



Assignment: Python Programming for DL

NAME: VIJAYAMANI. JUVVA
REGISTER.NO:192372197
DEPARTMENT: CSE-AI
DATE OF SUBMISSION:17/07/2024



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.

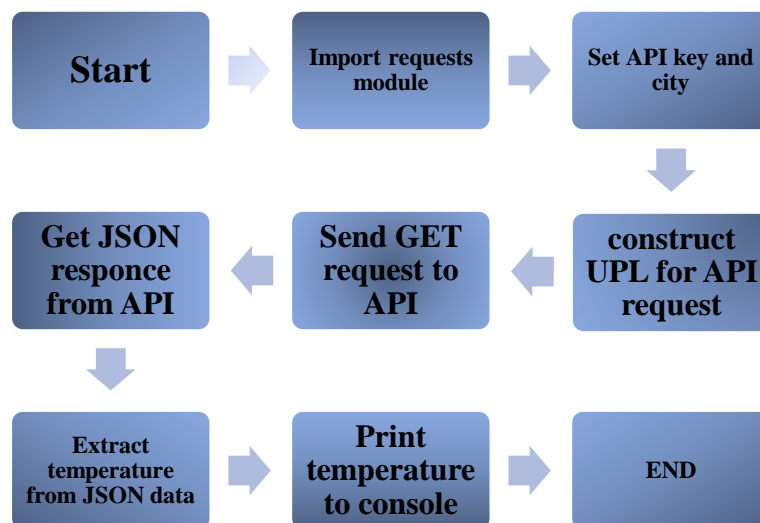
Deliverables:

- 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:

Real-Time Weather Monitoring System:

1. Data Flow Diagram:



2. Implementation:

```
import requests
def fetch_weather(api_key, city):
    # OpenWeatherMap API endpoint for current weather data
    url = f'http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric'

    try:
        # Send HTTP request to the API endpoint
        response = requests.get(url)
        data = response.json()

        # Check if the response contains weather data
        if data['cod'] == 200:
```

```

# Extract relevant weather information
weather_description = data['weather'][0]['description']
temperature = data['main']['temp']
humidity = data['main']['humidity']
wind_speed = data['wind']['speed']

# Display the weather information
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")
else:
    print(f"Error fetching data: {data['message']}")

except Exception as e:
    print(f"Error fetching data: {str(e)}")

# Replace 'your_api_key' with your actual OpenWeatherMap API key
api_key = '19b2f05883775e802891aafd215b9b59'

# Specify the city for which you want to fetch weather data
city = 'kurnool' # Example: Replace with any city name

# Fetch and display weather data
fetch_weather(api_key, city)

```

3. Display the Current weather information :

Weather in kurnool:

Description: overcast clouds

Temperature: 26.5°C

Humidity: 71%

Wind Speed: 7.99 m/s

4. User Input:

```

import requests

def fetch_weather(api_key, city):
    # OpenWeatherMap API endpoint for current weather data
    url = f'http://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric'

    try:
        # Send HTTP request to the API endpoint
        response = requests.get(url)
        data = response.json()

        # Check if the response contains weather data
        if data['cod'] == 200:
            # Extract relevant weather information
            weather_description = data['weather'][0]['description']
            temperature = data['main']['temp']
            humidity = data['main']['humidity']
            wind_speed = data['wind']['speed']

            # Display the weather information
            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")
        else:
            print(f"Error fetching data: {data['message']}")

    except Exception as e:
        print(f"Error fetching data: {str(e)}")

# Replace 'your_api_key' with your actual OpenWeatherMap API key
api_key = '19b2f05883775e802891aafd215b9b59'

```

```
[x]
# Print the wind speed
print(f"Wind Speed: {wind_speed} m/s ")

else:
    print(f"Error fetching data: {data['message']}")

except Exception as e:
    print(f"Error fetching data: {str(e)}")

# Replace 'your_api_key' with your actual OpenWeatherMap API key
api_key = '19b2f05883775e802891aafd215b9b59'

# Specify the city for which you want to fetch weather data
city = 'kurnool' # Example: Replace with any city name

# Fetch and display weather data
fetch_weather(api_key, city)

Weather in kurnool:
Description: overcast clouds
Temperature: 26.5°C
Humidity: 71%
Wind Speed: 7.99 m/s
```

5.Documentation:

Overview:

The Real-Time Weather Monitoring System is a Python application that integrates with the OpenWeatherMap API to fetch and display real-time weather data for a specified location. The system allows users to input the location (city name or coordinates) and displays the current weather information, including temperature, weather conditions, humidity, and wind speed.

