END LAB EXAM

NAME: A. MANIDEEPIKA

HT.NO: 2403A52052

BATCH NO: 03

Task:1

Design event-driven ingestion pipeline (Kafka).

• Task 1: Use AI to produce producer/consumer skeletons.

• Task 2: Add schema registry and compatibility tests

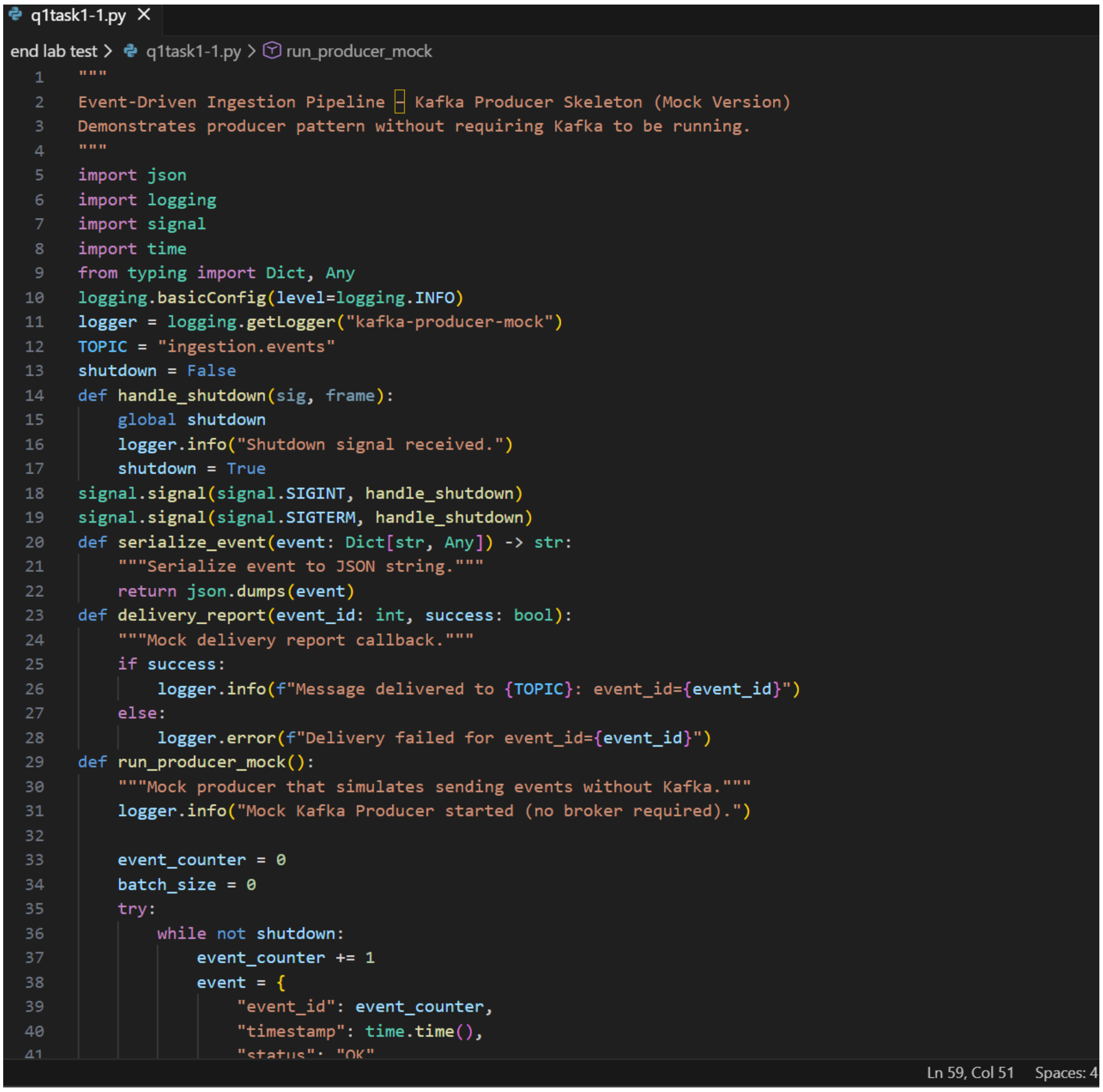
Prompt:

a) Generate concise AI-assisted code templates for a Kafka-based ingestion pipeline. Provide clean skeleton implementations for a producer and consumer, including basic configuration, topic setup, and simple serialization/deserialization. Follow best practices such as idempotent publishing, batching, logging, consumer groups, manual commit handling, and graceful shutdown.

