USN	1	R	V							

Department of Artificial Intelligence and Machine Learning

Course Code:	:	**AI372TA**	**Date**	:	
Semester	:	**VII**	**Time**	:	
Max Marks	:	**10+50**	**Duration**	:	**120 mins**

Stream Processing and Analytics
Common to AIML and CD

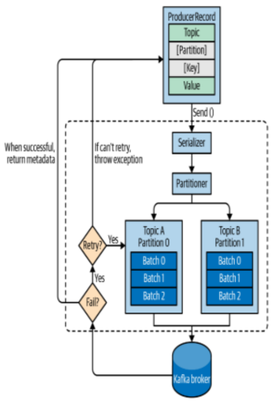
CIE I – Scheme and Solution

Note: Answer all the Questions

SL. No	**Questions**	**M**	**BT**	**CO**
	PART – A			
1	**Identify the most appropriate interaction pattern for the following applications and justify your choice:** a) An e-commerce system continuously collects clickstream data. b) A weather data collector sends periodic updates without expecting a reply. Ans **a) E-commerce clickstream data → *Stream Pattern*** • **Justification:** Clickstream data is generated continuously as users browse the website. The *stream pattern* allows real-time ingestion and processing of this continuous flow of events for analytics and personalization. **b) Weather data collector → *One-Way Pattern*** • **Justification:** The collector periodically sends sensor readings without needing acknowledgment or response. The *one-way pattern* suits this scenario as it prioritizes speed and simplicity over confirmation.	2	3	3
2	List any two needs for streaming data 1. Some data naturally comes as a never-ending stream of events. 2. Batch processing lets the data build up and try to process them at once while stream processing process data as they come in hence spread the processing over time 3. Data is huge and it is not even possible to store it. 4. There are a lot of streaming data available (e.g. customer transactions, activities, website visits) and they will grow faster with IoT use cases (all kind of sensors).	2	1	1
3	A data collection node loses messages due to a power failure. Which **message logging technique (RBML/SBML/HML)** would you use to ensure complete recovery, and why?	2	3	3

Ans :
RBML guarantees complete recovery by preserving all received messages prior to processing, making it ideal for data collection nodes prone to sudden failures.

4 Schematically represent the kafka producer components — 2 | 2 | 2



5 Summarize any two differences between early rebalance and cooperative rebalance in Kafka? — 2 | 2 | 2

Feature	Early Rebalance (Eager Rebalance)	Cooperative Rebalance Cooperative Sticky Reba
Rebalance Style	*Eager* — all consumers stop consuming during rebalance	*Incremental* — consume hand off partitions
Consumer Stop Behavior	All consumers revoke all partitions immediately	Only partitions that are another consumer are r
Downtime	Higher — complete stop of consumption during rebalance	Minimal — most partiti to be consumed
Partition Assignment	Full reassignment from scratch	Incremental reassignme necessary partitions mo
Impact on Throughput	Drop in throughput during rebalance	Smooth consumption w impact

# any two can be written

PART – B

1 a Consider an IoT-based environmental monitoring network that streams sensor data globally. — **10** | **4** | **1,3**

- Propose an architecture with a neat schematic diagram that balances **security**, **scalability**, and **fault tolerance**.

- Justify your choice of **interaction pattern** and **scaling model**
- Explain how **checkpointing** and **logging** can jointly ensure reliable recovery across multiple layers and technologies.

Ans :

1. Architecture (3 marks)
A layered, secure, and scalable architecture:

Sensors → Edge Gateways → Message Broker (Kafka/Pulsar) → Stream Processing (Flink/Spark) → Storage (Hot: Redis, Cold: S3/HDFS) → Dashboards/Alerts

Key Features:

- **Security:** TLS/mTLS for data transmission, device authentication, role-based access.
- **Scalability:** Kafka partitions + horizontally scalable stream processors.
- **Fault tolerance:** Replication, checkpoints, and message logging ensure reliability.

2. Interaction Pattern & Scaling Model (3 marks)

- **Pattern:** Publish/Subscribe + Stream pattern
  - Enables asynchronous, continuous data flow between producers (sensors) and consumers (analytics).
- **Scaling Model:** Horizontal scaling
  - Add nodes/partitions dynamically; stateless services allow auto-scaling.
  - Vertical scaling used only for high-load stateful nodes.

3. Fault Tolerance via Checkpointing & Logging (4 marks)

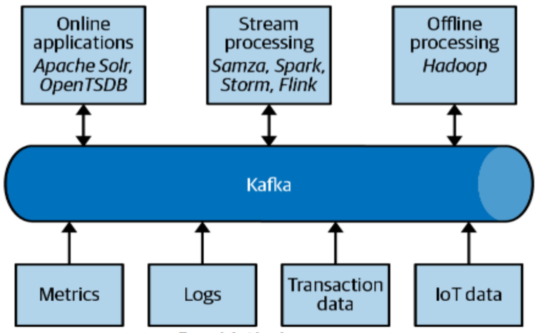
- **Checkpointing:** Periodic snapshots of application state and Kafka offsets (e.g., Flink checkpoints stored in S3).
  - Ensures system can resume from last consistent global state.
- **Logging:** Message brokers store and replay messages (RBML/SBML).
  - Enables recovery up to the last message before crash.
- **Combined Effect:** Checkpoint restores state; logs replay lost data — ensuring reliable recovery across all layers.

