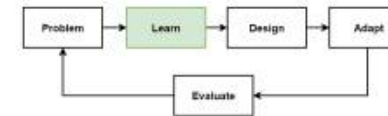


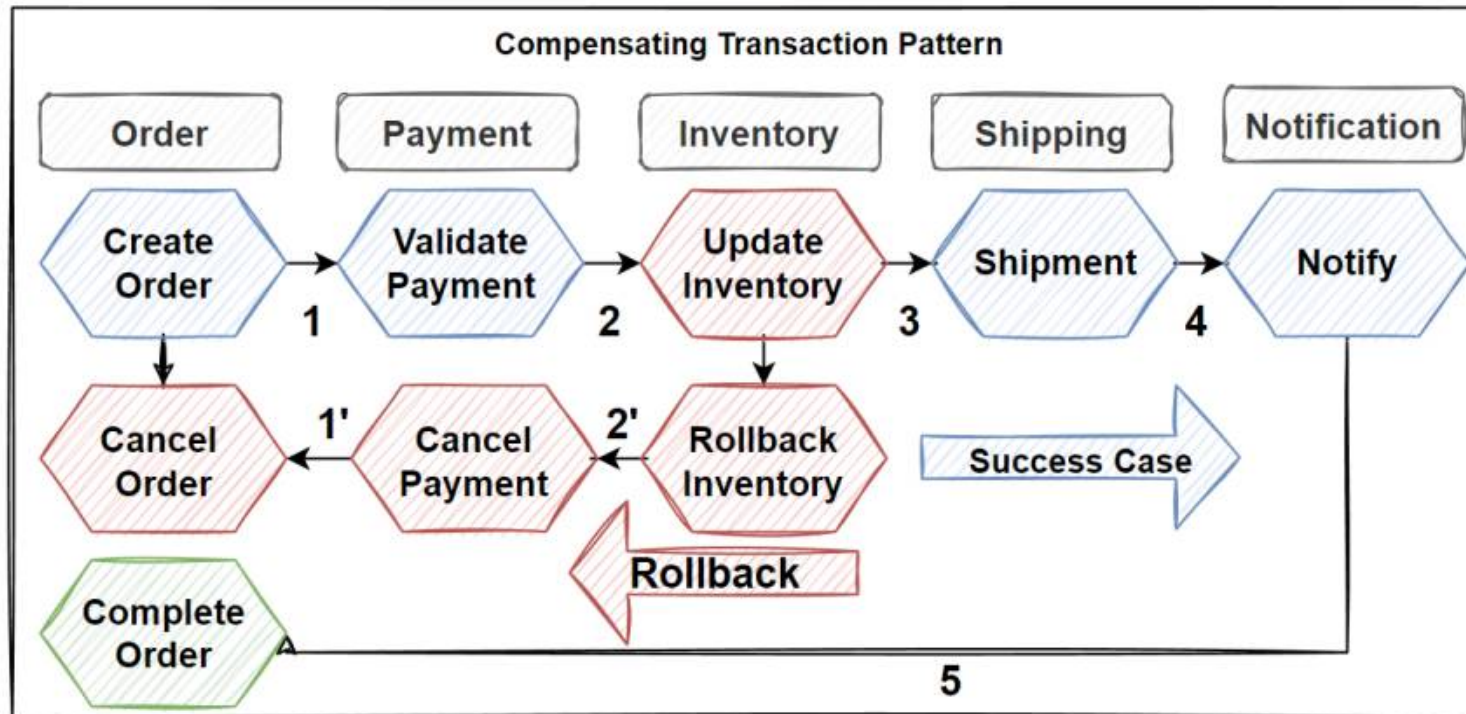
# Ecommerce Compensating Transactions Explained



## Ecommerce Order Fullfilment Compensating Transaction Pattern



- Customer places an order on the e-commerce application



Mehmet Ozkaya 520

Kindly explain this related to Compensating Transaction Pattern



This diagram illustrates an **Ecommerce Order Fulfillment process** using a **Compensating Transaction Pattern**. This pattern is particularly useful when dealing with distributed systems where multiple services (like order, payment, inventory, shipping, etc.) must work together, and failure in one service may require a rollback of prior actions to maintain consistency.

Here's the breakdown:

**1. Create Order** (Order Service):

- The customer initiates an order by placing it on the e-commerce application.

**2. Validate Payment** (Payment Service):

- Once the order is created, the system validates the payment. If this step fails, it triggers a **Cancel Order** process (shown as 1'), which rolls back the order.

**3. Update Inventory** (Inventory Service):

- After payment validation, the inventory is updated to reflect the purchased items. If there is a failure in inventory update, it triggers a **Cancel Payment** (2') and rolls back the inventory to its previous state.

**4. Shipment** (Shipping Service):

- After the inventory update, the system triggers the shipment of the product. This is a critical step in completing the order.

**5. Notify** (Notification Service):

- Once the shipment is successful, the customer is notified about the order status.

**6. Complete Order:**

- If all steps are successful, the system marks the order as complete, representing the successful case of the transaction.

## Compensating Transactions (Rollback)

If any part of the process fails (for example, if the payment fails or the inventory cannot be updated), the previous steps must be rolled back to maintain consistency. In this diagram, this is handled by the compensating transaction:

- **Cancel Order:** If the payment validation fails, the order creation is rolled back (1').
- **Cancel Payment:** If updating inventory fails, the payment validation is canceled (2').
- **Rollback Inventory:** If shipment fails, the inventory update is rolled back, undoing the changes to reflect the items back in stock.

The compensating transactions ensure that even in case of failure, the system can return to a consistent state by reversing the preceding operations.

This pattern is typically used in microservices architecture to maintain consistency in long-running transactions when it's difficult to lock resources across distributed services.

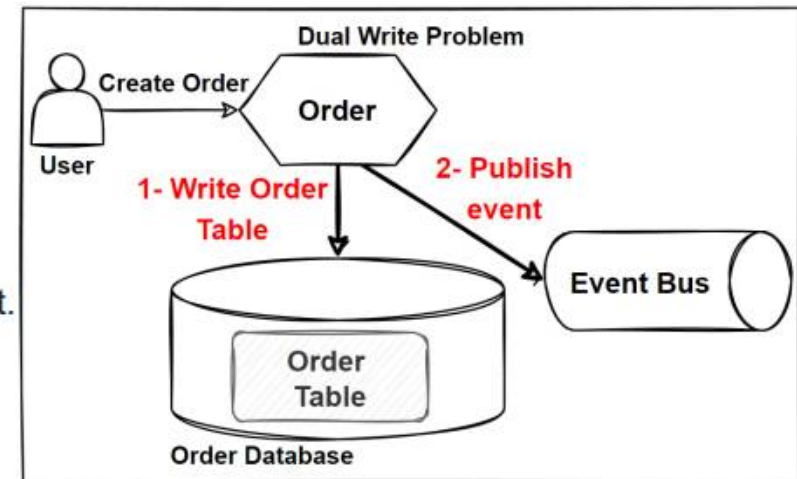


## Problem: Dual Write Problem

- When application **needs to change data in two different systems**, i.e. a database and a message queue, if one of the writes fails, it can result in inconsistent data.
- Happens when you use a **local transaction with each of the external systems** operations.
- I.e. app needs to **persist data in the database** and **send a message to Kafka** for notifying other systems.
- **If one of these two operations fails**, the **data** will be **inconsistent data** and these two systems becomes inconsistent.

### Problems

- Data loss or corruption.
- Difficult to resolve without proper error handling and recovery mechanisms.
- Dual writes can be hard to detect and fix.



Kindly explain in details related to dual write problemn



The diagram and description you provided highlight the **Dual Write Problem** in distributed systems. This problem occurs when an application needs to write data to two or more systems (for example, a database and a message queue) as part of a single transaction, but handles each write operation separately, often in the form of local transactions.

### Key points about the Dual Write Problem:

## 1. What is the Dual Write Problem?

- It happens when an application needs to update data in **two different systems**. If one of these operations fails while the other succeeds, it results in **inconsistent data**.
- For example, an e-commerce system needs to:
  - **Write to a database** (such as persisting order details).
  - **Send a message to a message queue** (such as Kafka) to trigger further actions like notifying the shipping system.

## 2. The Sequence of Operations:

- The first operation writes the order data to an **Order Table** in the database.
- The second operation publishes an **event to an event bus** (like Kafka) so that other services can take necessary actions.

Since these operations are **independent local transactions**, there's a risk if one fails and the other succeeds.

## 3. What Happens When It Fails?

- If one of the two operations fails, the data in the two systems (the database and the message bus) becomes inconsistent.
  - For example, if the order is written to the database but fails to publish the event to Kafka, the downstream services (like shipping or inventory) may never receive the message, leading to incorrect application behavior.
- Similarly, if the message is published successfully but the database write fails, other services might be notified of a non-existent or incomplete order.

## 4. Problems Caused by Dual Writes:

- **Data Loss or Corruption:** One system may believe that the transaction is successful, while the other system has no record of it, leading to a loss or corruption of data.
- **Difficult Recovery:** Without robust error handling and recovery mechanisms, it can be challenging to detect and fix these inconsistencies.
- **Hard to Detect:** Dual write failures might go unnoticed initially, but can later cause severe issues as other parts of the system rely on the incomplete or incorrect data.

## Why Does This Happen?

This issue arises because each write (to the database and message queue) is done in a **local transaction** (i.e., each system handles its operation independently without knowing about the other). There is no **global transaction management** to ensure both actions either succeed or fail together. This makes the system prone to inconsistency if one part of the operation fails.

## Solutions to the Dual Write Problem:

1. **Transactional Outbox Pattern:** In this pattern, instead of writing directly to both systems, the application writes all changes (like both the database update and the message) to a single local database table (outbox). A separate service then reads from the outbox and ensures that both operations (writing to the database and publishing the message) are done reliably.
2. **Two-Phase Commit (2PC):** This is a distributed transaction protocol that ensures both operations (writing to a database and publishing to a message queue) either both commit or both rollback. However, 2PC can be slow and is not always practical for high-performance systems.
3. **Eventual Consistency:** Instead of trying to synchronize everything immediately, systems embrace eventual consistency, where a compensating transaction corrects the state if an inconsistency is detected later.

