# Part 1 — Environment Setup and Basics

## 1. Start the environment

Download the repository and start the environment:

```
docker compose up -d
```

Check if the **four containers** are running:

- postgres
- kafka
- kafka-ui
- connect

## 2. Access PostgreSQL

```
docker exec -it postgres psql -U postgres
```

# Kafka Quick Start (Docker)

## A. Check Kafka is running

```
docker ps
```

**Explanation**
Confirms that the Kafka broker container is running and shows its container name (e.g. `kafka`).

```
[+] Running 4/4
 ✔ Container postgres  Running
 ✔ Container kafka     Running
 ✔ Container kafka-ui  Running
 ✔ Container connect   Running
```

---

## B. Create a topic with multiple partitions

```
docker exec -it kafka kafka-topics.sh \
  --bootstrap-server localhost:9092 \
```

```
  --create \
  --topic activity.streaming \
  --partitions 4 \
  --replication-factor 1
```

**Explanation**

- `--topic`: Name of the Kafka topic
- `--partitions 4`: Creates four partitions to allow parallelism
- `--replication-factor 1`: One replica per partition (suitable for local development)

---

## C. List all topics

```
docker exec -it kafka kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --list
```

**Explanation**
Displays all topics currently available in the Kafka cluster.

```
__consumer_offsets
activity.streaming
connect-configs
connect-offsets
connect-statuses
```

---

## D. Describe a topic

```
docker exec -it kafka kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --describe \
  --topic activity.streaming
```

**Explanation**
Shows partition count, leaders, replicas, and in-sync replicas (ISR).

```
Topic: activity.streaming       TopicId: hfRopfRGQ2exYpKwP1crPg PartitionCount: 4
ReplicationFactor: 1    Configs: segment.bytes=1073741824
        Topic: activity.streaming       Partition: 0    Leader: 1       Replicas:
1     Isr: 1  Elr:    LastKnownElr:
        Topic: activity.streaming       Partition: 1    Leader: 1       Replicas:
1     Isr: 1  Elr:    LastKnownElr:
```

```
        Topic: activity.streaming       Partition: 2    Leader: 1       Replicas:
1     Isr: 1  Elr:    LastKnownElr:
        Topic: activity.streaming       Partition: 3    Leader: 1       Replicas:
1     Isr: 1  Elr:    LastKnownElr:
```

---

## E. List topic configuration

```
docker exec -it kafka kafka-configs.sh \
  --bootstrap-server localhost:9092 \
  --entity-type topics \
  --entity-name activity.streaming \
  --describe
```

**Explanation**

Displays topic-level configurations such as retention and cleanup policies.
Configurations not listed inherit Kafka broker defaults.

---

## F. Produce messages to the topic

### F.1 Basic producer

```
docker exec -it kafka kafka-console-producer.sh \
  --bootstrap-server localhost:9092 \
  --topic activity.streaming
```

Example input:

```
{"id":1,"name":"Alice"}
{"id":2,"name":"Bob"}
```

**Explanation**

Messages are distributed across partitions in a round-robin fashion when no key is provided.

---

### F.2 Producer with keys

```
docker exec -it kafka kafka-console-producer.sh \
  --bootstrap-server localhost:9092 \
  --topic activity.streaming \
  --property parse.key=true \
  --property key.separator=:
```

Example input:

```
1:{"id":1,"name":"Alice"}
1:{"id":1,"name":"Alice-updated"}
2:{"id":2,"name":"Bob"}
```

**Explanation**

Messages with the same key are routed to the same partition, preserving per-key ordering.

---

# G. Consume messages from the topic

## G.1 Consume from the beginning

```
docker exec -it kafka kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 \
  --topic activity.streaming \
  --from-beginning
```

**Explanation**

Reads all messages from the beginning of the topic.

---

## G.2 Consume using a consumer group

```
docker exec -it kafka kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 \
  --topic activity.streaming \
  --group customers-service
```

**Explanation**

Consumers in the same group share partitions and automatically commit offsets.

