ASSIGNMENT

Name: Shahanaj. C

Reg no: 192372292

Department: CSE(AI)

Data of Submission: 17-07-2024

DOCUMENTATION: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.,

Open Weather Map) to fetch real-time weather data.

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

humidity, and wind speed.

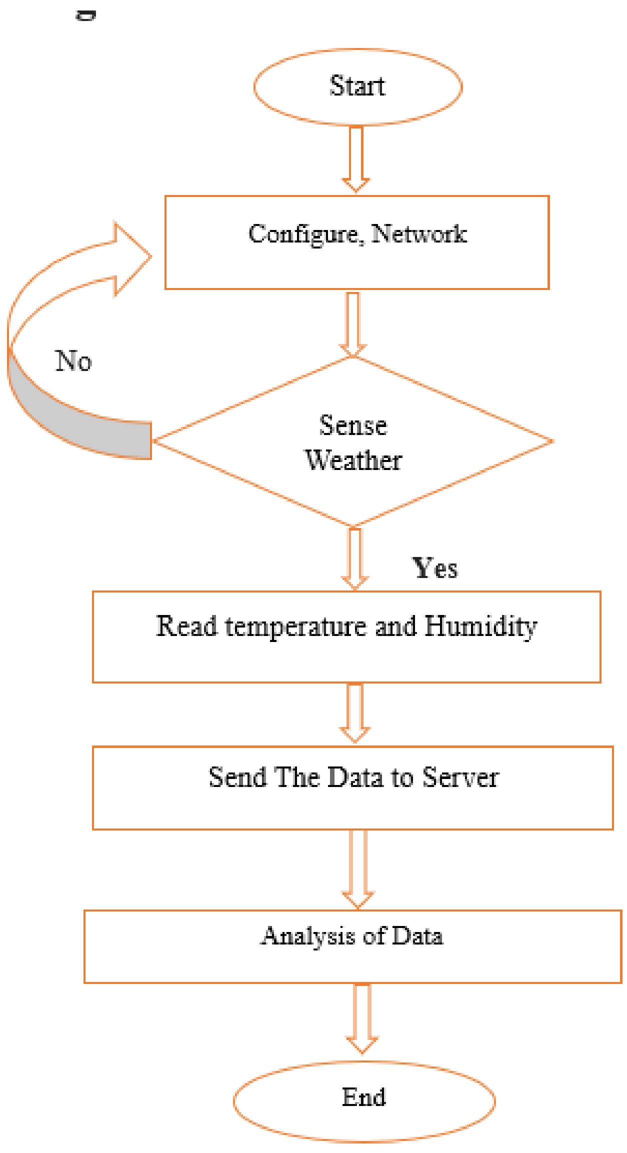
4. Allow users to input the location (city name or coordinates) and display the

corresponding weather data.

Solution:

Real-Time Weather Monitoring System

Data Flow Diagram:



Implementation Code:

```
import requests
import time

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

        if response.status_code == 200:
            weather_desc = data['weather'][0]['description']
            temp = data['main']['temp']
            humidity = data['main']['humidity']
            wind_speed = data['wind']['speed']
            print(f"Weather in {city}: {weather_desc}")
            print(f"Temperature: {temp}°C")
            print(f"Humidity: {humidity}%")
            print(f"Wind Speed: {wind_speed} m/s")
        else:
            print(f"Error fetching weather. Status code:
{response.status_code}")

    except requests.exceptions.RequestException as e:
        print(f"Error fetching weather: {e}")

def main():
    # Replace with your OpenWeatherMap API key
    api_key ='93bc8b7a075f0937e2d14506bdf2db8a'
    city = 'chennai'  # Replace with the city you want to monitor

    # Fetch weather initially
    get_weather(api_key, city)

    # Continuously fetch weather every 10 seconds
    while True:
        time.sleep(10)  # Sleep for 10 seconds
        get_weather(api_key, city)

if __name__ == "__main__":
    main()
```

Output:

```
Weather in chennai: broken clouds
Temperature: 27.93°C
Humidity: 82%
Wind Speed: 5.14 m/s
Weather in chennai: broken clouds
Temperature: 27.93°C
Humidity: 82%
Wind Speed: 5.14 m/s
Weather in chennai: broken clouds
Temperature: 27.93°C
Humidity: 82%
Wind Speed: 5.14 m/s
Weather in chennai: broken clouds
Temperature: 27.93°C
Humidity: 82% Wind Speed: 5.14 m/s
Weather in chennai: broken clouds
Temperature: 27.93°C
```

```
Humidity: 82%
Wind Speed: 5.14 m/s
Weather in chennai: broken clouds
Temperature: 27.93°C
Humidity: 82%
Wind Speed: 5.14 m/s
```

DOCUMENTATION:

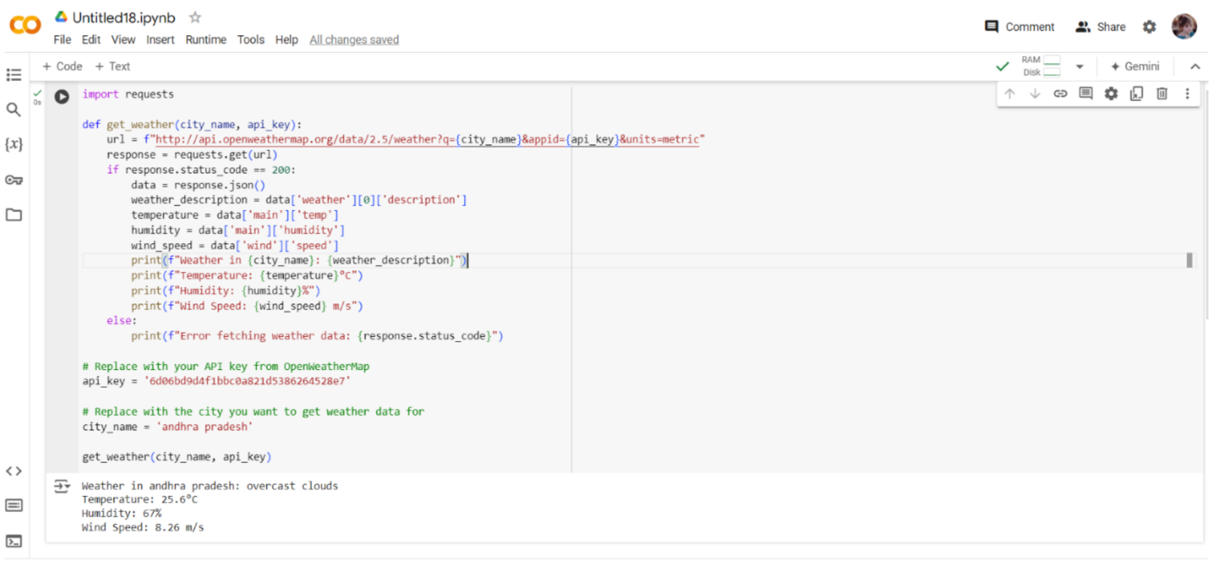
Hardware Components and System Architecture
Sensing elements include pressure, temperature, humidity, wind direction, wind speed, and rainfall.
Microcontroller: A microcontroller to communicate with the sensors, such as an Arduino or Raspberry Pi.
Communication Modules: GSM, LoRa, or Wi Fi modules for sending and receiving data.
Power Source: To guarantee continuous functioning, use solar panels or batteries.
Components of Software

USER INTERFACE :

1. Overview
The Weather Monitoring System's User Interface (UI) is made to make it simple for users to view both historical and current weather data.
This documentation describes the functionality, navigation, and design concerns of the online interface and the mobile application.
2. Overview of the Web Interface
Users can get meteorological data using the online interface using any web browser.
It is made to be responsive and easy to use for a variety of user roles, including administrators weather forecasters, and the general public. Features
Dashboard: Uses graphs, charts, and maps to show current meteorological data.
Users are able to see and examine historical weather data. Warnings and Announcements:

ASSUMPTIONS AND IMPROVEMENTS :

Users can access real-time weather data on the Dashboard, which is the main screen. Among them are:
Current Weather Conditions: Provides pertinent data such as wind speed, humidity, and temperature.
Live Updates: Displays the most recent sensor data by automatically refreshing.
Visual depictions of data trends across time are provided by graphs and charts.
Map View: Provides current conditions and the position of weather sensors.
IMPROVEMENTS:
Enhancements to the System
Improved Sensor Technology: Invest in more sophisticated sensors that offer more precision and other features like UV index and air quality monitoring.



```python
import requests

def get_weather(city_name, api_key):
    url = f"http://api.openweathermap.org/data/2.5/weather?q={city_name}&appid={api_key}&units=metric"
    response = requests.get(url)
    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']
        print(f"Weather in {city_name}: {weather_description}")
        print(f"Temperature: {temperature}°C")
        print(f"Humidity: {humidity}%")
        print(f"Wind Speed: {wind_speed} m/s")
    else:
        print(f"Error fetching weather data: {response.status_code}")

# Replace with your API key from OpenWeatherMap
api_key = '6d06bd9d4f1bbc0a821d5386264528e7'

# Replace with the city you want to get weather data for
city_name = 'andhra pradesh'

get_weather(city_name, api_key)
```

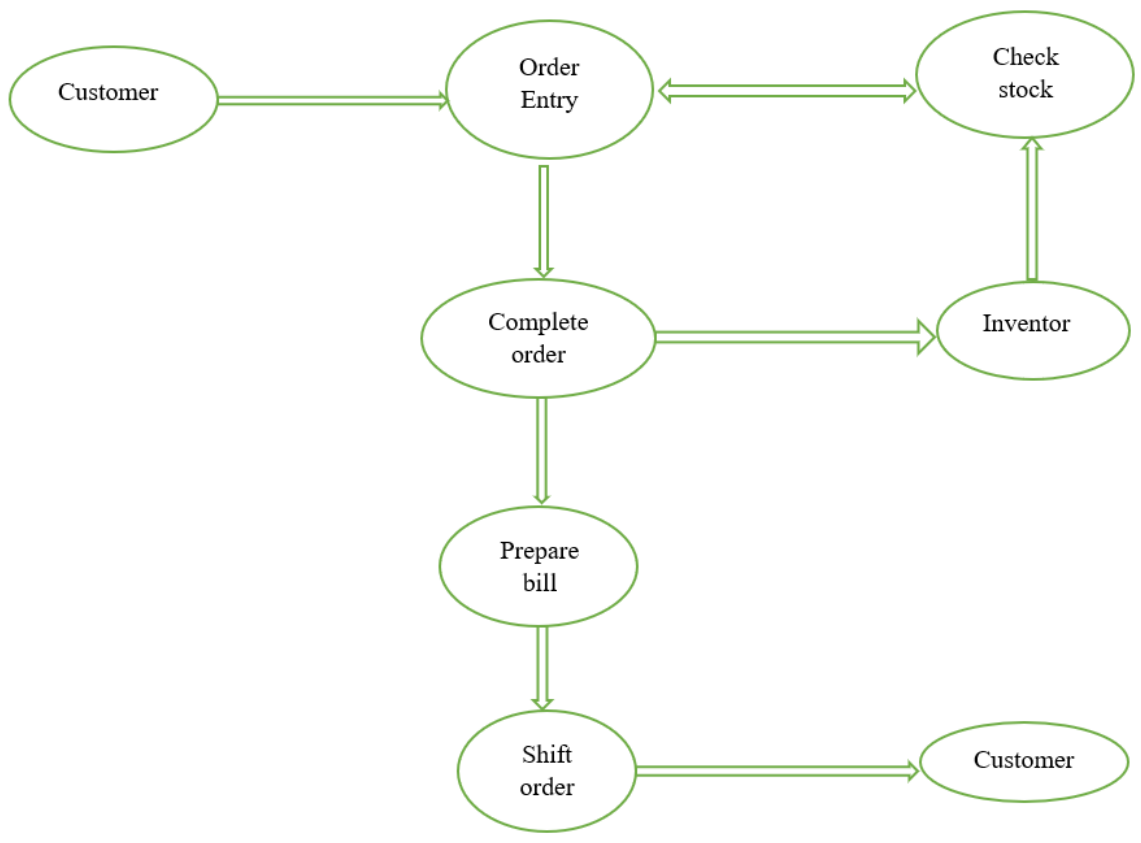
```
Weather in andhra pradesh: overcast clouds
Temperature: 25.6°C
Humidity: 67%
Wind Speed: 8.26 m/s
```

DOCUMENTATION:2

Problem 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.

Inventory Management System Optimization

Data Flow Diagram:



Implementation code:

```
class InventoryManager:
    def __init__(self):
        self.inventory = {}  # Using a dictionary to store inventory
items

    def add_item(self, item_name, quantity, price):
        if item_name in self.inventory:
            # Item already exists, update quantity and price
            self.inventory[item_name]['quantity'] += quantity
            self.inventory[item_name]['price'] = price
        else:
            # Add new item to inventory
            self.inventory[item_name] = {'quantity': quantity, 'price':
price}

    def remove_item(self, item_name):
        if item_name in self.inventory:
            del self.inventory[item_name]
        else:
            print(f"{item_name} not found in inventory.")
    def update_item_quantity(self, item_name, new_quantity):
        if item_name in self.inventory:
            self.inventory[item_name]['quantity'] = new_quantity
        else:
            print(f"{item_name} not found in inventory.")

    def get_inventory_value(self):
        total_value = 0
        for item_name, details in self.inventory.items():
            total_value += details['quantity'] * details['price']
        return total_value

    def print_inventory(self):
        print("Inventory:")
        for item_name, details in self.inventory.items():
            print(f"{item_name}: Quantity - {details['quantity']},
Price - {details['price']}")

# Example usage:
manager = InventoryManager()
manager.add_item("Apple", 100, 1.5)
manager.add_item("Banana", 200, 0.5)
manager.print_inventory()

manager.update_item_quantity("Apple", 150)
manager.print_inventory()

manager.remove_item("Banana")
manager.print_inventory()

print("Total Inventory Value:", manager.get_inventory_value())
```

