Solution for Question number 1:

To develop a 3-tier rule engine application, we need to break down the task into several steps, focusing on the data structure, storage, API design, and testing. Here's a systematic approach:

Step 1: Define the Data Structure for AST

- Objective:Create a data structure to represent rules as an Abstract Syntax Tree (AST).
- Structure:Use a Node class with fields for type, left, right, and value.

"Python code"

```
Class Node:
  Def __init__(self, type, left=None, right=None, value=None):
    Self.type = type  # "operator" or "operand"
    Self.left = left  # Left child Node
    Self.right = right  # Right child Node
    Self.value = value  # Value for operand nodes
```

Step 2: Choose a Database and Define Schema

- Objective:Store rules and metadata efficiently.
- Choice:Use a relational database like PostgreSQL for structured queries.

Schema Example:

"Sql"

```
CREATE TABLE rules (
  Id SERIAL PRIMARY KEY,
  Rule_string TEXT NOT NULL,
  Ast JSONB NOT NULL
);
CREATE TABLE metadata (
  Id SERIAL PRIMARY KEY,
  Key VARCHAR(255) NOT NULL,
  Value TEXT NOT NULL
```

);

Step 3: Implement API Functions

3.1: `` `create_rule(rule_string)` ``

- Objective: Convert a rule string into an AST.
- Implementation:
-

"Python code"

```
Def create_rule(rule_string):

   # Parse the rule_string and construct the AST

   # Example: "age > 30 AND department = 'Sales'"

   # Return the root Node of the AST

   Pass
```

3.2: `` `combine_rules(rules)` ``

- Objective:Combine multiple rules into a single AST.
- Implementation:

"Python code"

```
Def combine_rules(rules):

   # Combine rules using a heuristic like most frequent operator

   # Return the root Node of the combined AST

   Pass
```

3.3: `` `evaluate_rule(ast, data)` ``

- Objective: Evaluate the AST against user data.
- Implementation:

"Python code"

```
Def evaluate_rule(ast, data):

   # Traverse the AST and evaluate conditions against data
```

    # Return True if conditions are met, False otherwise

    Pass

Step 4: Test Cases

4.1: Test ` create_rule`

- Objective:Verify AST representation for individual rules.

Example:

"Python code"

Rule1 = "((age > 30 AND department = 'Sales') OR (age < 25 AND salary > 50000 OR experience > 5))"

Ast1 = create_rule(rule1)

# Verify the structure of ast1

4.2: Test ` combine_rules`

- Objective: Ensure combined AST reflects logic.

Example:

"Python code"

Combined_ast = combine_rules([rule1, rule2])

# Verify the structure of combined_ast

4.3: Test ` evaluate_rule`

- Objective:Evaluate rules against sample data.

Example:

"Python code"

Data = {"age": 35, "department": "Sales", "salary": 60000, "experience": 3}

Result = evaluate_rule(combined_ast, data)

# Check if result is True or False

Step 5: Implement Error Handling and Validations

- Objective:Handle invalid inputs and ensure data integrity.
- Implementation:Add checks in API functions for syntax errors and missing attributes.

Final Answer

Develop a 3-tier rule engine application using an AST-based data structure, PostgreSQL for storage, and Python functions for rule creation, combination, and evaluation. Implement comprehensive test cases and error handling to ensure robustness.

Solution for Question number 2

To develop a real-time data processing system for weather monitoring using the OpenWeatherMap API, follow these steps:

Step 1: Set Up Environment

- Objective: Prepare the development environment.
- Actions:

1. Sign up for an OpenWeatherMap API key.
2. Set up a programming environment (e.g., Python with libraries like `requests`, `pandas`, `matplotlib`).
3. Configure a database for storing weather summaries (e.g., SQLite, PostgreSQL).

```
“Python code”
Import requests
Import pandas as pd
Import matplotlib.pyplot as plt
Import sqlite3
Import schedule
Import time
Import smtplib
From email.mime.text import MIMEText

# Replace with your OpenWeatherMap API key
Api_key = “YOUR_API_KEY”

# Create a database connection
Conn = sqlite3.connect(“weather_data.db”)
Cursor = conn.cursor()

# Create a table to store weather data
Cursor.execute(“””
CREATE TABLE IF NOT EXISTS weather_data (
  Id INTEGER PRIMARY KEY AUTOINCREMENT,
  City TEXT,
  Timestamp INTEGER,
  Temp REAL,
  Feels_like REAL,
  Weather_description TEXT,
  Humidity REAL,
  Pressure REAL,
  Wind_speed REAL,
  Wind_deg INTEGER
)
“””)
Conn.commit()
```

Step 2: API Integration

- Objective:Retrieve weather data from the OpenWeatherMap API.
- Actions:

1. Write a function to call the OpenWeatherMap API using the API key.
2. Parse the JSON response to extract relevant data: `main`, `temp`, `feels_like`, `dt`.
3. Convert temperatures from Kelvin to Celsius: \(\text{Celsius} = \text{Kelvin} – 273.15 \).

