Activity 1

What it does:

It impliments the Change Data Capture which we learned in the lecture

The Setup:

PostgreSQL is being setup, we make the database and create the table. (It should also be the OLTP Database here)

Afterwards we need to Start the Debezium Connector because otherwise it does not run. The connector is there to watch our PostgresDatabase.

The next step was to check if the Debezium really is running and if everything is going good.

Afterwards we insert something in our table and look at Kafka if we get a message (the message is the change we did so the insertion into our table, it does not need to be a row insertion, it can be any database change.)

The last code snipped (with the consume) prints out all the messages from the Kafka topic.

So basically the Debezium looks at the Write Ahead Log and captures all the changes to our database. These changes get transformed into events for our topic in Kafka, which is our Event Log. We therefore just get the new changes which can go into our OLAP Database for example and do not need to take the changes once a day, or anything like that. It should have the data/the changes in near real-time.

Why it is relevant for Software Arhcitecture for Big Data in the AI era and for which use cases

It is relevent as we get reliable continuous changes, near real time features and we get event based triggers. These are all things we can use in an AI system.

We could with it do Fraud detection in real time, anomily detection, incremental model training and we can also monitor how the data changes over time.

I think other use cases are Event-Driven Microservices and Real Time analytical dashboards (with sales or clicks or whatever is useful for the people who need it). These are just some examples I can right now think of, but I am sure there's a lot more use cases for it.

Activity 2

Part 1

Because we have a very simple database, with low volume, low velocity and from what I can guess not a lot of variety we should just stay simple and just use a DB to query the data.

We can make it simply with Postgres DB and query what we want to query, nothing else. So basically a Monolith, as all can be done with one application.

This makes it also easy to debug.

Also, because we do not get a lot of data we will not need to worry about the scaling for a while. If the time comes one can consider if we need all the data or if we can throw some away, otherwise we can buy a new drive (where we get better ones and cheaper ones often)

Part 2

Firstly I needed to change some stuff, I needed to change the requirements to psycopg[binary] as they did not work for me otherwise. I also with that changed some things in the temperature data producer. I needed to use this connector conn = psycopg.connect(f"dbname={DB_USER} user={DB_USER} password={DB_PASSWORD} host={DB_HOST} port={DB_PORT}")

With this all worked.

2026-01-07 22:15:29.594990 - Inserted temperature: 21.25 °C 2026-01-07 22:16:29.598403 - Inserted temperature: 23.5 °C 2026-01-07 22:17:29.603338 - Inserted temperature: 23.27 °C 2026-01-07 22:18:29.610040 - Inserted temperature: 23.7 °C 2026-01-07 22:19:29.616556 - Inserted temperature: 21.16 °C 2026-01-07 22:20:29.621052 - Inserted temperature: 21.44 °C 2026-01-07 22:21:29.624420 - Inserted temperature: 22.25 °C 2026-01-07 22:22:29.629495 - Inserted temperature: 23.81 °C 2026-01-07 22:23:29.630129 - Inserted temperature: 22.77 °C 2026-01-07 22:24:29.636665 - Inserted temperature: 24.43 °C

This should be the last 10 minutes of recording for me when I executed the other script.

```
        conn = psycopg.connect(
            f"dbname={DB_NAME} user={DB_USER} password={DB_PASSWORD} host={DB_HOST} port={DB_PORT}"
        )

        with conn.cursor() as cur:
            cur.execute("""
                SELECT AVG(temperature) as avg_temp
                FROM temperature_readings
                WHERE recorded_at >= %s
            """, (ten_minutes_ago,)) ## Querying the data so it should give me the average of the last 10 minutes

            result = cur.fetchone()
            avg_temp = result[0] if result and result[0] is not None else None

        conn.close()
```

this is the code with which I connected to the database and made my query to check the average temperature of the last 10 minutes and my result was:

2026-01-07 22:24:42.743646 - Average temperature last 10 minutes: 22.76 °C

(I got by calculating 22.758° so this is a correct result.)

This.

2026-01-07 22:24:42.743646 - Average temperature last 10 minutes: 22.76 °C 2026-01-07 22:34:42.779524 - Average temperature last 10 minutes: 24.21 °C

The code was specifically ran for more than 10 minutes to check if it still gives me an average temperature.