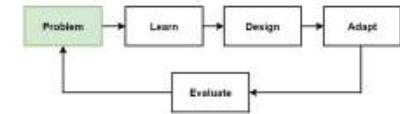
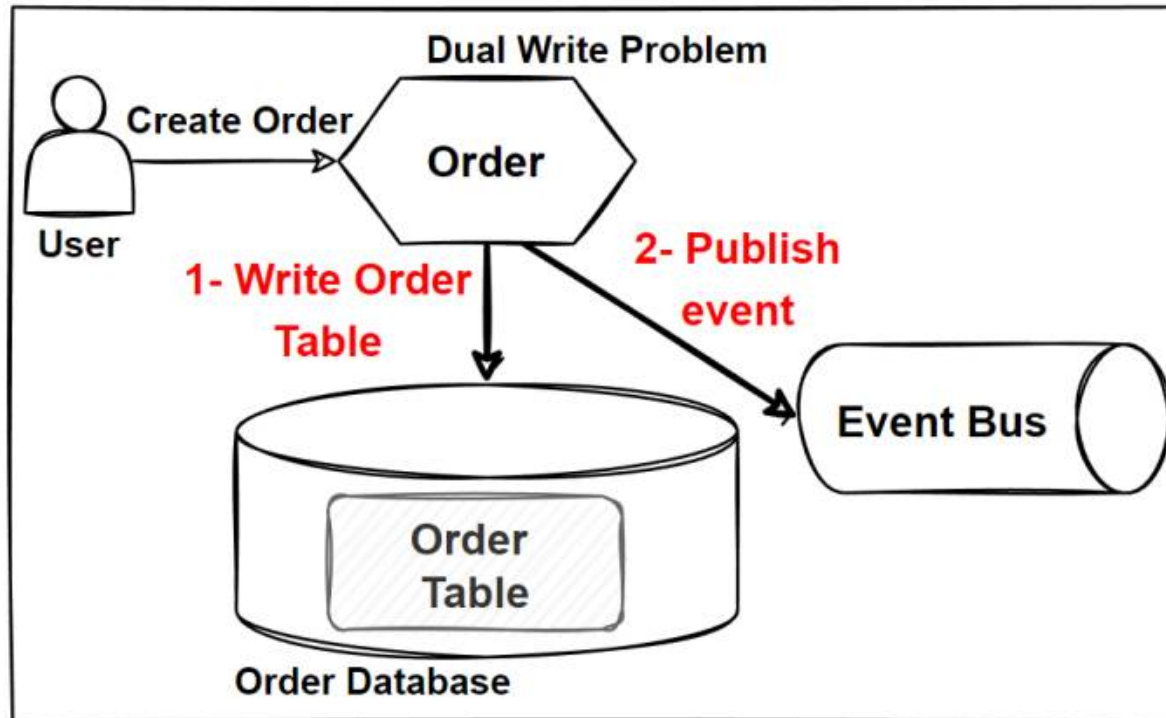
By understanding the dual write problem and using appropriate patterns, developers can avoid data inconsistencies in distributed systems.





## Example of Dual-write problem: Create Order

- 1- Change data in Order database with creating new record
- 2- Send an order\_created event to the EventBus like Apache Kafka.



Run Transactions

- 1, 2 or
- 2, 1
- Both becomes dual writes

Mehmet Ozkaya 522

# How to avoid dual write problems in microservices ?

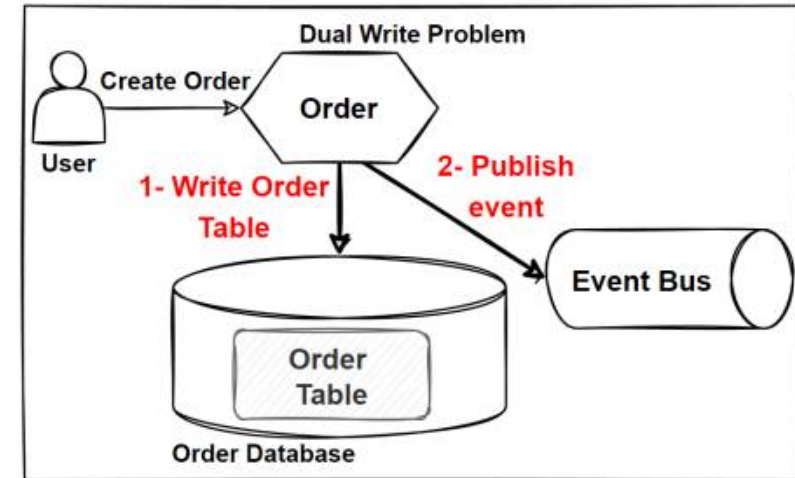
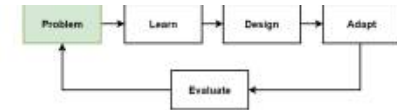
- **Monolith** applications use the **2 phase commit** protocol.
- It **splits the commit** process of the transaction into 2 steps and ensures the **ACID principles** for all systems.
- **Can't use 2-phase commit** transactions when building **microservices**.
- These transactions **require locks** and **don't scale well**.
- Need all systems to be **up and running** at the same time.

## Solutions

- Transactional Outbox Pattern
- CDC - Change Data Capture

## Best Practice

- Use Red Hat **Apache Kafka** and **CDC** using **Debezium** in event-driven applications.
- Use New databases like **CockroachDB** which has built-in **Change Data Capture** feature.



Mehmet Ozkaya 523

Kindly explain the above images in details



The two images expand on the **Dual Write Problem** with an example and potential solutions, especially in the context of microservices.

## Image 1: Example of Dual Write Problem – Create Order



### 1. Scenario:

- A user creates an order in an e-commerce system.
- There are **two operations** happening:
  - **Writing the order to the database** (Order Table).
  - **Publishing an event** (to an Event Bus like Apache Kafka), notifying other systems of the new order.

### 2. Dual Write Issue:

- These two operations are handled as **separate transactions**.
- If the first step (writing the order to the database) succeeds but the second step (publishing to the event bus) fails, or vice versa, it leads to an **inconsistent state**.
  - For example, the order might be recorded in the database, but the downstream systems that rely on the event might not know about it.
- The operations can happen in different orders (1 followed by 2, or 2 followed by 1), but both suffer from the **dual write problem**.

### 3. Problem Description:

- There's no **guarantee of atomicity** (i.e., both operations succeeding together or failing together).
- This lack of synchronization between systems causes **inconsistencies**, such as lost messages or mismatched data.

## Image 2: Avoiding Dual Write Problems in Microservices

This image outlines strategies to avoid the dual write problem, especially when building microservices.

### 1. 2-Phase Commit (2PC) in Monoliths:

- In traditional **monolithic applications**, a 2-phase commit protocol is often used to ensure that multiple systems (e.g., a database and a message queue) can **commit or roll back** together.
- This protocol splits the transaction commit into two steps, ensuring that all systems uphold the **ACID** (Atomicity, Consistency, Isolation, Durability) principles.
- However, **2PC cannot be used effectively** in microservices due to:
  - **Locking issues**: 2PC requires locks to coordinate between systems, which can slow down the system.
  - **Scalability issues**: Locking makes it difficult to scale across distributed systems.
  - **System availability**: 2PC requires all systems to be online and available at the same time.

### 2. Solutions:

- **Transactional Outbox Pattern:**
  - This is a design pattern where the system first writes both the database changes and the events (to be published) to an **outbox table** in the same database transaction.
  - A separate process or service (a **poller**) reads from the outbox and publishes the event to Kafka (or another event bus). This ensures that both the database write and event publish are handled reliably.
- **CDC (Change Data Capture):**
  - CDC is a mechanism to capture and record all the changes happening in the database and stream these changes to other systems.
  - **Debezium** is one popular tool that implements CDC, allowing changes in a database to be captured and sent as events in real-time to systems like Kafka.

### 3. Best Practices:

- Use **Apache Kafka** with **CDC** (like Debezium) for event-driven microservices.
- Consider using modern databases like **CockroachDB**, which has **built-in CDC features** that allow seamless capturing and streaming of database changes without requiring additional tools or patterns.

---

## Summary:

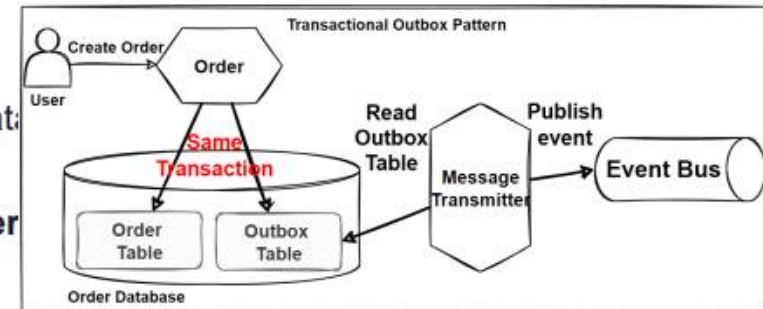
The **dual write problem** is particularly challenging in distributed microservice architectures, where the lack of atomic transactions between systems like databases and message queues can cause inconsistencies. Traditional 2-phase commit is not suitable for microservices, so patterns like **Transactional Outbox** and **Change Data Capture (CDC)** are recommended to maintain data consistency across distributed systems.

By using these best practices and tools like **Kafka**, **Debezium**, and **CockroachDB**, developers can effectively manage consistency and avoid the pitfalls of dual writes in modern microservice environments.

