# Documentation

## Documentation: Weather & Wine Recommendation System

### Overview

This project fetches weather and wine data, merges them, stores them in a PostgreSQL database, uses OpenAI's GPT to generate wine recommendations based on current weather, and exposes the functionality through a FastAPI backend.

### Modules Breakdown

#### `wine_fetch.py` – Fetch Wine Descriptions

- **Purpose**: Uses the Spoonacular API to fetch wine descriptions for a predefined list of wines.
- **Why**: Descriptive metadata about each wine helps the language model make more context-aware recommendations.
- **Key Features**:
    - Asynchronous API calls using `aiohttp`.
    - Saves data to `Wine_train.json`

#### `data_fetch.py` – Weather Data Collection

- **Purpose**: Uses the OpenWeather API to fetch **current weather data** for a predefined list of cities.
- **Why**: Weather conditions (e.g. temperature, feels-like temperature) influence wine preferences (e.g. bold reds vs. light whites). Supplying weather data helps the language model make **contextual and personalized wine recommendations**.

- **Key Features**:
  - Asynchronous weather data fetching using `aiohttp` for efficient parallel API calls.
  - Extracts relevant metrics such as temperature and feels-like temperature in both Celsius and Fahrenheit.
  - Saves:
    - Raw weather data to `Weather_train.json`
    - Cleaned and structured data to `weather_cleaned.json`

## `merge_data.py` – Merge Weather and Wine Data

- **Purpose**: Merges weather data and wine descriptions into a single JSON file (`merged_data.json`).
- **Why**: Centralizing the data allows the LLM module to access and reason over a unified structure.
- **Key Features**:
  - Asynchronous file I/O using `aiofiles`.
  - Structure ensures the merged output includes keys `"weather"` and `"wine"` for easy parsing downstream.

## `Database.py` – Store Merged Data in PostgreSQL

- **Purpose**: Inserts the content of `merged_data.json` into a PostgreSQL table (`merged_data`).
- **Why**: Persistent storage ensures data availability across sessions and scales better than keeping everything in memory.
- **Key Features**:
  - Auto-creates table if it doesn’t exist.
  - Stores entire JSON blob for flexibility in future querying or auditing.

### `llm.py` – Generate LLM-Based Wine Recommendations

- **Purpose**: Uses OpenAI's GPT to generate summaries that recommend a wine based on a user's weather-based query.
- **Why**: The power of LLMs allows for rich, contextual wine recommendations by interpreting temperature and wine features.
- **Key Features**:
    - Extracts the city from the query.
    - Looks up the weather for that city.
    - Constructs a natural-language prompt to GPT-4.
    - Stores unique summaries in the `analysis_summaries` table in PostgreSQL.
    - Avoids duplicate entries using a pre-check.

### `main.py` – FastAPI Web Server

- **Purpose**: Hosts three API endpoints to interact with the system.
- **Why**: Provides an interface for external clients to use the system programmatically.
- **Endpoints**:
    - `POST /fetch_and_process`: Accepts a query like *"What's the weather in Paris and what wine suits it?"* Calls the LLM and stores result.
    - `GET /results`: Returns all LLM-generated recommendations, optionally filtered by city.
    - `GET /analysis`: Returns only the latest summary (or latest per city).

## Data Flow Overview

1. `wine_fetch.py` + (external weather data source): Generate base JSONs.

2. `merge_data.py`: Combines both datasets into `merged_data.json`.
3. `Database.py`: Saves this merged data to the PostgreSQL database.
4. `llm.py`: Reads from the merged file, calls GPT, and stores summaries.
5. `main.py`: API interface to query the system and access summaries.

## Files Summary

| File | Role |
|---|---|
| `wine_fetch.py` | Fetch wine data from Spoonacular API |
| `merge_data.py` | Merge wine + weather JSON into unified format |
| `merged_data.json` | Output file from merge step (used throughout the system) |
| `Database.py` | Push merged data into PostgreSQL |
| `llm.py` | GPT logic to generate and store recommendations |
| `main.py` | FastAPI backend for querying and managing data |
| `.env` | Secure storage of API keys and DB credentials (not shared here) |

## Why Use GPT/LLM?

- The use of GPT enables **natural language interpretation** and **contextual wine pairing** that would be difficult to hardcode.
- By combining weather details (e.g., *"feels like 34°C"*) with nuanced wine descriptions, GPT provides thoughtful and dynamic recommendations.

## Environment Variables

The system relies on a `.env` file with:

```
env
CopyEdit
POSTGRES_DB_URL=your_postgres_connection_url
```

```
OPENAI_API_KEY=your_openai_api_key
SPOONACULAR_API_KEY=your_spoonacular_api_key
```

Kindly use own api-keys

Open-Weather = “”

spoonacular= ” ”

Open_API_key = “”

PostgressSQL = “”

## Why This Architecture?

- **Separation of concerns**: Each script handles one task (data fetching, merging, storing, generating, serving).
- **Scalability**: Modular design allows replacing components (e.g., new wine API or weather provider).
- **Efficiency**: Async operations for fetching and merging improve performance.
- **Reliability**: PostgreSQL provides durable storage, while LLM ensures rich content generation.

## LLM Integration Overview

This is a visual representation of how the **LLM (Large Language Model)** integrates the weather and wine data:

- The **weather API** provides city-specific temperature and climate data.
- The **wine API** offers descriptive profiles of various wines.
- When a user asks, for example, *“What wine should I have in Paris?”*, the system:
  1. Looks up the **current weather** in Paris.
  2. Uses GPT to interpret both the **weather** and the **wine dataset**.
  3. Generates a personalized **wine recommendation**.

## Adaptive Recommendations

The output varies based on city and weather. For example:

- In **Paris** with warm weather, it may suggest a light, crisp white wine like **Riesling**.
- In **Berlin** on a cooler day, it might recommend a bold red like **Cabernet Sauvignon**.

## Illustration

This diagram shows the flow of data and logic between the components: