Python WeatherDashboard

PROJECT REPORT

Submitted by:

Dibakar Mohanta

Reg No: 25BHI10036

Computer Science & Engineering(Health Informatics)

Date:

23 November 2025

Abstract- This project report details the development of a Python-based Weather Dashboard. The application leverages the OpenWeatherMap API for real-time data retrieval, Tkinter for the Graphical User Interface, and Pillow for dynamic image processing. The system provides a full-screen, visually immersive experience that adapts to current weather conditions.

1 Introduction

The ”Python Weather Dashboard” is a desktop application designed to provide real-time weather information in a visually engaging format. Unlike standard text-based weather reports, this application utilizes a Graphical

User Interface (GUI) to display weather data on a ”card” overlay, while dynamically changing the background image of the application to match current weather conditions (e.g., displaying a rainy background when it is raining).

The project leverages the **OpenWeatherMap API** to fetch live data and uses Python’s **Tkinter** library for the interface and **Pillow (PIL)** for advanced image processing.

2 Problem Statement

Users often find raw weather data (numbers and text) unengaging and difficult to visualize quickly. While many websites exist, they often require navigation through complex menus. The problem this project solves is the need for a **minimalist, full-screen visual dashboard** that provides immediate context about the weather through visual cues (dynamic backgrounds) alongside precise meteorological data.

3 Functional Requirements

The system is designed to fulfill the following functions:

- **Input Mechanism:** The system must accept a city name from the user via a clear input dialog box.
- **Data Retrieval:** The system must connect to the OpenWeatherMap API using a valid API key to fetch real-time weather data (JSON format).
- **Data Parsing:** The application must extract specific data points: Temperature, Humidity, Wind Speed, Weather Description, and Weather ID.
- **Dynamic Visualization:** The system must analyze the ”Weather ID” to categorize the weather (e.g., Rain, Snow, Clear) and update the background image accordingly.
- **Error Handling:** The system must display user-friendly error pop-ups for scenarios such as ”City Not Found” or ”Connection Error”.

4 Non-functional Requirements

- **Usability:** The interface is designed as a full-screen application to minimize distractions and focus solely on the data.
- **Performance:** Background images must be resized efficiently using high-quality resampling filters (LANCZOS) to fit the user’s screen resolution without pixelation.
- **Reliability:** The application includes exception handling to prevent crashes during network timeouts or invalid API responses.

5 System Architecture

The program follows a modular architecture where the Frontend (GUI) is decoupled from the Backend (Logic) and External Services. The flow is linear:

1. **Initialization:** The main() function triggers the application.
2. **User Input:** get_city_prompt() captures the target city.
3. **API Request:** get_weather(city) constructs the URL and sends an HTTP GET request.
4. **Logic Processing:** Data is parsed, and the Weather ID is mapped to a specific image filename.

5. **GUI Rendering:** The GUI() function builds the window, resizes the image using PIL, and places text widgets.

