Application 1 : Rule Engine with AST

Data Structure (AST Node)

```
class Node:
    def __init__(self, node_type, left=None, right=None,
value=None):
        self.node_type = node_type  # "operator" or "operand"
        self.left = left  # Left child (for operators)
        self.right = right  # Right child (for operators)
        self.value = value  # Value for operands (e.g., age >
30)
```

AST Construction (Create Rule)

```
import re

def create_rule(rule_string):
    tokens = tokenize(rule_string)
    return build_ast(tokens)

def tokenize(rule_string):
    # Tokenizing the rule string for operators and operands
    return re.findall(r'\w+|[()><=]|AND|OR', rule_string)

def build_ast(tokens):
    stack = []
    for token in tokens:
        if token == '(':
            stack.append(token)
        elif token == ')':
            nodes = []
            while stack and stack[-1] != '(':
                nodes.append(stack.pop())
            stack.pop()  # pop '('
            stack.append(combine_nodes(nodes[::-1]))
        else:
            stack.append(token)
    return combine_nodes(stack)

def combine_nodes(nodes):
    while len(nodes) > 1:
        if "AND" in nodes:
            idx = nodes.index("AND")
            left, right = nodes[idx-1], nodes[idx+1]
```

```
            nodes = nodes[:idx-1] + [Node("operator", left,
right, "AND")] + nodes[idx+2:]
        elif "OR" in nodes:
            idx = nodes.index("OR")
            left, right = nodes[idx-1], nodes[idx+1]
            nodes = nodes[:idx-1] + [Node("operator", left,
right, "OR")] + nodes[idx+2:]
    return nodes[0] if nodes else None
```

Combine Rules

```
def combine_rules(rules):
    asts = [create_rule(rule) for rule in rules]
    root = asts[0]
    for ast in asts[1:]:
        root = Node("operator", root, ast, "AND")  # Combine
with AND operator
    return root
```

Database (Example Schema)

For storing rules, a relational database like **PostgreSQL** or a NoSQL database like **MongoDB** can be used. Here's a sample schema for storing rules in **PostgreSQL**:

```
sql

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

Example Usage

```
python

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 ASTs for each rule
ast1 = create_rule(rule1)
ast2 = create_rule(rule2)

# Combine both rules
combined_ast = combine_rules([rule1, rule2])
```

Testing

```
python

def evaluate_rule(ast, user_data):
    if ast.node_type == "operator":
        if ast.value == "AND":
            return evaluate_rule(ast.left, user_data) and
evaluate_rule(ast.right, user_data)
        elif ast.value == "OR":
            return evaluate_rule(ast.left, user_data) or
evaluate_rule(ast.right, user_data)
    elif ast.node_type == "operand":
        return eval_condition(ast.value, user_data)

def eval_condition(condition, user_data):
    # Implement the evaluation of conditions (e.g., age > 30)
    key, operator, value = condition.split()
    if operator == ">":
        return user_data[key] > int(value)
    elif operator == "=":
        return user_data[key] == value.strip("'")
    # Add other operators like <, >=, etc.

# Test cases
user_data = {"age": 32, "department": "Sales", "salary":
60000, "experience": 6}
print(evaluate_rule(combined_ast, user_data))  # Should
return True/False based on conditions
```

Conclusion:

- **create_rule**: Converts a rule string into an AST.
- **combine_rules**: Combines multiple rules into a single AST.
- **evaluate_rule**: Evaluates user data against the rule to check eligibility.

This implementation is simplified for the given task, and can be further expanded with more robust features like error handling and optimization for larger rule sets.

Application 2. Real-Time Weather Monitoring System

Prerequisites:

1. Install required libraries:

```
pip install requests sqlite3 matplotlib
```

Python Code:

```
import requests
import sqlite3
import time

# OpenWeatherMap API
API_KEY = 'your_openweathermap_api_key'
BASE_URL = 'http://api.openweathermap.org/data/2.5/weather'

# SQLite setup
conn = sqlite3.connect('weather.db')
cursor = conn.cursor()

# Create table to store daily summaries
cursor.execute('''
CREATE TABLE IF NOT EXISTS daily_summary (
    city TEXT,
    date TEXT,
    avg_temp REAL,
    max_temp REAL,
    min_temp REAL,
    dominant_condition TEXT
)''')

# Function to get real-time weather data from OpenWeatherMap
def get_weather_data(city):
    params = {
        'q': city,
```

```
        'appid': API_KEY,
        'units': 'metric'  # Convert to Celsius
    }
    response = requests.get(BASE_URL, params=params)
    return response.json()

# Process weather data and store daily summary
def process_weather_data(city):
    data = get_weather_data(city)
    main_weather = data['weather'][0]['main']
    temp = data['main']['temp']
    temp_min = data['main']['temp_min']
    temp_max = data['main']['temp_max']

    date = time.strftime('%Y-%m-%d' ,
time.localtime(data['dt']))

    cursor.execute('''
    INSERT INTO daily_summary (city, date, avg_temp,
max_temp, min_temp, dominant_condition)
    VALUES (?, ?, ?, ?, ?, ?)
    ''', (city, date, temp, temp_max, temp_min,
main_weather))

    conn.commit()

# Rollups for daily summaries
def calculate_daily_summary(city):
    cursor.execute('''
    SELECT avg(avg_temp), max(max_temp), min(min_temp),
dominant_condition
    FROM daily_summary WHERE city = ? GROUP BY date ORDER BY
date DESC LIMIT 1
    ''', (city,))
    summary = cursor.fetchone()
    return summary

# Check for temperature threshold breaches
def check_threshold(city, threshold_temp):
    data = get_weather_data(city)
    temp = data['main']['temp']
    if temp > threshold_temp:
        print(f"Alert: Temperature in {city} exceeded
{threshold_temp}°C! Current: {temp}°C")
```

```
# Main function to fetch data every 5 minutes
def monitor_weather(city_list, threshold_temp):
    while True:
        for city in city_list:
            process_weather_data(city)
            check_threshold(city, threshold_temp)
        time.sleep(300)  # 5 minutes

# Sample usage
if __name__ == "__main__":
    city_list = ['Delhi', 'Mumbai', 'Bangalore', 'Kolkata',
'Hyderabad', 'Chennai']
    monitor_weather(city_list, 35)
```

Key Features:

1. **Weather Data**: Fetches real-time weather data from OpenWeatherMap using the API.
2. **SQLite Database**: Stores daily weather summaries including average, max, min temperatures, and dominant weather conditions.
3. **Threshold Alerts**: Checks if the current temperature exceeds a user-defined threshold and triggers an alert.
4. **Polling**: The script polls the OpenWeatherMap API every 5 minutes.

Testing:

1. **Temperature Conversion**: Temperature is already converted to Celsius using `units='metric'`.
2. **Daily Rollups**: Each weather update is stored, and daily summaries are calculated.
3. **Alerts**: Alerts trigger when temperature thresholds are breached.

Improvements:

1. Implement an email alert system using SMTP for email notifications.
2. Use a more advanced database like PostgreSQL for large-scale deployment.
3. Add data visualization with libraries like `matplotlib` to display daily trends.