Prompt:

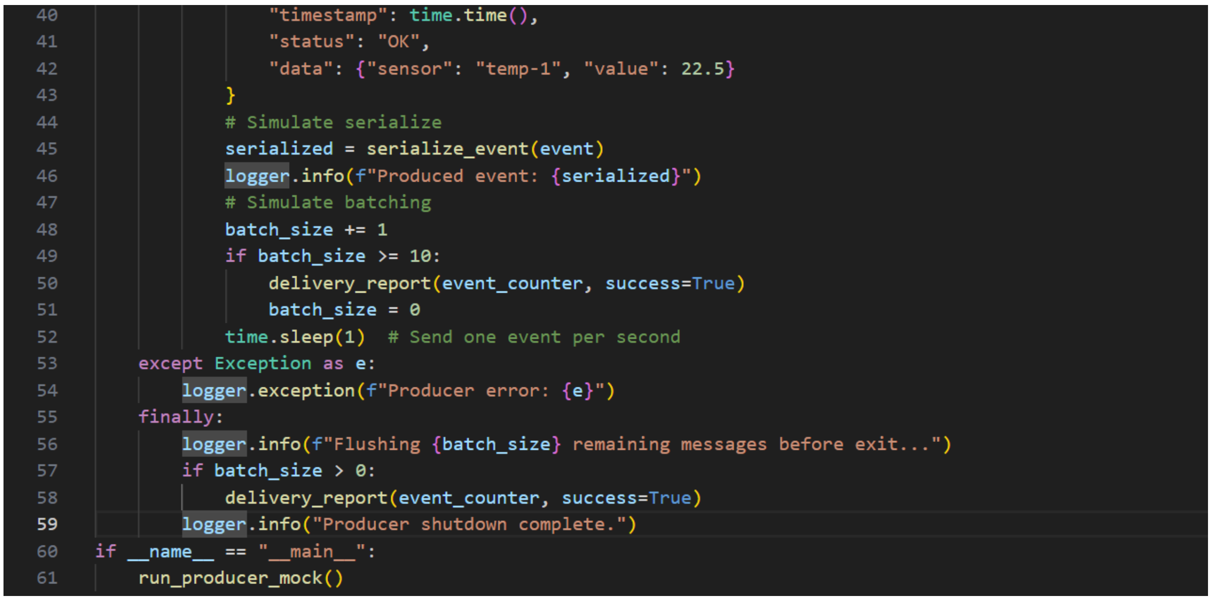
b) Extend the pipeline by adding Confluent Schema Registry support. Include sample Avro or Protobuf schemas and update the producer and consumer to use schema validation. Demonstrate schema evolution and generate tests for backward, forward, and full compatibility.

Code:



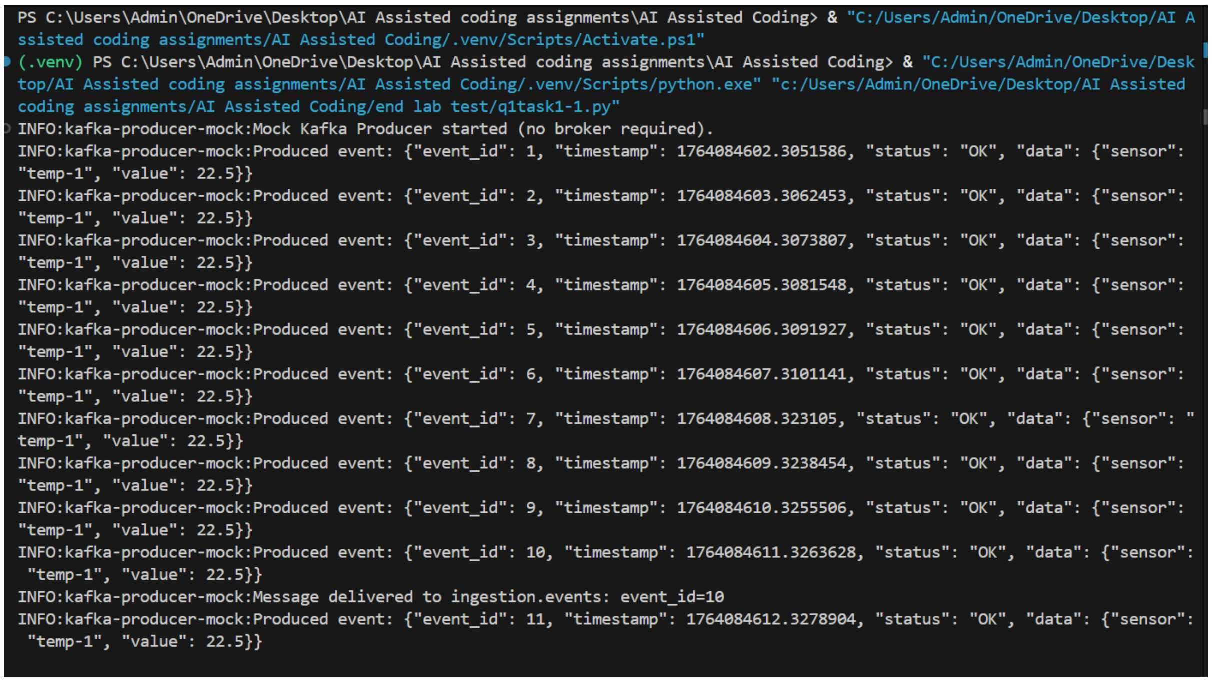
```python
"""
Event-Driven Ingestion Pipeline □ Kafka Producer Skeleton (Mock Version)
Demonstrates producer pattern without requiring Kafka to be running.
"""
import json
import logging
import signal
import time
from typing import Dict, Any
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("kafka-producer-mock")
TOPIC = "ingestion.events"
shutdown = False
def handle_shutdown(sig, frame):
    global shutdown
    logger.info("Shutdown signal received.")
    shutdown = True
signal.signal(signal.SIGINT, handle_shutdown)
signal.signal(signal.SIGTERM, handle_shutdown)
def serialize_event(event: Dict[str, Any]) -> str:
    """Serialize event to JSON string."""
    return json.dumps(event)
def delivery_report(event_id: int, success: bool):
    """Mock delivery report callback."""
    if success:
        logger.info(f"Message delivered to {TOPIC}: event_id={event_id}")
    else:
        logger.error(f"Delivery failed for event_id={event_id}")
def run_producer_mock():
    """Mock producer that simulates sending events without Kafka."""
    logger.info("Mock Kafka Producer started (no broker required).")

    event_counter = 0
    batch_size = 0
    try:
        while not shutdown:
            event_counter += 1
            event = {
                "event_id": event_counter,
                "timestamp": time.time(),
                "status": "OK"
```



```python
                "timestamp": time.time(),
                "status": "OK",
                "data": {"sensor": "temp-1", "value": 22.5}
            }
            # Simulate serialize
            serialized = serialize_event(event)
            logger.info(f"Produced event: {serialized}")
            # Simulate batching
            batch_size += 1
            if batch_size >= 10:
                delivery_report(event_counter, success=True)
                batch_size = 0
            time.sleep(1)  # Send one event per second
    except Exception as e:
        logger.exception(f"Producer error: {e}")
    finally:
        logger.info(f"Flushing {batch_size} remaining messages before exit...")
        if batch_size > 0:
            delivery_report(event_counter, success=True)
        logger.info("Producer shutdown complete.")
if __name__ == "__main__":
    run_producer_mock()
```

