Submitted by: Riya Dave (202318011)

Assignment Report on
Real-Time E-commerce Order Processing System Using Kafka

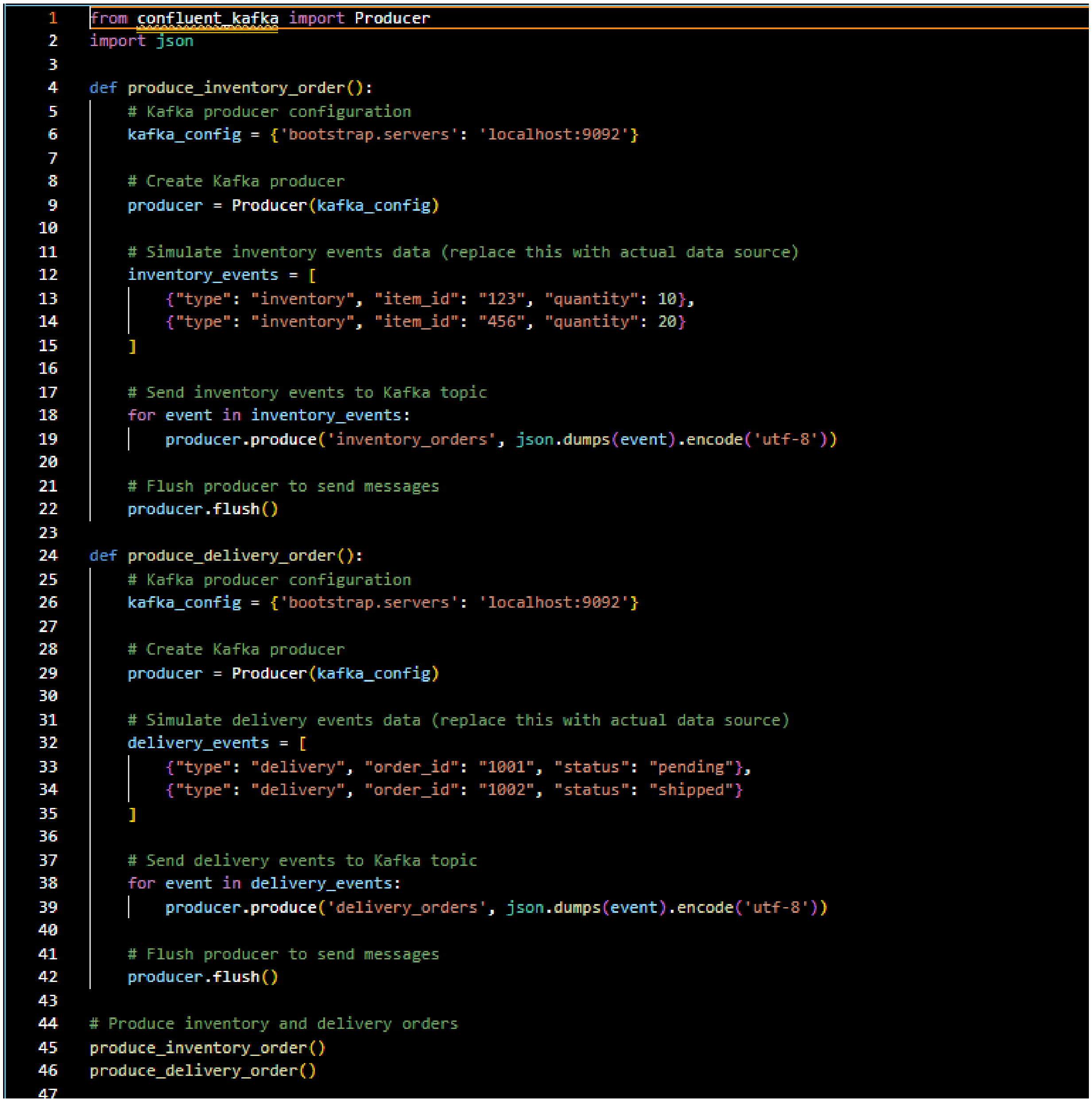
To develop a Kafka-based system for managing e-commerce orders in real-time, you'll need to set up producers, consumers, and implement message filtering logic. Below are the steps you can follow to achieve this:

Step 1: Set Up Kafka

1. **Install Kafka:** Ensure Kafka is installed and running on your system or a server.
2. **Create Kafka Topics:** Create Kafka topics named **inventory_orders** and **delivery_orders** for each producer to send messages to.

