Software Engineer Intern

Name – **Shubham Mishra**

Email id – shubhammishra6m@gmail.com

Github link: shubham6m/Zeotap-Assignment

Application 1 : Rule Engine with AST

- This application includes the creation of rules using Abstract Syntax Tree (AST), combination of multiple rules, and rule evaluation.

- The project is designed to work with a database (e.g., **SQLite** or **PostgreSQL**) to store the rules and metadata.

Step-by-Step Implementation of the Rule Engine with AST :

1. Data Structure for AST:

We will define a `Node` class to represent the tree structure for the rule engine. Each `Node` can be an operand (e.g., a condition like `age > 30`) or an operator (e.g., `AND`, `OR`).

```
class Node:
    def __init__(self, node_type, left=None, right=None,
value=None):
        self.type = node_type   "operator" for AND/OR or "operand"
for conditions
        self.left = left   Left child node (for operators)
        self.right = right   Right child node (for operators)
        self.value = value   Value for operand nodes (condition as
string)

    def __repr__(self):
        if self.type == "operand":
            return f"Operand({self.value})"
        elif self.type == "operator":
            return f"Operator({self.left} {self.value}
{self.right})"

     Evaluate the node based on the data provided
    def evaluate(self, data):
        if self.type == "operand":
            return eval(self.value, {}, data)
        elif self.type == "operator":
            if self.value == "AND":
                return self.left.evaluate(data) and
self.right.evaluate(data)
            elif self.value == "OR":
                return self.left.evaluate(data) or
self.right.evaluate(data)
        return False
```

2. Database Schema for Rule Storage :

The system will store the rules and metadata in a simple table. The rule strings are stored in the database, and when needed, they are parsed into AST structures.

Schema for PostgreSQL (or SQLite):

```sql
CREATE TABLE rules (
    id SERIAL PRIMARY KEY,
    rule_string TEXT NOT NULL,
    description TEXT
);
```

3. Rule Creation API:

The `create_rule` function parses a rule string into an AST by using operators and operands.

```
import re

def create_rule(rule_string):
     Remove extra spaces and tokenize the rule string based on
parentheses and operators
    tokens = re.findall(r'\w+|[><=()]|\band\b|\bor\b',
rule_string)
    return parse_expression(tokens)

 Helper function to parse a tokenized expression into AST
def parse_expression(tokens):
    stack = []
    while tokens:
        token = tokens.pop(0)
        if token == '(':
            stack.append(token)
        elif token == ')':
            right = stack.pop()
            operator = stack.pop()
            left = stack.pop()
            stack.pop()   pop '('
            stack.append(Node("operator", left, right, operator))
        elif token.lower() in ['and', 'or']:
            operator = token.upper()
            stack.append(operator)
        else:
            condition = token
            if tokens and tokens[0] in ['>', '<', '=']:
                condition += tokens.pop(0) +
tokens.pop(0)   Construct conditions like age > 30
            stack.append(Node("operand", value=condition))
    return stack[0] if stack else None
```

4. Combining Multiple Rules :

The `combine_rules` function allows combining multiple rule strings into a single AST using an operator like `AND` or `OR`.

```
def combine_rules(rules, operator='AND'):
    ast_list = [create_rule(rule) for rule in rules]
    while len(ast_list) > 1:
        left = ast_list.pop(0)
        right = ast_list.pop(0)
        combined_ast = Node("operator", left, right, operator)
        ast_list.append(combined_ast)
    return ast_list[0]
```

5. Rule Evaluation:

The `evaluate_rule` function takes the AST and a dictionary of user attributes to evaluate if the rule conditions are met.

```
def evaluate_rule(ast, data):
    return ast.evaluate(data)
```