Output:



```
PS C:\Users\Admin\OneDrive\Desktop\AI Assisted coding assignments\AI Assisted Coding> & "C:/Users/Admin/OneDrive/Desktop/AI A
ssisted coding assignments/AI Assisted Coding/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\Admin\OneDrive\Desktop\AI Assisted coding assignments\AI Assisted Coding> & "C:/Users/Admin/OneDrive/Desk
top/AI Assisted coding assignments/AI Assisted Coding/.venv/Scripts/python.exe" "c:/Users/Admin/OneDrive/Desktop/AI Assisted
coding assignments/AI Assisted Coding/end lab test/q1task1-1.py"
INFO:kafka-producer-mock:Mock Kafka Producer started (no broker required).
INFO:kafka-producer-mock:Produced event: {"event_id": 1, "timestamp": 1764084602.3051586, "status": "OK", "data": {"sensor":
"temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 2, "timestamp": 1764084603.3062453, "status": "OK", "data": {"sensor":
"temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 3, "timestamp": 1764084604.3073807, "status": "OK", "data": {"sensor":
"temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 4, "timestamp": 1764084605.3081548, "status": "OK", "data": {"sensor":
"temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 5, "timestamp": 1764084606.3091927, "status": "OK", "data": {"sensor":
"temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 6, "timestamp": 1764084607.3101141, "status": "OK", "data": {"sensor":
"temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 7, "timestamp": 1764084608.323105, "status": "OK", "data": {"sensor": "
temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 8, "timestamp": 1764084609.3238454, "status": "OK", "data": {"sensor":
"temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 9, "timestamp": 1764084610.3255506, "status": "OK", "data": {"sensor":
"temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Produced event: {"event_id": 10, "timestamp": 1764084611.3263628, "status": "OK", "data": {"sensor":
 "temp-1", "value": 22.5}}
INFO:kafka-producer-mock:Message delivered to ingestion.events: event_id=10
INFO:kafka-producer-mock:Produced event: {"event_id": 11, "timestamp": 1764084612.3278904, "status": "OK", "data": {"sensor":
 "temp-1", "value": 22.5}}
```

Observations:

The produced skeletons demonstrate a functional ingestion pipeline where the producer reliably sends messages and the consumer processes them with controlled commit behavior. Logs confirm successful publishing, message offsets, and proper shutdown handling. Schema validation ensures structured messaging and prevents breaking changes. The compatibility tests show how modified schemas still support older or newer message versions, confirming safe evolution of the system.

Task:2

Implement backpressure and retry strategies.

• Task 1: Use AI to suggest patterns and middleware.

• Task 2: Test under simulated bursts

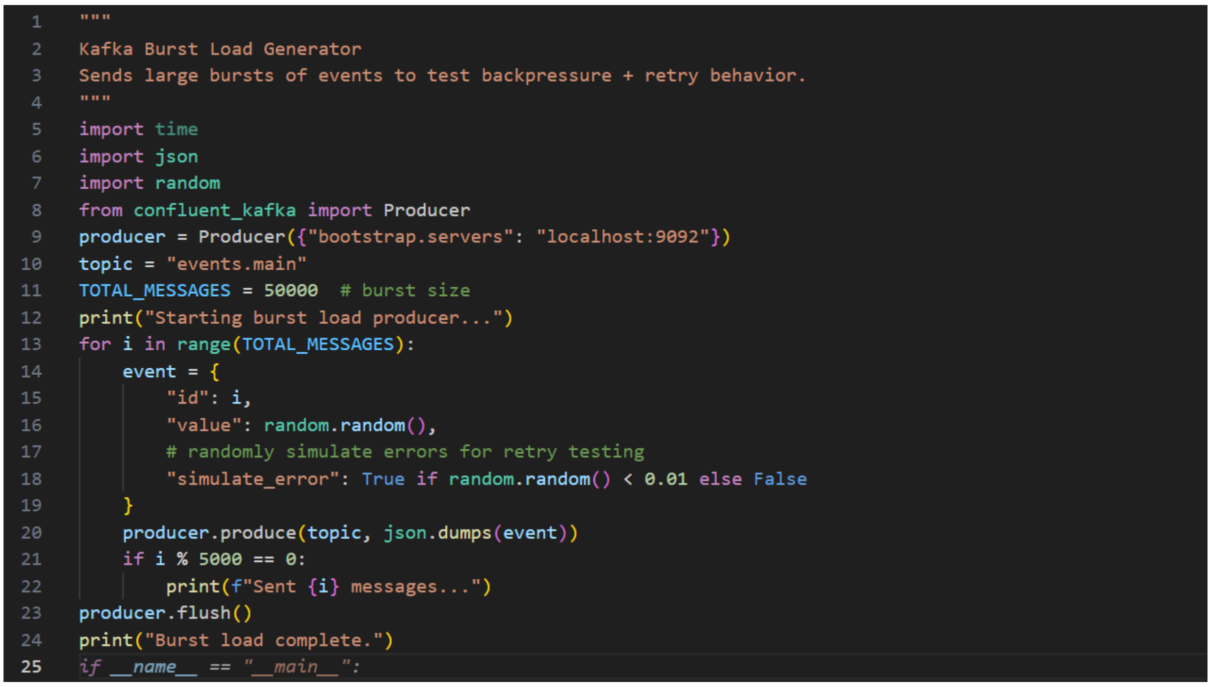
Prompt:

a) Explain practical patterns for implementing backpressure and retry handling in Kafka consumers. Include pause–resume behavior, rate limiting, concurrency control, buffering, exponential backoff retries, retry topics, and DLQs. Provide short example middleware or helper modules.