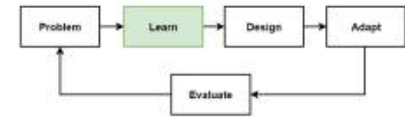


# Transactional Outbox Pattern

- The idea is to have an “**Outbox**” table in the microservice’s database. It provides to publish events reliably.
- **Dual write problem** happens when application needs to change data in two different systems.
- Instead of sending the data to two separate locations, **send a single transaction** that will **store two separate copies** of the data on the database.
- **One copy** is **stored** in the relevant **database table**, and the **other copy** is **stored** in an **outbox table** that will publish to event bus.
- When API **publishes event messages**, it doesn’t directly send them, Instead, the **messages** are **persisted** in a **database table**.
- After that, **a job publish events** to message broker system in predefined time intervals.
- **Events** are **not written directly** to a event bus, it is written to a table in the “**Outbox**” role of the service.

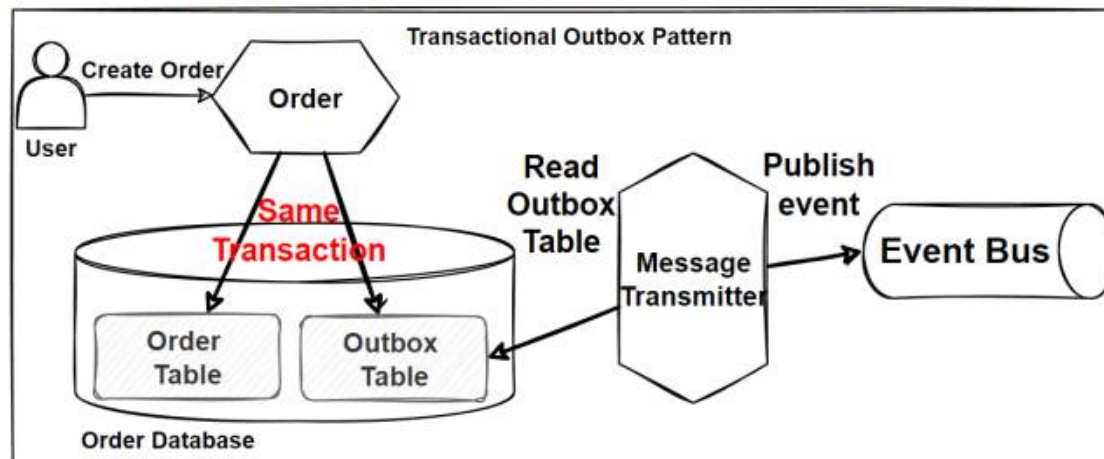


Mehmet Ozkaya 524



## Transactional Outbox Pattern - 2

- Transaction **performed before the event** and the **event written to the outbox table** are part of the **same transaction**.
- When a **new order is added** to the system, the process of adding the order and writing the **Order\_Created event** to the **Outbox table** is done **in the same transaction** to ensure the event is saved to the database.
- If one of the **process is fail**, this will **rollback the whole operations** with following ACID principles.
- The second step is to **receive these events** written to the **Outbox table** by an **independent service** and **write** them to the **Event bus**. Another service **listen** and **polls the Outbox table** records and publish events.



Mehmet Ozkaya 525

Kindly explain the above in details



The images explain the **Transactional Outbox Pattern**, a common solution to the **Dual Write Problem** in microservices architectures. Let's break down the explanation based on the two images provided.

## Image 1: Transactional Outbox Pattern – Overview

### 1. The Problem:

- The **Dual Write Problem** occurs when an application must write data to two separate systems (e.g., a database and an event bus like Kafka) and one of the writes fails, causing data inconsistencies.

### 2. The Idea of the Outbox Pattern:

- The solution to this problem is to use an **outbox table** within the microservice's database to reliably handle events.
- Instead of writing to the database and the event bus in two separate operations, **a single transaction** writes to both:
  - One part writes the relevant data to the **database table**.
  - The other part writes an event into an **outbox table** (also within the same database).

### 3. How it Works:

- The API first **persists event messages** in the **outbox table** instead of sending them directly to the event bus.
- A **job or a service** runs in the background to read from this outbox table and publishes the event to the event bus (Kafka, for example).
- This way, events are not written directly to the event bus in real-time but are handled **asynchronously** by the background job after the database operation is confirmed successful.

### 4. Advantage:

- Both the **database write and event creation** are part of the same atomic transaction. If either part fails, the transaction is rolled back, ensuring consistency and eliminating the dual write problem.

## Image 2: Transactional Outbox Pattern – Detailed Workflow

### 1. Transaction Execution:

- The key point is that the transaction which performs the business operation (like creating an order) and the process of writing an event to the outbox table are handled within the **same transaction**.
- For instance, when an order is created, the order data is saved to the **order table**, and an event describing this operation is written to the **outbox table**. Both actions occur as part of the same database transaction.

### 2. Rollback Mechanism:

- If any part of the transaction (either writing to the order table or the outbox table) fails, the entire operation is **rolled back**. This ensures that the database remains consistent, and no incomplete data is saved.
- This design follows the **ACID principles**: Atomicity, Consistency, Isolation, Durability.

### 3. Event Publishing:

- The next step involves a **separate service** (often called a **Message Transmitter**) that reads events from the **outbox table**.
- This service periodically **polls** the outbox table for new events and publishes these events to the event bus (Kafka, RabbitMQ, etc.).
- This separation of concerns ensures that the **order creation** (or any business logic) is not tied to the **event publishing** process. The events can be published in a **reliable and asynchronous manner**.

#### 4. How It Solves the Problem:

- Since both the database write and the event generation are part of the same transaction, if something fails, nothing is committed to either the database or the outbox, preventing inconsistencies.
- The asynchronous job can retry event publishing independently if something goes wrong during the event transmission.

---

### Summary of the Transactional Outbox Pattern:

The **Transactional Outbox Pattern** is a design pattern used to ensure consistency when a microservice needs to write data to both a **database** and an **event bus** (or other systems). By using an **outbox table** to store events as part of the same transaction that writes to the database, the pattern ensures that both actions (writing data and emitting events) either succeed together or fail together. This pattern avoids the dual write problem and allows for a reliable, consistent microservices architecture.

The process involves:

- Writing the event to the outbox table during the same transaction that modifies the database.
- A separate, independent service that reads from the outbox table and publishes the events to the event bus asynchronously.

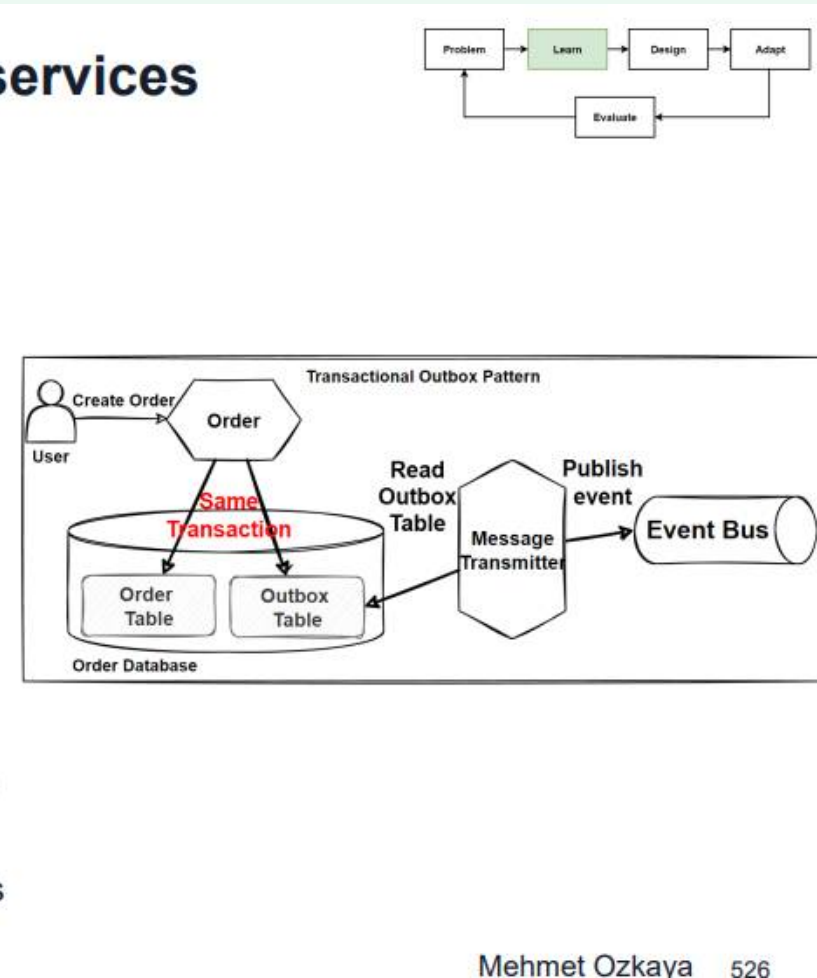
This pattern ensures **data integrity**, **atomicity**, and **reliable event publishing**, especially in distributed systems where direct dual writes to multiple systems could lead to inconsistencies.





# Transactional Outbox Pattern in Microservices

- **Microservice** provides an **outbox table** within its database. Outbox table will **include** all the **events**.
- There will be a **CDC (change data capture) plugin** that **reads** the **commit log** of the **outbox table** and **publish** the **events** to the relevant queue.
- It provides that **messages** are **reliably delivered** from a **microservice** to **another microservice** even if the transaction that triggered the message fails.
- It involves **storing** the **message** in a local "**Outbox**" table within the microservice, that message sent to the consumer after the transaction is committed.
- **Outbox pattern** can be used to ensure that **messages** are **delivered consistently**, even if the microservice that sent the message is **unavailable** or **experiencing errors**.
- Useful for **communicating important information** or updates between services.



Kindly explain below in details:



The image provides an explanation of how the **Transactional Outbox Pattern** is used in **microservices** to address the dual write problem and ensure **reliable event delivery** across different microservices.

## Breakdown of the Explanation:

### 1. Outbox Table within the Microservice:

- Each microservice maintains an **outbox table** within its own database.
- This table stores **all the events** that need to be published (e.g., new orders, updated inventory, etc.).
- Instead of sending the event directly to the event bus (like Kafka), the event is first recorded in the **outbox table** during the same transaction that updates the business data (such as writing the order to the order table).