Part 3

This code is very easy to impliment, does not need a lot of computing power, is not complex at all (the most complex thing was getting it to run, which was fixed with a new version of it.) It would be quite easy to debug as well. I think this is a quite good implimentation for this problem. Maybe it can still be optimized but as of now it already runs with little resources, does not need a lot of storage and it runs smoothly. What more could one want from it. the little difference in the timestamps shows that it needs like not a lot of time to calculate it, but still a small amount, over a lot of hours it might be a minute difference, but I do not think this should be a huge issue.

Activity 3

Part 1

Because we have very high data volume, high velocity, multiple independent consumers and a near real-time requirement, I think a Change Data Capture + Event-Driven Streaming Architecture would be the best solution.

It would look somewhat like this: Postgres as my database (OLTP database), Debezium to read the changes directly from the Postgres Database with a Write-Ahead Log (this will be the CDC), Kafka as my Event Streaming Platform. The Agents will be, as already in the Scenario specified, our Fraud detection Service.

Why this?

Because Kafka can handle all of our hundreds of thousands of transaction, we need real time data so a CDC is necessary, Postgres only needs to be focused on the OLTP workload and not on calculating anything next to it and our Consumer groups can process the same data without affecting each other. So adding more would not need to require changes to our database or to the existing ones as far as I know.

Part 2

curl -i -X POST
-H "Accept:application/json"
-H "Content-Type:application/json"
http://localhost:8083/connectors/
-d '{ "name": "transaction-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", "topic.prefix": "dbserver1", "plugin.name": "pgoutput", "slot.name": "debezium_slot", "publication.name": "dbz_publication" } }'

docker exec kafka env | grep KAFKA KAFKA_CFG_PROCESS_ROLES=broker,controller KAFKA_CFG_NODE_ID=1 KAFKA_CFG_INTER_BROKER_LISTENER_NAME=INTERNAL KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka:9093 KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT,CONTROLLER:PLAINTEXT KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER KAFKA_CFG_LISTENERS=INTERNAL://:9092,EXTERNAL://:9094,CONTROLLER://:9093

KAFKA_KRAFT_CLUSTER_ID=kraft-cluster-1
KAFKA_CFG_ADVERTISED_LISTENERS=INTERNAL://kafka:9092,EXTERNAL://localhost:9094

With this I figured out that i need in my code 9094 and not 9092

amount = float(data['amount']) ^^^^^^^^^^^^^^^^^^^^^ ValueError: could not convert string to float: 'B5V5'

This told me i needed to change my code and with this it now worked.

```
# Configuration
KAFKA_BROKER = "localhost:9094"
TOPIC = "dbserver1.public.transactions"

# Kafka Consumer Setup
consumer = KafkaConsumer(
    TOPIC,
    bootstrap_servers=[KAFKA_BROKER],
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    auto_offset_reset='latest',
    group_id='anomaly-detection-group'
)

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

def decode_decimal(encoded_bytes):
    """Decode Debezium's binary decimal format"""
    if not encoded_bytes:
        return 0.0
    try:
        # Decode base64
        decoded = base64.b64decode(encoded_bytes)
        # Convert bytes to integer (big-endian)
        value = int.from_bytes(decoded, byteorder='big', signed=True)
        # Debezium uses scale=2, so divide by 100
        return value / 100.0
    except Exception as e:
        print(f"Error decoding amount: {e}")
        return 0.0
```

```
# Configuration
KAFKA_BROKER = "localhost:9094"
TOPIC = "dbserver1.public.transactions"

# Kafka Consumer Setup
consumer = KafkaConsumer(
    TOPIC,
    bootstrap_servers=[KAFKA_BROKER],
    value_deserializer=lambda m: json.loads(m.decode('utf-8')),
    auto_offset_reset='latest',
    group_id='fraud-detection-group'
)

# Simulated In-Memory State
user_history = {}

def decode_decimal(encoded_bytes):
    """Decode Debezium's binary decimal format"""
    if not encoded_bytes:
        return 0.0
    try:
        decoded = base64.b64decode(encoded_bytes)
        value = int.from_bytes(decoded, byteorder='big', signed=True)
        return value / 100.0
    except Exception as e:
        print(f"Error decoding amount: {e}")
        return 0.0
```

This is my code for the consumer agent 2 and my decode decimal to get the right values

The transactions worked really fast and nice, they had no real issue the whole time which was nice.

