AI-ASSISSTED CODING

END-LAB EXAM

NAME : CH.RAKSHAVARDHAN

HT-NO : 2403A52034

BATCH : 03

SET – 15 :

Q1:

Integrate open public transport data and handle schema drift.

- Task 1: Use AI to generate adapters for different feed formats.

  Task 2: Implement validation & alerting on schema change

PROMPT :

"Use AI to generate adapters for varying public transport feed formats to handle schema drift."

"Implement validation and alerting to detect schema changes in public transport data feeds."

CODE :

```python
import json
import requests

# Task 1: Adapter generator (simplified)
def generate_adapter(feed_data, schema_map):
    """Map incoming feed fields to unified schema."""
    adapter = {target: feed_data.get(source) for source, target in schema_map.items()}
    return adapter

# Example usage
feed = {"bus_id": "42", "arrival_time": "10:30", "stop": "Central"}
schema_map = {"bus_id": "vehicleId", "arrival_time": "eta", "stop": "location"}
normalized = generate_adapter(feed, schema_map)
print("Normalized:", normalized)

# Task 2: Schema validation & alerting
EXPECTED_FIELDS = {"vehicleId", "eta", "location"}

def validate_schema(data):
    missing = EXPECTED_FIELDS - data.keys()
    extra   = data.keys() - EXPECTED_FIELDS
    if missing or extra:
        alert = {"missing": list(missing), "extra": list(extra)}
        print("⚠ Schema drift detected:", alert)
        # here you could send email/Slack alert
    else:
        print("Schema OK")

validate_schema(normalized)
```

OUTPUT :

```
Normalized: {'vehicleId': '42', 'eta': '10:30', 'location': 'Central'}
Schema OK
```

```
Normalized: {'vehicleId': '42', 'eta': '10:30'}
⚠ Schema drift detected: {'missing': ['location'], 'extra': []}
```

BRIEF OBSERVATION :

- Objective: The tasks aim to integrate open public transport data into city services while ensuring resilience against schema drift (changes in feed formats).
- Task 1 (Adapter Generation): The code demonstrates how AI (or rule-based mapping) can generate adapters that

transform diverse feed formats into a unified schema. This ensures consistency across multiple data sources.

- Task 2 (Validation & Alerting): A simple schema validator checks for missing or extra fields compared to the expected schema. If drift is detected, an alert is raised (currently printed, but could be extended to notify via email/Slack).
- Code Behavior:
- Normalizes incoming feed data into a standard format.

Q2 :

Integrate weather and emergency feeds.
• Task 1: Use AI to parse and prioritize alerts.
• Task 2: Route to downstream systems with backpressure handling

PROMPT :

Task 1 Prompt

"Use AI to parse incoming weather and emergency feeds, and prioritize alerts by severity."

Task 2 Prompt

"Route prioritized alerts to downstream systems with backpressure handling to avoid overload."

CODE :

```python
import queue
import threading

# Task 1: Parse & prioritize alerts
def parse_and_prioritize(feed):
    """
    Simple AI-like prioritization: emergency > severe weather > normal
    """
    priority_map = {"emergency": 0, "severe": 1, "normal": 2}
    parsed = [{"type": item["type"], "msg": item["msg"],
               "priority": priority_map.get(item["type"], 3)} for item in feed]
    # sort by priority (lower = higher priority)
    return sorted(parsed, key=lambda x: x["priority"])

# Example feed
feed = [
    {"type": "normal", "msg": "Light rain expected"},
    {"type": "emergency", "msg": "Flood warning issued"},
    {"type": "severe", "msg": "Heatwave alert"}
]

alerts = parse_and_prioritize(feed)
print("Prioritized Alerts:", alerts)

# Task 2: Route downstream with backpressure handling
alert_queue = queue.Queue(maxsize=2)  # small size to simulate backpressure

def producer(alerts):
    for alert in alerts:
        try:
```

OUTPUT :

```
Prioritized Alerts: [{'type': 'emergency', 'msg': 'Flood warning issued', 'pr
                     {'type': 'severe', 'msg': 'Heatwave alert', 'priority':
                     {'type': 'normal', 'msg': 'Light rain expected', 'priori
Produced: {'type': 'emergency', 'msg': 'Flood warning issued', 'priority': 0}
Produced: {'type': 'severe', 'msg': 'Heatwave alert', 'priority': 1}
Consumed: {'type': 'emergency', 'msg': 'Flood warning issued', 'priority': 0}
Consumed: {'type': 'severe', 'msg': 'Heatwave alert', 'priority': 1}
Produced: {'type': 'normal', 'msg': 'Light rain expected', 'priority': 2}
Consumed: {'type': 'normal', 'msg': 'Light rain expected', 'priority': 2}
```

BRIEF OBSERVATION :

-
- **Task 1 (Parsing & Prioritization):**
    - The function `parse_and_prioritize` assigns numeric priorities (`emergency=0`, `severe=1`, `normal=2`).
    - Alerts are sorted so the most critical ones are processed first.
    - Output shows a clear ordering: *Flood warning* → *Heatwave alert* → *Light rain*.
- **Task 2 (Routing with Backpressure):**
    - A bounded queue (`maxsize=2`) simulates downstream system capacity.
    - The **producer** pushes alerts into the queue, respecting limits.
    - The **consumer** pulls alerts out, ensuring processing continues.
    - If the queue fills faster than it empties, the code prints a backpressure warning and drops alerts.