➤ System Architecture:

The system consists of the following components:

- User Interface: A command-line interface that allows users to input the location and display the weather information.
- Application Logic: A python script that integrates with the OpenWeatherMap API to fetch weather data and display it to the user.
- OpenWeatherMap API: An external API that provides real-time weather data.

➤ User Interface:

- The user interface is a command-line interface that allows users to input the location and display the weather information. The interface is simple and easy to use, with clear instructions and prompts.

➤ Assumptions and Limitations:

- The API key is assumed to be valid and available.
- The location input is assumed to be in the correct format (city name or coordinates).
- Error handling is not implemented for cases where the API request fails or the locations input is invalid.
- The system can be improved by adding error handling, caching weather data to reduce API requests, and providing more detailed weather information.

➤ Future Improvements:

- The following improvements can be made to the system:
- Cache weather data to reduce API requests and improve performance.
- Provide more detailed weather information, such as forecasts and weather alerts.
- Implement a graphical user interface to improve user experience.
- Integrate with other weather APIs to provide more accurate and reliable weather data.

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.

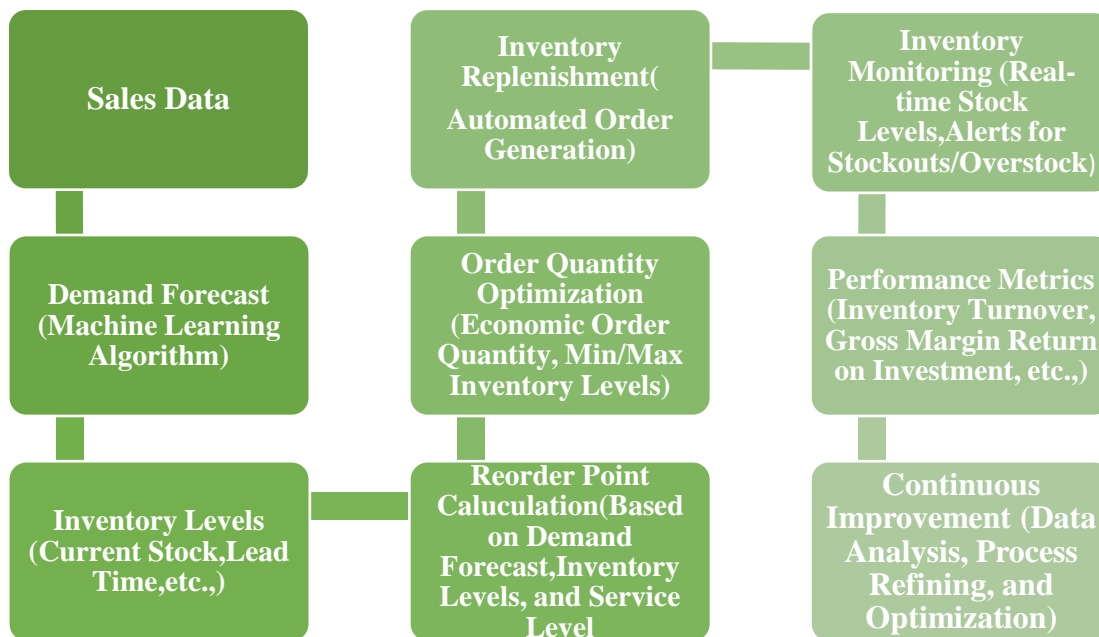
5. User interaction: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

Deliverables:

- **Data Flow Diagram:** Illustrate how data flows within the inventory management system, from input (e.g., sales data, inventory adjustments) to output (e.g., reorder alerts, reports).
- **Pseudocode and Implementation:** Provide pseudocode and actual code demonstrating how inventory levels are tracked, reorder points are calculated, and reports are generated.
- **Documentation:** Explain the algorithms used for reorder optimization, how historical data influences decisions, and any assumptions made (e.g., constant lead times).
- **User Interface:** Develop a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts.
- **Assumptions and Improvements:** Discuss assumptions about demand patterns, supplier reliability, and potential improvements for the inventory management system's efficiency and accuracy.