### 2. CDC (Change Data Capture) Plugin:

- A **Change Data Capture (CDC)** tool (such as **Debezium**) is used to monitor changes to the **outbox table**.
- The CDC tool reads the **commit log** (which records all changes to the database) and identifies when new events are added to the outbox table.
- The CDC plugin then publishes these events to the relevant **event queue** or message broker (e.g., Kafka, RabbitMQ, etc.).
- This decouples the process of writing business data from publishing events, improving the system's reliability.

### 3. Reliable Event Delivery:

- The transactional outbox ensures that events are **reliably delivered** from one microservice to another, even if the transaction that triggered the message fails or if there are transient issues.
- Since the event is written to the outbox table within the same transaction as the business operation (e.g., creating an order), both operations succeed or fail together. This ensures **consistency**.

### 4. How It Works:

- When a message (event) needs to be published, instead of sending it immediately, the message is **stored in the outbox table**.
- Once the **transaction is committed**, the **message transmitter** (or CDC plugin) will read the outbox table and send the message to the event bus.
- This approach ensures that events are only sent to the message broker after the transaction is successful.

### 5. Handling Errors and Unavailability:

- The outbox pattern can handle situations where the microservice responsible for sending messages (the one that originally created the event) is **unavailable** or experiencing errors.
- Since the events are stored in the **outbox table**, they are persisted and can be published when the service or the message broker becomes available again.

### 6. Use Case:

- The **Transactional Outbox Pattern** is particularly useful when **communicating important information** between services, ensuring that messages (events) are delivered **consistently** even in the face of errors, microservice failures, or communication issues between services.
- For instance, when a new order is created, the order creation event is stored in the outbox table, and even if there's a delay or issue in sending the event to the event bus, it will eventually be sent, ensuring downstream systems are notified.

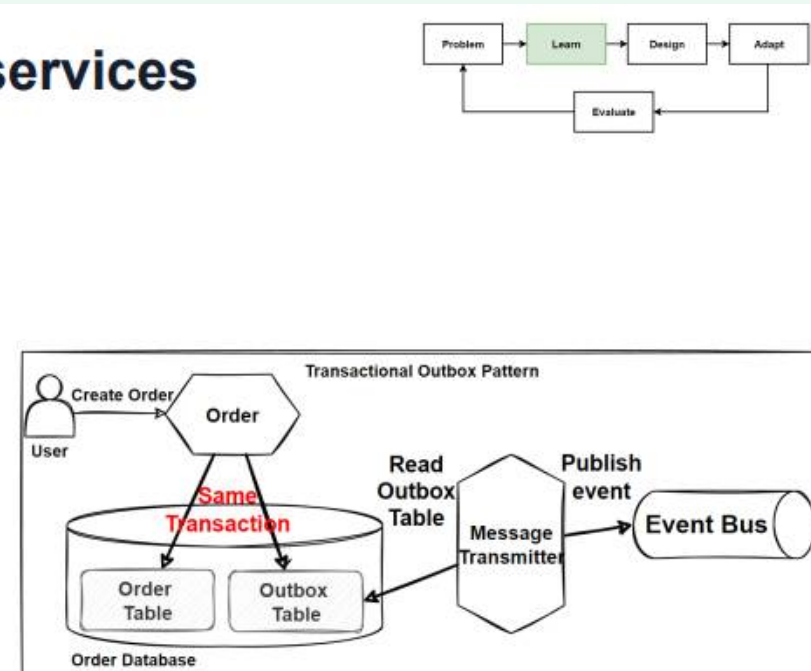
## Summary:

The **Transactional Outbox Pattern** solves the **dual write problem** in microservices by storing events in a dedicated **outbox table** in the same transaction as the business operation (e.g., writing order data). A **CDC plugin** or **message transmitter** then reliably reads the outbox and sends the events to an event bus like Kafka. This ensures **eventual consistency** and **reliable communication** between microservices, even if there are failures or delays in communication. This pattern is highly effective in distributed systems where consistent event propagation is crucial.



# Transactional Outbox Pattern in Microservices

- **Microservice** provides an **outbox table** within its database. Outbox table will **include** all the **events**.
- There will be a **CDC (change data capture) plugin** that **reads** the **commit log** of the **outbox table** and **publish the events** to the relevant queue.
- It provides that **messages** are **reliably delivered** from a **microservice** to **another microservice** even if the transaction that triggered the message fails.
- It involves **storing the message** in a local **"Outbox"** table within the microservice, that message sent to the consumer after the transaction is committed.
- **Outbox pattern** can be used to ensure that **messages** are **delivered consistently**, even if the microservice that sent the message is **unavailable** or **experiencing errors**.
- Useful for **communicating important information** or updates between services.



Mehmet Ozkaya 526

Kindly explain above image in details



The image explains how the **Transactional Outbox Pattern** is implemented in microservices to ensure reliable communication between services. Let's go through the details:

## Key Components of the Transactional Outbox Pattern in Microservices:

### 1. Outbox Table:

- Each **microservice** includes an **outbox table** within its **own database**.
- This table holds all the **events** that need to be published to other services (e.g., a new order event).
- Instead of publishing the event directly to the event bus (e.g., Kafka or RabbitMQ), the event is first stored in the outbox table as part of the same database transaction that involves updating business data (like creating an order).

### 2. CDC (Change Data Capture) Plugin:

- A **CDC tool** (like **Debezium**) is used to monitor changes in the outbox table.
- The **CDC plugin** reads the **commit log** from the outbox table and publishes the recorded events to the appropriate **message queue** (event bus).
- This decouples the process of modifying data and publishing events, improving system reliability.

### 3. Reliable Event Delivery:

- The main benefit of this pattern is that it ensures that events are **reliably delivered** from one microservice to another.
- Even if the transaction that triggered the event fails or there are issues with the event bus, the event is stored persistently in the outbox table and can be processed later when the system is available again.

### 4. Local Outbox Table:

- Events are **stored locally** within the microservice's **outbox table**.
- After the **business transaction is committed**, a **message transmitter** (or CDC plugin) reads from the outbox and sends the event to the **event bus**.
- This separation between the business transaction and the event publication ensures that both succeed or fail together, maintaining consistency.

### 5. Handling Service Unavailability:

- The **outbox pattern** is designed to handle scenarios where the microservice or the message bus is **unavailable** or experiencing **errors**.
- Even if the service fails to publish the message immediately, it remains stored in the **outbox table** and can be processed later by the message transmitter.

### 6. Consistency and Error Handling:

- The **outbox pattern** guarantees that messages (or events) are **delivered consistently**, even if the microservice or message bus encounters issues.

- The event will not be lost or ignored, as it is already recorded in the outbox table. This is crucial for maintaining **data integrity** and preventing missed or duplicated messages.

## 7. Use Cases:

- The **Transactional Outbox Pattern** is particularly useful for **communicating important information** or updates between services.
- For example, if a new order is created in an e-commerce system, this event (order creation) can be saved in the outbox table and later sent to downstream services like inventory or shipping through the event bus.

---

## Summary:

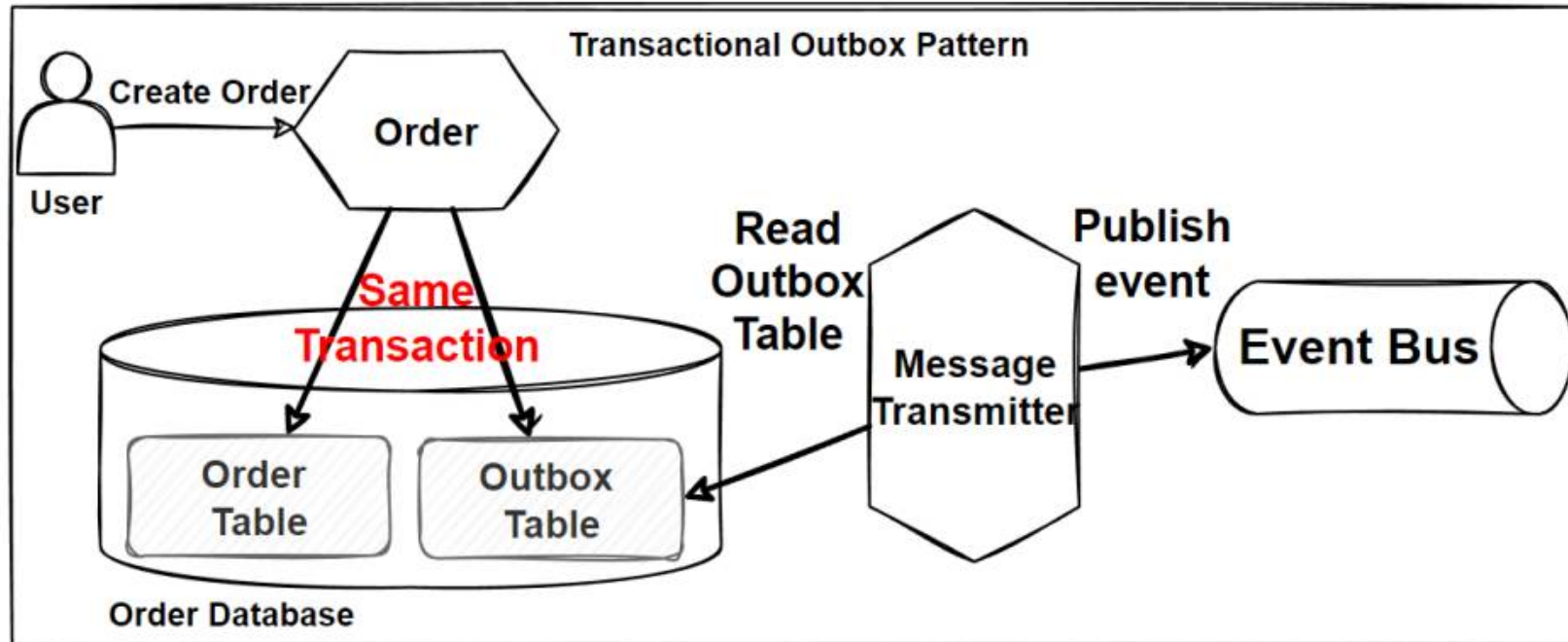
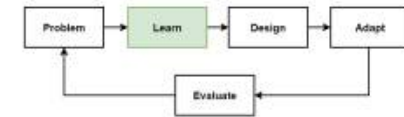
The **Transactional Outbox Pattern** in microservices solves the **dual write problem** by using a dedicated **outbox table** within the microservice's database to store events. A **CDC tool** reads from this outbox and reliably publishes the events to an **event bus** or **message queue**. This pattern ensures **consistent and reliable event delivery** even in the face of microservice or message broker unavailability, making it a robust solution for distributed systems where different services need to communicate via event-based messaging.