“Python code”

```
Def get_weather_data(city):
  url = fhttps://api.openweathermap.org/data/2.5/weather?q={city}&appid={api_key}&units=metric
  response = requests.get(url)
  data = response.json()

  if data[“cod”] != 200:
    print(“Error fetching weather data:”, data[“message”])
    return None
  weather_data = {
    “city”: city,
    “timestamp”: data[“dt”],
    “temp”: data[“main”][“temp”],
    “feels_like”: data[“main”][“feels_like”],
    “weather_description”: data[“weather”][0][“description”],
    “humidity”: data[“main”][“humidity”],
    “pressure”: data[“main”][“pressure”],
    “wind_speed”: data[“wind”][“speed”],
    “wind_deg”: data[“wind”][“deg”]
  }

  Return weather_data
```

Step 3: Continuous Data Retrieval

- Objective:Automate data collection at regular intervals.
- Actions:

1. Implement a scheduler (e.g., using `schedule` or `cron`) to call the API every 5 minutes.
2. Store the retrieved data in a temporary storage or directly in the database.

“Python code”

```
Def retrieve_data():
  City = “YourCity”  # Replace with your desired city
  Weather_data = get_weather_data(city)

  If weather_data:
    Cursor.execute(“””
    INSERT INTO weather_data (city, timestamp, temp, feels_like, weather_description, humidity,
pressure, wind_speed, wind_deg)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
    “””, (weather_data[“city”], weather_data[“timestamp”], weather_data[“temp”],
weather_data[“feels_like”], weather_data[“weather_description”], weather_data[“humidity”],
weather_data[“pressure”], weather_data[“wind_speed”], weather_data[“wind_deg”]))

    Conn.commit()


Schedule.every(5).minutes.do(retrieve_data)


While True:

  Schedule.run_pending()

  Time.sleep(1)
```

Step 4: Data Processing and Rollups

- Objective:Process daily weather data and compute aggregates.
- Actions:

1. At the end of each day, calculate:
2. Average temperature.
3. Maximum temperature.
4. Minimum temperature.
5. Dominant weather condition (most frequent condition).
6. Store these aggregates in the database.

“Python code”

```
Def process_daily_data():

  Today = datetime.date.today()

  Start_of_day = datetime.datetime.combine(today, datetime.time.min)

  End_of_day = datetime.datetime.combine(today, datetime.time.max)


  Cursor.execute(“””

  SELECT temp, weather_description

  FROM weather_data

  WHERE timestamp BETWEEN ? AND ?

  “””, (start_of_day.timestamp(), end_of_day.timestamp()))

  Data = cursor.fetchall()
  # Calculate daily aggregates

  Average_temp = sum(row[0] for row in data) / len(data)

  Max_temp = max(row[0] for row in data)

  Min_temp = min(row[0] for row in data)

  Dominant_weather = max(set(row[1] for row in data), key=lambda x: data.count(x))


  # Store daily aggregates

  Cursor.execute("""

  INSERT INTO daily_aggregates (date, average_temp, max_temp, min_temp, dominant_weather)

  VALUES (?, ?, ?, ?, ?)

  """, (today.strftime("%Y-%m-%d"), average_temp, max_temp, min_temp, dominant_weather))

  Conn.commit()


# Schedule daily processing

Schedule.every().day.at("00:00").do(process_daily_data)
```

Step 5: Alerting Mechanism

- Objective: Monitor weather conditions against user-defined thresholds.
- Actions:

1. Allow users to set thresholds (e.g., temperature > 35°C).
2. Continuously compare current data with thresholds.
3. Trigger alerts (console message or email) if thresholds are breached.
   "Python code"

```
Def send_alert(message):
  Sender_email = your_email@example.com
  Receiver_email = recipient_email@example.com
  Password = "your_email_password"

  Message = MIMEText(message)
  Message["Subject"] = "Weather Alert"
  Message["From"] = sender_email
  Message["To"] = receiver_email

  With smtplib.SMTP("smtp.gmail.com", 587) as server:
    Server.starttls()
    Server.login(sender_email, password)
```

Server.sendmail(sender_email, receiver_email, message.as_string())

Step 6: Visualization

- Objective:Create visual representations of weather data.
- Actions:

1. Use visualization libraries (e.g., ` matplotlib` , ` seaborn`) to plot:
2. Daily summaries.
3. -Historical trends.
4. Alerts.
5. Display these visualizations on a dashboard or save them as images.

“Python code”

```
# ... (visualization code using matplotlib or seaborn)
```

Step 7: Testing

- Objective: Ensure the system functions correctly.
- Actions:

1. Verify API connectivity and data retrieval.
2. Test temperature conversion accuracy.
3. Simulate multiple days of data to check daily summary calculations.
4. Test alert functionality by simulating threshold breaches.

“Python code”

```
# ... (testing code)
```

Final Answer

Develop a real-time weather monitoring system by integrating with the OpenWeatherMap API, processing data for daily summaries, implementing alert mechanisms, and visualizing results. Ensure the system is robust through comprehensive testing.