---

# H. Inspect consumer group status

```
docker exec -it kafka kafka-consumer-groups.sh \
  --bootstrap-server localhost:9092 \
  --describe \
  --group customers-service
```

**Explanation**

Shows partition assignments, current offsets, and consumer lag.

---

## I. Delete the topic (optional)

```
docker exec -it kafka kafka-topics.sh \
  --bootstrap-server localhost:9092 \
  --delete \
  --topic activity.streaming
```

**Explanation**
Deletes the topic and all stored data (requires `delete.topic.enable=true` on the broker).

# Debezium CDC with PostgreSQL and Kafka

## Verify the services

- Kafka UI: http://localhost:8080
- Connector plugins endpoint: http://localhost:8083/connector-plugins

Ensure that the Connect service responds successfully.

## Example: Insert a row in PostgreSQL

### Create a new database

```
CREATE DATABASE activity;
```

### Connect to the new database

```
\c activity
```

### Create the table

```
CREATE TABLE activity (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  email VARCHAR(255)
);
```

## Register the Debezium Connector

The Docker Compose file only starts the Kafka Connect engine.
You must explicitly register a Debezium connector so it starts watching PostgreSQL.

In **another terminal**, run:

```
curl -i -X POST   -H "Accept:application/json"   -H "Content-
Type:application/json"   localhost:8083/connectors/   -d '{
    "name": "activity-connector",
    "config": {
      "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
      "tasks.max": "1",
      "database.hostname": "postgres",
      "database.port": "5432",
      "database.user": "postgres",
      "database.password": "postgrespw",
      "database.dbname": "activity",
      "slot.name": "activityslot",
      "topic.prefix": "dbserver1",
      "plugin.name": "pgoutput",
      "database.replication.slot.name": "debeziumactivity"
    }
  }'
```

### Check Debezium status

The connector and its tasks should be in the `RUNNING` state:

```
curl -s http://localhost:8083/connectors/activity-connector/status | jq
```

In the Kafka UI (http://localhost:8080), verify that new topics appear.

## Insert a record into PostgreSQL

Back in the PostgreSQL console, insert a record:

```
INSERT INTO activity(id, name) VALUES (1, 'Alice');
```

Debezium will produce a Kafka message on the topic:

```
dbserver1.public.activity
```

With a payload similar to:

```
{
  "op": "c",
  "after": {
    "id": 1,
```

```
    "name": "Alice"
  }
}
```

### Consume from the Kafka topic

```
docker exec -it kafka kafka-console-consumer.sh   --bootstrap-server
localhost:9092   --topic dbserver1.public.activity  --from-beginning
```

# Activity 1

Considering the above part `Debezium CDC with PostgreSQL and Kafka`, explain with your own words what it does and why it is a relevant software architecture for Big Data in the AI era and for which use cases.

### What it does:

Debezium CDC (Change Data Capture) with PostgreSQL and Kafka creates a streaming pipeline that automatically captures and publishes database changes in real-time.

### How it works:

Change Detection: Debezium connects to PostgreSQL and monitors the database's transaction log to detect all data modifications. Event Streaming: Each database change is captured and transformed into an event message that includes:

- The operation type (op: create, update, delete)
- The data before the change (before)
- The data after the change (after)
- Metadata (timestamp, transaction info)

These change events are published to Kafka topics, making them available to multiple downstream consumers without impacting the source database. Applications can consume these change streams independently, enabling real-time reactions to data changes without directly querying the database

### Why this is relevant:

- Low Latency: Changes propagate within milliseconds, enabling real-time ML model inference and decision-making
- Event-Driven AI: ML systems can react to data changes immediately rather than waiting for batch ETL jobs Scalability & Decoupling
- Multiple Consumers: Multiple AI/ML systems can consume the same change stream independently (e.g., feature stores, training pipelines, inference engines)
- Database Protection: Prevents overwhelming the OLTP database with analytical queries from ML workloads
- Horizontal Scaling: Kafka's partitioning allows parallel processing of high-volume data streams Data Consistency & Reliability