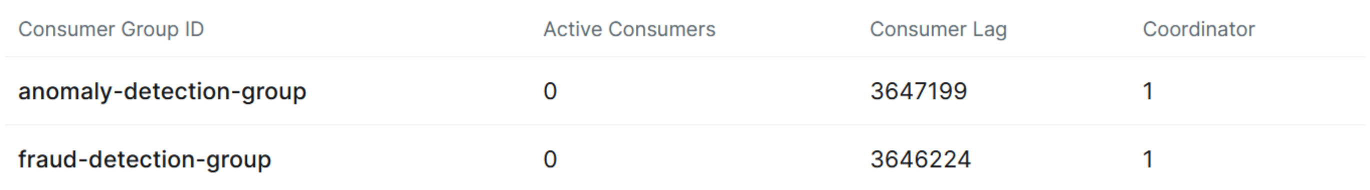
📊 Profile updated for User 2284 - Amount: $4093.71 📊 Profile updated for User 5581 - Amount: $4031.67 📊 Profile updated for User 3896 - Amount: $1819.35 🚨 ANOMALY DETECTED: User 4010 spent $3684.14 (Significantly higher than average) 📊 Profile updated for User 8190 - Amount: $1380.05 📊 Profile updated for User 3172 - Amount: $2303.57 📊 Profile updated for User 7349 - Amount: $1352.97 📊 Profile updated for User 5649 - Amount: $3708.57 📊 Profile updated for User 3212 - Amount: $542.19 📊 Profile updated for User 4948 - Amount: $4707.91 📊 Profile updated for User 5486 - Amount: $3829.94

There you can see the anomaly detection from the agent1

⚠ HIGH FRAUD ALERT: User 2354 | Score: 90 | Amt: $4488.34 ✅ Transaction OK: 39304 (Score: 40, Amt: $1331.43) ⚠ HIGH FRAUD ALERT: User 2603 | Score: 90 | Amt: $4281.49 ✅ Transaction OK: 39306 (Score: 40, Amt: $2752.43) ✅ Transaction OK: 39307 (Score: 50, Amt: $4936.67) ✅ Transaction OK: 39308 (Score: 0, Amt: $230.52) ✅ Transaction OK: 39309 (Score: 40, Amt: $3899.53) ✅ Transaction OK: 39310 (Score: 40, Amt: $797.75) ✅ Transaction OK: 39311 (Score: 0, Amt: $2179.72) ✅ Transaction OK: 39312 (Score: 0, Amt: $1400.69) ✅ Transaction OK: 39313 (Score: 0, Amt: $897.65) ✅ Transaction OK: 39314 (Score: 0, Amt: $3499.75) ✅ Transaction OK: 39315 (Score: 0, Amt: $23.22) ⚠ HIGH FRAUD ALERT: User 3085 | Score: 90 | Amt: $4786.87 ⚠ HIGH FRAUD ALERT: User 4131 | Score: 90 | Amt: $4260.21 ✅ Transaction OK: 39318 (Score: 0, Amt: $1799.31) ⚠ HIGH FRAUD ALERT: User 6541 | Score: 90 | Amt: $4894.28 ✅ Transaction OK: 39320 (Score: 0, Amt: $1772.88) ✅ Transaction OK: 39321 (Score: 0, Amt: $1296.47) ✅ Transaction OK: 39322 (Score: 0, Amt: $987.84) ✅ Transaction OK: 39323 (Score: 40, Amt: $704.37) ✅ Transaction OK: 39324 (Score: 0, Amt: $2608.86) ✅ Transaction OK: 39325 (Score: 40, Amt: $3188.53) ✅ Transaction OK: 39326 (Score: 0, Amt: $1752.43)

Here from Agent 2

which shows that it works.



Consumer Group ID	Active Consumers	Consumer Lag	Coordinator
anomaly-detection-group	0	3647199	1
fraud-detection-group	0	3646224	1

This is also from the UI for Apache Kafka from the consumers.

Part 3

Discussion of the proposed architecture

The proposed architecture is based on PostgreSQL + CDC (Debezium) + Kafka + multiple independent consumer agents. This design fits very well with the requirements of high throughput, near real-time processing, and multiple fraud detection consumers.