6. Sample Rules:

We can now define the sample rules and test the rule engine with real data.

```
 Sample rules
rule1 = "((age > 30 AND department = 'Sales') OR (age < 25 AND
department = 'Marketing')) AND (salary > 50000 OR experience > 5)"
rule2 = "((age > 30 AND department = 'Marketing')) AND (salary >
20000 OR experience > 5)"

 Create individual ASTs for the rules
ast1 = create_rule(rule1)
ast2 = create_rule(rule2)

 Combine the rules using AND operator
combined_rule_ast = combine_rules([rule1, rule2], operator="AND")

 Example user data
user_data = {"age": 35, "department": "Sales", "salary": 60000,
"experience": 3}

 Evaluate the rule against user data
result = evaluate_rule(combined_rule_ast, user_data)
print("User is eligible:", result)
```

7. Unit Tests:

Unit tests to verify that the system behaves as expected for different rule scenarios.

```
def test_create_rule():
    rule_string = "age > 30 AND salary > 50000"
    ast = create_rule(rule_string)
    assert ast.type == "operator"
    assert ast.left.value == "age > 30"
    assert ast.right.value == "salary > 50000"

def test_combine_rules():
    rule1 = "age > 30 AND salary > 50000"
    rule2 = "experience > 5 OR department = 'Sales'"
    combined_ast = combine_rules([rule1, rule2], "AND")
    assert combined_ast.type == "operator"

def test_evaluate_rule():
    rule_string = "age > 30 AND salary > 50000"
    ast = create_rule(rule_string)
    user_data = {"age": 35, "salary": 60000}
    assert evaluate_rule(ast, user_data) == True

    user_data = {"age": 25, "salary": 40000}
    assert evaluate_rule(ast, user_data) == False

 Run tests
test_create_rule()
test_combine_rules()
test_evaluate_rule()
print("All tests passed.")
```

8. Bonus Features:

- **Error Handling**: We can add basic error handling for invalid rule strings and data formats.

- **Modification of Rules:** Modify existing rules using additional functionalities such as changing operators or operand values.

```
def modify_rule(ast, new_value):
    if ast.type == "operand":
        ast.value = new_value
    return ast
```

9. Running the code in System’s CMD prompt:

1. Install required libraries:

pip install psycopg2 re

2. Start the rule engine application:

Python rule_engine.py

Conclusion:

This Rule Engine with AST supports rule creation, combination, and evaluation with a simple API and backend storage. The system can be easily extended for additional features, such as user-defined functions, and integrated with a UI for rule management.

Combined all code that is runnable code in my system :