- Guaranteed Delivery: Captures every change without data loss, crucial for training accurate AI models
- Temporal Ordering: Preserves the sequence of events, essential for time-series ML models
- Replayability: Historical change logs can be replayed for model retraining or debugging

### Use cases:

- Real-time ML Feature Stores
  - Database changes update feature values in real-time
  - ML models always access fresh features for inference
  - Example: Credit scoring models that need up-to-date customer transaction data
- Fraud Detection Systems
  - Transaction changes trigger immediate fraud detection algorithms
  - Multiple ML models analyze the same transaction stream
  - Example: Banking systems detecting suspicious activities as they occur
- Recommendation Engines
  - User behavior changes (clicks, purchases) stream to recommendation models
  - Real-time personalization based on latest user actions
  - Example: E-commerce platforms updating product recommendations instantly
- Microservices Communication
  - Database changes in one microservice trigger events in others
  - Enables event-driven architectures for AI-powered applications
  - Example: Inventory updates triggering demand forecasting models

# Activity 2

## Scenario:

You run a temperature logging system in a small office. Sensors report the temperature once per minute and write the sensor readings into a PostgreSQL table

## Running instructions

It is recommended to run the scripts (e.g., `temperature_data_producer.py` file) in a Python virtual environments venv, basic commands from the `activity.streaming` folder:

```
python3 -m venv venv
source venv/bin/activate   # or venv\Scripts\activate on Windows
pip install --upgrade pip
pip install -r requirements.txt
```

Then one can run the python scripts.

## Characteristics:

Low volume (~1 row per minute)

Single consumer (reporting script)

No real-time streaming needed

# Part 1

In a simple use case where sensor readings need to be processed every 10 minutes to calculate the average temperature over that time window, describe which software architecture would be most appropriate for fetching the data from PostgreSQL, and explain the rationale behind your choice.

## Architecture: Simple Scheduled Batch Processing

For this low-volume temperature logging scenario, the most appropriate architecture is a simple scheduled batch job that directly queries PostgreSQL every 10 minutes

**Why not Kafka/Debezium CDC:**

- Volume Mismatch
  - Kafka/Debezium designed for high-throughput streaming
  - Massive over-engineering for the workload
- Infrastructure Overhead
  - Requires 3 additional services
  - Continuous resource consumption even when idle
  - More failure points and monitoring complexity
- Processing Pattern Mismatch
  - Need 10-minute batch aggregates, not real-time event processing
  - No benefit from sub-second latency capabilities

