**Final Project**

**“Real Time Alert Mechanism System”**

Name: Harsh Vardhan Singh

Reg ID: CU23MSD0011A

Class: M.Sc. Data science

Course: Real Time Analytics

Course Code: MAT503



**Assignment Guide**

**Sarath B Sribhasyam**

**School of Mathematics & Natural Sciences**

**CHANAKYA UNIVERSITY, BANGALORE**

**January 2025**

**Synopsis**

1. **Use Case**

The main purpose of the real-time weather monitoring and alerting system is to ensure users receive timely and accurate weather information to help them prepare for extreme weather conditions like storms, heatwaves, or heavy rainfall.

Detailed Use Case Breakdown:

* **Real-time Weather Monitoring:**
  + Weather data (e.g., temperature, humidity, pressure, rainfall) is collected at frequent intervals from OpenWeatherMap API.
  + The data is continuously updated to provide a live feed of weather conditions to the system.
  + The system can adapt to the user’s location, providing weather data specific to that region.
* **Alerts for Extreme Conditions:**
  + When specific weather thresholds are breached (e.g., temperature > 35°C, 100mm of rain in an hour), alerts are triggered.
  + Alerts are sent in real-time to users via Telegram.
  + Notifications contain information about the severity of the weather, what actions to take, and safety tips.
* **Data Visualization:**
  + The weather data is aggregated and displayed on Apache Superset dashboards.
  + Visualizations include graphs, heat maps, and alerts indicating extreme weather events.
  + Real-time visual updates help users monitor trends and conditions on an ongoing basis.

1. **Problem Statement**

Weather alerts are often generalized and delayed, failing to cater to individual regions and the timing required to take action. The problem can be broken down into:

1. **Delayed Alerts:**
   * Alerts may arrive too late for users to act accordingly, especially in situations like thunderstorms or extreme heat. For instance, a storm warning might be issued after the storm has already impacted the area.
2. **Lack of Personalization:**
   * Current weather systems often deliver broad alerts that might not be region-specific or suitable for the user's actual location. For example, an alert for heavy rain might be irrelevant for users living in an area that does not experience precipitation.
3. **Challenges in Data Processing and Visualization:**
   * Aggregating, processing, and visualizing live weather data in a format that is both user-friendly and actionable is complex. Storing and analyzing this large volume of data in real-time is another challenge that must be addressed.
4. **Approach to Build the Real-Time Analytics Application**

The application will utilize modern technologies to solve these problems:

1. **Data Collection:**
   * The OpenWeatherMap API will be used to fetch live weather data. API calls will be made at regular intervals to capture the latest data, with detailed weather parameters (temperature, humidity, etc.).
2. **Kafka for Real-Time Data Streaming:**
   * Kafka acts as the backbone for real-time data ingestion and processing.
     + Kafka Producer: A Python script will pull data from OpenWeatherMap and send it to Kafka topics for real-time processing.
     + Kafka Consumer: Another Python script will listen for data from Kafka, process it (extract weather data), and forward it to PostgreSQL for storage.
3. **PostgreSQL for Data Storage:**
   * Weather data will be stored in a PostgreSQL database, enabling efficient querying, analytics, and history tracking. The database schema will be designed to hold large volumes of weather data.
4. **Alerting System via Telegram:**
   * A Telegram bot will listen for specific triggers (e.g., extreme temperature or precipitation) and send instant alerts to subscribed users.
5. **Data Visualization via Apache Superset:**
   * Apache Superset will be used to build interactive dashboards for visualizing weather trends and conditions. This enables easy monitoring of data over time.