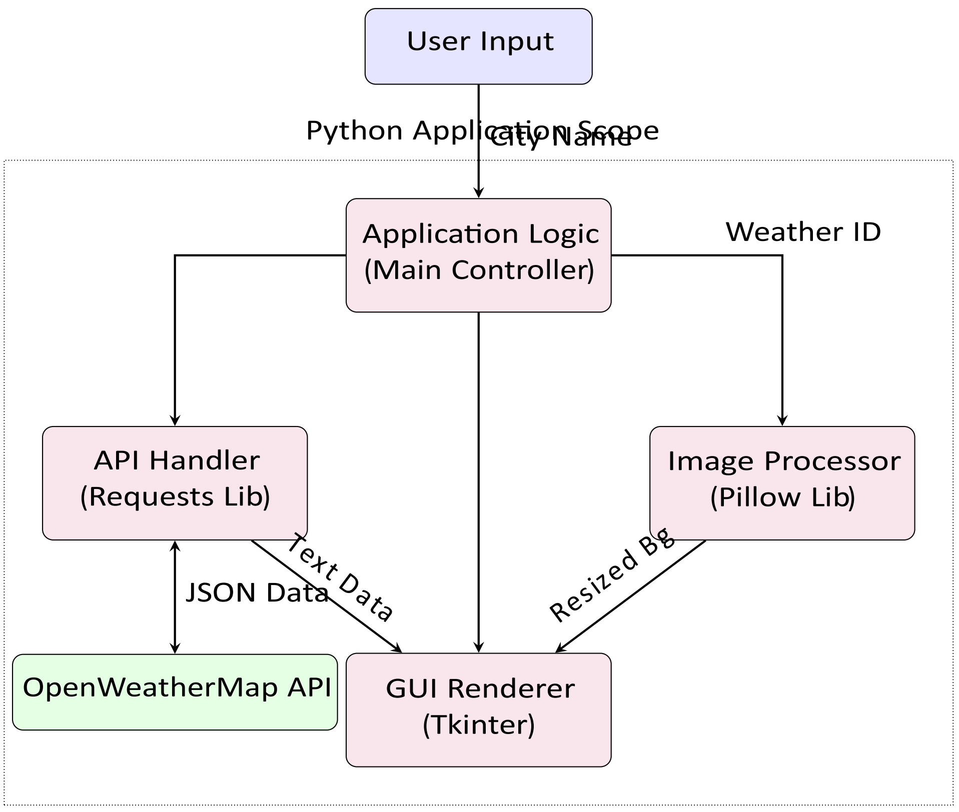


Figure 1: System Architecture Block Diagram

6 Design Diagrams

6.1 A. Use Case Diagram

The following diagram illustrates the interactions between the User and the System functionalities.

6.2 B. Workflow Diagram

The flow of logic from start to finish.

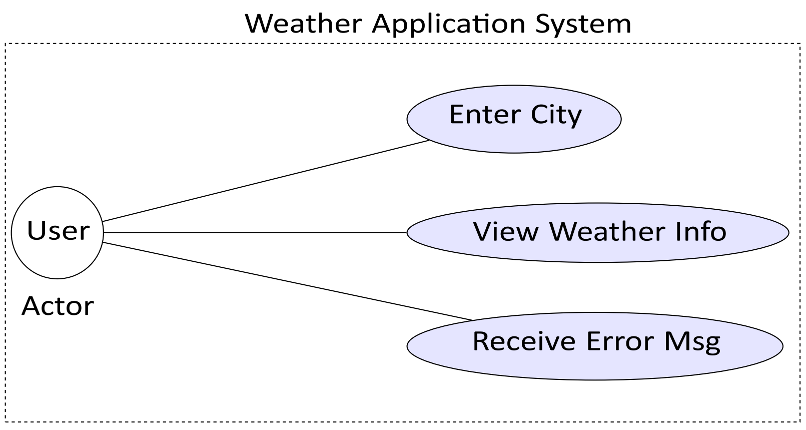


Figure 2: Use Case Diagram

6.3 C. Sequence Diagram

The chronological sequence of messages between the User, System, and API.

7 Design Decisions & Rationale

- **Language: Python** - Chosen for its simplicity and powerful libraries for API handling and GUIs.
- **Library: Tkinter** - Selected over generic frameworks because it is built into Python and sufficient for creating lightweight desktop GUIs.
- **Library: Requests** - Used for HTTP calls because it simplifies the process of sending headers and parameters compared to the standard urllib.
- **Library: Pillow (PIL)** - Essential for the project because standard Tkinter cannot handle dynamic image resizing based on screen width/height, which is a core requirement of this dashboard.

8 Implementation Details

The application is implemented in a modular fashion using Python functions:

- **get_city_prompt()**: Uses simpledialog to capture user input cleanly.
- **get_weather(city_name)**:
- Constructs the query URL using BASE_URL and API_KEY.
- Uses atry...exceptblock to catchrequests.exceptions.HTTPErrorandConnectionError.
- Extracts the specific Weather ID (e.g., 500 for Rain) to determine visual logic.
- **GUI(...)**:
- Initializes a full-screen tk.Tk() window.
- Implements a custom grouping logic (e.g., ID 500-531 maps to ”Rainy”) to handle the API’s 50+ weather codes.