# Part 2

From the architectural choice made in `Part 1`, implement the solution to consume and processing the data generated by the `temperature_data_producer.py` file (revise its features!). The basic logic from the file `temperature_data_consumer.py` should be extended with the conection to data source defined in `Part 1`'s architecture..

```
python ./temperature_data_producer.py
Database office_db created.
Table ready.
2026-01-07 17:26:21.442094 - Inserted temperature: 29.7 °C
2026-01-07 17:27:21.459701 - Inserted temperature: 21.98 °C
2026-01-07 17:28:21.472986 - Inserted temperature: 28.38 °C
2026-01-07 17:29:21.485414 - Inserted temperature: 26.35 °C
2026-01-07 17:30:21.496384 - Inserted temperature: 25.49 °C
2026-01-07 17:31:21.503462 - Inserted temperature: 19.64 °C
2026-01-07 17:32:21.524653 - Inserted temperature: 18.58 °C
2026-01-07 17:33:21.543366 - Inserted temperature: 21.36 °C
2026-01-07 17:34:21.556207 - Inserted temperature: 25.72 °C
2026-01-07 17:35:21.568861 - Inserted temperature: 19.79 °C
2026-01-07 17:36:21.590253 - Inserted temperature: 18.48 °C
2026-01-07 17:37:21.606072 - Inserted temperature: 24.88 °C
2026-01-07 17:38:21.623544 - Inserted temperature: 27.3 °C
2026-01-07 17:39:21.639253 - Inserted temperature: 25.97 °C
```

```
2026-01-07 17:40:21.651498 - Inserted temperature: 26.5 °C
2026-01-07 17:41:21.666399 - Inserted temperature: 20.67 °C
```

```
import os
import subprocess
import sys
import time
import psycopg2
from datetime import datetime, timedelta

DB_NAME = "office_db"
DB_USER = "postgres"
DB_PASSWORD = "postgrespw"
DB_HOST = "localhost"
DB_PORT = 5432

# Step 1: Connect to default database
conn = psycopg2.connect(
    dbname=DB_NAME,
    user=DB_USER,
    password=DB_PASSWORD,
    host=DB_HOST,
    port=DB_PORT
)

# -------------------------
# Periodically compute average over last 10 minutes
# -------------------------
try:
    while True:
        ten_minutes_ago = datetime.now() - timedelta(minutes=10)
        ## Fetch the data from the choosen source (to be implemented)

        cursor = conn.cursor()
        query = """
            SELECT AVG(temperature) as avg_temperature
            FROM temperature_readings
            WHERE recorded_at >= %s;
        """
        cursor.execute(query, (ten_minutes_ago,))
        result = cursor.fetchone()
        conn.close()

        avg_temp = result[0] if result else None
        if avg_temp is not None:
            print(f"{datetime.now()} - Average temperature last 10 minutes:
{avg_temp:.2f} °C")
        else:
            print(f"{datetime.now()} - No data in last 10 minutes.")
        time.sleep(600)  # every 10 minutes
except KeyboardInterrupt:
```

```
    print("Stopped consuming data.")
finally:
    print("Exiting.")
```

```
python ./temperature_data_consumer.py
2026-01-07 17:30:41.027669 - Average temperature last 10 minutes: 26.38 °C
```

## Part 3

Discuss the proposed architecture in terms of resource efficiency, operability, and deployment complexity. This includes analyzing how well the system utilizes compute, memory, and storage resources; how easily it can be operated, monitored, and debugged in production.

### Resource Efficiency:

- Compute
  - Script runs only few seconds every 10 minutes
  - No continuous processes draining CPU
  - PostgreSQL handles aggregation natively
- Memory
  - Really low (to none when idle)
  - No message buffers or queues in memory
- Storage
  - Uses existing database storage
  - No duplicate data storage needed

### Operability:

- Monitoring
  - Simple log files show all activity
  - Standard PostgreSQL tools work
  - Single script to check
- Debugging
  - One Python file to troubleshoot
  - Run SQL queries manually to verify
  - No distributed system issues
- Operation
  - Start/stop with one command
  - Single configuration file
  - Restart on failure is trivial

### Deployment Complexity:

- Setup
  - Installing python environment
  - Setup autostart + python script

    - Runs on same machine as PostgreSQL
- Infrastructure
    - PostgreSQL
    - Autostart + python script

# Activity 3

## Scenario:

A robust fraud detection system operating at high scale must be designed to handle extremely high data ingestion rates while enabling near real-time analysis by multiple independent consumers. In this scenario, potentially hundreds of thousands of transactional records per second are continuously written into an OLTP PostgreSQL database (see an example simulating it with a data generator inside the folder `Activity3`), which serves as the system of record and guarantees strong consistency, durability, and transactional integrity. Moreover, the records generated are needed by many consumers in near real-time (see inside the folder `Activity3` two examples simulating agents consuming the records and generating alerts). Alerts or enriched events generated by these agents can then be forwarded to downstream systems, such as alerting services, dashboards, or case management tools.