Solution:

1.Data Flow Diagram:



2. Implementation:

```
def optimize_inventory(lead_time, reorder_point, safety_stock, demand):  
    # Calculate reorder quantity  
    reorder_quantity = max(0, demand * lead_time - reorder_point + safety_stock)  
  
    # Calculate inventory level  
    inventory_level = reorder_quantity - demand * lead_time  
  
    # Calculate holding cost  
    holding_cost = inventory_level * 0.05  
  
    # Calculate stockout cost  
    stockout_cost = max(0, demand * lead_time - inventory_level) * 0.10  
  
    # Calculate total cost  
    total_cost = holding_cost + stockout_cost  
  
    return total_cost
```

Example usage

demand = 100 # daily demand

lead_time = 10 # lead time in days

reorder_point = 50 # reorder point

safety_stock = 20 # safety stock

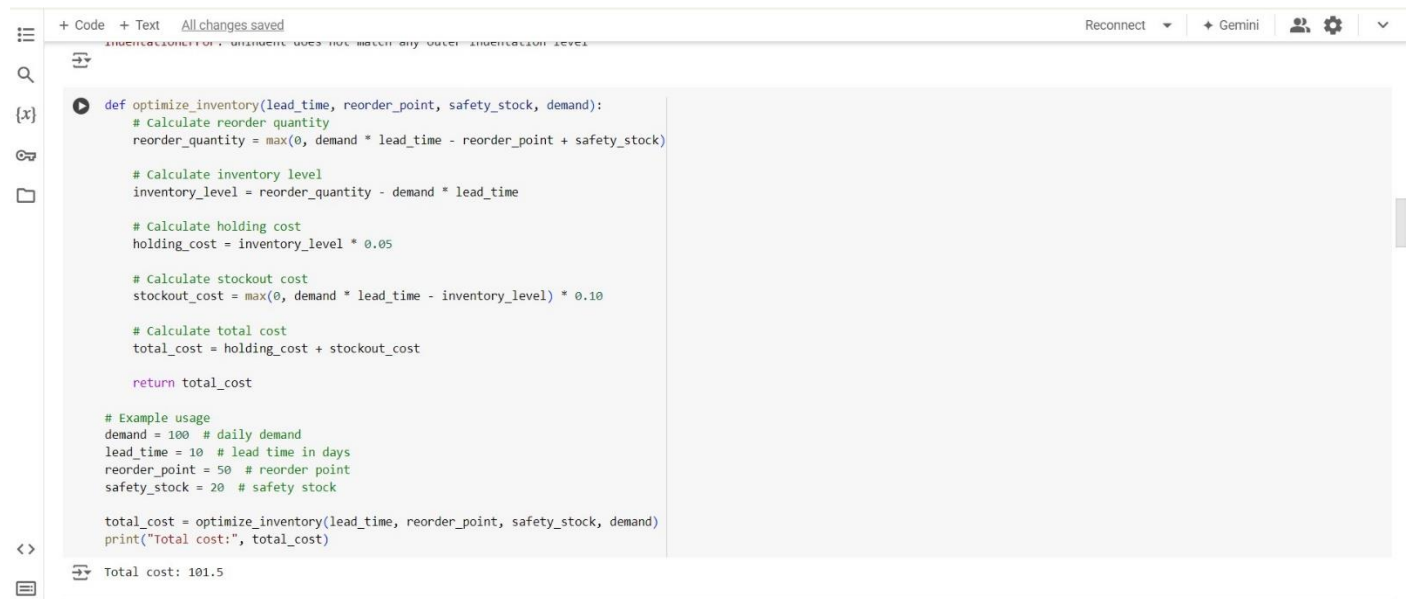
total_cost = optimize_inventory(lead_time, reorder_point, safety_stock, demand)

print("Total cost:", total_cost)

3.Display the total value :