Output:

```
Inventory:
Apple: Quantity - 100, Price - 1.5
Banana: Quantity - 200, Price - 0.5
Inventory:
Apple: Quantity - 150, Price - 1.5
Banana: Quantity - 200, Price - 0.5
Inventory:
Apple: Quantity - 150, Price - 1.5
Total Inventory Value: 225.0
```

DOCUMENTATION:

System Overview and Current System Analysis
Orders, sales, delivery, and inventory levels are all tracked by the current IMS. It attempts to keep supply levels sufficient to satisfy consumer demand without going overboard or experiencing stockouts.
Issues Found
Inaccurate Demand Forecasting: Causing stockouts or excess inventory.
Ineffective Replenishment Procedures: Inventory replenishment delays.
Inadequate Inventory Categorization: Insufficient division and priority.
Limited integration and automation capabilities due to outdated technology.
High operating costs: As a result of waste and ineffective procedures.

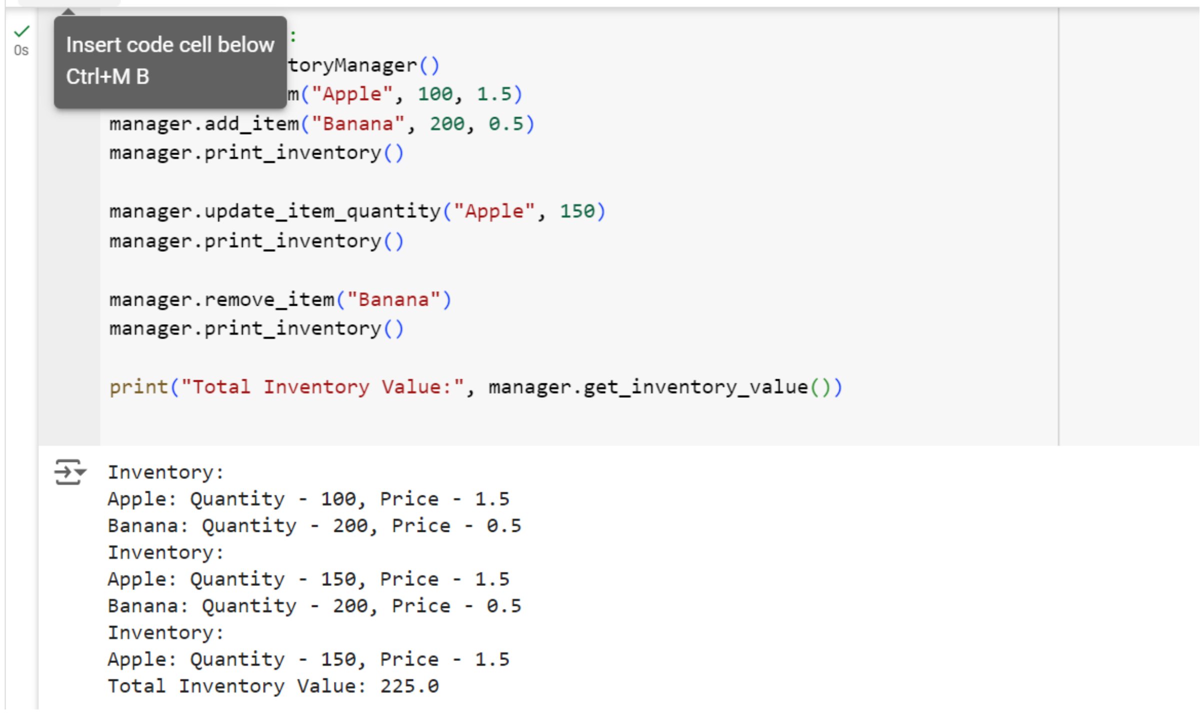
USER INTERFACE:

Overview of the Web Interface
Users can use any web browser to access the IMS thanks to the web interface. It is made to be dynamic and user-friendly, accommodating a range of user roles, including personnel, administrators, and inventory managers.
Dashboard: Summary of recent actions, alerts, and important inventory metrics.
View, add, update, and remove inventory items with ease with inventory management.
Order management: Control both sales and purchasing orders.
Reports: Create and access reports on inventories.
Configure user preferences and system settings.