File name : **rule_engine.py**

```
import re
import sqlite3

 Define the AST node structure
class Node:
    def __init__(self, node_type, left=None, right=None,
value=None):
        self.type = node_type   "operator" for AND/OR or "operand"
for conditions
        self.left = left   Left child node (for operators)
        self.right = right   Right child node (for operators)
        self.value = value   Value for operand nodes (condition as
string)

    def __repr__(self):
        if self.type == "operand":
            return f"Operand({self.value})"
        elif self.type == "operator":
            return f"Operator({self.left} {self.value}
{self.right})"

     Evaluate the node based on the data provided
    def evaluate(self, data):
        if self.type == "operand":
            return eval(self.value, {}, data)
        elif self.type == "operator":
            if self.value == "AND":
                return self.left.evaluate(data) and
self.right.evaluate(data)
            elif self.value == "OR":
                return self.left.evaluate(data) or
self.right.evaluate(data)
        return False


 Function to create rule (AST) from a rule string
```

```
def create_rule(rule_string):
     Remove extra spaces and tokenize the rule string based on
parentheses and operators
    tokens = re.findall(r'\w+|[><=()]|\band\b|\bor\b', rule_string)
    return parse_expression(tokens)

 Helper function to parse a tokenized expression into AST
def parse_expression(tokens):
    stack = []
    while tokens:
        token = tokens.pop(0)
        if token == '(':
            stack.append(token)
        elif token == ')':
            right = stack.pop()
            operator = stack.pop()
            left = stack.pop()
            stack.pop()   pop '('
            stack.append(Node("operator", left, right, operator))
        elif token.lower() in ['and', 'or']:
            operator = token.upper()
            stack.append(operator)
        else:
            condition = token
            if tokens and tokens[0] in ['>', '<', '=']:
                condition += tokens.pop(0) +
tokens.pop(0)   Construct conditions like age > 30
            stack.append(Node("operand", value=condition))
    return stack[0] if stack else None


 Function to combine multiple rules into a single AST
def combine_rules(rules, operator='AND'):
    ast_list = [create_rule(rule) for rule in rules]
    while len(ast_list) > 1:
        left = ast_list.pop(0)
        right = ast_list.pop(0)
        combined_ast = Node("operator", left, right, operator)
        ast_list.append(combined_ast)
    return ast_list[0]
```

```
 Function to evaluate a rule (AST) against user data
def evaluate_rule(ast, data):
    return ast.evaluate(data)


 Database Functions: Connecting to SQLite to store rules
def init_db():
    conn = sqlite3.connect('rules.db')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS rules (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            rule_string TEXT NOT NULL,
            description TEXT
        )
    ''')
    conn.commit()
    conn.close()

 Function to save rule to the database
def save_rule(rule_string, description):
    conn = sqlite3.connect('rules.db')
    cursor = conn.cursor()
    cursor.execute("INSERT INTO rules (rule_string, description)
VALUES (?, ?)", (rule_string, description))
    conn.commit()
    conn.close()

 Function to retrieve rules from the database
def get_rules():
    conn = sqlite3.connect('rules.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM rules")
    rules = cursor.fetchall()
    conn.close()
    return rules

 Modify existing rule (Bonus feature)
def modify_rule(ast, new_value):
    if ast.type == "operand":
        ast.value = new_value
    return ast
```

```
 Test Cases
def test_create_rule():
    rule_string = "age > 30 AND salary > 50000"
    ast = create_rule(rule_string)
    assert ast.type == "operator"
    assert ast.left.value == "age > 30"
    assert ast.right.value == "salary > 50000"

def test_combine_rules():
    rule1 = "age > 30 AND salary > 50000"
    rule2 = "experience > 5 OR department = 'Sales'"
    combined_ast = combine_rules([rule1, rule2], "AND")
    assert combined_ast.type == "operator"

def test_evaluate_rule():
    rule_string = "age > 30 AND salary > 50000"
    ast = create_rule(rule_string)
    user_data = {"age": 35, "salary": 60000}
    assert evaluate_rule(ast, user_data) == True

    user_data = {"age": 25, "salary": 40000}
    assert evaluate_rule(ast, user_data) == False

 Running Tests
if __name__ == '__main__':
    init_db()

     Test the system by creating, saving, and evaluating rules
    print("Running unit tests...")
    test_create_rule()
    test_combine_rules()
    test_evaluate_rule()
    print("All tests passed.")

     Sample rules
    rule1 = "((age > 30 AND department = 'Sales') OR (age < 25 AND
department = 'Marketing')) AND (salary > 50000 OR experience > 5)"
    rule2 = "((age > 30 AND department = 'Marketing')) AND (salary >
20000 OR experience > 5)"

     Create individual ASTs for the rules
```

```
    ast1 = create_rule(rule1)
    ast2 = create_rule(rule2)

     Combine the rules using AND operator
    combined_rule_ast = combine_rules([rule1, rule2],
operator="AND")

     Save rule to the database
    save_rule(rule1, "Rule 1 for Sales and Marketing")
    save_rule(rule2, "Rule 2 for Marketing")

     Retrieve and print saved rules
    rules = get_rules()
    print("Saved rules in the database:")
    for rule in rules:
        print(rule)

     Example user data
    user_data = {"age": 35, "department": "Sales", "salary": 60000,
"experience": 3}

     Evaluate the rule against user data
    result = evaluate_rule(combined_rule_ast, user_data)
    print("User is eligible:", result)

     Modify a rule (optional)
    modified_ast = modify_rule(ast1, "age > 25")
    result = evaluate_rule(modified_ast, user_data)
    print("Modified rule eligibility:", result)
```