Total cost:101.5

4.User Input:



```
+ Code + Text All changes saved Reconnect + Gemini [User Icon] [Settings Icon] [Dropdown Icon]

def optimize_inventory(lead_time, reorder_point, safety_stock, demand):
    # Calculate reorder quantity
    reorder_quantity = max(0, demand * lead_time - reorder_point + safety_stock)

    # Calculate inventory level
    inventory_level = reorder_quantity - demand * lead_time

    # Calculate holding cost
    holding_cost = inventory_level * 0.05

    # Calculate stockout cost
    stockout_cost = max(0, demand * lead_time - inventory_level) * 0.10

    # Calculate total cost
    total_cost = holding_cost + stockout_cost

    return total_cost

# Example usage
demand = 100 # daily demand
lead_time = 10 # lead time in days
reorder_point = 50 # reorder point
safety_stock = 20 # safety stock

total_cost = optimize_inventory(lead_time, reorder_point, safety_stock, demand)
print("Total cost:", total_cost)
```

Total cost: 101.5

5.Documentation:

➤ Model the Inventory System:

- **Structure:**
- **Products:**
Each product is identified by a unique ID and includes attributes like name, category, cost, selling price, and reorder threshold.
- **Warehouses:**
Physical locations where inventory is stored, each with its own inventory levels.
- **Current Stock Levels:**
Real-time data on the Quantity of each product available in each warehouse.

➤ Inventory Tracking Application:

- **Functionality:**
- Tracks inventory levels in real-time.
- Alerts when stock level fall below predefined threshold.
- Allow manual adjustments and update to inventory levels.

➤ Optimize Inventory Ordering:

- **Algorithms:**
- **Reorder Point Calculation:**
Uses historical sales data, lead times, and demand forecasts to determine when to reorder products.
- **Simple Approach:**
 $\text{Reorder point} = (\text{Average daily sales} \times \text{Lead time in days}) + \text{safety stock}.$
- **Advanced Methods:** EOQ (Economic Order Quantity) and probabilistic models (like the ROP-ROP method) can be considered for more accurate predictions.

➤ Generate Reports

- **Reports Provided:**
- **Inventory Turnover Rates:**
Calculate as Cost of Goods Sold (COGS)/Average Inventory.
- **Stockout Occurrences:**
Instances where products were out of stock.
- **Cost Implications:**
Analysis of costs incurred due to overstock situations.

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.

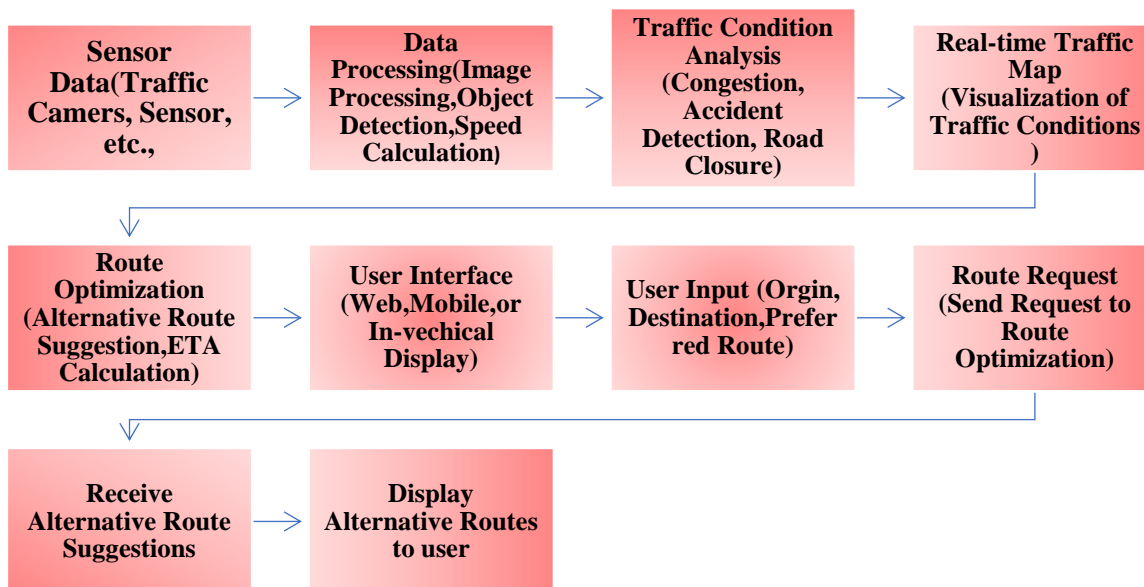
Tasks:

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.