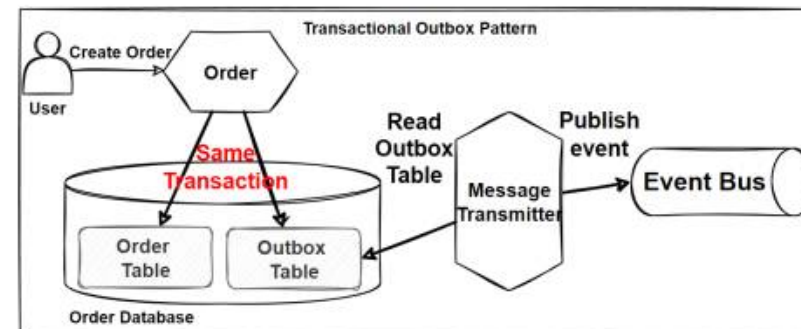
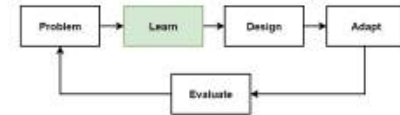
## Customer Places an Order Use Case

### Transactional Outbox pattern in Microservices



## Why Use Outbox Pattern ?

- When **working with critical data** that **need to consistent** and **need to accurate** to catch all requests.
- When the **database update** and **sending of the message** should be **atomic** to make sure **data consistency**.
- For example the **order sale transactions**, because they are about **financial business**. Thus, the calculations **must be correct 100%**.
- To access this **accuracy**, must be sure that our system is not **losing any event messages**.
- **The Outbox Pattern** should be applied this kind of cases.



Kindly explain above in details



The two images describe the **Transactional Outbox Pattern** in microservices, particularly for use cases like processing customer orders, and explain why this pattern is important for ensuring data consistency and reliability.

### Image 1: Customer Places an Order Use Case - Transactional Outbox Pattern in Microservices

This diagram depicts the workflow of how the **Transactional Outbox Pattern** operates in the scenario where a customer places an order.

#### 1. Customer Creates an Order:

- The user initiates an order in the system.

- The **Order microservice** receives the request to create an order.

## 2. Order and Outbox Tables:

- The **Order service** writes the new order into the **Order Table** in the database.
- Simultaneously, it writes a corresponding event (representing the order creation) into the **Outbox Table**.
- Both of these actions are part of the **same database transaction**, meaning they either succeed or fail together. If the order is successfully created, the event is also stored in the outbox table.

## 3. Message Transmitter:

- Once the transaction is successfully committed, a **Message Transmitter** or a **CDC (Change Data Capture) tool** reads from the outbox table.
- It retrieves the event data and then **publishes the event** to the **Event Bus** (e.g., Kafka, RabbitMQ, etc.).

## 4. Event Bus:

- The **Event Bus** acts as a central hub for distributing the event to other microservices (such as inventory management, shipping, or payment services) that need to know about the order creation.

This approach ensures **data consistency** and **reliable event publication**. If something goes wrong during the transaction (e.g., the database fails), neither the order nor the event is saved.

## Image 2: Why Use the Outbox Pattern?

This image explains the reasons for using the **Outbox Pattern** in microservices, especially when dealing with **critical data** and ensuring **accuracy**.

### 1. Working with Critical Data:

- When handling critical operations like **order processing** or **financial transactions**, **data consistency** is vital.
- The **Outbox Pattern** helps ensure that all requests are captured and handled accurately, preventing data loss.

### 2. Atomicity for Database Update and Message Sending:

- The **database update** (e.g., recording a new order) and the **sending of the event** (e.g., publishing the "order created" event) must be atomic. This means that they must either both happen or neither happens.
- Ensuring atomicity is crucial for maintaining **data consistency** across different systems and microservices.

### 3. Sales or Financial Transactions:

- For **order sale transactions**, particularly in **financial business**, data must be **100% correct**. Any discrepancies can lead to significant issues, such as incorrect billing or inventory problems.
- In these cases, the **Outbox Pattern** guarantees that all events related to order sales are processed without loss or duplication.

#### 4. No Loss of Event Messages:

- One of the key reasons for using the outbox pattern is to ensure that no **event messages** are lost.
- By recording events in the **outbox table** within the same transaction as the business operation, the system ensures that even if the service temporarily fails, the event will eventually be sent when the service is restored.

#### 5. Applicability of the Outbox Pattern:

- The **Outbox Pattern** should be applied whenever:
  - Critical business operations need to be reliably communicated between microservices.
  - There's a need for **accuracy**, such as in financial transactions or other important business processes.

---

### Summary:

The **Transactional Outbox Pattern** is crucial for ensuring that data is consistently handled across distributed systems. When a customer places an order, this pattern allows the order service to:

- Record the order in the **Order Table**.
- Store a corresponding event in the **Outbox Table** as part of the same transaction.
- Later, the event is read from the outbox and published to an **event bus**, ensuring reliable communication with other services.

This pattern ensures **atomicity**, **data consistency**, and **reliable event publishing**, making it essential for critical business processes where missing or inconsistent data can have significant consequences.



## Problem: Listen and Polling Outbox Table

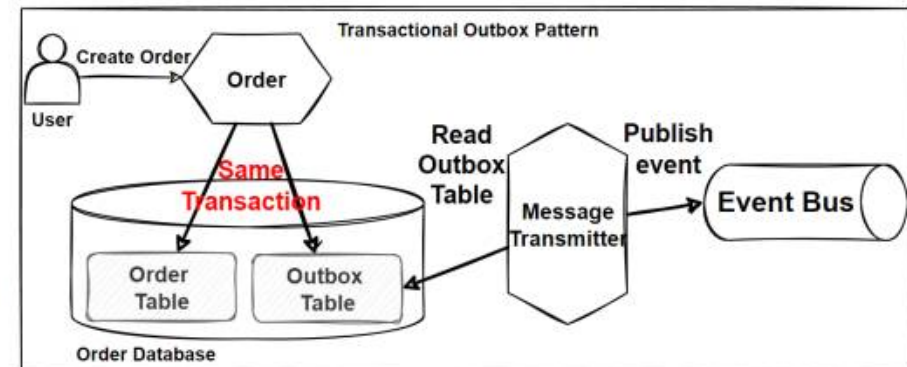
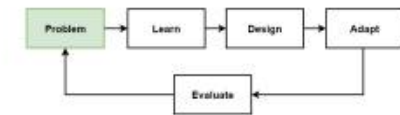
- **Creating new order record** into order table and **creating new order\_created event** into outbox table **are in the same transactions**.
- Another service **listen** and **polls** the **outbox table records** and publish events into separate processes.
- The current architecture **required additional microservices** that listen and polls records from the outbox table.

### Problems

- Reduces the performance and error-prone operations.
- Time is wasted during Polling, wasted resources.
- Consuming system resource unnecessary pull requests.
- Come with a latency overhead, reach system limitations.
- Can cause additional Dual-write problem.

### How solve these problems when applied outbox pattern ?

- CDC - Change Data Capture with Outbox Pattern.



Mehmet Ozkaya 529

Kindly explain above in details



The image discusses a **problem** associated with the **Listen and Polling Outbox Table** approach, used in the **Transactional Outbox Pattern**, and how **Change Data Capture (CDC)** can be applied to improve it.

## Problem: Listen and Polling Outbox Table

### Process Explanation:

#### 1. Transactional Outbox Mechanism:

- When a customer creates an order, the **order record** is written into the **order table**, and at the same time, the corresponding **order\_created event** is written into the **outbox table**. These two actions occur within the **same transaction**, ensuring atomicity (both actions succeed or fail together).

#### 2. Polling Process:

- To publish the events recorded in the **outbox table** to the **event bus** (e.g., Kafka, RabbitMQ), a separate microservice or process **polls** the outbox table to look for new records (events).
- The polled events are then published to the event bus.

#### 3. Additional Microservices for Polling:

- The architecture typically requires **additional microservices** or processes that **listen** and **poll the outbox table** to detect new events and publish them.

### Problems with Polling the Outbox Table:

The polling mechanism has several drawbacks, including performance and resource inefficiencies:

#### 1. Reduced Performance:

- Polling reduces overall system performance because the system must frequently check the outbox table for new records.

#### 2. Wasted Time and Resources:

- **Polling** can consume unnecessary resources, as the system is constantly pulling for updates, even when there are no new events to process. This wastes **CPU** and **I/O** resources.

#### 3. Unnecessary Pull Requests:

- Continuous polling means the system issues **frequent database queries** to the outbox table, which can lead to inefficiencies and unnecessary use of system resources.

#### 4. Latency Overhead:



- Polling introduces **latency** because there may be a delay between when the event is written to the outbox and when the polling process detects it and publishes it. This latency can cause the system to reach its limitations, especially under heavy load.

### 5. Potential Dual-Write Problem:

- Without careful management, the polling mechanism can inadvertently introduce an additional **dual-write problem**, leading to inconsistencies between systems if events are missed or processed incorrectly.

## Solution: CDC (Change Data Capture) with Outbox Pattern

To solve these problems, the **Change Data Capture (CDC)** approach is applied with the outbox pattern.