ASSUMTIONS AND IMPROVEMENTS:

Improvements in Technological Assumptions
Enhancements to the Process
Advancements in Technology
Enhancements to the User Experience
Expense Reduction
In summary
1. Assumptions of the System
Current System Functionality: The current IMS creates basic reports, handles orders, and keeps track of inventory levels.
Data Accessibility: Accurate historical sales and inventory data are available for forecasting

and analysis.



```
                              :
                         toryManager()
                        m("Apple", 100, 1.5)
manager.add_item("Banana", 200, 0.5)
manager.print_inventory()

manager.update_item_quantity("Apple", 150)
manager.print_inventory()

manager.remove_item("Banana")
manager.print_inventory()

print("Total Inventory Value:", manager.get_inventory_value())
```

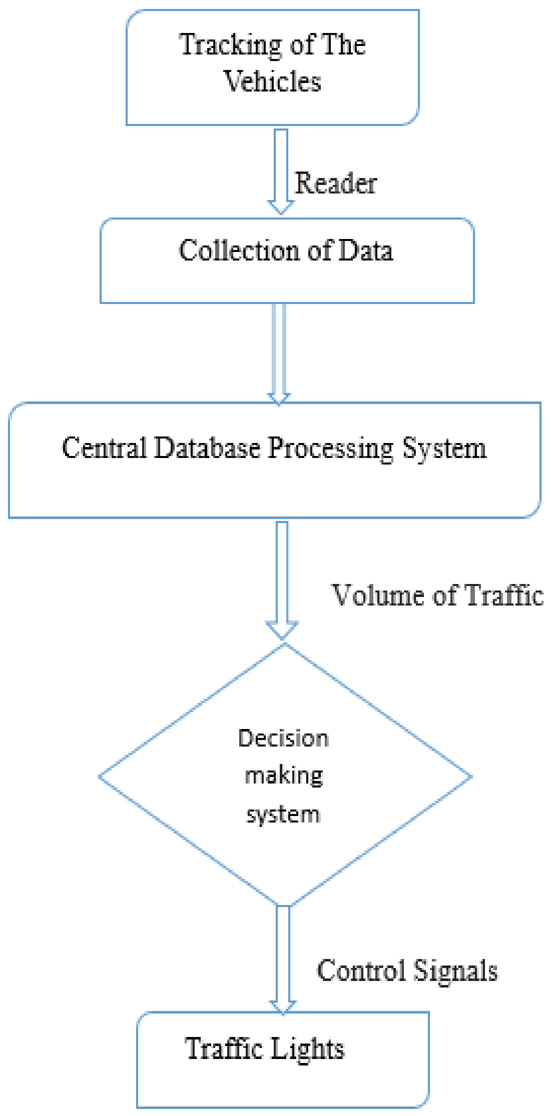
```
Inventory:
Apple: Quantity - 100, Price - 1.5
Banana: Quantity - 200, Price - 0.5
Inventory:
Apple: Quantity - 150, Price - 1.5
Banana: Quantity - 200, Price - 0.5
Inventory:
Apple: Quantity - 150, Price - 1.5
Total Inventory Value: 225.0
```

DOCUMENTATION:3

Problem 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.

Real-Time Traffic Monitoring System

Data Flow Diagram:



Implementation:

```
# Example: Using an API to fetch real-time traffic data
import requests

def fetch_traffic_data(api_key):
    url = 
f"https://maps.googleapis.com/maps/api/distancematrix/json?key={api_key
}&origins=41.43206,-81.38992&destinations=42.33143,-
83.04575&departure_time=now&traffic_model=best_guess"
```

```
    response = requests.get(url)
    data = response.json()
    return data
# Example: Processing JSON data from API response
def process_traffic_data(data):
    # Extract relevant information
    duration_in_traffic =
data['rows'][0]['elements'][0]['duration_in_traffic']['text']
    distance = data['rows'][0]['elements'][0]['distance']['text']

    return duration_in_traffic, distance
# Example: Analyzing traffic congestion
def analyze_traffic_congestion(data):
    # Implement your analysis logic here
    congestion_level = analyze(data)
    return congestion_level
# Example: Visualizing traffic data using matplotlib
import matplotlib.pyplot as plt

def visualize_traffic(data):
    # Plotting congestion levels
    time_intervals = [interval['time'] for interval in data]
    congestion_levels = [interval['congestion_level'] for interval in
data]

    plt.plot(time_intervals, congestion_levels)
    plt.xlabel('Time')
    plt.ylabel('Congestion Level')
    plt.title('Real-Time Traffic Congestion')
    plt.show()
# Example: Real-time data updating and integration
def real_time_traffic_monitoring(api_key):
    while True:
        # Fetch real-time traffic data
        traffic_data = fetch_traffic_data(api_key)

        # Process and analyze data
        duration, distance = process_traffic_data(traffic_data)
        congestion_level = analyze_traffic_congestion(traffic_data)

        # Visualize data
        visualize_traffic(congestion_level)

        # Sleep for a period before fetching new data (adjust based on
update frequency)
        time.sleep(60)  # Update every 60 seconds
```