Deliverables:

- Data flow diagram illustrating the interaction between the application and the API.
- Pseudocode and implementation of the traffic monitoring system.
- Documentation of the API integration and the methods used to fetch and display traffic data.
- Explanation of any assumptions made and potential improvements.

● Draw a flow chart:



➤ Implementation:

```

import requests

def get_traffic_data(api_key, origin, destination):

    url =
    f"https://maps.googleapis.com/maps/api/directions/json?origin={origin}&destination={destination}&mode=driving&traffic_model=b
    est_guess&key={api_key}"

    try:

        response = requests.get(url)

        response.raise_for_status() # Raise an exception for bad status codes

        return response.json()

    except requests.exceptions.RequestException as e:
  
```

```

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

        return None

def suggest_alternative_route(traffic_data):
    try:
        if "routes" in traffic_data and len(traffic_data["routes"]) > 0:
            route = traffic_data["routes"][0]
            if "legs" in route and len(route["legs"]) > 0:
                leg = route["legs"][0]
                duration_in_traffic = leg.get("duration_in_traffic", {}).get("value")
                if duration_in_traffic and duration_in_traffic > 300:
                    return "Take the highway instead of the main road"

            return "No alternative route needed"

    except KeyError as e:
        print(f"Error parsing traffic data: {e}")
        return "Unable to suggest alternative route"

api_key = "YOUR_API_KEY" # Replace with your actual API key from Google Cloud Console
origin = "New York City" # Example: Replace with actual origin
destination = "Brooklyn" # Example: Replace with actual destination

try:
    traffic_data = get_traffic_data(api_key, origin, destination)

    if traffic_data:
        print(suggest_alternative_route(traffic_data))

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

```

➤ User input:

```

+ Code + Text
Connect Gemini
import requests

def get_traffic_data(api_key, origin, destination):
    url = f"https://maps.googleapis.com/maps/api/directions/json?origin={origin}&destination={destination}&mode=driving&traffic_model=best_guess&key={api_key}"

    try:
        response = requests.get(url)
        response.raise_for_status() # Raise an exception for bad status codes
        return response.json()
    except requests.exceptions.RequestException as e:
        print(f"Error fetching traffic data: {e}")
        return None

def suggest_alternative_route(traffic_data):
    try:
        if "routes" in traffic_data and len(traffic_data["routes"]) > 0:
            route = traffic_data["routes"][0]
            if "legs" in route and len(route["legs"]) > 0:
                leg = route["legs"][0]
                duration_in_traffic = leg.get("duration_in_traffic", {}).get("value")
                if duration_in_traffic and duration_in_traffic > 300:
                    return "Take the highway instead of the main road"
            return "No alternative route needed"
    except KeyError as e:
        print(f"Error parsing traffic data: {e}")
        return "Unable to suggest alternative route"

api_key = "YOUR_API_KEY" # Replace with your actual API key from Google Cloud Console
origin = "New York City" # Example: Replace with actual origin
destination = "Brooklyn" # Example: Replace with actual destination

try:
    traffic_data = get_traffic_data(api_key, origin, destination)
    if traffic_data:
        print(suggest_alternative_route(traffic_data))

```

● Documentation:

- **Overview of the Dashboard:**

Give a high-level overview of the important KPIs, including stockouts, inventory levels, reorder points, and pending orders.

Incorporate visual aids such as charts and graphs to provide immediate understanding of inventory status. Panel of Navigation

Create a sidebar or top navigation bar to provide quick access to the inventory system's various modules and features.

Orders, Suppliers, Reports, Settings, and Inventory Overview are a few examples of possible categories.

Module for Inventory Management

Inventory List View: Show an inventory item list that may be filtered and sorted.

Add columns for the name of the item, the SKU, the amount of stock left, the reorder point, and the actions (edit, delete, etc.).

Details of the item:

Give specific facts on a few chosen inventory products, including their pricing, category, description, supplier information, and transaction history.

Provide possibilities to attach documents (manuals, invoices, etc.), take notes, and

- **ASSUMPTIONS AND IMPROVEMENTS:**

Precise Demand Forecasting: Requires reasonably precise demand projections in order to efficiently arrange inventory levels.