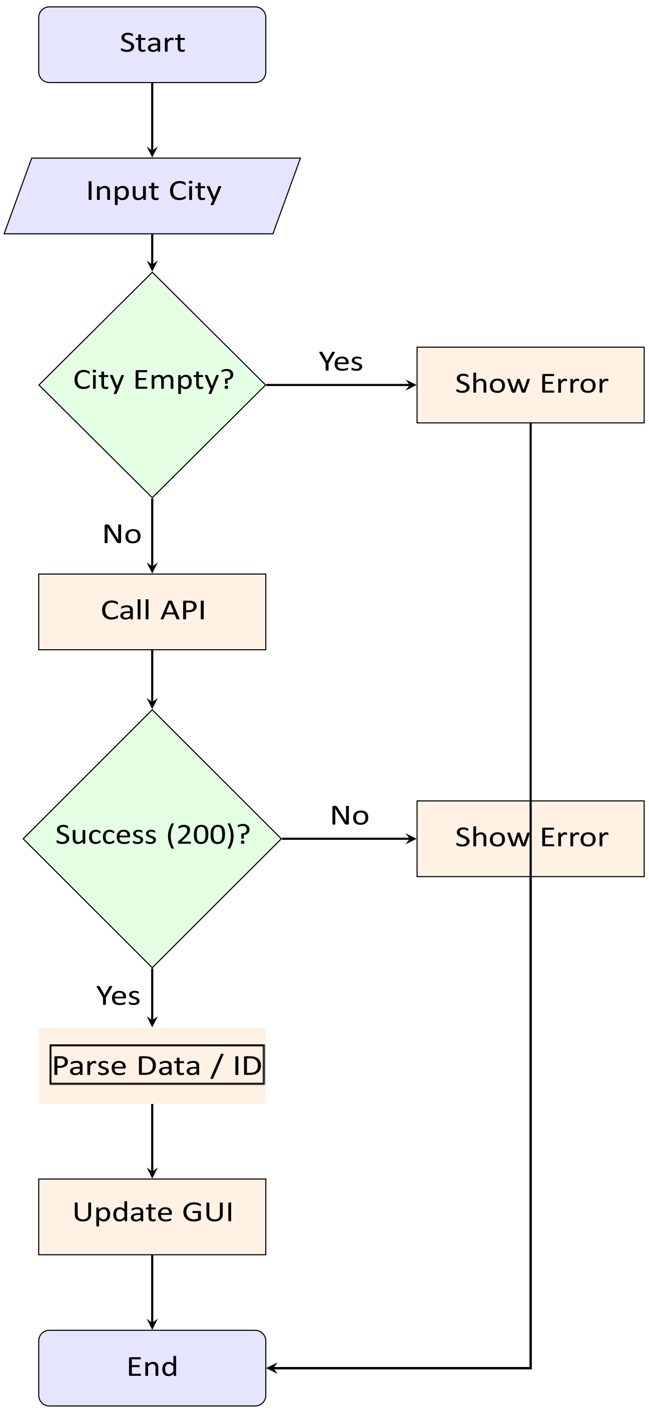


Figure 3: Flowchart Activity Diagram

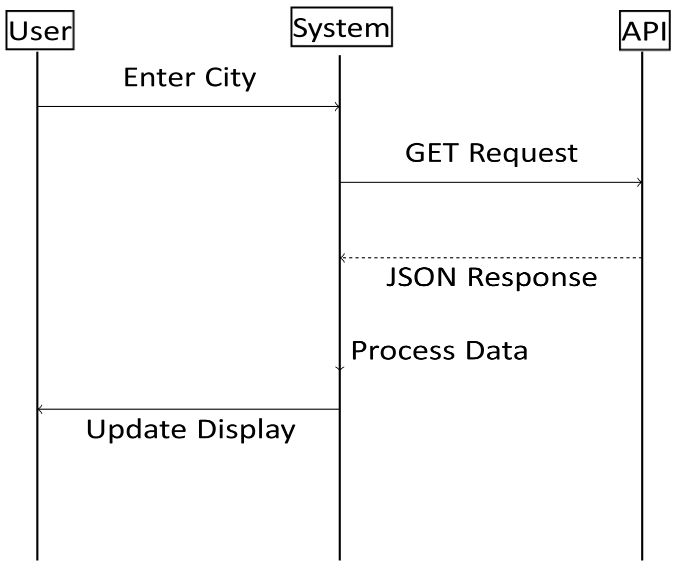
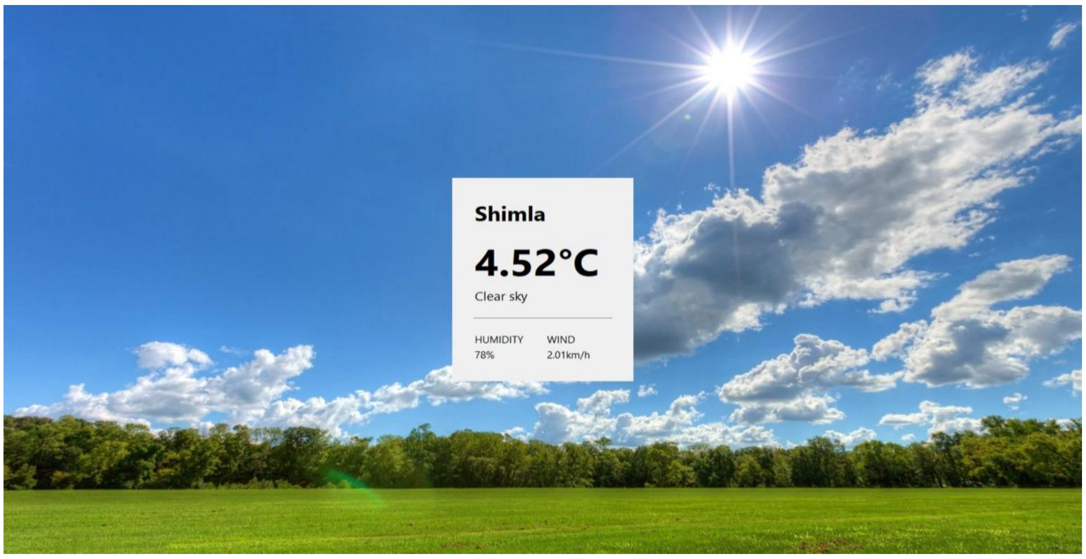


Figure 4: Sequence Diagram

- Uses Image.resize((screen_width, screen_height)) to ensure the background covers the whole screen regardless of monitor size.

9 Screenshots / Results



Shimla

4.52°C

Clear sky

HUMIDITY 78%

WIND 2.01km/h

Figure 5: Clear Sky Condition

10 Testing Approach

The application underwent the following testing strategies:

- **Unit Testing:** Tested the get_weather function with known cities (London, New York) to verify data accuracy.
- **Boundary Testing:** Tested the input prompt with empty strings, numbers, and gibberish to ensure the app handles ”404 City Not Found” errors gracefully.