Application 2 : Real-Time Data Processing System for Weather Monitoring with Rollups and Aggregates

To create **a Real-Time Data Processing System** for Weather Monitoring as specified, we will break the project down into smaller components, focusing on the core requirements such as weather data retrieval, real-time data processing, rollup calculations, alert thresholds, and visualization. Here's an overview of the solution design and code implementation.

High-Level Design:

1. **Weather Data Retrieval:** I Used Open Weather Map API to retrieve real-time weather data for six cities in India (Delhi, Mumbai, Chennai, Bangalore, Kolkata, Hyderabad) as the rule for creating this application.

2. **Data Processing**: Process the retrieved weather data, converting temperatures from Kelvin to Celsius or Fahrenheit (based on user preference).

3. Aggregations:

- Calculate daily summaries including:
  - Average temperature
  - Maximum temperature
  - Minimum temperature
  - Dominant weather condition

4. Alerting System: Set up user-configurable thresholds for temperature or weather conditions. If thresholds are exceeded for a certain period, trigger alerts.

5. Storage and Visualization: Store data in a database (e.g., PostgreSQL or MongoDB) and visualize weather trends and alerts.

6. Configurable Intervals: The system should poll the API every 5 minutes (or user-defined interval).

Tech Stack used :

- Python for data processing and logic
- OpenWeatherMap API as the weather data source
- PostgreSQL or MongoDB for data storage (can use Docker for setup)
- Matplotlib or Plotly for visualizations
- Flask to serve a simple web interface for viewing weather summaries
- SMTP for email alerts **(bonus point)**

Steps and Code:

1. API Key Setup:

```
API_KEY = '0df472a1c4ab65341b538215d9efc4c7'
CITIES = ['Delhi', 'Mumbai', 'Chennai', 'Bangalore', 'Kolkata',
'Hyderabad']
```

2. Weather Data Retrieval:

This function retrieves real-time weather data using the **Open Weather Map API**.

```
import requests
import time
import json

def get_weather_data(city, api_key):
    base_url = "http://api.openweathermap.org/data/2.5/weather"
    params = {
        'q': city,
        'appid': api_key
    }
    response = requests.get(base_url, params=params)
    if response.status_code == 200:
        return response.json()
    else:
        return None
```

3. Temperature Conversion:

Convert temperature values from Kelvin to Celsius or Fahrenheit.

```
def kelvin_to_celsius(kelvin):
    return kelvin - 273.15

def kelvin_to_fahrenheit(kelvin):
    return (kelvin - 273.15) * 9/5 + 32

def convert_temperature(kelvin, unit="C"):
    if unit == "C":
        return kelvin_to_celsius(kelvin)
    elif unit == "F":
        return kelvin_to_fahrenheit(kelvin)
    else:
        return kelvin  # return Kelvin by default
```

4. Processing and Storing Weather Data:

Here, we define a function to process the weather data for each city, store the relevant parameters, and perform necessary aggregations. We'll also store this data in a PostgreSQL database :

```
import psycopg2
from datetime import datetime

# Connect to PostgreSQL
conn = psycopg2.connect(
    host="localhost",
    database="weather_db",
    user="my_userid",
    password="password"
)
cur = conn.cursor()

# Initialize a table for weather data
def create_table():
    cur.execute("""
        CREATE TABLE IF NOT EXISTS weather_data (
            id SERIAL PRIMARY KEY,
            city VARCHAR(50),
            temperature FLOAT,
            feels_like FLOAT,
            main VARCHAR(50),
            dt TIMESTAMP
        )
    """)
    conn.commit()

# Function to insert weather data into the database
def insert_weather_data(city, temperature, feels_like, main, dt):
    cur.execute("""
        INSERT INTO weather_data (city, temperature, feels_like,
main, dt)
        VALUES (%s, %s, %s, %s, %s)
    """, (city, temperature, feels_like, main, dt))
    conn.commit()

# Processing data for a city
def process_weather_data(data, unit="C"):
    city = data['name']
    temp = convert_temperature(data['main']['temp'], unit)
    feels_like = convert_temperature(data['main']['feels_like'],
unit)
    main_weather = data['weather'][0]['main']
```

```
    timestamp = datetime.fromtimestamp(data['dt'])

    # Insert processed data into DB
    insert_weather_data(city, temp, feels_like, main_weather,
timestamp)
```