### 1. CDC (Change Data Capture):

- **CDC tools** (such as **Debezium**) track changes in the database in real-time by **monitoring the commit log**.
- Instead of polling the outbox table for new records, the CDC tool directly captures changes (new rows in the outbox table) and **immediately triggers the event**.
- This eliminates the need for continuous polling and reduces latency, improving the efficiency of the system.

### 2. Improved Performance and Efficiency:

- Using CDC improves the performance of the outbox pattern by **removing the need for frequent polling**.
- **Resources are conserved** since the system no longer constantly checks the outbox table for updates, and events are processed as soon as they are created.

### 3. No More Latency Overhead:

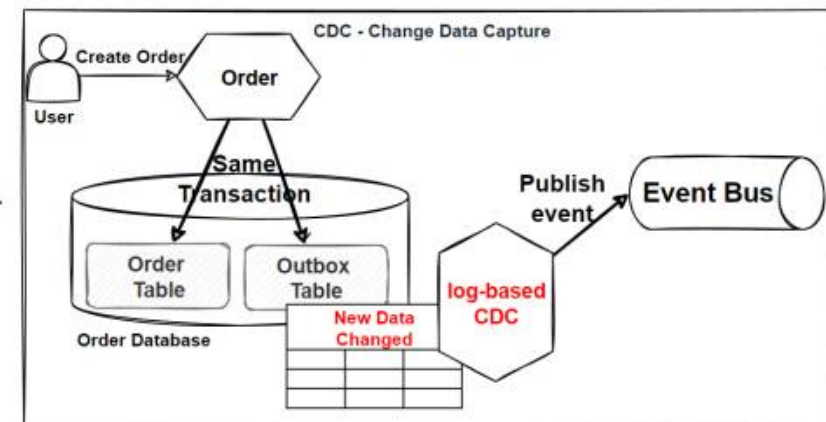
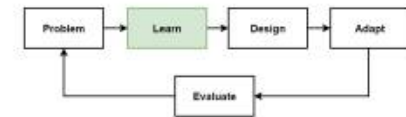
- CDC allows **real-time event propagation**, as changes in the outbox table are detected immediately and the corresponding events are published without delay.

## Summary:

The **Listen and Polling Outbox Table** method has several performance and resource-related problems, including wasted resources, latency overhead, and unnecessary pull requests. To address these, the **Change Data Capture (CDC)** approach is introduced. CDC tools monitor database changes in real time, capturing new events from the outbox table as soon as they are created, which improves efficiency, reduces latency, and ensures that the system is more performant without the need for continuous polling. This approach ensures **real-time event delivery** with fewer system resource demands.

# CDC - Change Data Capture

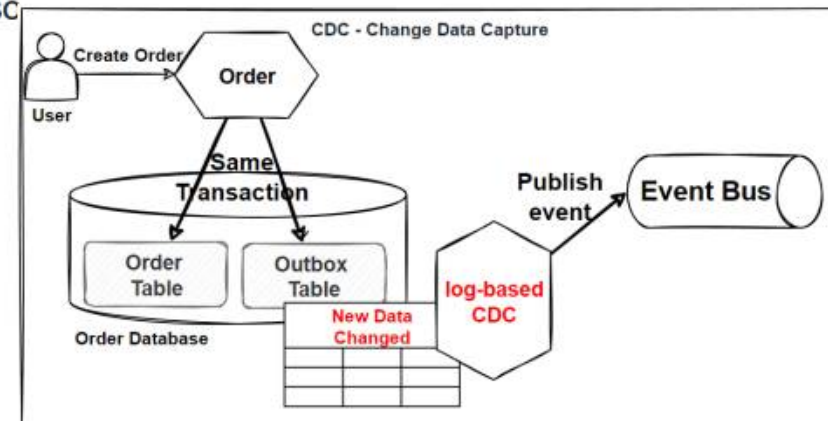
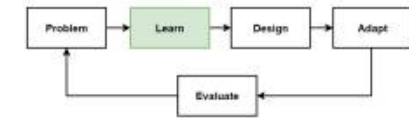
- **Change Data Capture (CDC)** is a technology that captures insert, update, and delete activity on a database.
- CDC typically works by **continuously monitoring the transaction log** of a **database for changes**, and then **extracting** and **propagating** those **changes** to the target system.
- This allows the target system to **stay up-to-date** with the source system **in near real-time**, instead of relying on batch-based data synchronization processes.
- CDC can be used in **replicating data between databases**, **synchronizing data** between systems in a microservices architecture, and **enabling real-time data analytics**.
- CDC is a way to **track changes** that happen to data in a database that **captures insert, update, and delete activity** and makes this information available to other systems.
- This allows those systems to **stay up-to-date** with the data in the **database in real-time**.



Mehmet Ozkaya 530

# CDC - Change Data Capture with Outbox Pattern

- **Outbox pattern** is ensuring **data changes** made by a microservice are **eventually propagated** to other microservices.
- Whenever a microservice **updates data** in its database, it also writes a **record to the outbox table** with the details of the change.
- **CDC** can then be used to **monitor the outbox table** for new records, **extract the data changes** that **propagated** to the **target microservices** to be **kept up-to-date** with the data.
- Using **CDC** with the **Outbox pattern** allows microservices to **decouple their data updates** from the **process of propagating** those **updates** to other microservices.
- This can make it **easier to scale** and **maintain** a microservices architecture.
- Each microservice can **focus on its own data updates** and let **CDC handle the synchronization** of data between services.



Mehmet Ozkaya 531

Kindly explain the above image in details



The two images provide an explanation of **Change Data Capture (CDC)** and its integration with the **Outbox Pattern** in microservices. Here's a detailed breakdown:

## Image 1: CDC - Change Data Capture

### What is CDC (Change Data Capture)?

- **CDC** is a technology used to **capture** changes (insert, update, delete) in a database. It continuously monitors database transactions and logs and then propagates those changes to other systems in **real-time**.

### How CDC Works:

#### 1. Continuous Monitoring:

- CDC operates by monitoring the **transaction log** of a database for changes (such as inserts, updates, or deletes). This allows it to **extract** those changes and propagate them to a target system (like another microservice or an event bus).

#### 2. Near Real-Time Data Synchronization:

- CDC ensures that the target system stays **up-to-date** with the source system in near real-time, unlike batch-based synchronization methods, which may delay updates.

#### 3. Use Cases for CDC:

- **Data Replication:** CDC can replicate data between databases, keeping them synchronized.
- **Microservice Synchronization:** In a microservices architecture, CDC helps synchronize data between services, enabling **real-time data analytics** and consistency across the system.

#### 4. CDC's Role in Tracking Changes:

- CDC tracks changes like **insert, update, and delete** activities in the database and makes this information available for other systems to use.
- This ensures that the systems can stay in sync with **real-time updates**.

### Diagram in Image 1:

The diagram shows the flow where a customer creates an order. The **order data** is stored in both the **Order Table** and the **Outbox Table** as part of the same transaction. The **CDC system** then reads the **commit log** to detect the changes and immediately publishes the event to the **Event Bus**.

## Image 2: CDC - Change Data Capture with Outbox Pattern

This image explains how the **Outbox Pattern** is enhanced by **CDC** to handle data propagation between microservices efficiently.

## How the Outbox Pattern Works with CDC:

### 1. Outbox Pattern for Data Propagation:

- The **Outbox Pattern** ensures that any **data changes** (such as creating a new order) are **eventually propagated** to other microservices.
- When a microservice updates its data (e.g., an order is placed), it writes both to its **business table** (Order Table) and the **Outbox Table** (which contains details of the event).

### 2. Monitoring the Outbox Table:

- **CDC** is used to **monitor the outbox table** for any new records (events).
- When changes are detected in the outbox table, CDC **extracts** these changes and **propagates** them to the target microservices, ensuring they stay up-to-date with the latest data.

### 3. Decoupling Data Updates and Event Propagation:

- By using **CDC with the Outbox Pattern**, microservices can **decouple** the actual **data update** (writing to the Order Table) from the process of **propagating the event** to other microservices.
- This decoupling improves system performance, reduces latency, and makes the architecture more scalable.

### 4. Scaling and Maintenance:

- CDC simplifies scaling and maintaining a microservices architecture by taking responsibility for ensuring data synchronization across services.
- Each microservice can focus on **updating its own data**, while CDC handles the synchronization with other services.

## Diagram in Image 2:

The diagram shows the flow:

1. A **customer places an order**, which triggers a transaction.
2. The **order data** is stored in both the **Order Table** and the **Outbox Table** as part of the same transaction.
3. **CDC** continuously monitors the **log-based changes** in the outbox table, and when it detects a new event, it publishes that event to the **Event Bus**, propagating it to other microservices that rely on this data.