## Running instructions

It is recommended to run the scripts in a Python virtual environments venv, basic commands from the `Activity3` folder:

```
python3 -m venv venv
source venv/bin/activate   # or venv\Scripts\activate on Windows
pip install --upgrade pip
pip install -r requirements.txt
```

Then one can run the python scripts.

## Characteristics:

High data volume (potentially hundreds of thousands of records per second)

Multiple consumer agents

Near real-time streaming needed

## Part 1

Describe which software architecture would be most appropriate for fetching the data from PostgreSQL and generate alerts in real-time. Explain the rationale behind your choice.

### Architecture: Debezium CDC with PostgreSQL and Kafka

For this high-volume fraud detection scenario with multiple independent consumers requiring near real-time alerts, the most appropriate architecture is **Debezium CDC (Change Data Capture) integrated with Apache Kafka**.

**Why Debezium CDC + Kafka:**

- High Throughput Handling
    - Kafka can easily handle hundreds of thousands of messages per second
    - Avoids overwhelming PostgreSQL with polling queries from multiple consumers
- Multiple Independent Consumers
    - Kafka's pub/sub model allows multiple fraud detection agents to consume the same stream independently
    - Each consumer maintains its own offset, enabling parallel processing without coordination
- Near Real-Time Processing
    - Change events propagate to Kafka within milliseconds
    - Fraud alerts can be generated within seconds of transaction occurrence
- Database Protection
    - CDC captures changes from PostgreSQL's transaction log, not through polling queries
    - OLTP database focus remains on transactional integrity, not analytical queries
    - Prevents consumer workloads from impacting transaction processing performance
- Event-Driven Architecture
    - Enables reactive fraud detection workflows triggered immediately upon transaction creation
    - Events contain before/after state, allowing agents to analyze transaction patterns
    - Facilitates alerts to be forwarded to downstream systems
- Fault Tolerance & Reliability
    - Kafka provides message durability and replayability
    - Failed consumers can replay missed messages from their last offset
    - Guarantees no transaction data is lost

**Why not Polling/Scheduled Batch Processing:**

- Introduces unacceptable latency
- Hundreds of thousands of polling queries would overload PostgreSQL
- Cannot detect fraud in real-time before transaction completion
- Waste of compute resources checking for changes repeatedly

# Part 2

From the architectural choice made in `Part 1`, implement the 'consumer' to fetch and process the records generated by the `fraud_data_producer.py` file (revise its features!). The basic logic from the files

`fraud_consumer_agent1.py.py` and `fraud_consumer_agent2.py.py` should be extended with the conection to data source defined in `Part 1`'s architecture.

Delete connector if already existant:

```
curl -X DELETE http://localhost:8083/connectors/fraud-detection-connector
```

Setup connector:

```
curl -i -X POST \
  -H "Accept:application/json" \
  -H "Content-Type:application/json" \
  localhost:8083/connectors/ \
  -d '{
    "name": "fraud-detection-connector",
    "config": {
      "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
      "tasks.max": "1",
      "database.hostname": "postgres",
      "database.port": "5432",
      "database.user": "postgres",
      "database.password": "postgrespw",
      "database.dbname": "mydb",
      "slot.name": "fraudslot",
      "topic.prefix": "fraud-detection",
      "plugin.name": "pgoutput",
      "database.replication.slot.name": "debeziumfraud",
      "table.include.list": "public.transactions"
    }
  }'
```