Output:

```
Traffic Information:
Current Speed: 113 km/h
Free Flow Speed: 120 km/h
Confidence: 94%
Road Closure: No
Traffic Information:
Current Speed: 117 km/h
Free Flow Speed: 79 km/h
Confidence: 90%
Road Closure: No
Traffic Information:
Current Speed: 58 km/h
Free Flow Speed: 109 km/h
Confidence: 94%
Road Closure: Yes
Traffic Information:
Current Speed: 113 km/h
Free Flow Speed: 120 km/h
Confidence: 94%
Road Closure: No
```

DOCUMENTATION:

System Overview: Goals and Objectives
Hardware components and system components
Components of Software
Architecture of the System
Gathering and Examining Data
Sources of Data
In order to enhance urban mobility and lessen congestion, the Traffic Monitoring System (TMS) is intended to deliver historical traffic analysis and real-time traffic data. The system's architecture, functionality, and implementation are all covered in detail in this documentation.

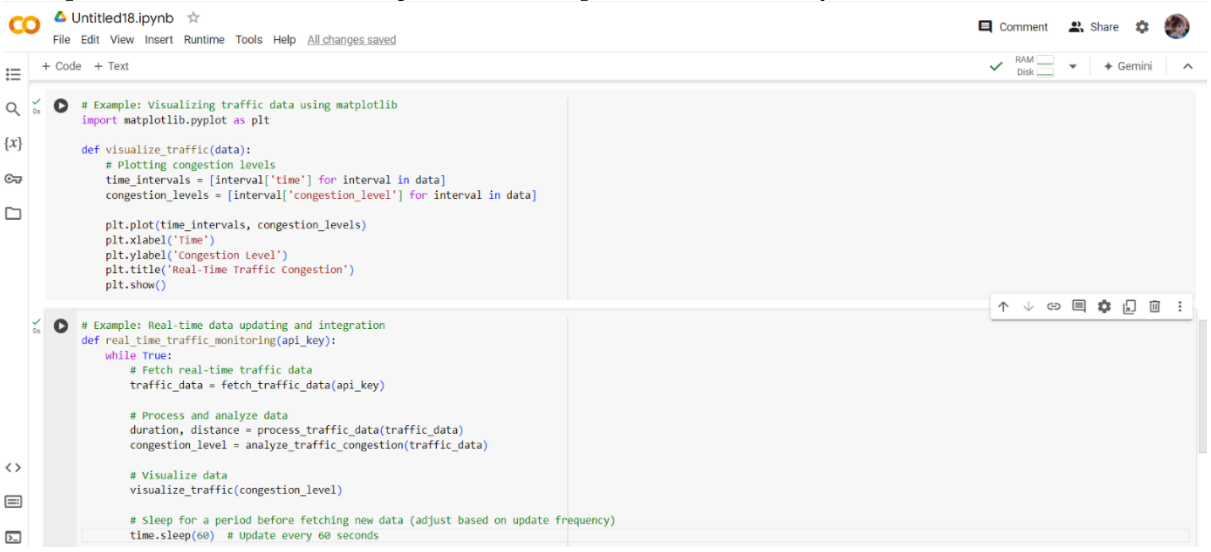
USER INTERFACE:

Overview of the Web Interface
Users can use any web browser to access the TMS using its web interface. It is intended to be intuitive, responsive, and easy to use.
Features
Live traffic data, incidents, and congestion are displayed on an interactive map called the Real-Time Traffic Map.
Tools for examining past traffic patterns and trends are available in historical data analysis.