Keywords:
Pub/Sub, stream pattern, horizontal scaling, checkpointing, message logging, Kafka, Flink, TLS, replication, recovery consistency.

2	**a**	Examine the conceptual overlap and differences between **soft**, **near**, and **non-hard real-time systems** and **streaming data systems**.	**5**	**2**	**1**

		• **Overlap:** All aim for timely data processing and delivery; used in real-time analytics, IoT, and monitoring. • **Differences:** o **Soft real-time:** Delays acceptable; timing preferred (e.g., video streaming). o **Near real-time:** Very low delay; almost instantaneous (e.g., stock trading). o **Non-hard real-time:** No strict deadlines; delay tolerated (e.g., weather updates). o **Streaming system:** Continuous data flow; clients consume data as needed — focuses on scalability and availability, not strict timing. **In short:** Streaming systems process data in real time but relax strict deadline constraints seen in traditional real-time systems.			
	b	**Describe and compare the common interaction patterns** with suitable examples. **1. Request/Response:** Client sends a request and waits for an immediate reply. *Example:* HTTP requests, REST APIs. *Use:* When instant feedback is required. **2. Request/Acknowledge:** Client sends a request and receives an acknowledgment token for tracking or later response. *Example:* Job submission systems, order tracking. **3. Publish/Subscribe:** Producers publish messages to topics; subscribers receive updates asynchronously. *Example:* Kafka topics, message queues, IoT systems. **4. One-Way:** Client sends data without expecting a reply. *Example:* System logs, telemetry data. **5. Stream Pattern:** Continuous, asynchronous exchange of data chunks over time. *Example:* Live video feeds, real-time analytics. **Comparison:** • *Request/Response* is synchronous; others support asynchronous communication. • *Publish/Subscribe* and *Stream* patterns are best for scalable, real-time streaming systems.	**5**	**2**	**2**

3	a	A global financial services company processes billions of transactions daily to detect fraudulent activity. The company leverages a real-time fraud detection pipeline using streaming data. The company wants to implement different fault tolerant mechanisms to have secure transactions. What are the potential challenges that occur due to the failure in the stream processing? Discuss how the different stream processing tools handle failures in communication? What happens if a broker crashes? **1. Potential Challenges due to Stream Processing Failures (3 marks)** Failures in a real-time fraud detection pipeline can lead to: • **Data loss:** Transactions not processed or lost during node crashes. • **Duplicate processing:** Same transaction processed multiple times after restart. • **Inconsistent state:** Partial updates in streaming operators or aggregations. • **Latency spikes / downtime:** Reprocessing delays while recovering. • **Security and compliance risks:** Missing or incomplete fraud analysis affecting transaction integrity. **2. Handling Failures in Communication – Mechanisms (4 marks)** From the slides: Two major fault-tolerance techniques — **Checkpointing** and **Logging protocols** — are applied to recover from such failures. • **Checkpointing (Global Snapshot):** o Periodically saves the *global state* of the system (e.g., operator states + message offsets). o On recovery, system restores to the last saved state and resumes processing. o *Example:* Flink stores checkpoints to durable storage like S3 or HDFS. o **Limitation:** Data between failure and last checkpoint may be lost. • **Logging Protocols:** o Every message is logged either at the **receiver** (RBML) or **sender** (SBML) side. o Messages can be **replayed** after failure to restore consistent state without a full global snapshot. o **Hybrid Message Logging (HML)** combines both for improved reliability and reduced overhead. **Fault recovery flow:** After failure → restore from checkpoint → replay unprocessed messages from log → ensure consistent state recovery. **3. Broker Crash Scenario (3 marks)** If the **message broker (e.g., Kafka)** crashes: • **Replication** ensures no data loss — messages are stored in multiple brokers (ISR group).	**10**	**4**	**3**

		• Producers and consumers automatically reconnect to available brokers. • **Unacknowledged messages** can be replayed from the broker's log. • **Offset management** ensures consumers resume from last committed position. • In case of permanent broker failure, leader election assigns a new broker for each partition.			
4	a	With a neat diagram elaborate the kafka data eco system  Online applications Apache Solr, OpenTSDB Stream processing Samza, Spark, Storm, Flink Offline processing Hadoop Kafka Metrics Logs Transaction data IoT data Figure 1-9. A big data ecosystem # Elaborate the diagram o Defined inputs in the form of applications that create data or otherwise introduce it to the system. o Defined outputs in the form of metrics, reports, and other data products. o Create loops, with some components reading data from the system, transforming it using data from other sources, and then introducing it back into the data infrastructure to be used elsewhere. o This is done for numerous types of data, with each having unique qualities of content, size, and usage	1+ 4	3	3

	b	Assume a real-time e-commerce platform where every order placed by a customer must be published to a Kafka topic named orders. The orders are later consumed by:
		• An inventory service (to reduce stock),
		• A billing service (to generate invoices), and
		• A notification service (to send order confirmations).
		This system handles thousands of orders per second. The system needs high throughput and no duplicate orders.
		Analyse and demonstrate the kafka producer using a program.
		Ans:
		