Run data producer:

```
python ./fraud_data_producer.py
Inserted 1000 transactions...
Inserted 1000 transactions...
Inserted 1000 transactions...
Inserted 1000 transactions...
Inserted 1000 transactions...
Inserted 1000 transactions...
Inserted 1000 transactions...
Inserted 1000 transactions...
Inserted 1000 transactions...
Inserted 1000 transactions...
```

```
# This agent calculates a running average for each user and flags transactions
that are significantly higher than their usual behavior (e.g., $3\sigma$
outliers).

import json
import statistics
import base64
from kafka import KafkaConsumer

# Configuration
KAFKA_BOOTSTRAP_SERVERS = ['localhost:9094']
KAFKA_TOPIC = 'fraud-detection.public.transactions'
CONSUMER_GROUP = 'fraud-detection-agent1'

# In-memory store for user spending patterns
user_spending_profiles = {}

def decode_decimal(encoded_bytes):
    """Decode Debezium DECIMAL bytes to float"""
    if isinstance(encoded_bytes, str):
        try:
            # Try base64 decode first
            decoded = base64.b64decode(encoded_bytes)
            # Convert bytes to integer (big-endian)
            value = int.from_bytes(decoded, byteorder='big', signed=True)
            # Apply scale of 2 (DECIMAL(10,2))
            return float(value) / 100
        except:
            return None
    elif isinstance(encoded_bytes, (int, float)):
        return float(encoded_bytes)
    return None

def analyze_pattern(data):
    user_id = data['user_id']
    amount = decode_decimal(data['amount'])

    if user_id not in user_spending_profiles:
        user_spending_profiles[user_id] = []

    history = user_spending_profiles[user_id]

    # Analyze if transaction is an outlier (Need at least 3 transactions to judge)
    is_anomaly = False
    if len(history) >= 3:
        avg = statistics.mean(history)
        stdev = statistics.stdev(history) if len(history) > 1 else 0

        # If amount is > 3x the average (Simple heuristic)
        if amount > (avg * 3) and amount > 500:
            is_anomaly = True

    # Update profile
```

```
    history.append(amount)
    # Keep only last 50 transactions per user for memory efficiency
    if len(history) > 50: history.pop(0)

    return is_anomaly

print("🧬 Anomaly Detection Agent started...")

consumer = KafkaConsumer(
        KAFKA_TOPIC,
        bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
        group_id=CONSUMER_GROUP,
        value_deserializer=lambda m: json.loads(m.decode('utf-8')),
        auto_offset_reset='earliest', # start reading from the beginning
        enable_auto_commit=True
    )

for message in consumer: #consumer has to be implemented before!
    payload = message.value.get('payload', {})
    data = payload.get('after')

    if data:
        # Match the variable name here...
        is_fraudulent_pattern = analyze_pattern(data)

        # ...with the variable name here
        if is_fraudulent_pattern:
            print(f"🚨 ANOMALY DETECTED: User {data['user_id']} spent
${decode_decimal(data['amount'])} (Significantly higher than average)")
        else:
            print(f"📊 Profile updated for User {data['user_id']}")
```

