ASSIGNMENT-3

NAME: **GOWTHAMI MOPURI**

REG.NO: **192311287**

DEPART: **CSE**

DATE OF SUBMISSION: 17-07-2024

DOCUMENT:1

Problem 1: Real-Time Weather Monitoring System

Scenario:

You are developing a real-time weather monitoring system for a weather forecasting company.

The system needs to fetch and display weather data for a specified location.

Tasks:

1. Model the data flow for fetching weather information from an external API and

displaying it to the user.

2. Implement a Python application that integrates with a weather API (e.g.,

OpenWeatherMap) to fetch real-time weather data.

3. Display the current weather information, including temperature, weather conditions,

humidity, and wind speed.

4. Allow users to input the location (city name or coordinates) and display the

corresponding weather data.

Deliverables:

• Data flow diagram illustrating the interaction between the application and the API.

• Pseudocode and implementation of the weather monitoring system.

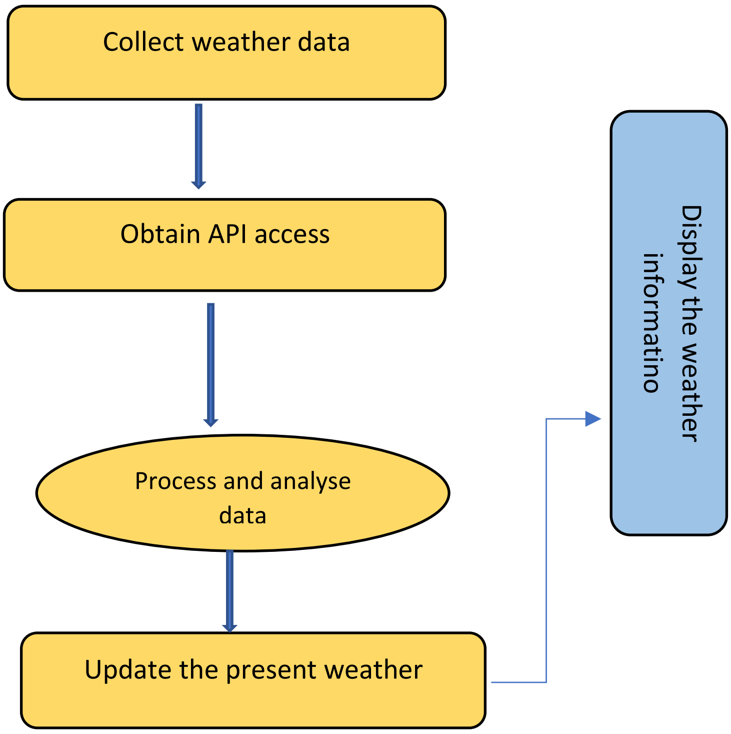
• Documentation of the API integration and the methods used to fetch and display

weather data.

• Explanation of any assumptions made and potential improvements.

Solution:

Data flowchart:



Implementation code:

```
import requests
import json
from datetime import datetime

# Replace with your API key
API_KEY = 'a7c7a0c4dd1561efd041f0e52609bd2c'

# Replace with your city and country code
CITY = 'chennai'

def get_weather(api_key, city):
    url = f'http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric'
    response = requests.get(url)
```

```python
    if response.status_code == 200:
        data = response.json()
        weather_description = data['weather'][0]['description']
        temperature = data['main']['temp']
        humidity = data['main']['humidity']
        wind_speed = data['wind']['speed']
        sunrise = datetime.fromtimestamp(data['sys']['sunrise']).strftime('%Y-%m-%d %H:%M:%S')
        sunset = datetime.fromtimestamp(data['sys']['sunset']).strftime('%Y-%m-%d %H:%M:%S')
        timezone = data['timezone']

        print(f'Weather in {city}:')
        print(f'Description: {weather_description}')
        print(f'Temperature: {temperature}°C')
        print(f'Humidity: {humidity}%')
        print(f'Wind Speed: {wind_speed} m/s')
        print(f'Sunrise time: {sunrise} (UTC{timezone // 3600})')
        print(f'Sunset time: {sunset} (UTC{timezone // 3600})')
    else:
        print(f'Error fetching data: {response.status_code}')

if __name__ == '__main__':
    get_weather(API_KEY, CITY)
```

Input: Chennai

Output:

Weather in chennai:

Description: drizzle

Temperature: 27.23°C

Humidity: 86%

Wind Speed: 5.14 m/s

Sunrise time: 2024-07-15 00:19:58 (UTC5)

Sunset time: 2024-07-15 13:09:20 (UTC5)

addCode

addText

Documentation:

• **Purpose:** Describe the purpose and goals of the Real-Time Weather Monitoring System.

• **Scope:** Define the scope of the system, including the geographical area covered and types of weather data monitored.

• **Audience:** Identify the intended audience for the documentation (e.g., developers, system administrators, stakeholders).

User and interface:

• **Admin:** Responsible for system configuration, user management, and overall system maintenance.

• **Meteorologist:** Analyzes weather data, creates forecasts, and generates reports.

• **General User:** Accesses weather information for personal or professional use.

• **Emergency Response Personnel:** Monitors weather conditions for disaster preparedness and response.

• **Developer:** Manages system integrations, customizations, or enhancements.

Assumptions and improvements:

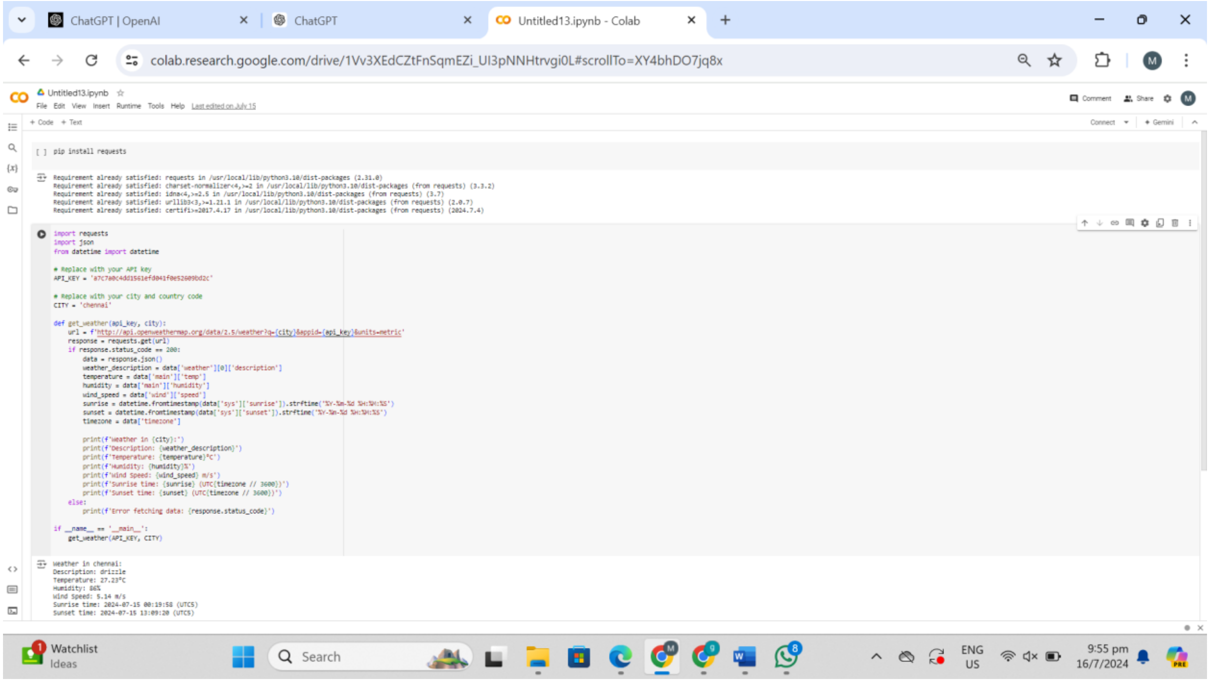
1. Data Accuracy and Reliability: Assumption that weather data collected from sensors (e.g., temperature, humidity) is accurate and reliable. This assumes sensors are properly calibrated and maintained.

2. Data Transmission: Assumption that data transmission from sensors to the central database or server occurs without significant delays or interruptions, ensuring real-time updates.

3. System Scalability: Assuming the system can handle varying data loads during extreme weather events or peak usage periods without performance degradation.

4. User Accessibility: Assuming users have access to reliable internet connectivity and compatible devices (e.g., smartphones, tablets, computers) to access the system.

5. Security: Assuming adequate security measures are in place to protect sensitive weather data from unauthorized access or breaches.



2.Inventory Management System Optimization:

Scenario:

You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.

Tasks:

1. Model the inventory system: Define the structure of the inventory system, including products, warehouses, and current stock levels.

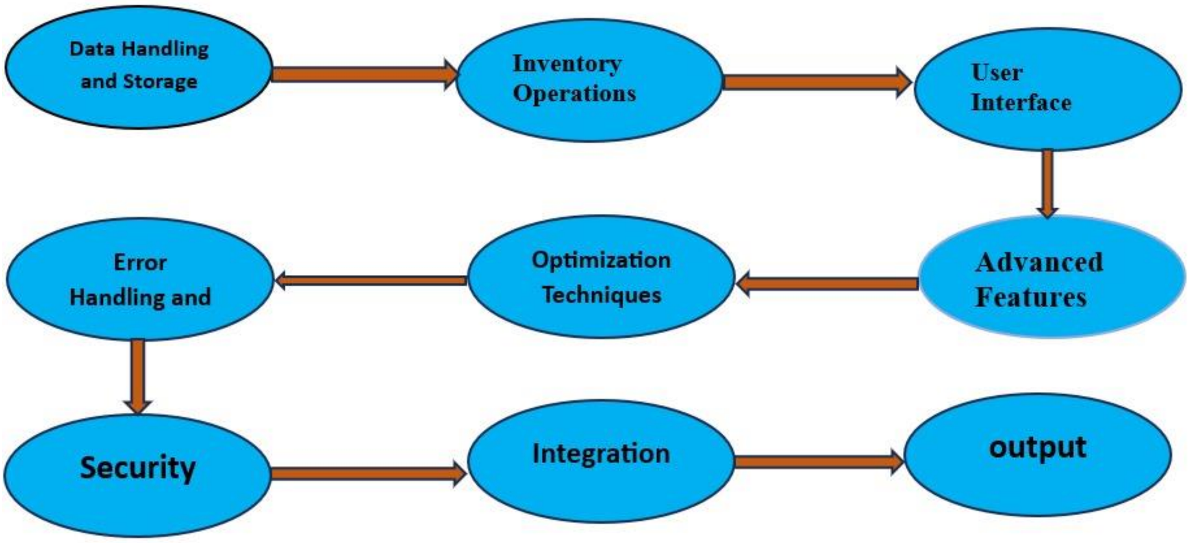
2. Implement an inventory tracking application: Develop a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain threshold.

3. Optimize inventory ordering: Implement algorithms to calculate optimal reorder points and quantities based on historical sales data, lead times, and demand forecasts.

4. Generate reports: Provide reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.

5. User interaction: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

Data flowchart:



Implementation code:

```
import pulp

# Define the data
products = ['product_1', 'product_2', 'product_3']
demand = {'product_1': 100, 'product_2': 150, 'product_3': 200}
costs = {'product_1': 5, 'product_2': 8, 'product_3': 7}

# Create a LP minimization problem
prob = pulp.LpProblem("Inventory_Optimization", pulp.LpMinimize)

# Create a dictionary of decision variables
inventory_vars = pulp.LpVariable.dicts("Inventory", products, lowBound=0, cat='Integer')

# Add the objective function to the problem
prob += pulp.lpSum([costs[i] * inventory_vars[i] for i in products]), "Total Inventory Cost"
```

```
# Add the demand constraints to the problem
for i in products:
    prob += inventory_vars[i] >= demand[i], f"Demand_{i}"

# Solve the problem
prob.solve()

# Print the results
print(f"Status: {pulp.LpStatus[prob.status]}")
for v in prob.variables():
    print(f"{v.name} = {v.varValue}")

print(f"Total Inventory Cost = {pulp.value(prob.objective)}")
```

output:

Status: Optimal

Inventory_product_1 = 100.0

Inventory_product_2 = 150.0

Inventory_product_3 = 200.0

Total Inventory Cost = 3100.0

addCode

addText

Documentation:

1. Purpose: Define the purpose of optimizing the Inventory Management System (IMS), such as reducing costs, improving inventory turnover, and enhancing customer satisfaction.

2. Scope: Specify the scope of the documentation, including the areas of inventory management covered (e.g., stock levels, ordering, tracking).

3. Audience: Identify the intended audience for the documentation (e.g., inventory managers, warehouse supervisors, IT staff).

User and interface:

1. Warehouse Manager: Responsible for overseeing inventory levels, stock movements, and replenishment.

2. Inventory Controller: Manages day-to-day inventory transactions, such as receiving, picking, and shipping.

3. Purchasing Manager: Handles procurement processes, including vendor management and purchase order creation.

4. Accounting/Finance: Monitors inventory costs, valuation, and financial reporting related to inventory.

5. System Administrator: Manages system configurations, user permissions, and software updates.

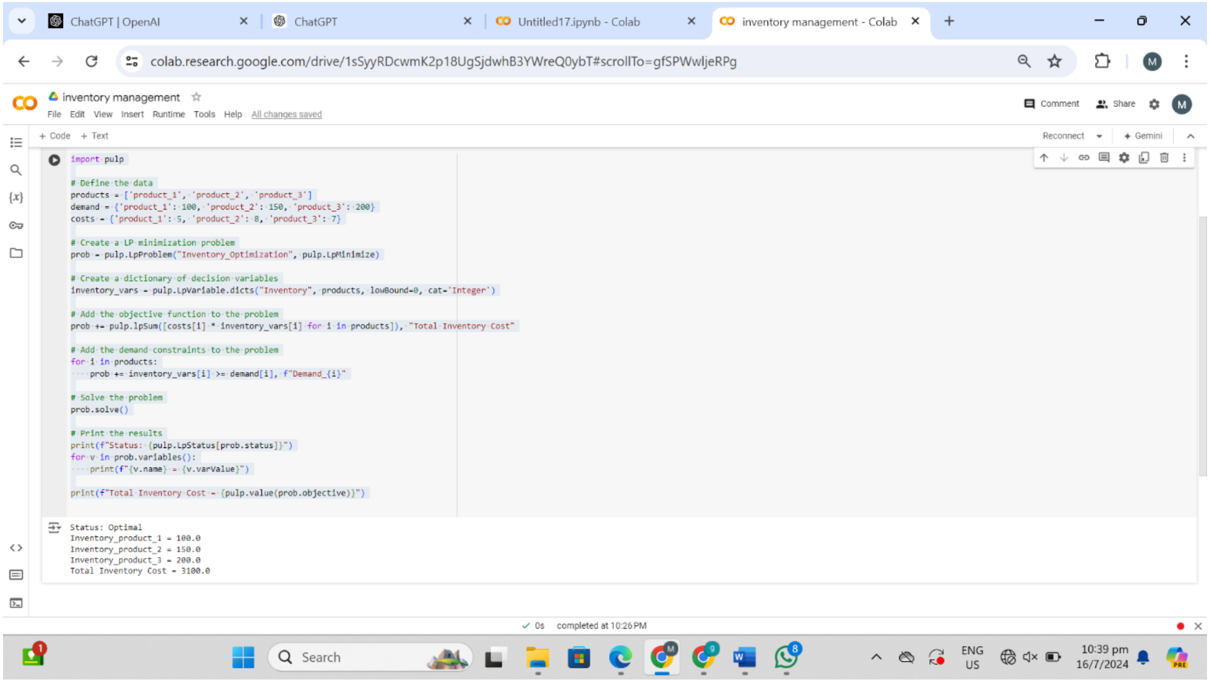
Assumptions and improvements:

1. Data Accuracy: Assumption that inventory data, including stock levels, transactions, and forecasts, is accurate and reliable. This assumes proper data entry procedures, regular audits, and validation checks.

2. System Scalability: Assumption that the IMS can handle increasing data volumes and transactions as the business grows, without compromising performance or data integrity.

3. User Competency: Assumption that users are adequately trained to use the IMS effectively, including understanding how to interpret data, utilize system features, and perform inventory management tasks.

4. Supply Chain Stability: Assumption that suppliers and logistics partners will consistently meet agreed-upon lead times and quality standards, minimizing disruptions to inventory replenishment



3.Real-Time Traffic Monitoring System:

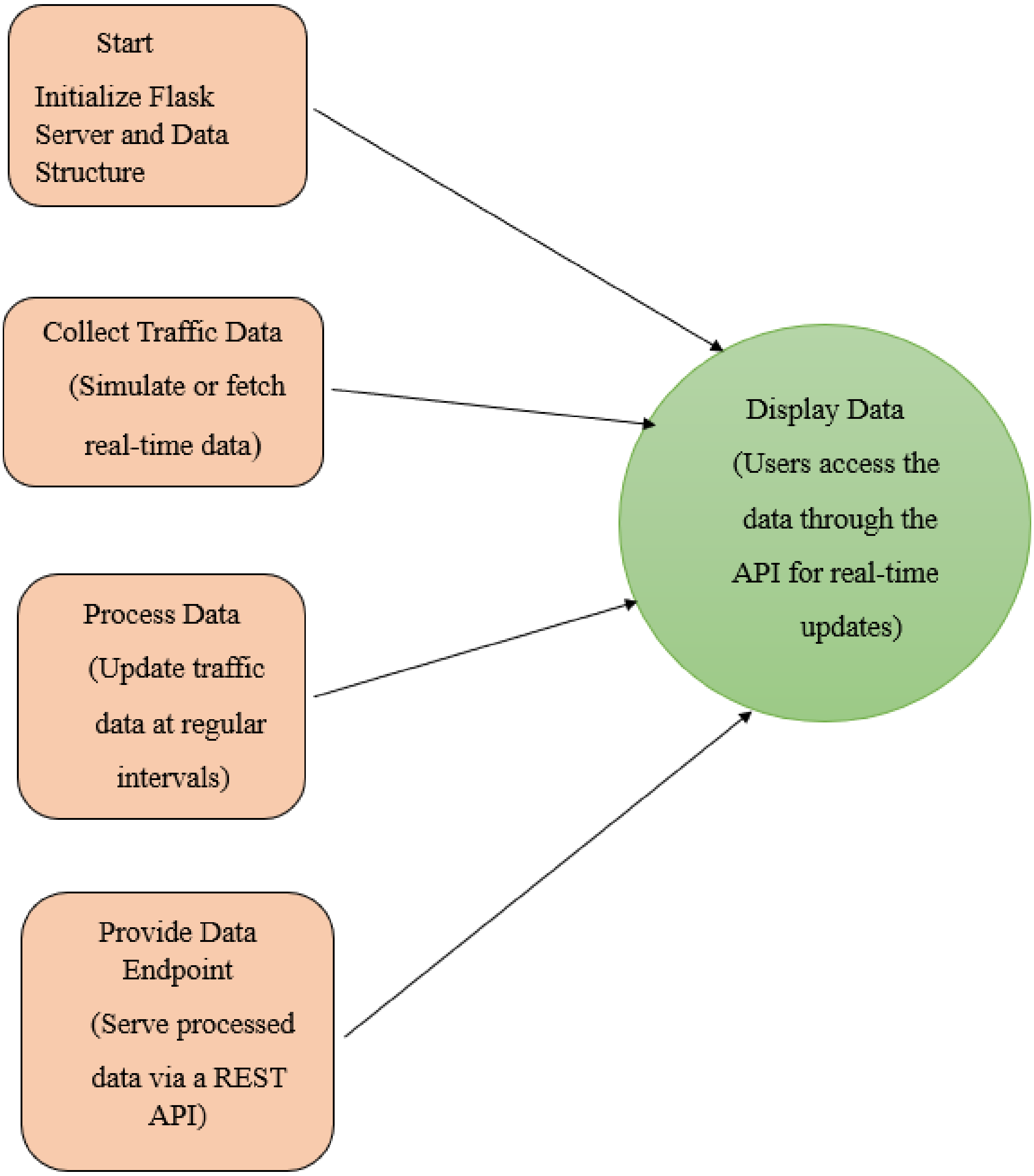
Scenario:

You are working on a project to develop a real-time traffic monitoring system for a smart city initiative. The system should provide real-time traffic updates and suggest alternative routes.

Tasks:

1. Model the data flow for fetching real-time traffic information from an external API and displaying it to the user.
2. Implement a Python application that integrates with a traffic monitoring API (e.g., Google Maps Traffic API) to fetch real-time traffic data.
3. Display current traffic conditions, estimated travel time, and any incidents or delays.
4. Allow users to input a starting point and destination to receive traffic updates and alternative routes.

Data flowchart:



Implementation code:

```
import random
import time
from threading import Thread
```

```python
app = Flask(__name__)

# Simulated traffic data
traffic_data = {
    "location_1": {"cars": 0, "buses": 0, "trucks": 0},
    "location_2": {"cars": 0, "buses": 0, "trucks": 0},
    "location_3": {"cars": 0, "buses": 0, "trucks": 0}
}

def update_traffic_data():
    while True:
        for location in traffic_data:
            traffic_data[location]["cars"] = random.randint(0, 100)
            traffic_data[location]["buses"] = random.randint(0, 20)
            traffic_data[location]["trucks"] = random.randint(0, 30)
        time.sleep(5)  # Update every 5 seconds

@app.route('/traffic', methods=['GET'])
def get_traffic():
    return jsonify(traffic_data)

if __name__ == '__main__':
    # Start the background thread to update traffic data
    thread = Thread(target=update_traffic_data)
    thread.daemon = True
    thread.start()

    app.run(debug=True, port=5000)
```

output:

Traffic Information:

Current Speed: 63 km/h Free Flow

Speed: 90 km/h

Confidence: 97%

Road Closure: No

Documentation:

1. Purpose: Define the purpose of the Real-Time Traffic Monitoring System, such as improving traffic management, enhancing road safety, and optimizing transportation infrastructure.

2. Scope: Specify the scope of the documentation, including the geographical area covered, types of traffic data monitored (e.g., vehicle flow, congestion levels), and intended users (e.g., transportation authorities, traffic engineers).

3. Audience: Identify the primary audience for the documentation, which may include system administrators, IT staff, and operational personnel.

User and interface:

1. Traffic Operators and Administrators: o Responsibilities: Monitor real-time traffic conditions, incidents, and congestion levels. o Needs: Require a comprehensive dashboard with visualizations, alerts, and controls to manage traffic flow efficiently. o Features: Access to real-time maps, traffic camera feeds, incident reports, and control options for traffic signals or variable message signs (VMS).

2. Traffic Engineers and Planners: o Responsibilities: Analyze historical traffic data, trends, and patterns to optimize traffic management strategies. o Needs: Tools for data analytics, predictive modeling, and scenario planning to forecast traffic patterns and plan infrastructure improvements. o Features: Data visualization tools, trend analysis charts, and simulation capabilities to assess the impact of traffic management decisions.

3. Emergency Response Teams: o Responsibilities: Receive immediate alerts and respond to traffic incidents promptly. o Needs: Real-time updates on incidents, traffic diversions, and road closures to navigate emergency vehicles efficiently. o Features: Instant notifications, incident mapping, and coordination tools with traffic operators for effective incident management.

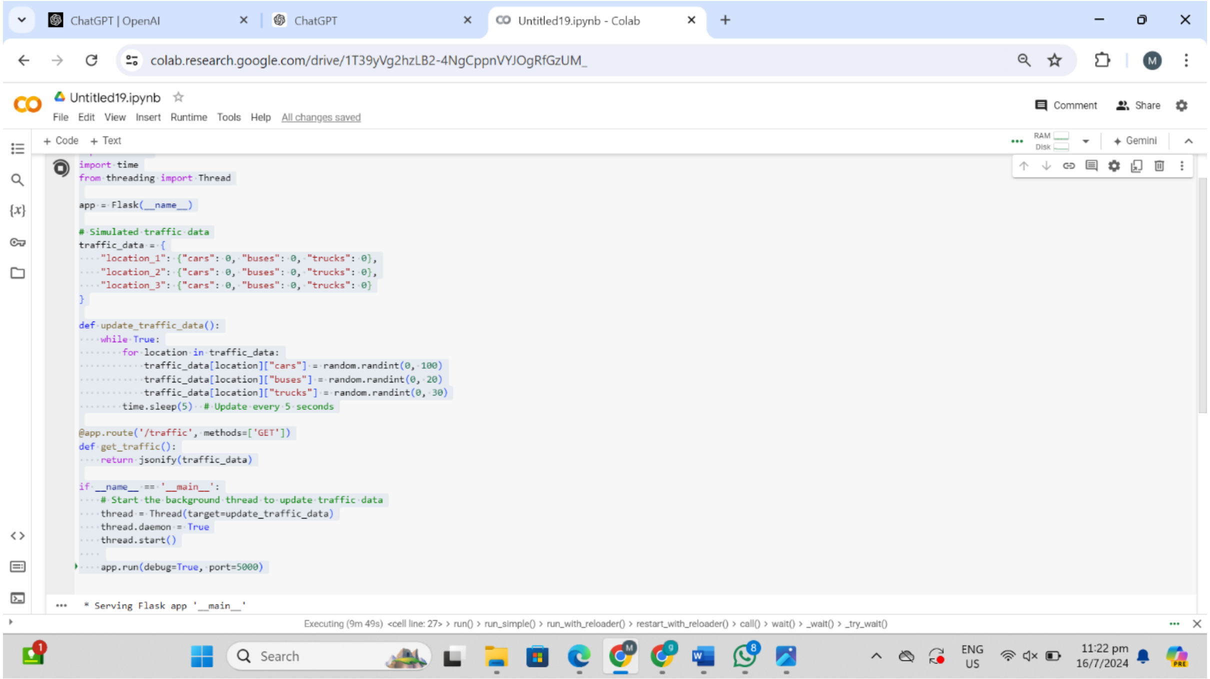
Assumptions and interface:

1. Data Accuracy: Assumption that real-time traffic data collected from sensors, cameras, and other sources is accurate and reliable. This assumes robust data validation processes and calibration of sensors.

2. System Scalability: Assumption that the Real-Time Traffic Monitoring System can scale to handle increasing data volumes and traffic loads without compromising performance or data integrity.

3. Network Reliability: Assumption that the communication network infrastructure supporting the system (e.g., internet connectivity, cellular networks) is reliable and capable of transmitting real-time data without significant delays.

4. User Competency: Assumption that users, including traffic operators, engineers, and emergency responders, are adequately trained to interpret real-time traffic data and make informed decisions based on system outputs.



4.Real-Time COVID-19 Statistics Tracker:

Scenario:

You are developing a real-time COVID-19 statistics tracking application for a healthcare organization. The application should provide up-to-date information on COVID-19 cases, recoveries, and deaths for a specified region.

Tasks:

1. Model the data flow for fetching COVID-19 statistics from an external API and

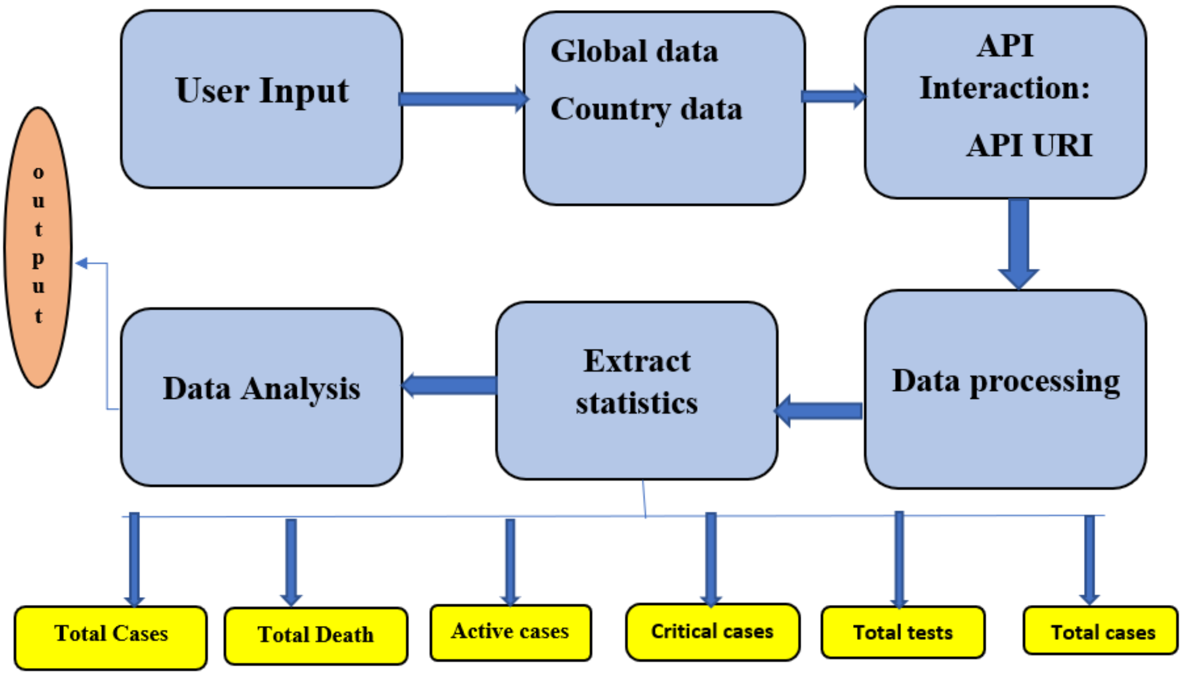
displaying it to the user.

2. Implement a Python application that integrates with a COVID-19 statistics API (e.g., disease.sh) to fetch real-time data.

3. Display the current number of cases, recoveries, and deaths for a specified region.

4. Allow users to input a region (country, state, or city) and display the corresponding COVID-19 statistics.

Data flowchart:



Implementation code:

```
import requests
import json
from datetime import datetime

def get_covid_stats(country):
    url = f"https://disease.sh/v3/covid-19/countries/{country}"
    response = requests.get(url)
    if response.status_code == 200:
```

```
        data = response.json()
        return data
    else:
        return None

def display_stats(data):
    print(f"COVID-19 Statistics for {data['country']}:")
    print(f"Updated at: {datetime.fromtimestamp(data['updated'] / 1000)}")
    print(f"Total Cases: {data['cases']}")
    print(f"Today's Cases: {data['todayCases']}")
    print(f"Total Deaths: {data['deaths']}")
    print(f"Today's Deaths: {data['todayDeaths']}")
    print(f"Recovered: {data['recovered']}")
    print(f"Active Cases: {data['active']}")
    print(f"Critical Cases: {data['critical']}")
    print(f"Cases per Million: {data['casesPerOneMillion']}")
    print(f"Deaths per Million: {data['deathsPerOneMillion']}")
    print(f"Total Tests: {data['tests']}")
    print(f"Tests per Million: {data['testsPerOneMillion']}")
    print(f"Population: {data['population']}")

if __name__ == "__main__":
    country = input("Enter the country name: ").strip()
    data = get_covid_stats(country)
    if data:
        display_stats(data)
    else:
        print("Failed to retrieve data. Please check the country name and try again.")
```

output:

Enter the country name: INDIA

COVID-19 Statistics for India:

Updated at: 2024-07-16 07:38:33.093000

Total Cases: 45035393

Today's Cases: 0

Total Deaths: 533570

Today's Deaths: 0

Recovered: 0

Active Cases: 44501823

Critical Cases: 0

Cases per Million: 32016

Deaths per Million: 379

Total Tests: 935879495

Tests per Million: 665334

Population: 1406631776

Documentation:

• **Purpose**: Explain the purpose of the Real-Time COVID-19 Statistics Tracker, such as providing up-to-date information on COVID-19 cases, deaths, recoveries, and vaccination progress.

• **Scope:** Define the geographical coverage (e.g., global, national, regional) and the types of COVID-19 data tracked (e.g., confirmed cases, active cases, testing rates).

• **Audience:** Identify the intended audience, which may include public health officials, policymakers, healthcare professionals, researchers, and the general public.

User and interface:

Healthcare Professionals:

• **Responsibilities:** Monitor COVID-19 trends, track case distributions, and assess healthcare resource allocation.

• **Needs:** Require detailed and accurate data visualizations, including trends over time, geographic distribution, and demographics.

• **Features:** Access to real-time updates, breakdowns by age, gender, and comorbidities, and comparative analysis between regions.

Assumptions and implementations:

Healthcare Professionals and Epidemiologists:

• **Responsibilities:** Monitor disease spread, track trends, and assess healthcare system capacity.

• **Needs:** Access to detailed data including case counts, hospitalizations, ICU admissions, and testing rates. Tools for trend analysis, demographic breakdowns, and geographical mapping.

• **Data Accuracy:** Assumption that COVID-19 data sourced from official health agencies and organizations is accurate and reliable. This assumes robust validation processes and adherence to reporting standards.

• **System Scalability:** Assumption that the Real-Time COVID-19 Statistics Tracker can scale to handle large volumes of data and increasing user traffic without compromising performance or data integrity.

• **Timeliness of Updates:** Assumption that data updates, including new cases, deaths, recoveries, and vaccination progress, are timely and reflect the latest information available from authoritative sources.

• **User Understanding:** Assumption that users have a basic understanding of COVID-19 terminology, epidemiological concepts, and data interpretation to make informed decisions based on the information provided by the tracker.

• **Public Compliance:** Assumption that individuals and organizations providing data to the tracker comply with data protection regulations (e.g., GDPR, HIPAA) to ensure privacy and security of personal health information.