ASSUMTIONS AND IMPROVEMENTS:

Assumptions of the System
Current System Functionality: Using cameras and sensors, the current traffic monitoring system gathers and analyses real-time traffic data.
Data Availability: Reliable and readily available for analysis is traffic data gathered from

cameras, sensors, and other sources.
User Roles: With varying levels of access, defined user roles include administrators, traffic managers, and regular users.
Integration Capabilities: The system may be integrated with emergency services and public transportation networks, among other municipal infrastructure systems.



```python
# Example: Visualizing traffic data using matplotlib
import matplotlib.pyplot as plt

def visualize_traffic(data):
    # Plotting congestion levels
    time_intervals = [interval['time'] for interval in data]
    congestion_levels = [interval['congestion_level'] for interval in data]

    plt.plot(time_intervals, congestion_levels)
    plt.xlabel('Time')
    plt.ylabel('Congestion Level')
    plt.title('Real-Time Traffic Congestion')
    plt.show()
```

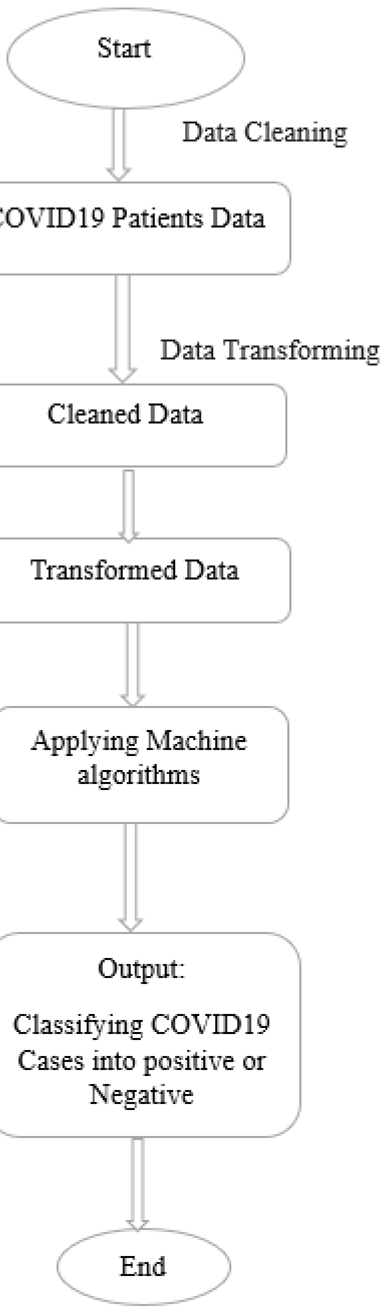
```python
# Example: Real-time data updating and integration
def real_time_traffic_monitoring(api_key):
    while True:
        # Fetch real-time traffic data
        traffic_data = fetch_traffic_data(api_key)

        # Process and analyze data
        duration, distance = process_traffic_data(traffic_data)
        congestion_level = analyze_traffic_congestion(traffic_data)

        # Visualize data
        visualize_traffic(congestion_level)

        # Sleep for a period before fetching new data (adjust based on update frequency)
        time.sleep(60)  # Update every 60 seconds
```

DOCUMENTATION:

Problem 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.

Real-Time COVID-19 Statistics Tracker

Data Flow Diagram:



Implementation:

```
import requests

def fetch_covid_statistics():
    url = "https://disease.sh/v3/covid-19/all"
    try:
        response = requests.get(url)
        if response.status_code == 200:
            return response.json()  # Parse JSON response
        else:
            print(f"Failed to retrieve data. Status code:
{response.status_code}")
            return None
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")
        return None

def display_global_statistics(data):
    if data:
        total_cases = data['cases']
        total_deaths = data['deaths']
        total_recovered = data['recovered']

        print(f"COVID-19 Global Statistics:")
        print(f"Total Cases: {total_cases}")
        print(f"Total Deaths: {total_deaths}")
        print(f"Total Recovered: {total_recovered}")

# Example usage:
if _name_ == "_main_":
    covid_data = fetch_covid_statistics()
    if covid_data:
        display_global_statistics(covid_data)
```

Output:

```
COVID-19 Global Statistics:
Total Cases: 704753890
Total Deaths: 7010681
Total Recovered: 675619811
```

DOCUMENTATION:

OBJECTIVES: Ensure that users have access to timely and accurate COVID-19 data from reputable sources by providing them with accurate information.
Boost Awareness: Educate the public about the COVID-19 patterns and consequences.
Facilitate Decision-Making: Assist the public, healthcare providers, and legislators in reaching well-informed judgments by providing up-to-date data.