## Summary:

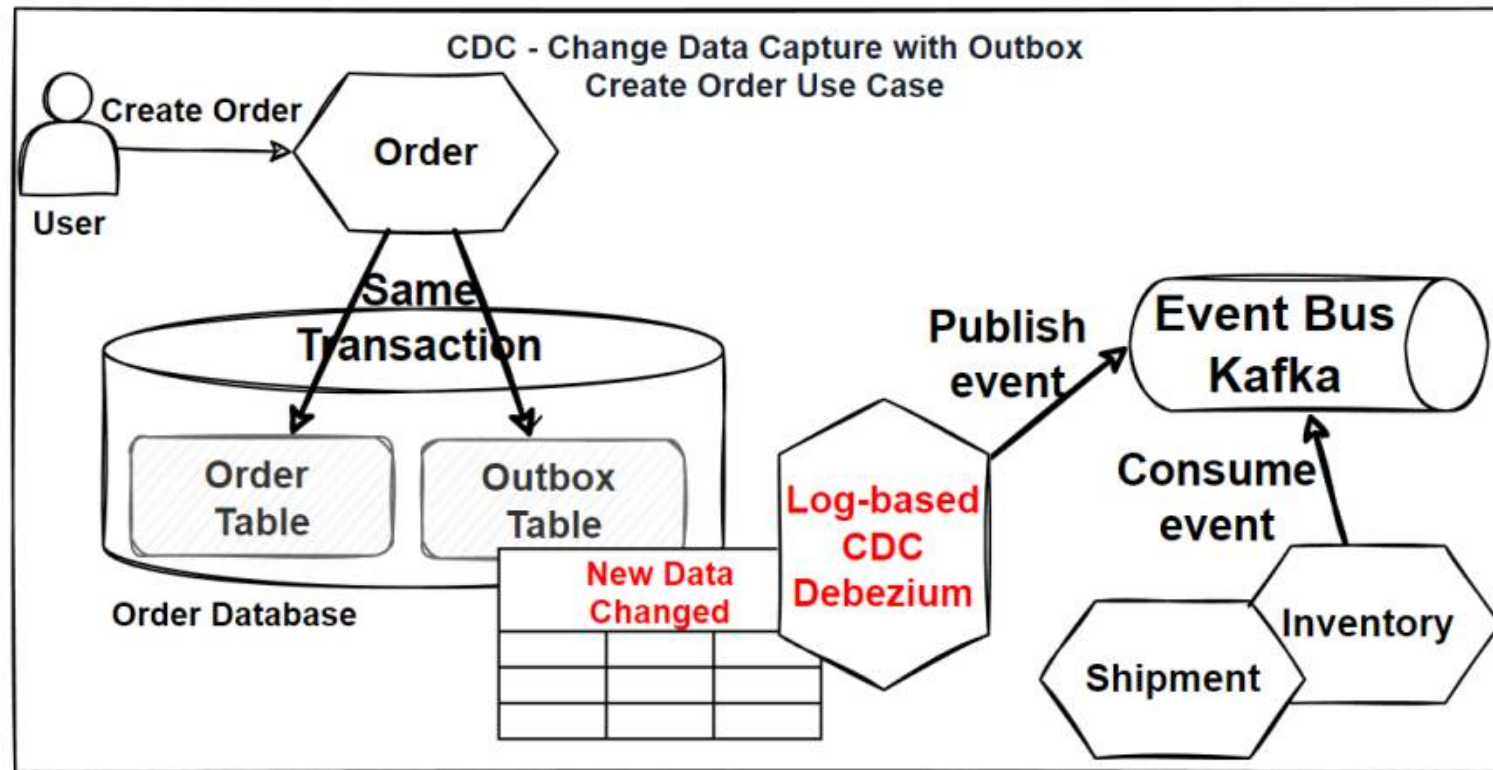
- **CDC (Change Data Capture)** is a powerful mechanism for tracking and propagating database changes in real time. It helps keep systems synchronized and up-to-date.
- **Outbox Pattern** ensures reliable event publishing by recording changes in a separate table (outbox) alongside business data.
- When **CDC** is used with the **Outbox Pattern**, it continuously monitors the outbox table for new events and propagates them to other systems without the need for manual polling. This approach helps achieve **scalability**, **real-time data synchronization**, and a more efficient microservices architecture.

This combination is particularly effective in systems where **reliability**, **data consistency**, and **real-time event propagation** are critical for smooth operations.





## CDC - Change Data Capture with Outbox Pattern E-commerce Create Order Use Case



Mehmet Ozkaya 532

Kindly explain the above usecase in details



The diagram explains the **E-commerce Create Order Use Case** using the **CDC (Change Data Capture) with Outbox Pattern**. It illustrates how the **Outbox Pattern** combined with **CDC** ensures reliable event-driven communication across microservices during an order creation process. Let's break it down step

by step:

## Step-by-Step Breakdown of the E-commerce Create Order Use Case:

### 1. Customer Creates an Order:

- A user initiates the process by placing an order in the e-commerce system.
- This request triggers the **Order microservice**, which handles the creation of the order.

### 2. Order Written to the Order and Outbox Tables (Same Transaction):

- The **Order Service** stores the new order in the **Order Table** of its database.
- At the same time, it also writes an **order\_created event** into the **Outbox Table** within the same transaction.
- This ensures that both the business operation (order creation) and the event (that will notify other systems) either succeed or fail together, maintaining data consistency.

### 3. CDC Monitors the Transaction Log:

- **Change Data Capture (CDC)** technology (implemented using **Debezium** in this example) is configured to **monitor the transaction logs** of the database.
- Whenever a new record (such as an event in the Outbox Table) is added, CDC detects the **new data change** in real time.
- **Log-based CDC** uses the database's **commit log** to capture these changes and propagate them to other systems.

### 4. Publishing Event to Event Bus (Kafka):

- Once **CDC** detects the new event in the **Outbox Table**, it triggers the process of **publishing the event** to an **Event Bus** (e.g., **Kafka**).
- The event bus acts as a message broker, ensuring that the event is delivered to all downstream services that need to consume this information.

### 5. Consumption of the Event by Other Microservices:

- Various other microservices in the e-commerce system, such as **Inventory** and **Shipment** services, are subscribed to the event bus (Kafka).
- They **consume the event** (order\_created), triggering their own processes.
  - **Inventory Service:** Updates inventory levels to reflect the new order.
  - **Shipment Service:** Initiates the shipping process for the new order.

These services act based on the events they receive, ensuring a decoupled, event-driven architecture.

## How CDC with the Outbox Pattern Helps:

- **Consistency and Atomicity:**

- By writing the business data (order) and the event into the outbox table within the same transaction, this architecture guarantees **atomicity**. Both actions succeed or fail together, avoiding inconsistencies.

- **Decoupling with Real-Time Event Delivery:**

- **CDC** ensures that events are **propagated in near real-time** by reading from the **transaction log**. This eliminates the need for traditional polling mechanisms, reducing system overhead.
- Using the **event bus (Kafka)** enables a **decoupled** architecture, where the **Order Service** doesn't need to know about the downstream services that will consume the events. They simply publish the event, and subscribers like **Inventory** and **Shipment** services handle the rest.

- **Scalability:**

- This architecture makes it easy to scale. New microservices can subscribe to the event bus to consume the events without needing to change the **Order Service** itself. For example, if a new **Payment Service** is introduced, it can consume the `order_created` event without impacting the existing system.

## Summary:

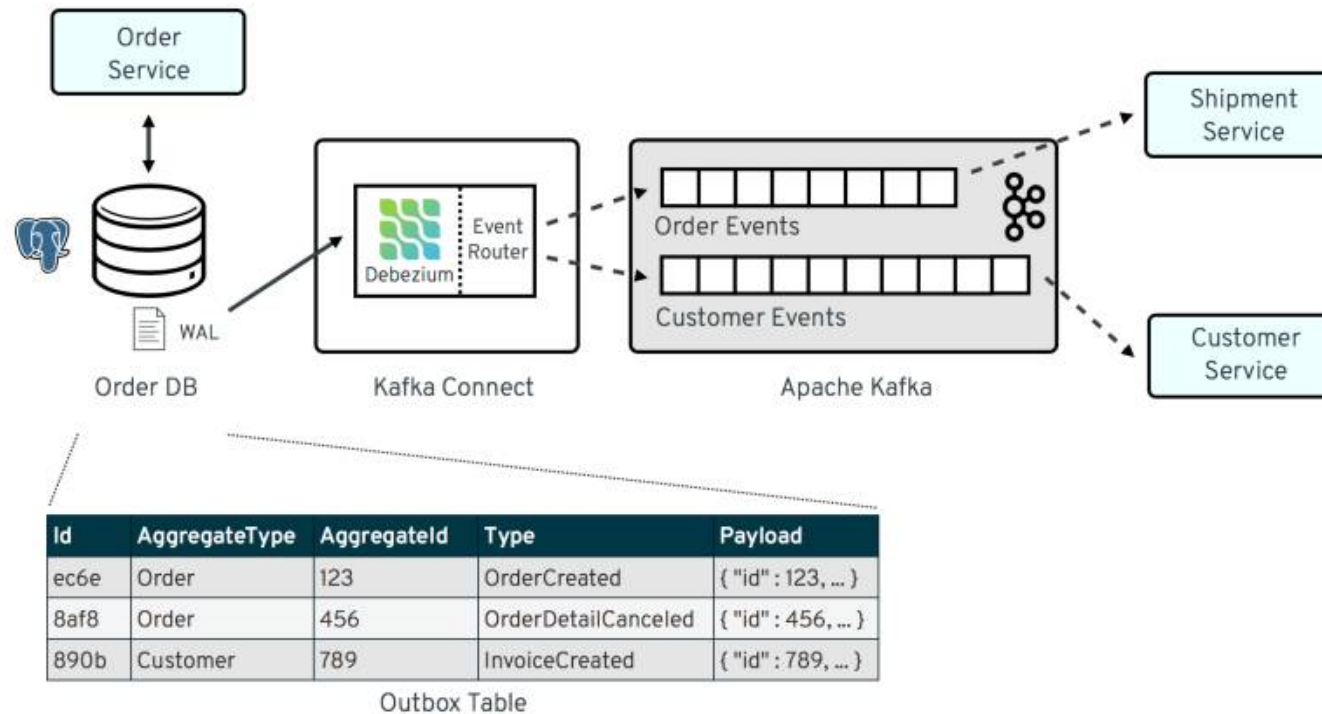
The **E-commerce Create Order Use Case** with the **CDC and Outbox Pattern** ensures that when an order is created:

- The order data is reliably stored and an event is generated.
- **Change Data Capture (CDC)** detects changes in the outbox table and publishes the event to **Kafka** in real-time.
- Downstream microservices like **Inventory** and **Shipment** consume the event and perform their tasks based on the new order.

This combination of **CDC** and **Outbox Pattern** ensures data consistency, enables decoupled communication between services, and supports scalability in a microservices architecture.