5. Daily Rollups and Aggregates:

Calculate daily weather summaries for each city (average temperature, max, min, dominant weather condition). Here I summarize the data for a day and store it.

```
def calculate_daily_summary():
    # Group by city and date
    cur.execute("""
        SELECT city,
               DATE(dt) as day,
               AVG(temperature),
               MAX(temperature),
               MIN(temperature),
               mode() WITHIN GROUP (ORDER BY main) as
dominant_weather
        FROM weather_data
        GROUP BY city, day
    """)
    summaries = cur.fetchall()
    return summaries

# Example of how to fetch summaries
daily_summaries = calculate_daily_summary()
for summary in daily_summaries:
    print(summary)
```

6. Alerting System:

Define user-configurable thresholds and alert when certain conditions are met (e.g., temp > 35°C for 2 consecutive updates).

```
ALERT_THRESHOLD_TEMP = 35.0  # User-configurable threshold for
temperature

def check_alerts(city, current_temp):
    cur.execute("""
        SELECT temperature
        FROM weather_data
        WHERE city=%s
        ORDER BY dt DESC
        LIMIT 2
    """, (city,))
    temps = cur.fetchall()

    if len(temps) == 2 and all(temp[0] > ALERT_THRESHOLD_TEMP for
temp in temps):
        print(f"ALERT: {city} temperature has exceeded
{ALERT_THRESHOLD_TEMP}°C for 2 consecutive updates.")
```

7. Visualization:

Use libraries like Matplotlib or Plotly to display weather trends and alerts.

```
import matplotlib.pyplot as plt

def plot_temperature_trend(city):
    cur.execute("""
        SELECT dt, temperature
        FROM weather_data
        WHERE city=%s
        ORDER BY dt
    """, (city,))
    data = cur.fetchall()
    timestamps, temps = zip(*data)

    plt.plot(timestamps, temps, label=f"Temperature in {city}")
    plt.xlabel('Time')
    plt.ylabel('Temperature (°C)')
    plt.title(f"Temperature Trend for {city}")
    plt.legend()
    plt.show()

# Plotting temperature trend for Delhi
plot_temperature_trend('Delhi')
```

Running the System:

1. Clone the code repository
2. Build and run the services at cmd:

   docker-compose up --build

3. Set up the database and start collecting weather data from the **Open Weather Map** API.

```python
import requests
import time
from datetime import datetime
import psycopg2
import matplotlib.pyplot as plt

# API Key and City List
API_KEY = '0df472a1c4ab65341b538215d9efc4c7'
CITIES = ['Delhi', 'Mumbai', 'Chennai', 'Bangalore', 'Kolkata',
'Hyderabad']

# PostgreSQL Connection
conn = psycopg2.connect(
    host="db",  # The hostname matches the service name in docker-
compose.yml
    database="weather_db",
    user="my_user_id",
    password="password"
)
cur = conn.cursor()

# Create table for weather data
def create_table():
    cur.execute("""
        CREATE TABLE IF NOT EXISTS weather_data (
            id SERIAL PRIMARY KEY,
            city VARCHAR(50),
            temperature FLOAT,
            feels_like FLOAT,
            main VARCHAR(50),
            dt TIMESTAMP
        )
    """)
    conn.commit()

# Fetch weather data for a specific city
def get_weather_data(city, api_key):
    base_url = "http://api.openweathermap.org/data/2.5/weather"
    params = {'q': city, 'appid': api_key}
    response = requests.get(base_url, params=params)
    if response.status_code == 200:
        return response.json()
```

```
    else:
        return None

# Temperature conversion from Kelvin to Celsius/Fahrenheit
def kelvin_to_celsius(kelvin):
    return kelvin - 273.15

def kelvin_to_fahrenheit(kelvin):
    return (kelvin - 273.15) * 9/5 + 32

def convert_temperature(kelvin, unit="C"):
    if unit == "C":
        return kelvin_to_celsius(kelvin)
    elif unit == "F":
        return kelvin_to_fahrenheit(kelvin)
    else:
        return kelvin  # return Kelvin by default

# Insert weather data into PostgreSQL
def insert_weather_data(city, temperature, feels_like, main, dt):
    cur.execute("""
        INSERT INTO weather_data (city, temperature, feels_like,
main, dt)
        VALUES (%s, %s, %s, %s, %s)
    """, (city, temperature, feels_like, main, dt))
    conn.commit()

# Process the weather data and store it in the database
def process_weather_data(data, unit="C"):
    city = data['name']
    temp = convert_temperature(data['main']['temp'], unit)
    feels_like = convert_temperature(data['main']['feels_like'],
unit)
    main_weather = data['weather'][0]['main']
    timestamp = datetime.fromtimestamp(data['dt'])

    insert_weather_data(city, temp, feels_like, main_weather,
timestamp)

# Calculate daily summaries (average, max, min, dominant weather)
def calculate_daily_summary():
    cur.execute("""
```

```
        SELECT city,
               DATE(dt) as day,
               AVG(temperature),
               MAX(temperature),
               MIN(temperature),
               mode() WITHIN GROUP (ORDER BY main) as
dominant_weather
        FROM weather_data
        GROUP BY city, day
    """)
    summaries = cur.fetchall()
    return summaries

# Alert if temperature exceeds threshold for two consecutive updates
ALERT_THRESHOLD_TEMP = 35.0

def check_alerts(city, current_temp):
    cur.execute("""
        SELECT temperature
        FROM weather_data
        WHERE city=%s
        ORDER BY dt DESC
        LIMIT 2
    """, (city,))
    temps = cur.fetchall()

    if len(temps) == 2 and all(temp[0] > ALERT_THRESHOLD_TEMP for
temp in temps):
        print(f"ALERT: {city} temperature has exceeded
{ALERT_THRESHOLD_TEMP}°C for 2 consecutive updates.")

# Plot temperature trend
def plot_temperature_trend(city):
    cur.execute("""
        SELECT dt, temperature
        FROM weather_data
        WHERE city=%s
        ORDER BY dt
    """, (city,))
    data = cur.fetchall()
    timestamps, temps = zip(*data)
```

```
    plt.plot(timestamps, temps, label=f"Temperature in {city}")
    plt.xlabel('Time')
    plt.ylabel('Temperature (°C)')
    plt.title(f"Temperature Trend for {city}")
    plt.legend()
    plt.show()

# Main function to run the real-time monitoring system
def run_weather_monitoring():
    create_table()

    # Loop to fetch and process weather data every 5 minutes
    while True:
        for city in CITIES:
            weather_data = get_weather_data(city, API_KEY)
            if weather_data:
                process_weather_data(weather_data)
                check_alerts(city, weather_data['main']['temp'])

        # Wait for 5 minutes before the next update
        time.sleep(300)

        # Display daily summaries (Optional for testing)
        summaries = calculate_daily_summary()
        for summary in summaries:
            print(summary)

# Run the monitoring system
if __name__ == "__main__":
    run_weather_monitoring()
```