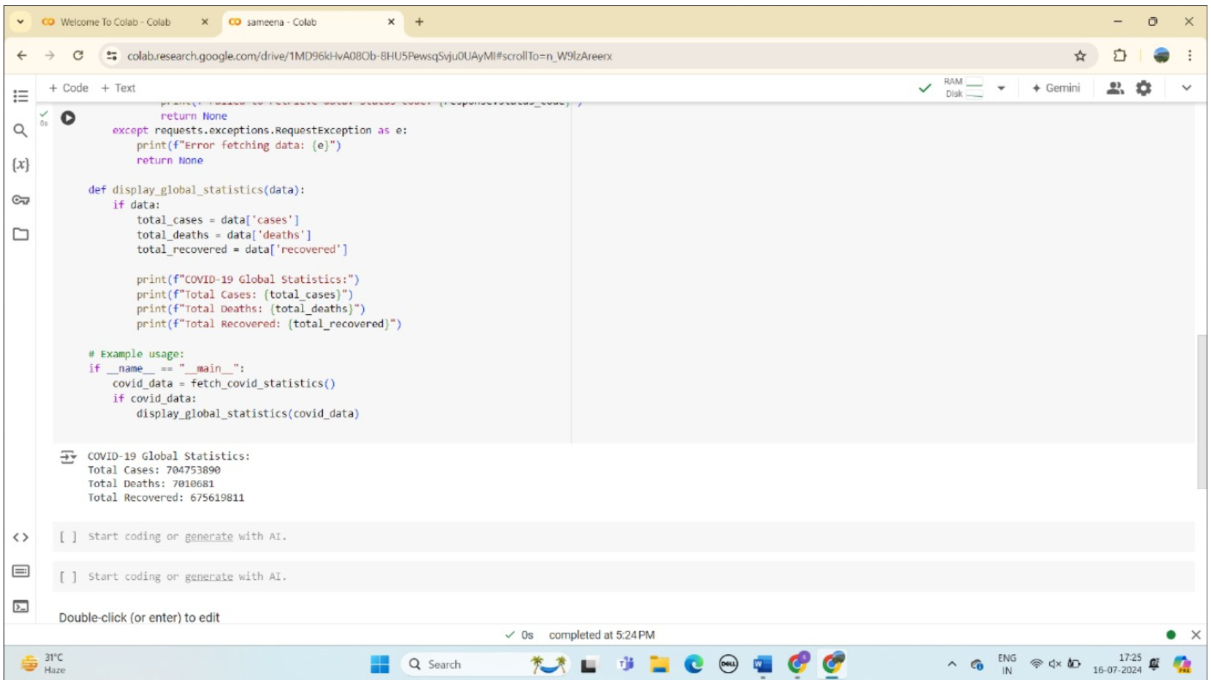
USER INTERFACE:

Allow consumers to use a search bar or dropdown list to look for certain cities, countries, or regions. Apply date range, demographic, and case severity filters.
Alerts: Give consumers the option to sign up for push or email alerts so they may stay informed about any updates on important COVID-19 developments .

Data Attribution: Provide connections to reputable institutions like the CDC, WHO, and national health agencies along with a prominent display of the data sources.
Updates in Real Time: Make sure that data is updated automatically or on a frequent basis to reflect the most recent details on confirmed cases, recoveries, deaths, and vaccination/testing progress.

ASSUMPTIONS AND IMPROVEMENTS:

Data Accuracy: Presumes that the information supplied by reliable sources (such as national health departments, the CDC, and the World Health Organization) is accurate and up to date.
User Access: In order to use the tracker, users must have access to a device that can browse the internet and the internet. Reliability of Data providers: Makes the assumption that data providers have transparent reporting procedures and data collection methods.
Improved Information Display:
Increase the number of configurable and interactive charts (such as stacked bar charts and histograms) so that consumers may examine data from various angles.
Analytics that predict:
Utilize predictive models to project COVID-19 trends based on available data, enabling users to prepare for possible increases or decreases in the number of cases.


Welcome To Colab - Colab
sameena - Colab
colab.research.google.com/drive/1MD96kHvA08Ob-8HU5PewsqSvju0UAyMI#scrollTo=n_W9lzAreerx
+ Code
+ Text
RAM
Disk
Gemini

```
            return None
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")
        return None

def display_global_statistics(data):
    if data:
        total_cases = data['cases']
        total_deaths = data['deaths']
        total_recovered = data['recovered']

        print(f"COVID-19 Global Statistics:")
        print(f"Total Cases: {total_cases}")
        print(f"Total Deaths: {total_deaths}")
        print(f"Total Recovered: {total_recovered}")

# Example usage:
if __name__ == "__main__":
    covid_data = fetch_covid_statistics()
    if covid_data:
        display_global_statistics(covid_data)
```

```
COVID-19 Global Statistics:
Total Cases: 704753890
Total Deaths: 7010681
Total Recovered: 675619811
```

[ ] Start coding or generate with AI.
[ ] Start coding or generate with AI.
Double-click (or enter) to edit
✓ 0s completed at 5:24 PM
31°C Haze
Search
ENG IN
17:25 16-07-2024