Step 2: Implement Kafka Producers

1. **Inventory Orders Producer (inventory_orders_producer):**
   - This producer should filter messages where the **type** field is **inventory**.
   - Implement a Kafka producer that reads inventory-related events from a data source (like a database or event stream) and sends messages with **type** set to **inventory** to the **inventory_orders** topic.
2. **Delivery Orders Producer (delivery_orders_producer):**
   - This producer should filter messages where the **type** field is **delivery**.
   - Develop a Kafka producer that reads delivery-related events and sends messages with **type** set to **delivery** to the **delivery_orders** topic.

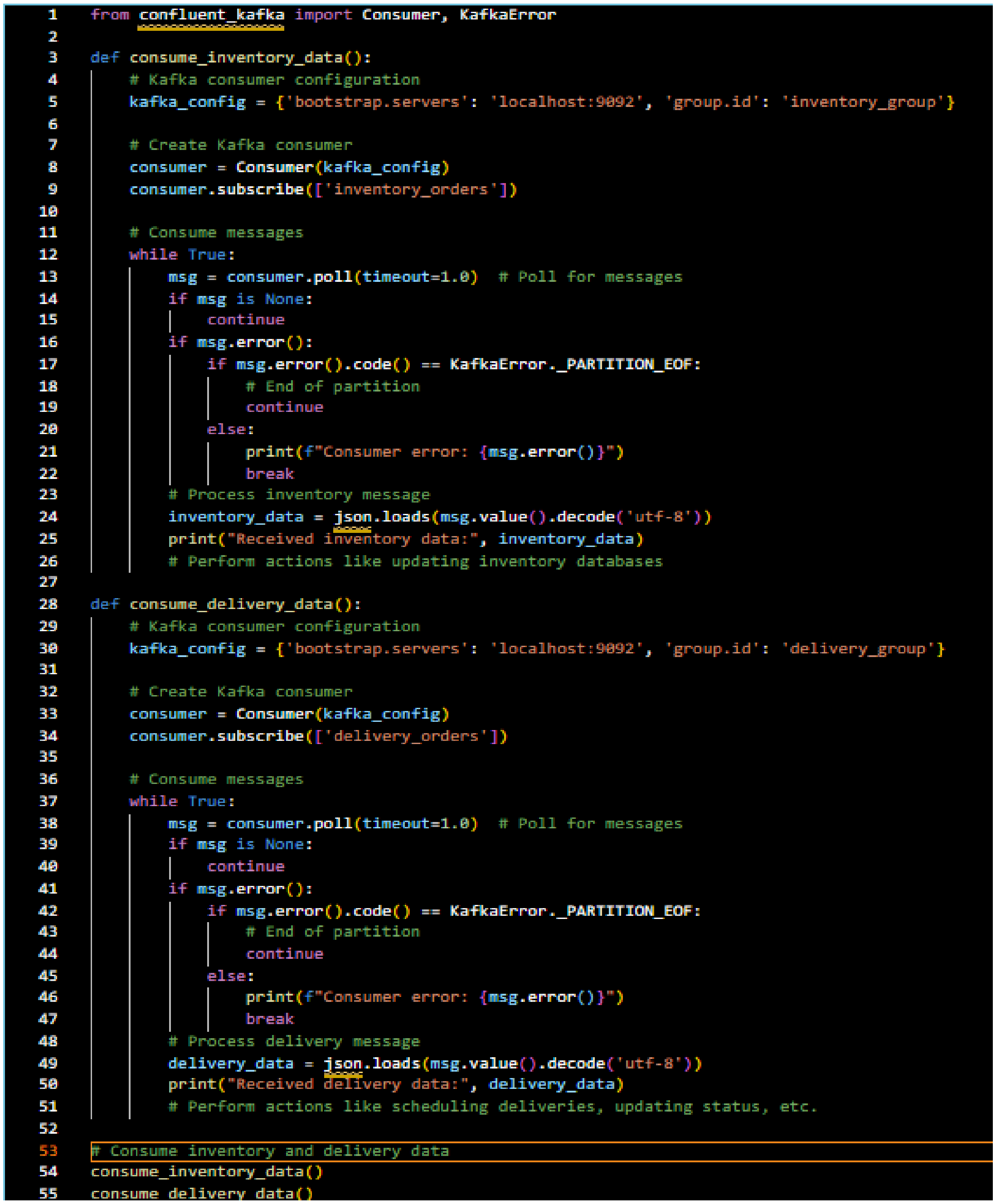


```python
from confluent_kafka import Producer
import json

def produce_inventory_order():
    # Kafka producer configuration
    kafka_config = {'bootstrap.servers': 'localhost:9092'}

    # Create Kafka producer
    producer = Producer(kafka_config)

    # Simulate inventory events data (replace this with actual data source)
    inventory_events = [
        {"type": "inventory", "item_id": "123", "quantity": 10},
        {"type": "inventory", "item_id": "456", "quantity": 20}
    ]

    # Send inventory events to Kafka topic
    for event in inventory_events:
        producer.produce('inventory_orders', json.dumps(event).encode('utf-8'))

    # Flush producer to send messages
    producer.flush()

def produce_delivery_order():
    # Kafka producer configuration
    kafka_config = {'bootstrap.servers': 'localhost:9092'}

    # Create Kafka producer
    producer = Producer(kafka_config)

    # Simulate delivery events data (replace this with actual data source)
    delivery_events = [
        {"type": "delivery", "order_id": "1001", "status": "pending"},
        {"type": "delivery", "order_id": "1002", "status": "shipped"}
    ]

    # Send delivery events to Kafka topic
    for event in delivery_events:
        producer.produce('delivery_orders', json.dumps(event).encode('utf-8'))

    # Flush producer to send messages
    producer.flush()

# Produce inventory and delivery orders
produce_inventory_order()
produce_delivery_order()