Resource Efficiency

- Compute
  - Compute resources are used efficiently because:
  - PostgreSQL only handles transactional writes and does not run analytical queries.
  - Fraud detection logic is executed in lightweight consumer services instead of a heavy analytics engine.
  - Each consumer only processes the data it needs.
- Memory
  - Consumers use in-memory state only for short-term history (sliding windows, recent averages).
  - Kafka handles buffering and backpressure, so consumers do not need to keep large amounts of data in memory.
  - Memory usage scales linearly with the number of active users per consumer.
- Storage
  - PostgreSQL stores only the transactional data.
  - Kafka stores events for a configurable retention period, which is much more efficient than storing intermediate results in databases.
  - No data duplication is required between consumers.

Overall, resources are well separated by responsibility, which avoids waste and contention.

Operability (Monitoring, Debugging, Day-2 Operations)

- Kafka provides:
  - Consumer offsets for tracking progress
  - Replayability for debugging or reprocessing
- Each fraud detection agent can be:
  - Logged independently
  - Restarted without affecting others

- CDC guarantees that no committed transactions are missed.

This makes the system easier to operate in production compared to polling-based or batch-based solutions.

Maintainability and Evolvability

- Each component has a single responsibility:
  - PostgreSQL → transactions
  - Debezium → change capture
  - Kafka → event distribution
  - Consumers → fraud logic
- Adding a new fraud detection strategy:
  - Does not require changes to the database
  - Does not affect existing consumers
- Fraud logic can evolve independently (rules → ML models → hybrid approaches).

Deployment Complexity

- The architecture is more complex than a monolith because it requires: Kafka, Debezium and Multiple services

- However:

  - All components are well-known, production-proven tools
  - The complexity is justified by the scale and requirements
  - Containerization (e.g., Docker Compose or Kubernetes) makes deployment manageable

So while deployment complexity is moderate to high, it is acceptable and common for real-time systems.

Performance and Scalability

- Ingestion scalability

  - Kafka can handle very high ingestion rates using partitions.

- Consumer scalability

  - Consumer groups allow horizontal scaling.

- Latency

  - CDC streams changes almost immediately after commit, enabling near real-time alerts.

- Reliability

  - Kafka guarantees ordered, durable event delivery.

The system can scale to:

- Higher transaction rates
- More fraud detection agents
- More complex fraud logic

- without significant degradation in latency or reliability.

Part 4

In Exercise 3 from the previous lecture, the data was loaded from PostgreSQL into Spark using the JDBC connector and then analyzed using Spark SQL. This approach works well for analytical workloads, but it is quite different from the CDC-based streaming architecture proposed in this activity.

With the Spark + JDBC approach, Spark connects directly to PostgreSQL and reads the table data in batches. This means that Spark usually needs to scan large parts of the table, which puts additional load on the database. The data is only as fresh as the last load, so the latency is much higher compared to streaming. For use cases like fraud detection, where decisions should be made almost immediately after a transaction happens, this batch-oriented model is not ideal.

In contrast, the CDC + Kafka architecture only processes incremental changes. Instead of repeatedly loading full tables, Debezium streams each committed transaction from the PostgreSQL WAL into Kafka. Consumers then process these events one by one in near real-time. This results in much lower latency and avoids unnecessary work on both the database and the consumers.

From a resource efficiency perspective, Spark is a heavy system that requires a lot of CPU and memory, which makes sense for complex aggregations, joins, and large analytical queries (as shown in the Spark notebook). However, for simple per-transaction fraud checks, this is often overkill. The streaming consumers used in this activity are lightweight, use little memory, and scale more naturally with the event rate.

Deployment complexity is also different. Running Spark requires a cluster, job configuration, and tuning, while the CDC-based approach requires Kafka and Debezium but allows fraud detection logic to run in small, independent services. Adding a new fraud detection agent in the streaming architecture is much easier than deploying a new Spark job.

Overall, the Spark JDBC approach is better suited for batch analytics and historical analysis, while the CDC-based architecture is clearly a better fit for near real-time fraud detection. Spark answers analytical questions about existing data, whereas CDC and Kafka enable reacting to events as they happen.