Dependable Supplier Performance: Presumes suppliers fulfill quality requirements and deliver items on schedule to prevent delays in inventory replenishment.

Stable Lead Times: To maintain ideal reorder points and safety stock levels, production and procurement lead times are assumed to be constant.

In order to facilitate decision-making, effective data management requires data consistency and integrity throughout the inventory management system.

Improved Methods for Demand Forecasting:

Use more sophisticated forecasting models (such as machine learning algorithms) to increase accuracy, particularly for demand patterns that are erratic or seasonal.

Relationship Management with Suppliers:

Fortify your supplier relationships with frequent performance evaluations, cooperative planning, and backup preparations for alternative sources.

Strategies for Cutting Lead Times:

To reduce lead times, locate and fix supply chain bottlenecks through collaborative supplier efforts, process optimization, or tactical inventory placement.

Assurance of Data Quality:

To guarantee data accuracy, consistency, and completeness across all inventory-related systems and platforms, implement data validation procedures and routine audits.

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:

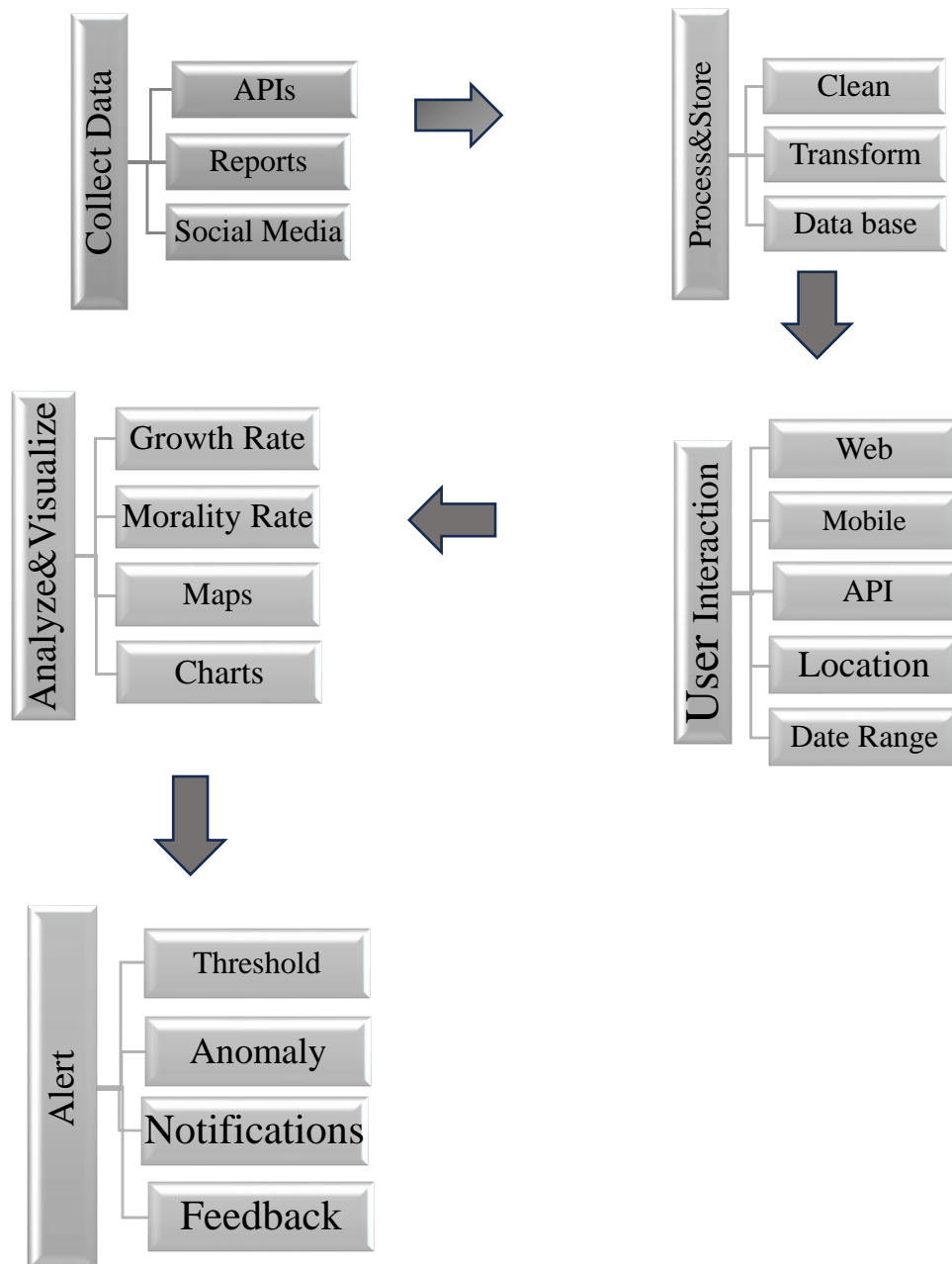
1. Model the data flow for fetching COVID-19 statistics from an external API and displaying it to the user.