Run agent 1:

```
python ./fraud_consumer_agent1.py
🧬 Anomaly Detection Agent started...
📊 Profile updated for User 3280
📊 Profile updated for User 4826
📊 Profile updated for User 4197
📊 Profile updated for User 2239
📊 Profile updated for User 1622
📊 Profile updated for User 2687
📊 Profile updated for User 1363
📊 Profile updated for User 9835
📊 Profile updated for User 3315
📊 Profile updated for User 8526
📊 Profile updated for User 5354
📊 Profile updated for User 2306
📊 Profile updated for User 2179
📊 Profile updated for User 5979
📊 Profile updated for User 8021
🚨 ANOMALY DETECTED: User 9157 spent $3927.52 (Significantly higher than average)
```

```
📊 Profile updated for User 7651
📊 Profile updated for User 2599
📊 Profile updated for User 1721
📊 Profile updated for User 6357
📊 Profile updated for User 1904
📊 Profile updated for User 5256
📊 Profile updated for User 1126
📊 Profile updated for User 2582
```

```
#This agent uses a sliding window (simulated) to perform velocity checks and score
the transaction
import json
from collections import deque
import time
import base64
from kafka import KafkaConsumer

# Configuration
KAFKA_BOOTSTRAP_SERVERS = ['localhost:9094']
KAFKA_TOPIC = 'fraud-detection.public.transactions'
CONSUMER_GROUP = 'fraud-detection-agent2'

# Simulated In-Memory State for Velocity Checks.
user_history = {}

def decode_decimal(encoded_bytes):
    """Decode Debezium DECIMAL bytes to float"""
    if isinstance(encoded_bytes, str):
        try:
            # Try base64 decode first
            decoded = base64.b64decode(encoded_bytes)
            # Convert bytes to integer (big-endian)
            value = int.from_bytes(decoded, byteorder='big', signed=True)
            # Apply scale of 2 (DECIMAL(10,2))
            return float(value) / 100
        except:
            return None
    elif isinstance(encoded_bytes, (int, float)):
        return float(encoded_bytes)
    return None

def analyze_fraud(transaction):
    user_id = transaction['user_id']
    amount = decode_decimal(transaction['amount'])

    # 1. Velocity Check (Recent transaction count)
    now = time.time()
    if user_id not in user_history:
        user_history[user_id] = deque()

    # Keep only last 60 seconds of history
```

```
    user_history[user_id].append(now)
    while user_history[user_id] and user_history[user_id][0] < now - 60:
        user_history[user_id].popleft()

    velocity = len(user_history[user_id])

    # 2. Heuristic Fraud Scoring
    score = 0
    if velocity > 5: score += 40  # Too many transactions in a minute
    if amount > 4000: score += 50 # High value transaction

    # 3. Simulate ML Model Hand-off
    # model.predict([[velocity, amount]])

    return score

consumer = KafkaConsumer(
    KAFKA_TOPIC,
    bootstrap_servers=KAFKA_BOOTSTRAP_SERVERS,
    group_id=CONSUMER_GROUP,
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    auto_offset_reset='earliest', # start reading from the beginning
    enable_auto_commit=True
)

print("Agent started. Listening for CDC events...")
for message in consumer:  #consumer has to be implemented before!
    # Debezium wraps data in an 'after' block
    payload = message.value.get('payload', {})
    data = payload.get('after')

    if data:
        fraud_score = analyze_fraud(data)
        if fraud_score > 70:
            print(f"⚠️ HIGH FRAUD ALERT: User {data['user_id']} | Score:
{fraud_score} | Amt: {decode_decimal(data['amount'])}")
        else:
            print(f"✅ Transaction OK: {data['id']} (Score: {fraud_score})")
```

Run agent 2:

```
python ./fraud_consumer_agent2.py
Agent started. Listening for CDC events...
✅ Transaction OK: 138919 (Score: 0)
✅ Transaction OK: 138920 (Score: 0)
✅ Transaction OK: 138921 (Score: 0)
✅ Transaction OK: 138922 (Score: 50)
✅ Transaction OK: 138923 (Score: 0)
✅ Transaction OK: 138924 (Score: 0)
✅ Transaction OK: 138925 (Score: 0)
✅ Transaction OK: 138926 (Score: 0)
✅ Transaction OK: 138927 (Score: 0)
```

```
☑ Transaction OK: 138928 (Score: 0)
⚠ HIGH FRAUD ALERT: User 7690 | Score: 90 | Amt: 4178.74
☑ Transaction OK: 138930 (Score: 0)
☑ Transaction OK: 138931 (Score: 50)
☑ Transaction OK: 138932 (Score: 0)
☑ Transaction OK: 138933 (Score: 0)
☑ Transaction OK: 138934 (Score: 0)
☑ Transaction OK: 138935 (Score: 0)
☑ Transaction OK: 138936 (Score: 50)
```

## Part 3

Discuss the proposed architecture in terms of resource efficiency, operability, maintainability, deployment complexity, and overall performance and scalability. This includes discussing how well the system utilizes compute, memory, and storage resources; how easily it can be operated, monitored, and debugged in production; how maintainable and evolvable the individual components are over time; the effort required to deploy and manage the infrastructure; and the system's ability to sustain increasing data volumes, higher ingestion rates, and a growing number of fraud detection agents without degradation of latency or reliability.