**Project Plan**

**Phase 1: Initial Setup and Planning**

1. **Research & Requirements Gathering**:
   * Thoroughly review the OpenWeatherMap API documentation to understand the data structure and possible limits (e.g., how frequently API calls can be made).
   * Understand the thresholds for triggering alerts based on different weather conditions like temperature, wind speed, and rainfall.
2. **Tool & Technology Selection**:
   * OpenWeatherMap API (weather data).
   * Kafka (for real-time data streaming).
   * PostgreSQL (for data storage).
   * Telegram Bot API (for alerting).
   * Apache Superset (for data visualization).
3. **Set Up Environment**:
   * Install and configure necessary software like Kafka, PostgreSQL, Superset, and required Python libraries (e.g., psycopg2, python-telegram-bot, kafka-python).
   * Set up the necessary environment variables, API keys, and configuration files.

**Phase 2: Data Collection and Ingestion**

1. **Set Up Kafka Producer**:
   * Write the Kafka producer script (producer.py) that will:
     + Fetch data from OpenWeatherMap API.
     + Parse the JSON response from OpenWeatherMap and extract relevant weather data.
     + Send the parsed data to Kafka as messages to a topic (e.g., weather\_data).
2. **Set Up Kafka Consumer**:
   * Write the Kafka consumer script (consumer.py) that will:
     + Consume data from the weather\_data Kafka topic.
     + Extract useful data fields (e.g., city, temperature, humidity).
     + Store the data in the PostgreSQL database.
3. **Create Database Schema**:
   * **Create a weather\_data table** in PostgreSQL with the following columns:

CREATE TABLE weather\_data (

id SERIAL PRIMARY KEY, -- Unique identifier for each row

city VARCHAR(100) NOT NULL, -- City name

timestamp DOUBLE PRECISION NOT NULL, -- UNIX timestamp

temperature FLOAT NOT NULL, -- Temperature value

weather VARCHAR(100) NOT NULL -- Weather description

);

* + Implement logic in the Kafka consumer to insert processed data into this table.

**Phase 3: Alert System**

1. **Set Up Telegram Bot**:
   * Create a Telegram bot using BotFather on Telegram.
   * Retrieve the bot token and configure the bot in a Python script (weather\_alert\_bot.py).
   * Implement logic to send alerts when certain thresholds are met (e.g., temperature > 35°C).
   * Define user groups or lists to subscribe users and send them location-specific alerts.

**Phase 4: Data Presentation**

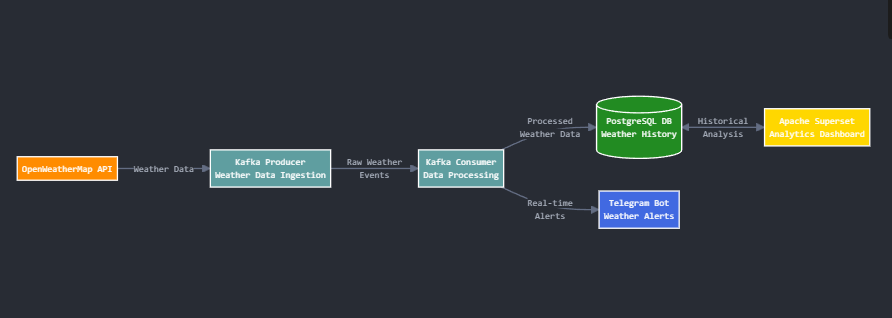
1. **Install Apache Superset**:
   * Set up Apache Superset and configure it to connect to PostgreSQL.
   * Install required dependencies using:

pip install apache-superset

* + Create a database connection in Superset to pull data from PostgreSQL.

1. **Create Dashboards**:
   * Build visualizations in Superset to display weather data, including charts for temperature, humidity, and extreme conditions.
   * Create filters for users to focus on their specific region.
2. **Test and Refine**:
   * Run tests for each component (producer, consumer, alert system, and visualization).
   * Refine the integration to ensure seamless data flow from OpenWeatherMap to Kafka to PostgreSQL and finally to Superset and Telegram bot.

**System Design and Architecture**

****

The system design and architecture represent how each component of the system interacts with each other. Here’s a detailed breakdown of the flow and key components:

**Components and Workflow:**

1. **OpenWeatherMap API**:
   * The data source that provides real-time weather information via an HTTP-based API. The system fetches the data at regular intervals (e.g., every minute) from OpenWeatherMap for specific cities or regions.
   * **Key Data**: Temperature, humidity, pressure, wind speed, precipitation levels, and condition status (e.g., sunny, rainy, stormy).
2. **Kafka Producer**:
   * **Purpose**: The Kafka producer is responsible for fetching the weather data from OpenWeatherMap API and sending it to a Kafka topic.
   * **Script**: producer.py
     + The script fetches data from the API.
     + The fetched data is serialized into JSON format and pushed to a Kafka topic (weather\_data).
     + A **producer** sends the weather data in real-time.
     + Kafka topics allow multiple consumers to read the same data independently. This enables flexibility in handling the data for various purposes (like alerting, storage, and analytics).
   * **Kafka Producer:**

import json

import time

import requests

from kafka import KafkaProducer

# Kafka configuration

KAFKA\_TOPIC = 'weather-data'

KAFKA\_BROKER = 'localhost:9092' # Change if your broker address is different

# OpenWeatherMap API configuration

API\_KEY = 'b2b7e34216a77579888422b4e36a7c1e' # Replace with your OpenWeatherMap API key

BASE\_URL = 'http://api.openweathermap.org/data/2.5/weather'

# Create a Kafka producer

producer = KafkaProducer(

bootstrap\_servers=KAFKA\_BROKER,

value\_serializer=lambda v: json.dumps(v).encode('utf-8')

)

def fetch\_and\_send\_weather\_data(city):

"""

Fetch weather data for a city and send it to Kafka.

"""

params = {

'q': city,

'appid': API\_KEY,

'units': 'metric' # Use 'imperial' for Fahrenheit

}

try:

response = requests.get(BASE\_URL, params=params)

if response.status\_code == 200:

data = response.json()

message = {

'city': city,

'timestamp': time.time(),

'temperature': data['main']['temp'],

'weather': data['weather'][0]['description']

}

producer.send(KAFKA\_TOPIC, value=message)

print(f"Sent: {message}")

else:

print(f"Failed to fetch data for {city}: HTTP {response.status\_code}")

except Exception as e:

print(f"Error fetching data for {city}: {e}")

# List of cities to fetch weather data for

cities = [

'London', 'New York', 'Tokyo', 'Berlin', 'Sydney', 'Paris', 'Dubai', 'Mumbai',

'Beijing', 'Moscow', 'Cape Town', 'Buenos Aires', 'Cairo', 'Singapore', 'Los Angeles',

'Toronto', 'Mexico City', 'Jakarta', 'Istanbul', 'Rome', 'Bangkok', 'Seoul', 'Sao Paulo',

'Nairobi', 'Melbourne', 'Vancouver', 'Madrid', 'Johannesburg', 'Lagos', 'Rio de Janeiro',

'Kuala Lumpur', 'Chicago', 'Hanoi', 'Baghdad', 'Karachi', 'Lisbon', 'Tehran', 'Casablanca',

'Athens', 'Warsaw', 'Hong Kong', 'Manila', 'Zurich', 'Lima', 'Stockholm', 'Dublin',

'Bogota', 'Brisbane', 'Vienna', 'Oslo', 'Kabul', 'Doha', 'Prague', 'Accra'

]

if \_\_name\_\_ == "\_\_main\_\_":

while True:

for city in cities:

fetch\_and\_send\_weather\_data(city)

time.sleep(60) # Sleep for a minute before fetching data again for all cities.

1. **Kafka Consumer**:
   * **Purpose**: The consumer listens to the Kafka topic and processes the incoming weather data.
   * **Script**: consumer.py
     + The consumer deserializes the incoming JSON messages.
     + Extracts key information (e.g., temperature, humidity, city).
     + The processed data is stored in a PostgreSQL database for analysis and historical records.
     + The data is also passed to an alert mechanism if it meets specific criteria (e.g., temperature > 35°C or heavy rainfall).
   * **Kafka Consumer**:

import json

from kafka import KafkaConsumer

import psycopg2

# PostgreSQL configuration

DB\_HOST = 'localhost'

DB\_NAME = 'weather\_data'

DB\_USER = 'harsh'

DB\_PASSWORD = 'harsh'

# Create PostgreSQL connection

try:

conn = psycopg2.connect(

host=DB\_HOST,

database=DB\_NAME,

user=DB\_USER,

password=DB\_PASSWORD,

port='5432'

)

cur = conn.cursor()

print("Connected to the database.")

except psycopg2.Error as e:

print(f"Database connection error: {e}")

exit(1)

# Kafka Consumer configuration

try:

consumer = KafkaConsumer(

'weather-data',

bootstrap\_servers='localhost:9092',

auto\_offset\_reset='earliest',

enable\_auto\_commit=True,

group\_id='weather-group',

value\_deserializer=lambda x: x.decode('utf-8') if x else None

)

print("Connected to Kafka broker.")

except Exception as e:

print(f"Error connecting to Kafka broker: {e}")

exit(1)

def is\_valid\_json(data):

"""

Validate if the input is a valid JSON string and contains the required keys.

"""

try:

parsed\_data = json.loads(data)

required\_keys = {'city', 'timestamp', 'temperature', 'weather'}

if isinstance(parsed\_data, dict) and required\_keys.issubset(parsed\_data.keys()):

return parsed\_data

except json.JSONDecodeError:

pass

return None

if \_\_name\_\_ == "\_\_main\_\_":

print("Starting Kafka consumer...")

try:

for message in consumer:

# Debug raw message

print(f"Raw message: {message.value}")

# Skip empty or invalid messages

if not message.value:

print("Skipping empty message...")

continue

# Validate and parse JSON

weather\_data = is\_valid\_json(message.value)

if not weather\_data:

print(f"Invalid JSON message: {message.value}")

continue

print(f"Valid message: {weather\_data}")

# Insert into PostgreSQL

try:

cur.execute("""

INSERT INTO weather\_info (city, timestamp, temperature, weather)

VALUES (%s, %s, %s, %s)

""", (

weather\_data['city'],

weather\_data['timestamp'],

weather\_data['temperature'],

weather\_data['weather']

))

conn.commit()

print("Data inserted successfully.")

except psycopg2.Error as db\_err:

print(f"Database error: {db\_err}")

conn.rollback()

except KeyboardInterrupt:

print("\nConsumer interrupted. Exiting...")

except Exception as e:

print(f"Unexpected error: {e}")

finally:

print("Closing database connection...")

cur.close()

conn.close()

1. **Alert Mechanism**:
   * **Purpose**: After the Kafka consumer processes the weather data, the alert mechanism comes into play. This ensures that the user is notified when weather conditions meet predefined thresholds for extreme conditions.
   * **Telegram Bot**:
     + If any threshold for extreme conditions is met (e.g., heavy rain, high temperature), the system triggers an alert.
     + A Telegram bot sends messages to users subscribed to weather alerts for a particular city or region.
     + **Telegram Bot Logic**:
       - Check for conditions like temperature > 35°C or extreme cold

< -10.

* + - * Send customized alerts via Telegram messages (e.g., "⚠️ Heavy Rain in [City]! Stay Safe!").
  + **Telegram Bot**:

import os

import requests

import json

from kafka import KafkaConsumer

import psycopg2

from psycopg2 import pool

from telegram import Bot

from telegram.ext import Application

import time

import asyncio

# Load environment variables

TELEGRAM\_BOT\_TOKEN = '7560823758:AAF84YiVI0qEdWuI\_TBQgRH4XEz2H4mpGwc'

CHAT\_ID = '1246065946'

# Initialize Telegram bot using Application class

application = Application.builder().token(TELEGRAM\_BOT\_TOKEN).build()

# Kafka Consumer setup

consumer = KafkaConsumer('weather-data',

bootstrap\_servers='localhost:9092',

auto\_offset\_reset='earliest',

enable\_auto\_commit=True,

group\_id='weather-group',

value\_deserializer=lambda x: json.loads(x.decode('utf-8')))

# PostgreSQL connection pool setup

db\_pool = psycopg2.pool.SimpleConnectionPool(10, 50,

dbname="postgres", user="postgres", password="harsh", host="localhost", port="5432"

)

# Function to handle database inserts and alerts

async def process\_weather\_data(weather\_data):

try:

conn = db\_pool.getconn() # Get a connection from the pool

with conn.cursor() as cursor: # Use context manager for the cursor

cursor.execute("INSERT INTO weather\_data (city, timestamp, temperature, weather) VALUES (%s, %s, %s, %s)",

(weather\_data['city'], weather\_data['timestamp'], weather\_data['temperature'], weather\_data['weather']))

conn.commit()

# Check for extreme weather conditions

if weather\_data['temperature'] > 35: # Example threshold for heatwave

alert\_message = f"Alert! Extreme heat detected in {weather\_data['city']}: {weather\_data['temperature']}°C."

await application.bot.send\_message(chat\_id=CHAT\_ID, text=alert\_message)

elif weather\_data['temperature'] < -10: # Example threshold for extreme cold

alert\_message = f"Alert! Extreme cold detected in {weather\_data['city']}: {weather\_data['temperature']}°C."

await application.bot.send\_message(chat\_id=CHAT\_ID, text=alert\_message)

# Check for clear sky condition

if weather\_data['weather'] == 'clear sky': # Check if the sky is clear

alert\_message = f"Clear sky detected in {weather\_data['city']}. Enjoy the beautiful day!"

await application.bot.send\_message(chat\_id=CHAT\_ID, text=alert\_message)

# Check for storm condition

if 'storm' in weather\_data['weather'].lower() or 'thunderstorm' in weather\_data['weather'].lower(): # Check for stormy weather

alert\_message = f"Storm detected in {weather\_data['city']}! Please stay safe."

await application.bot.send\_message(chat\_id=CHAT\_ID, text=alert\_message)

# Additional conditions can be added here (e.g., heavy rain or other weather conditions)

except Exception as e:

print(f"Error processing weather data: {e}")

time.sleep(5) # Optionally, wait for some time before retrying

finally:

db\_pool.putconn(conn) # Return the connection to the pool

# Main loop to consume messages from Kafka

async def main():

try:

for message in consumer:

weather\_data = message.value

await process\_weather\_data(weather\_data)

except KeyboardInterrupt:

print("Consumer interrupted, shutting down...")

finally:

consumer.close()

print("Kafka consumer closed.")

# Run the asyncio loop

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(main())

1. **Data Visualization (Apache Superset)**:
   * **Purpose**: Apache Superset is used for visualizing weather data in real-time via dashboards. This allows users to see key metrics like temperature trends, humidity levels, and storm occurrences.
   * **Setup**: Superset connects to the PostgreSQL database where the processed weather data is stored.
   * **Dashboards**: These display charts for the weather trends, alerts, and live updates, allowing users to interact with the data, filter by city or region, and track extreme weather events.
   * **Types of Visuals**:
     + **Line charts** for temperature and humidity trends.
     + **Heatmaps** to show regions with extreme conditions.
     + **Alert Status**: A dynamic chart showing whether the current weather meets alert criteria.

**Define Metrics**

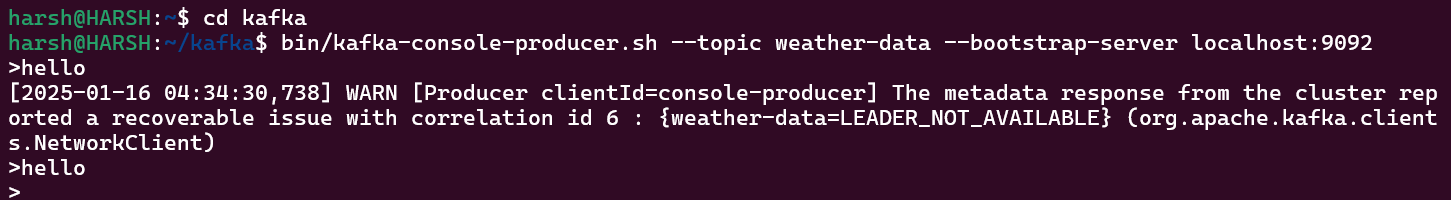
**Metrics** are crucial for evaluating the performance, effectiveness, and reliability of the system. Here's a breakdown of the key metrics:

1. **Data Latency**:
   * **Definition**: The delay between the time when weather data is fetched from OpenWeatherMap and when it appears on Superset dashboards.
   * **Goal**: Ensure that the system is near real-time with minimal latency. This should be below **1 second** to ensure that users get up-to-the-minute information.
   * **Measurement Method**: Record the time taken from when the data is retrieved from OpenWeatherMap (via the producer) to when it is visible on the Superset dashboard.
2. **System Uptime**:
   * **Definition**: The percentage of time the system is operational and serving weather data without failure.
   * **Goal**: Aim for **99.9%** uptime, which is industry-standard for critical systems.
   * **Measurement Method**: Monitor the system's health using tools like **Prometheus** or **Grafana** for system monitoring. Ensure automatic recovery in case of failure.
3. **Alert Accuracy**:
   * **Definition**: The accuracy of weather alerts, ensuring that the system does not generate too many false alarms or miss any important extreme weather events.
   * **Goal**: Minimize **false positives** (alerting for weather conditions that don't actually exist) and **false negatives** (missing actual weather extremes).
   * **Measurement Method**: Track actual weather events (using OpenWeatherMap API or trusted weather sources) and compare them with system alerts. Record false positives/negatives.
4. **User Engagement**:
   * **Definition**: How often users interact with the Telegram bot and engage with the alerts.
   * **Goal**: Track user interactions like clicks, responses, or forwarding alerts.
   * **Measurement Method**: Use Telegram API's analytics tools to track interactions with the bot (e.g., how many users respond to alerts or acknowledge them).
5. **Data Consistency**:
   * **Definition**: The consistency between the data retrieved from OpenWeatherMap and the data stored in PostgreSQL and displayed in Superset.
   * **Goal**: Ensure that the data in the database accurately represents what was fetched from the OpenWeatherMap API.
   * **Measurement Method**: Periodically verify the data using **checksums** or direct comparisons between the API and the database entries.

**Build End-to-End System**

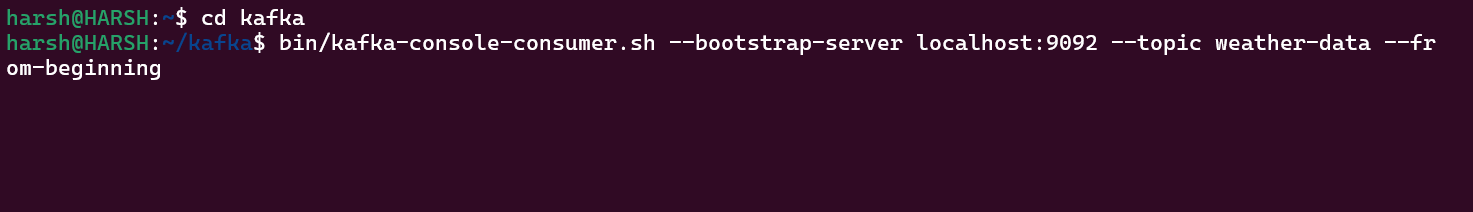
This section describes how to build the system step-by-step, ensuring that each component works together seamlessly.

1. **Data Receiving (Kafka Producer)**:
   * **Producer Code**: The producer.py script is placed in the **kafka\_producers** folder. This script fetches weather data from OpenWeatherMap at regular intervals and sends it to a Kafka topic (weather\_data).
   * **Command to Run Producer**:



Ensure that the API key is securely stored in a configuration file or environment variable to avoid hardcoding sensitive data.

1. **Data Transformation (Kafka Consumer)**:
   * **Consumer Code**: The consumer.py script is placed in the **kafka\_consumers** folder. This script listens to the Kafka topic, processes the incoming data, and stores it in PostgreSQL.
   * **Command to Run Consumer**:

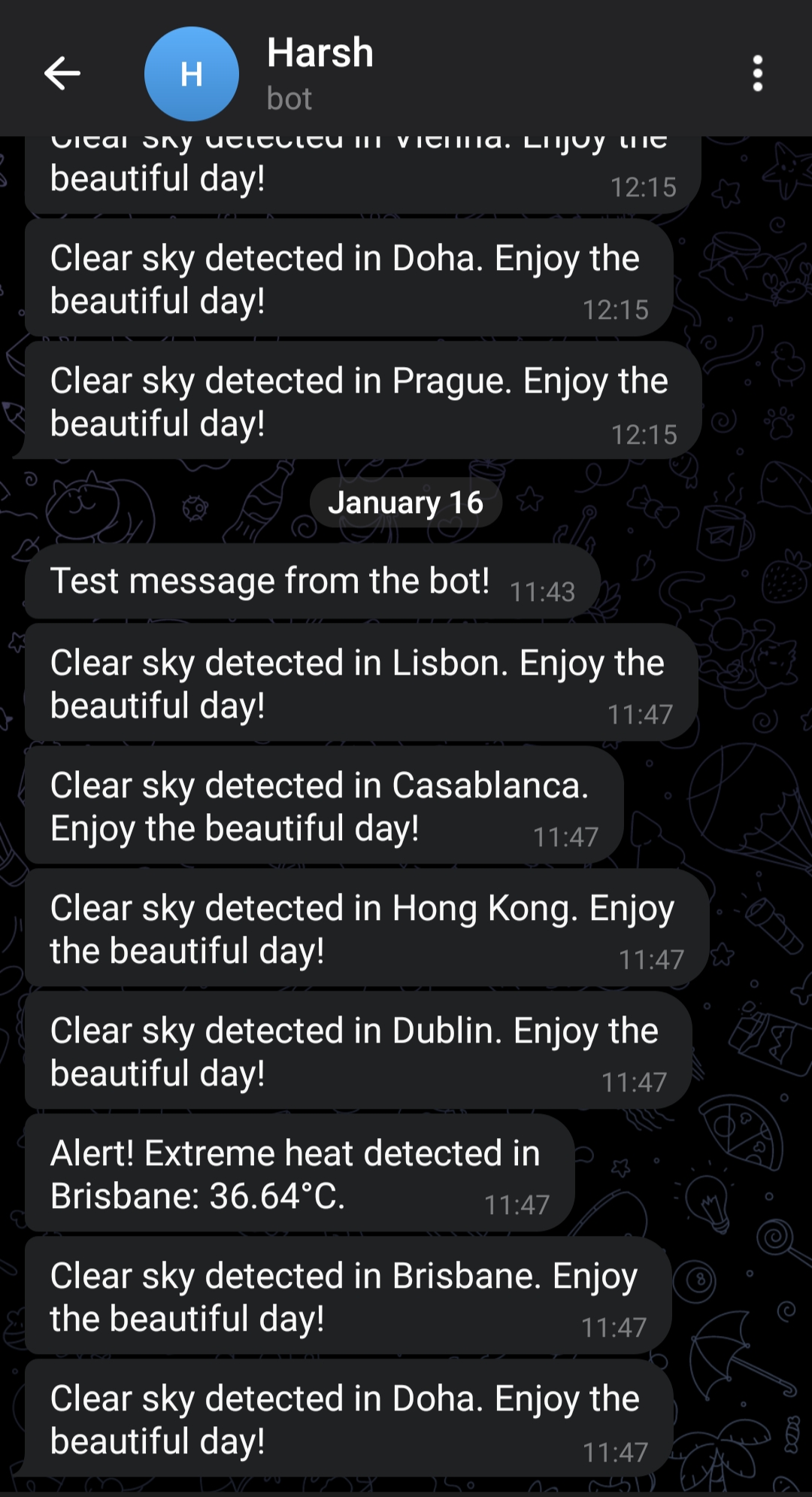
****



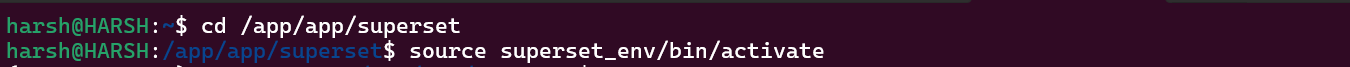
* + **Data Storage**: Data is stored in the PostgreSQL weather\_data table. The consumer processes each incoming message and inserts relevant data into the database.

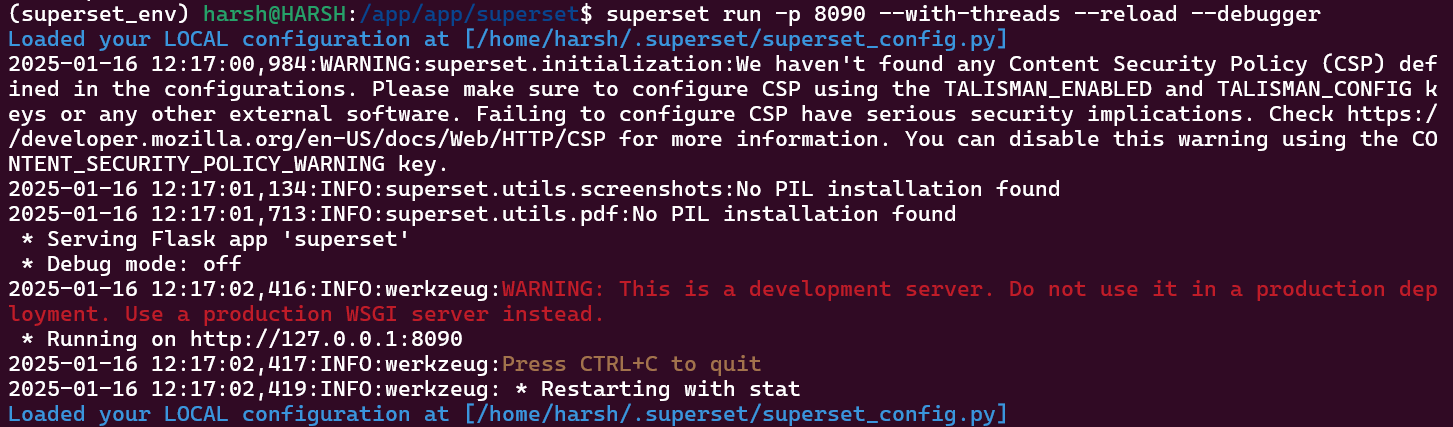
1. **Alert Mechanism (Telegram Bot)**:
   * **Alert Script**: The weather\_alert\_bot.py script is placed in the **alert\_system** folder.
     + It subscribes to certain weather thresholds (e.g., temperature > 35°C).
     + When the threshold is reached, the bot sends an alert to the specified users.
   * **Command to Run Bot**:

**Python weather\_alert\_bot.py**

****

1. **Data Presentation (Apache Superset)**:
   * **Superset Setup**: Connect Superset to the PostgreSQL database. After installation, log in to Superset and configure it to point to the weather\_data table in the database.





* + **Dashboard Creation**: Create multiple dashboards in Superset to visualize data:
    - **Temperature and Humidity Trends**.
    - **Extreme Weather Events**.
    - **Alerts Status**.