```

Step 3: Implement Kafka Consumers

1. **Inventory Data Consumer (inventory_data_consumer):**
    - Configure a Kafka consumer that subscribes to the **inventory_orders** topic.
    - Implement logic to process inventory messages received by updating inventory databases or systems accordingly.
2. **Delivery Data Consumer (delivery_data_consumer):**

- Set up a Kafka consumer for the **delivery_orders** topic.
- Develop logic to handle delivery-related messages such as scheduling deliveries, updating delivery status, and notifying customers.



```python
from confluent_kafka import Consumer, KafkaError

def consume_inventory_data():
    # Kafka consumer configuration
    kafka_config = {'bootstrap.servers': 'localhost:9092', 'group.id': 'inventory_group'}

    # Create Kafka consumer
    consumer = Consumer(kafka_config)
    consumer.subscribe(['inventory_orders'])

    # Consume messages
    while True:
        msg = consumer.poll(timeout=1.0)  # Poll for messages
        if msg is None:
            continue
        if msg.error():
            if msg.error().code() == KafkaError._PARTITION_EOF:
                # End of partition
                continue
            else:
                print(f"Consumer error: {msg.error()}")
                break
        # Process inventory message
        inventory_data = json.loads(msg.value().decode('utf-8'))
        print("Received inventory data:", inventory_data)
        # Perform actions like updating inventory databases

def consume_delivery_data():
    # Kafka consumer configuration
    kafka_config = {'bootstrap.servers': 'localhost:9092', 'group.id': 'delivery_group'}

    # Create Kafka consumer
    consumer = Consumer(kafka_config)
    consumer.subscribe(['delivery_orders'])

    # Consume messages
    while True:
        msg = consumer.poll(timeout=1.0)  # Poll for messages
        if msg is None:
            continue
        if msg.error():
            if msg.error().code() == KafkaError._PARTITION_EOF:
                # End of partition
                continue
            else:
                print(f"Consumer error: {msg.error()}")
                break
        # Process delivery message
        delivery_data = json.loads(msg.value().decode('utf-8'))
        print("Received delivery data:", delivery_data)
        # Perform actions like scheduling deliveries, updating status, etc.

# Consume inventory and delivery data
consume_inventory_data()
consume_delivery_data()
```

Step 4: Develop Message Filtering Logic

1. **Producer Message Filtering:**

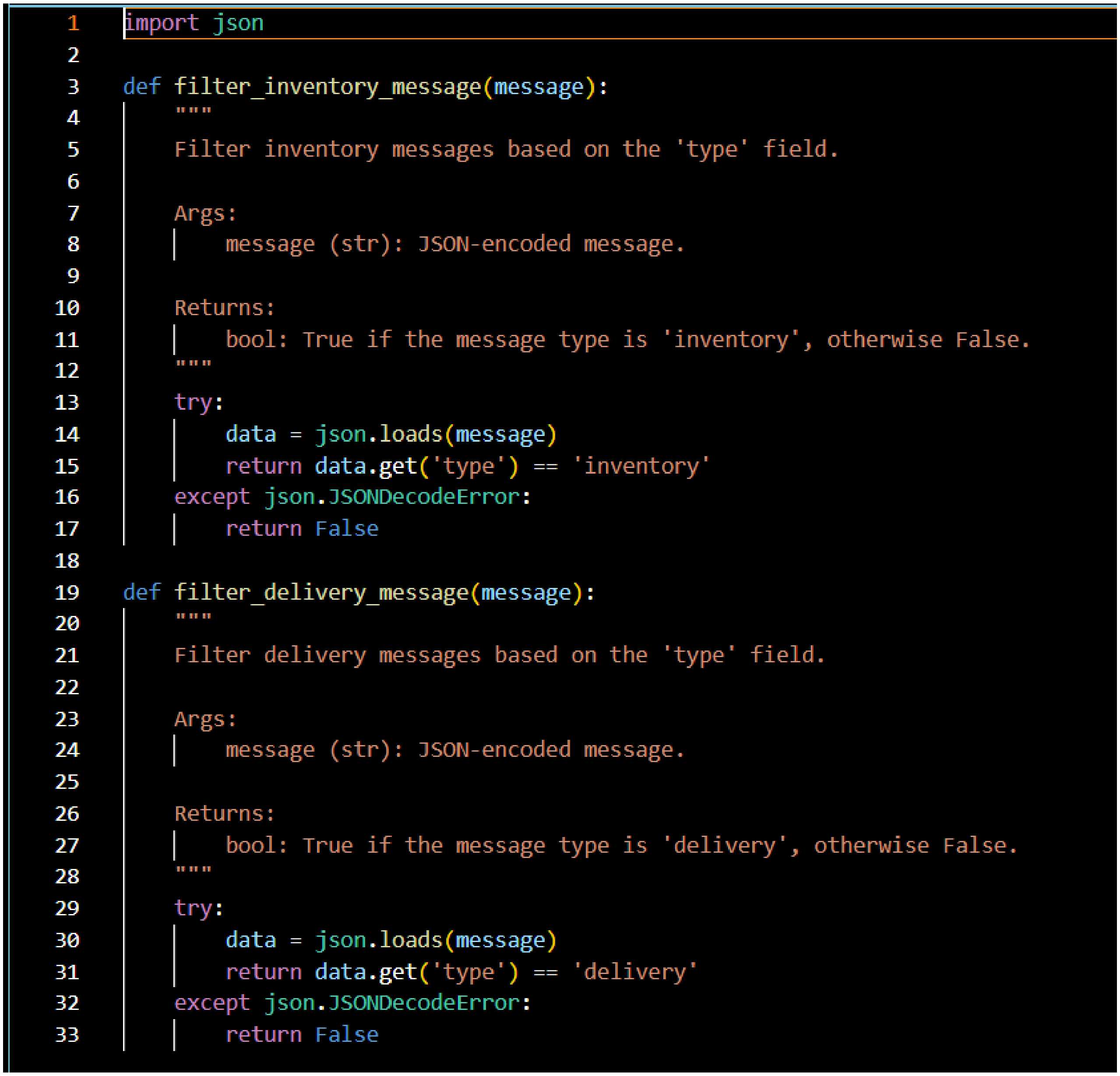
    - Implement logic within each producer (**inventory_orders_producer** and **delivery_orders_producer**) to filter messages based on the **type** field from the incoming data source.

    - Only send messages to Kafka if they match the desired **type** (i.e., **inventory** or **delivery**).

Additional Considerations

- **Error Handling:** Implement error handling within producers and consumers to manage exceptions or failed operations gracefully.
- **Scalability:** Design your system to handle increasing loads by considering Kafka partitioning, consumer groups, and scaling strategies.
- **Monitoring and Logging:** Utilize Kafka monitoring tools and logging frameworks to monitor system performance and troubleshoot issues effectively.

By following these steps and best practices, you'll be able to develop a robust Kafka-based e-commerce order management system capable of real-time inventory management and delivery processing.



```python
import json

def filter_inventory_message(message):
    """
    Filter inventory messages based on the 'type' field.

    Args:
        message (str): JSON-encoded message.

    Returns:
        bool: True if the message type is 'inventory', otherwise False.
    """
    try:
        data = json.loads(message)
        return data.get('type') == 'inventory'
    except json.JSONDecodeError:
        return False

def filter_delivery_message(message):
    """
    Filter delivery messages based on the 'type' field.

    Args:
        message (str): JSON-encoded message.

    Returns:
        bool: True if the message type is 'delivery', otherwise False.
    """
    try:
        data = json.loads(message)
        return data.get('type') == 'delivery'
    except json.JSONDecodeError:
        return False
```