### Resource Efficiency:

- Compute
  - Debezium CDC runs continuously but low CPU footprint
  - Kafka broker requires sustained CPU for message handling
  - Multiple agents run continuously
  - PostgreSQL write load increases with high-volume transactions
  - Scales better than polling but uses more resources than Activity 2
- Memory
  - Kafka buffers messages in memory
  - Agents maintain in-memory state
  - Each agent stores last 50 transactions times thousands of users
  - More memory-intensive than batch processing
  - Manageable at current scale
- Storage
  - Kafka stores messages on disk
  - PostgreSQL write overhead from high-volume inserts
  - Additional storage for Kafka topic logs
  - More storage needed than Activity 2 query-only approach

### Operability:

- Monitoring
  - Debezium: REST API provides status endpoint
  - Kafka: CLI tools available but not centralized
  - Agents: Console logging only
  - No centralized dashboard or alerting
- Debugging
  - Multiple components to troubleshoot

    - Consumer group offset issues not obvious
    - No distributed tracing across components
    - Much harder than Activity 2 single Python script
- Operation
    - Start/stop requires multiple docker-compose commands
    - Connector registration requires manual curl commands
    - Consumer group reset requires CLI knowledge
    - Failures in one component cascade to others
    - More operationally complex than Activity 2

### Deployment Complexity:

- Setup
    - Docker Compose containers (PostgreSQL, Kafka, Connect)
    - Manual Debezium connector registration via curl
    - Python venv with multiple agent scripts
    - Database creation and table setup required
- Infrastructure
    - PostgreSQL
    - Kafka broker + Zookeeper
    - Kafka Connect with Debezium plugin
    - Multiple Python agent processes

## Part 4

Compare the proposed architecture to Exercise 3 from previous lecture where the data from PostgreSQL was loaded to Spark (as a consumer) using the JDBC connector. Discuss both approaches at least in terms of performance, resource efficiency, and deployment complexity.

### Architecture Overview:

- Debezium CDC + Kafka
    - PostgreSQL, Debezium CDC, Kafka, Multiple Python Agents
    - Real-time change capture via transaction log
- Spark JDBC Connector
    - PostgreSQL, Spark JDBC Polling, Spark Cluster
    - Periodic batch queries via JDBC connection

### Performance Comparison:

- Latency
    - Debezium CDC + Kafka: Real-time streaming
    - Spark JDBC: Batch interval dependent
- Throughput
    - Debezium CDC + Kafka: 100K+ tps sustained
    - Spark JDBC: Limited by polling frequency and database connection pool
- Database Load
    - Debezium CDC + Kafka: Minimal

    - Spark JDBC: High

### Resource Efficiency:

- Compute Resources
    - Debezium CDC + Kafka: Moderate continuous CPU
    - Spark JDBC: High continuous CPU (JVM-based Spark executors always running)

### Deployment Complexity:

- Infrastructure Components
    - Debezium CDC + Kafka: Postgres, Kafka, Connect
    - Spark JDBC: Postgres, Spark Master, Executors
- Operational Complexity
    - Setup Difficulty: Both high (distributed systems knowledge required)
    - Monitoring: CDC has REST API; Spark has UI dashboard
    - Debugging: CDC has 4-component flow; Spark has DAG tracing
    - Failure Recovery: CDC has automatic consumer offsets; Spark has manual checkpointing

### Use Cases:

- Fraud Detection (Kafka)
    - Real-time alerts
    - Multiple consumers
    - High volume
- Batch Analytics (Spark JDBC)
    - Complex transformations
    - Historical aggregations
    - Data lake integration

# Submission

Send the exercises' resolution on Moodle and be ready to shortly present your solutions (5-8 minutes) in the next Exercise section (14.01.2026).