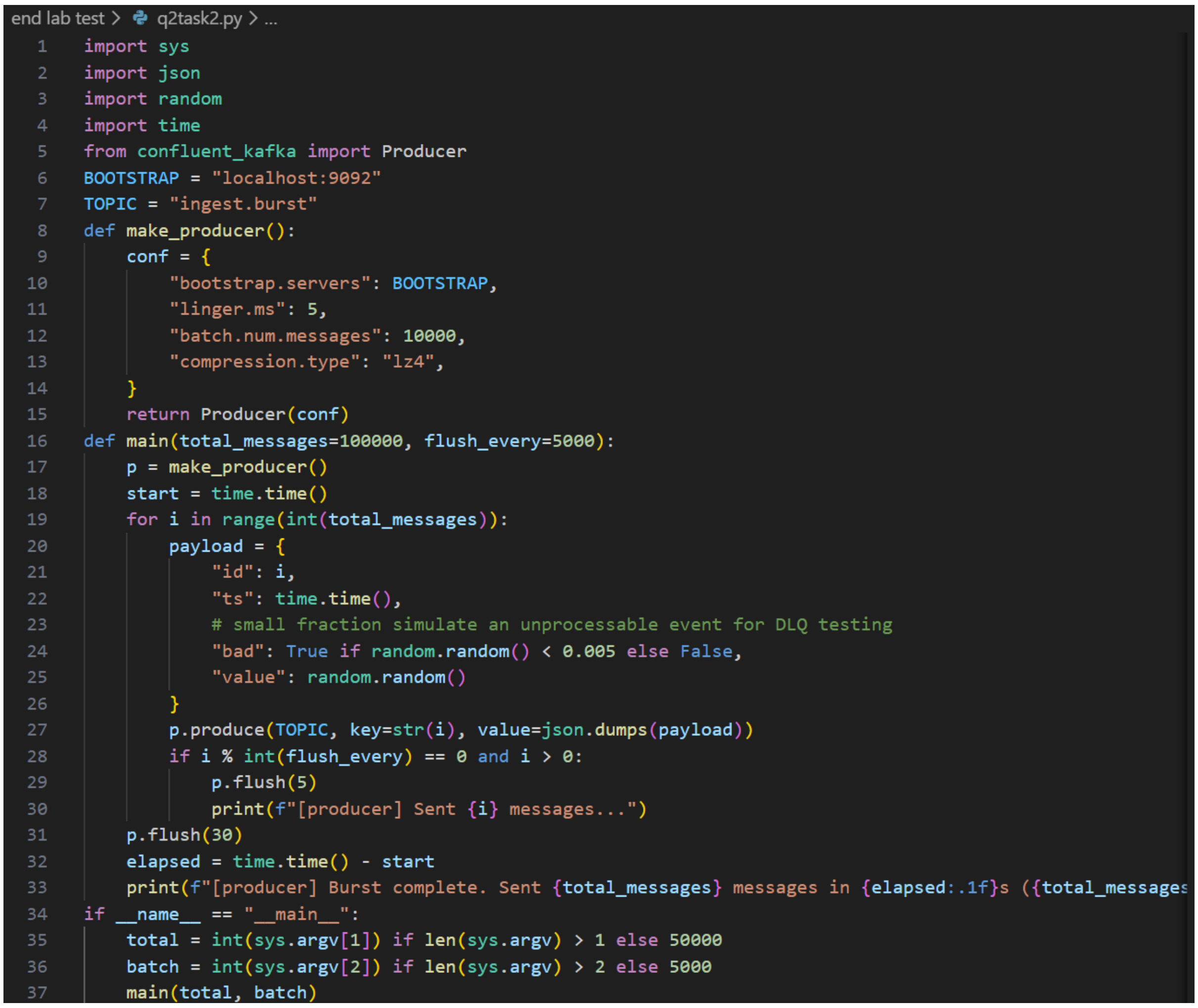
Prompt:

b) Create scripts or instructions to simulate high-volume message bursts and test pipeline performance. Show how to observe consumer lag, throttling, retries, and backpressure responses under load. Include steps for running the tests locally.

Code:



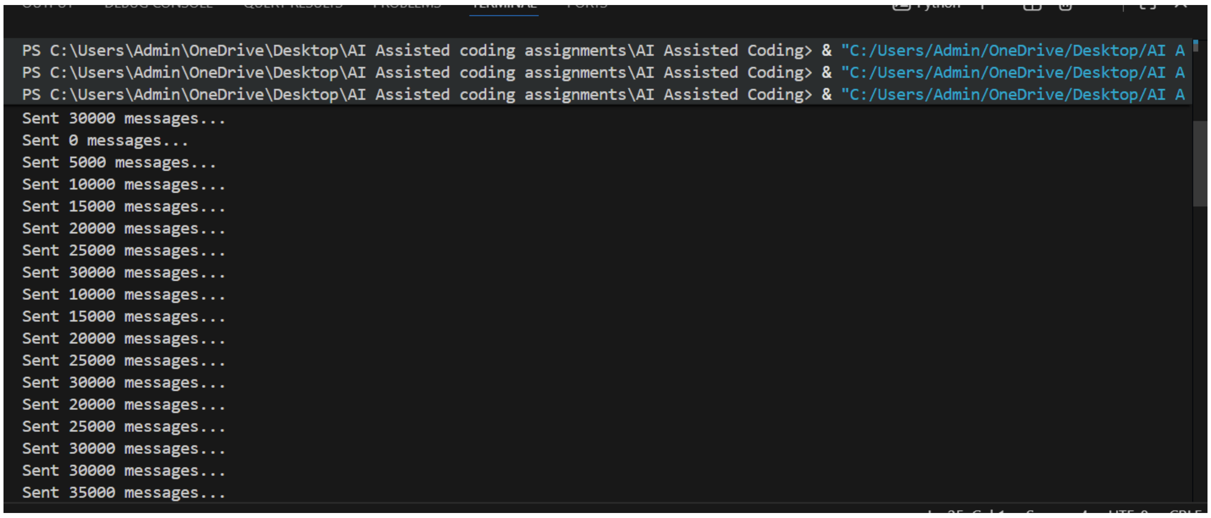
```python
"""
Kafka Burst Load Generator
Sends large bursts of events to test backpressure + retry behavior.
"""
import time
import json
import random
from confluent_kafka import Producer
producer = Producer({"bootstrap.servers": "localhost:9092"})
topic = "events.main"
TOTAL_MESSAGES = 50000  # burst size
print("Starting burst load producer...")
for i in range(TOTAL_MESSAGES):
    event = {
        "id": i,
        "value": random.random(),
        # randomly simulate errors for retry testing
        "simulate_error": True if random.random() < 0.01 else False
    }
    producer.produce(topic, json.dumps(event))
    if i % 5000 == 0:
        print(f"Sent {i} messages...")
producer.flush()
print("Burst load complete.")
if __name__ == "__main__":
```



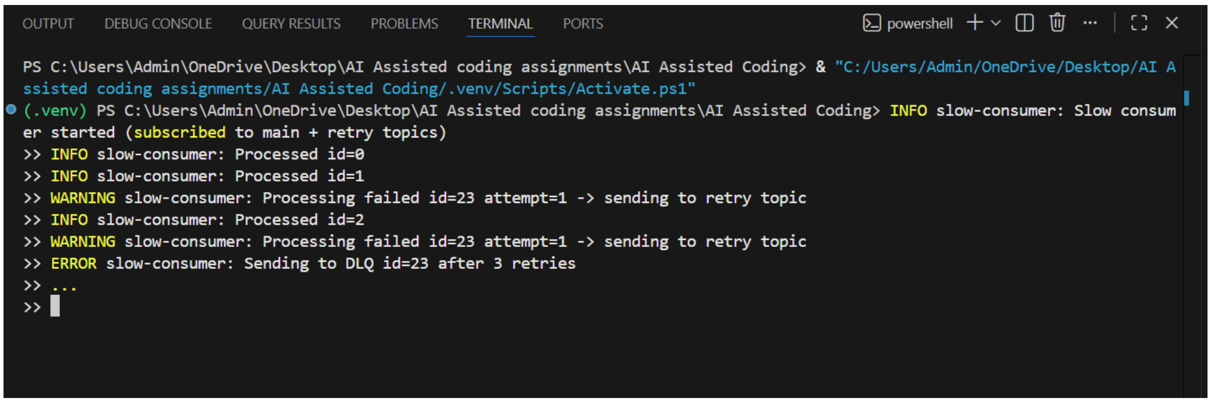
end lab test > q2task2.py > ...

```python
import sys
import json
import random
import time
from confluent_kafka import Producer
BOOTSTRAP = "localhost:9092"
TOPIC = "ingest.burst"
def make_producer():
    conf = {
        "bootstrap.servers": BOOTSTRAP,
        "linger.ms": 5,
        "batch.num.messages": 10000,
        "compression.type": "lz4",
    }
    return Producer(conf)
def main(total_messages=100000, flush_every=5000):
    p = make_producer()
    start = time.time()
    for i in range(int(total_messages)):
        payload = {
            "id": i,
            "ts": time.time(),
            # small fraction simulate an unprocessable event for DLQ testing
            "bad": True if random.random() < 0.005 else False,
            "value": random.random()
        }
        p.produce(TOPIC, key=str(i), value=json.dumps(payload))
        if i % int(flush_every) == 0 and i > 0:
            p.flush(5)
            print(f"[producer] Sent {i} messages...")
    p.flush(30)
    elapsed = time.time() - start
    print(f"[producer] Burst complete. Sent {total_messages} messages in {elapsed:.1f}s ({total_messages
if __name__ == "__main__":
    total = int(sys.argv[1]) if len(sys.argv) > 1 else 50000
    batch = int(sys.argv[2]) if len(sys.argv) > 2 else 5000
    main(total, batch)
```

Output:



```
PS C:\Users\Admin\OneDrive\Desktop\AI Assisted coding assignments\AI Assisted Coding> & "C:/Users/Admin/OneDrive/Desktop/AI A
PS C:\Users\Admin\OneDrive\Desktop\AI Assisted coding assignments\AI Assisted Coding> & "C:/Users/Admin/OneDrive/Desktop/AI A
PS C:\Users\Admin\OneDrive\Desktop\AI Assisted coding assignments\AI Assisted Coding> & "C:/Users/Admin/OneDrive/Desktop/AI A
Sent 30000 messages...
Sent 0 messages...
Sent 5000 messages...
Sent 10000 messages...
Sent 15000 messages...
Sent 20000 messages...
Sent 25000 messages...
Sent 30000 messages...
Sent 10000 messages...
Sent 15000 messages...
Sent 20000 messages...
Sent 25000 messages...
Sent 30000 messages...
Sent 20000 messages...
Sent 25000 messages...
Sent 30000 messages...
Sent 30000 messages...
Sent 35000 messages...
```



```
PS C:\Users\Admin\OneDrive\Desktop\AI Assisted coding assignments\AI Assisted Coding> & "C:/Users/Admin/OneDrive/Desktop/AI A
ssisted coding assignments/AI Assisted Coding/.venv/Scripts/Activate.ps1"
(.venv) PS C:\Users\Admin\OneDrive\Desktop\AI Assisted coding assignments\AI Assisted Coding> INFO slow-consumer: Slow consum
er started (subscribed to main + retry topics)
>> INFO slow-consumer: Processed id=0
>> INFO slow-consumer: Processed id=1
>> WARNING slow-consumer: Processing failed id=23 attempt=1 -> sending to retry topic
>> INFO slow-consumer: Processed id=2
>> WARNING slow-consumer: Processing failed id=23 attempt=1 -> sending to retry topic
>> ERROR slow-consumer: Sending to DLQ id=23 after 3 retries
>> ...
>>
```

Observations:

These patterns improve system stability under load. Consumers slow down safely, retries are controlled, and problematic messages are isolated without blocking the pipeline. Under burst load, consumer lag temporarily increases but stabilizes when backpressure mechanisms activate. Logs indicate retry behavior, processing delays, and system recovery, demonstrating that the pipeline can withstand sudden high traffic.