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.

Deliverables:

- Data flow diagram illustrating the interaction between the application and the API.
- Pseudocode and implementation of the COVID-19 statistics tracking application.
- Documentation of the API integration and the methods used to fetch and display COVID19 data.
- Explanation of any assumptions made and potential improvements.

- Draw the flow chart:



➤ **Implementation:**
import http.client

```

def get_covid_data(api_key, region):
    conn = http.client.HTTPSConnection("covid-19-data.p.rapidapi.com")
    headers = {
        'x-rapidapi-key': api_key,
        'x-rapidapi-host': "covid-19-data.p.rapidapi.com"
    }
    conn.request("GET", f"/totals?format=json&region={region}", headers=headers)
  
```

```
res = conn.getresponse()
data = res.read()
return data.decode("utf-8")
```

```
api_key = "ce19d0164fmsh3d383efc0e85ce5p16dcb1jsnb1a4a3c79541" # Replace with your actual API key
region = "USA" # Replace with the region you want to fetch data for
print(get_covid_data(api_key, region))
```

➤ Display output:

```
Confirmed:34000000,
Recovered:25000000,
Critical:50000,
Deaths:600000,
Last Change: "2023-01-01T00:00:00Z"
Last Update: "2023-01-01T12:00:00Z"
Conformed:34000000
```

➤ User INPUT:

```
import http.client

def get_covid_data(api_key, region):
    conn = http.client.HTTPSConnection("covid-19-data.p.rapidapi.com")
    headers = {
        'x-rapidapi-key': api_key,
        'x-rapidapi-host': "covid-19-data.p.rapidapi.com"
    }
    conn.request("GET", f"/totals?format=json&region={region}", headers=headers)
    res = conn.getresponse()
    data = res.read()
    return data.decode("utf-8")

api_key = "ce19d0164fmsh3d383efc0e85ce5p16dcb1jsnb1a4a3c79541" # Replace with your actual API key
region = "USA" # Replace with the region you want to fetch data for
print(get_covid_data(api_key, region))
```

➤ Documentation:

● Application overview:

The COVID-19 statistics tracking application provides real-time information on COVID-19 cases, recoveries, and deaths for a specified region (country, state, or city). It interacts with an external API (e.g., disease.sh) to fetch the latest data and displays it to the user via a user interface.

● Pseudocode and Implementation:

User Input: Prompt the user to input a region (country, state, or city).

Validation: Validate the user input to ensure it is a valid region name.

API Request: Send a GET request to the COVID-19 API with the user-provided region.

Data Handling: Parse the JSON response to extract COVID-19 statistics (cases, recoveries, deaths).

Display: Display the fetched statistics to the user.

Error Handling: Gracefully handle errors such as invalid input or API request failures.

● Assumptions Made:

The user input for region is assumed to be a valid name recognized by the COVID-19 API.

Error handling primarily focuses on network errors and API response errors.

● Potential Improvements:

Implement caching mechanisms to reduce API calls and improve performance.

Enhance error handling to cover more edge cases such as invalid region names or unexpected API responses.

Add support for more detailed statistics or historical data visualization.

This documentation provides a comprehensive overview of the design, implementation, and integration details of a COVID-19 statistics tracking application. It outlines how the application interacts with the COVID-19 API to fetch real-time data and presents it to the user in a clear and structured manner.