parses filenames to derive unique database table names using the `get_table_name()` function.
  - Constructs and executes `scripts/load.py` commands using dynamically derived table names and database credentials from `config.yaml`.

7.4 Flexibility and Maintainability

- Modularizes the ETL process by separating extraction, cleaning, and loading steps into individual scripts.
- Allows easy modification of extraction targets, timeframes, and database settings by editing the `config.yaml` file, avoiding hardcoded changes within the orchestration script.