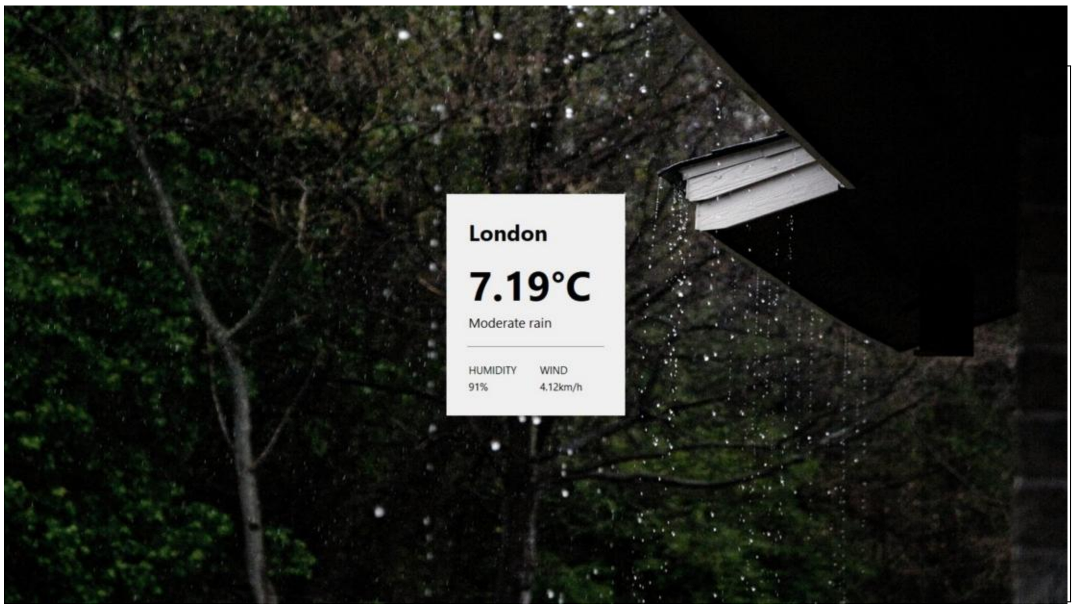
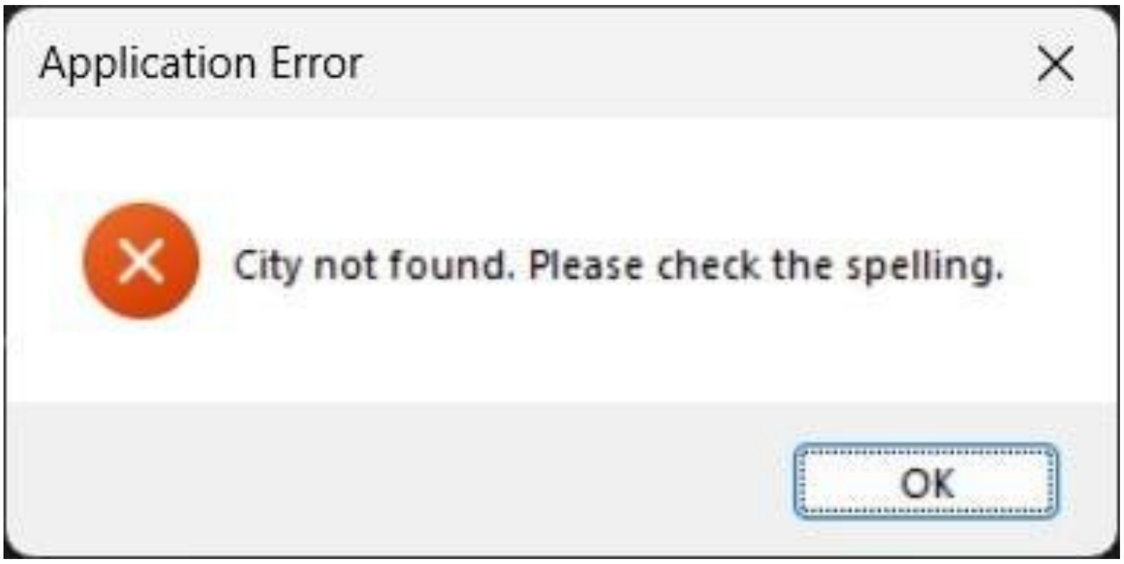


Figure 6: Rain Condition



Application Error

City not found. Please check the spelling.