		High throughput → so you'll send messages asynchronously.
		No duplicate orders → so you'll enable idempotence in the producer.
		from confluent_kafka import Producer
		import json
		import time
		
		# Define Kafka topic
		TOPIC = "orders"
		
		# Configure Kafka Producer
		conf = {
			5	4	5

```
    'bootstrap.servers': 'localhost:9092',
    'acks': 'all',                      # wait for all replicas
    'enable.idempotence': True,         # prevent duplicate messages
    'retries': 1000000,
    'max.in.flight.requests.per.connection': 5,
    'compression.type': 'snappy',
    'linger.ms': 5,
    'batch.num.messages': 1000
}

# Create Producer instance
producer = Producer(conf)

# Delivery report callback
def delivery_report(err, msg):
    if err is not None:
        print(f"❌ Delivery failed for {msg.key()}: {err}")
    else:
        print(f"✅ Delivered {msg.key().decode('utf-8')} "
              f"to {msg.topic()} [{msg.partition()}] @ offset {msg.offset()}")

# Produce messages asynchronously
for i in range(1, 6):
    order_id = f"ORD{i:03d}"
    order = {
        "orderId": order_id,
        "item": f"Product-{i}",
        "price": i * 1000.0,
        "timestamp": int(time.time() * 1000)
    }

    producer.produce(
        topic=TOPIC,
        key=order_id.encode('utf-8'),
        value=json.dumps(order).encode('utf-8'),
        callback=delivery_report
    )

    # Serve delivery callbacks
    producer.poll(0)

# Wait for all messages to be delivered
producer.flush()
print(" All messages sent successfully.")
```

5	**a**	Assume you have a kafka topic named Sales_data that contains 6 partitions (P0-P5). A consumer group with two consumers (C1 and C2). Assuming kafka uses RangeAssignor strategy, answer the following questions : (i) Explain how partitions will be assigned between C1 and C2. (ii) What issue can arise if you add a third consumer (C3) to the group? (iii) Give the solution to resolve the issues Ans :	**2+ 2+ 2**	4	**5**

		(i) With RangeAssignor, partitions are assigned *contiguously*: • C1 → p0, p1, p2 • C2 → p3, p4, p5 (ii) When C3 joins, Kafka rebalances and assigns: • C1 → p0, p1 • C2 → p2, p3 • C3 → p4, p5 □ **Issue:** Range assignor may lead to **uneven load** if topic has few partitions or unbalanced partition sizes. (e.g., C1 may get heavier partitions than C2). (iii)Use an assignor that balances partitions more evenly or preserves stickiness. • **RoundRobinAssignor** — evens out partition counts across consumers. • **CooperativeStickyAssignor** — balances AND minimizes movement with low-downtime rebalances. • **UniformStickyAssignor** (Kafka ≥ 3.5) — combines strict uniformity and stickiness (best if available).			

	b	Consider e a Kafka topic named **sales_data** that contains messages in **JSON format.** Each message is produced as a **UTF-8 encoded JSON string**, and there is a need for the **Kafka consumer** to automatically convert it into a Python/Java dictionary when consuming. Demonstrate the example consumer program.
		
		# Student can write the code in python or java
		Python Code
		from kafka import KafkaConsumer
		import json
		
		# Define a JSON deserializer function
		def json_deserializer(data):
		    if data is None:
		        return None
		    return json.loads(data.decode('utf-8'))
		
		# Create Kafka Consumer
		consumer = KafkaConsumer(
		    'sales_data', # topic name
		    bootstrap_servers='localhost:9092', # broker
		    group_id='sales_group', # consumer group
		    auto_offset_reset='earliest', # start from beginning
		    enable_auto_commit=True, # commit offsets automatically
		    value_deserializer=json_deserializer # apply custom deserializer
		)
		
		print("Consumer is listening to 'sales_data' topic...")
		
		# Consume and print messages
		for message in consumer:
		    print(f"Topic: {message.topic}, Partition: {message.partition}, Offset: {message.offset}")
			4	**4**	**5**

```
    print(f"Message Value (Deserialized): {message.value}")
```

Java Code // File: SalesRecord.java

```
public class SalesRecord {
    private int order_id;
    private String item;
    private double price;

    // Getters and setters
    public int getOrder_id() { return order_id; }
    public void setOrder_id(int order_id) { this.order_id = order_id; }

    public String getItem() { return item; }
    public void setItem(String item) { this.item = item; }

    public double getPrice() { return price; }
    public void setPrice(double price) { this.price = price; }

    @Override
    public String toString() {
        return "SalesRecord{" +
                "order_id=" + order_id +
                ", item='" + item + '\'' +
                ", price=" + price +
                '}';
    }
}
```