Figure 7: Error Handling

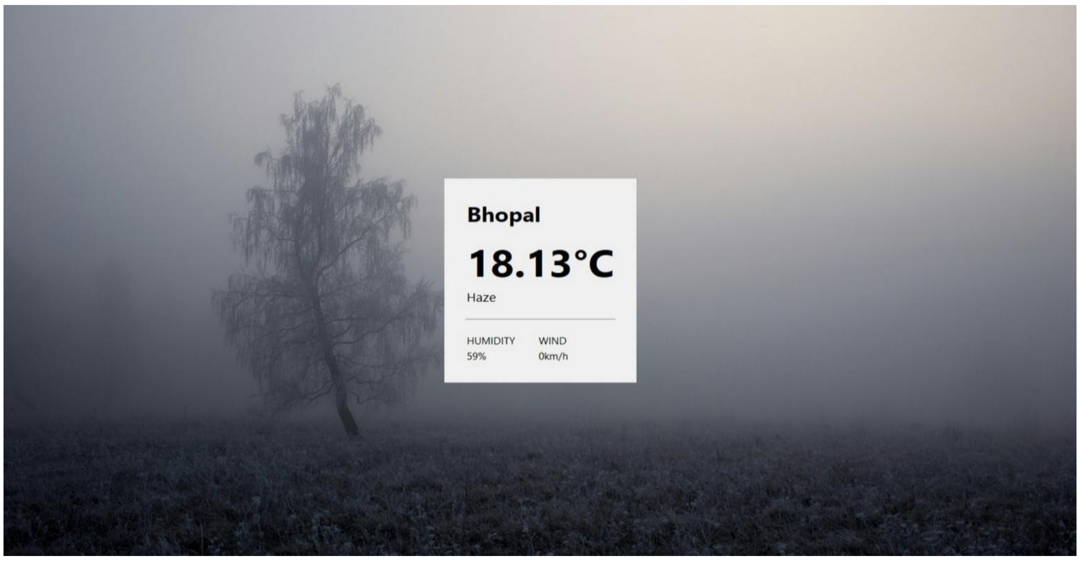


Figure 8: Mist/Fog Condition

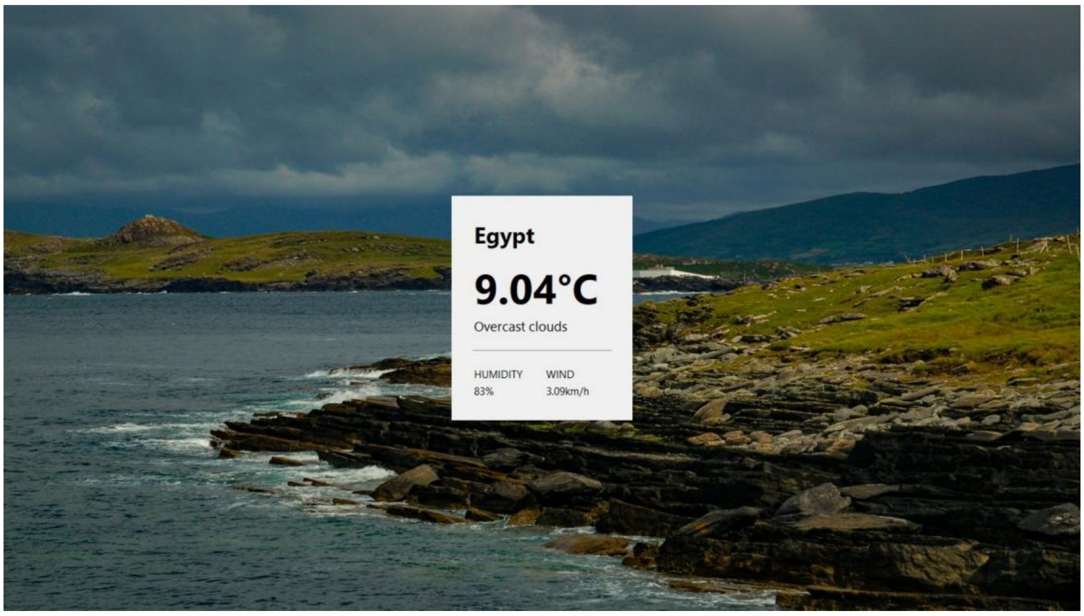


Figure 9: Cloudy Condition

- **Network Failure Testing:** Disconnected the system from the internet to verify that the `requests.exceptions.ConnectionError`block triggers the correct error popup.

11 Challenges Faced

- **Image Resizing:** Initially, images were static and didn't ft diferent monitors. This was solved by integrating the`PIL` library to dynamically detect screen resolution (`root.winfo_screenwidth`) and resize images on the fy.
- **Weather ID Mapping:** The API provides over 50 specifc weather codes (e.g., light rain, heavy rain, freezing rain). Creating an image for every single one was inefcient. This was solved by implementing logic to "group" IDs into broad categories (e.g., all 500-level codes use the "Rainy" image).

12 Future Enhancements

1. **5-Day Forecast:** Expanding the API call to fetch forecast data, not just current weather.
2. **GPS Integration:** Automatically detecting the user’s location via IP address instead of asking for manual input.
3. **Search History:** Saving recently searched cities for quick access.

13 References

1. **OpenWeatherMap API Docs:** https://openweathermap.org/api
2. **Python Requests Library:** https://pypi.org/project/requests/
3. **Tkinter Documentation:** https://docs.python.org/3/library/tkinter.html
4. **Pillow (PIL) Handbook:** https://pillow.readthedocs.io/