# CDC - Change Data Capture with Outbox Pattern E-commerce Create Order Use Case



<https://debezium.io/blog/2019/02/19/reliable-microservices-data-exchange-with-the-outbox-pattern/>

Mehmet Ozkaya

Kindly explain the above in details



The diagram illustrates an **E-commerce Create Order Use Case** using **Change Data Capture (CDC) with the Outbox Pattern**, and it demonstrates how events are captured, processed, and propagated using Kafka in an event-driven architecture. Let's break it down step by step:

## Key Components in the Diagram:

### 1. Order Service:

- This is the microservice responsible for handling customer orders. When a user places an order, this service handles the logic for creating the order and saving it to the **Order Database**.

### 2. Order Database (Order DB):

- The **Order DB** is the database where all order-related data is stored. In this example, it uses **PostgreSQL**, as indicated by the icon.
- The database keeps a **Write-Ahead Log (WAL)**, which is a standard method used by databases to track changes before they are written to the main database files.

### 3. Debezium (CDC):

- **Debezium** is a popular **CDC (Change Data Capture)** tool that continuously monitors the database changes by reading the **WAL (Write-Ahead Log)**.
- In this use case, Debezium detects when a new order is inserted into the database. It monitors any changes made to the **outbox table** where events are stored.
- As soon as Debezium detects a change (e.g., a new order created), it forwards that change as an **event** to the **event router** (Kafka Connect).

### 4. Kafka Connect:

- **Kafka Connect** is used to integrate Kafka with external systems, such as databases and other microservices.
- Kafka Connect takes the changes detected by Debezium and pushes them into the **Apache Kafka** event stream.

### 5. Apache Kafka (Event Bus):

- **Apache Kafka** is used as an **event bus** to store and route events. Kafka organizes these events into different **topics** (streams of messages).
- In this diagram, there are two streams of events:
  - **Order Events** (e.g., OrderCreated, OrderDetailCanceled).
  - **Customer Events** (e.g., InvoiceCreated).
- Kafka acts as the **central hub** where other services (such as **Shipment Service** and **Customer Service**) can subscribe to and consume events they are interested in.

### 6. Event Consumers:

- Once the event is published to Kafka, other microservices that are subscribed to these Kafka topics can consume the events. In this example:
  - The **Shipment Service** listens to **Order Events** (such as OrderCreated) and takes the necessary steps to fulfill the order.

- The **Customer Service** listens to **Customer Events** (such as InvoiceCreated) and may update customer information or handle billing.

## 7. Outbox Table:

- The **Outbox Table** holds the events that were created as part of the same transaction when the order was placed. This ensures that both the order data and the event are stored together, maintaining atomicity.
- The table contains columns like:
  - **Id**: A unique identifier for each event.
  - **AggregateType**: The type of entity that triggered the event (e.g., Order or Customer).
  - **AggregateId**: The unique ID of the entity.
  - **Type**: The specific event type (e.g., OrderCreated or InvoiceCreated).
  - **Payload**: The actual data related to the event, typically in JSON format.

## Example Events in the Outbox Table:

- **ID: ec6e**: Represents an **OrderCreated** event for **Order ID 123**.
- **ID: 8af8**: Represents an **OrderDetailCanceled** event for **Order ID 456**.
- **ID: 890b**: Represents an **InvoiceCreated** event for **Customer ID 789**.

These events will be processed by Kafka and consumed by the respective services that are interested in these events.

---

## Summary of the Flow:

1. A user creates an order through the **Order Service**.
2. The **Order Service** writes the order data to the **Order Table** and creates a corresponding **event** in the **Outbox Table** within the **Order DB**.
3. **Debezium** (CDC) monitors the **WAL** for changes and detects new records in the **Outbox Table**.
4. The detected change (new event) is forwarded to **Kafka Connect**, which publishes the event to **Apache Kafka**.
5. **Apache Kafka** holds these events in separate topics (streams) like **Order Events** and **Customer Events**.
6. **Microservices** (like **Shipment Service** and **Customer Service**) consume these events to take further actions, such as processing orders and handling customer invoices.

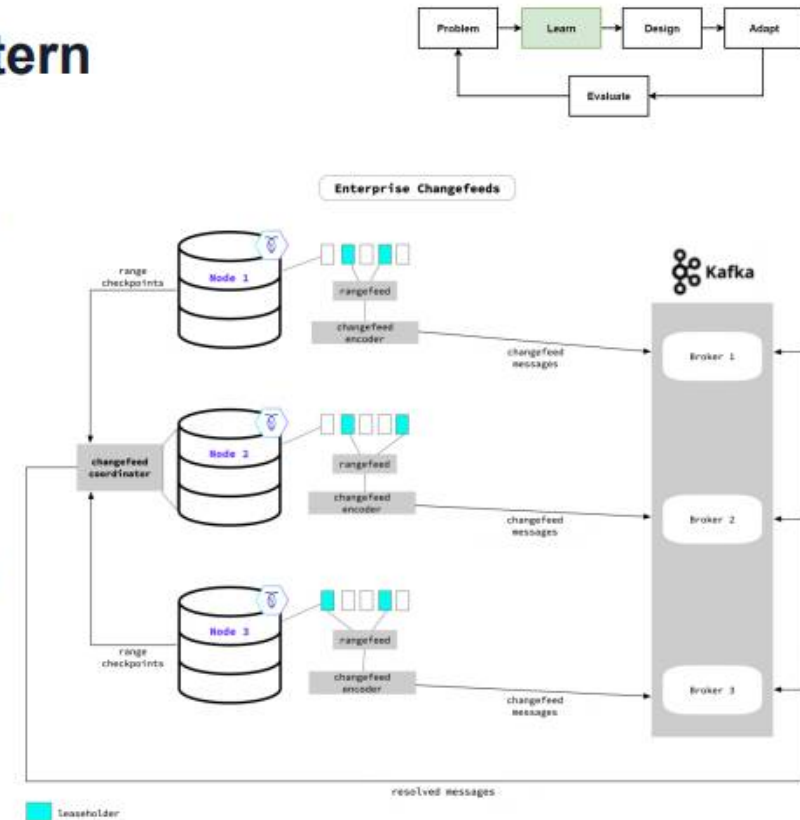
This architecture ensures **data consistency**, **real-time event propagation**, and **decoupled communication** between microservices using the **Outbox Pattern** with **CDC** and **Kafka**.





# CockroachDB for CDC and Outbox Pattern

- **CockroachDB** is a **distributed database** management system that is designed to be **scalable**, **resilient**, and easy to use.
- **Cockroach cluster**, which is a **group of database nodes** that work together to form a single, highly available database.
- Ability to **scale horizontally** by **adding more nodes** to the **cluster** as the **workload increases**.
- **CockroachDB** also has strong support for **data consistency** and **durability**, with features such as **multi-active availability** and **distributed transactions**.
- **CockroachDB** is written in the **Go programming** language.
- **CockroachDB** has **built-in Change Data Capture** feature, that you can build the **Transactional Outbox Pattern** with **CDC** into your own application.

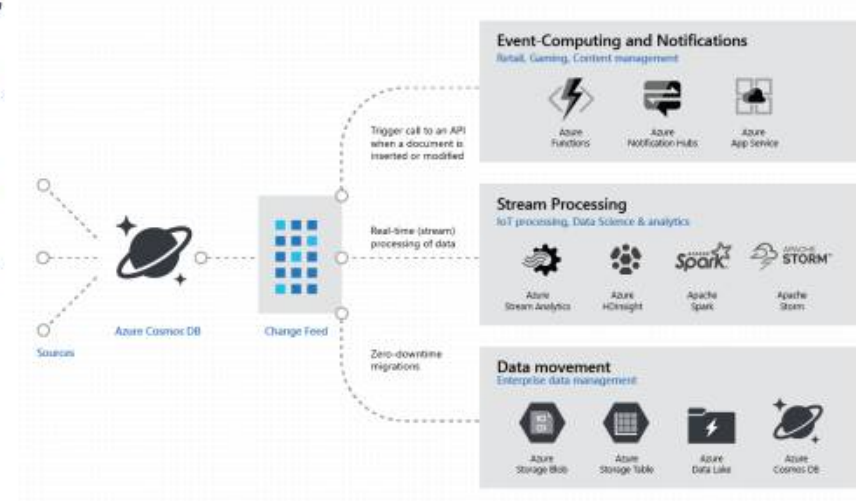


<https://www.cockroachlabs.com/docs/stable/change-data-capture-overview.html>

Mehmet Ozkaya 534

# Azure Cosmos DB for CDC and Outbox Pattern

- **Azure Cosmos DB** is a **globally distributed, multi-model** database service offered by **Microsoft Azure**.
- It supports various **database models**, including document, **key-value**, **column-family**, and **graph**, and can be accessed through multiple APIs, such as **SQL**, **MongoDB**, **Cassandra**, and **Azure Table Storage**.
- **Azure Cosmos DB** has built-in support for **Change Data Capture (CDC)**, allows it to **track and propagate data changes** made to the **database** to **other systems** in near real-time.
- **CDC in Azure Cosmos DB** works by **continuously monitoring the transaction logs** of the **database for changes** and **extracting those changes** to be sent to target systems.
- **Azure Cosmos DB** change **feed API** to access the data changes and process them in your application.



<https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/change-feed-design-patterns>

Mehmet Ozkaya 535

Kindly explain the given images in details



The two images explain how **CockroachDB** and **Azure Cosmos DB** can be integrated with the **CDC (Change Data Capture)** and **Outbox Pattern** for handling event-driven architecture in distributed microservices systems.

## Image 1: CockroachDB for CDC and Outbox Pattern

**CockroachDB** is a distributed, highly resilient, and horizontally scalable database. Here's a detailed explanation of how it can be used with **CDC and the Outbox Pattern**:

### 1. CockroachDB Overview:

- CockroachDB is a **distributed database management system** designed to be **scalable, resilient**, and easy to use.
- It operates as a **Cockroach cluster**, which is a group of database nodes that work together to form a single, highly available database. This cluster is resilient to failure and can maintain its state across geographically distributed regions.

### 2. Horizontal Scalability:

- **CockroachDB** has the ability to **scale horizontally**, meaning that as the workload increases, you can add more nodes to the cluster, ensuring higher availability and performance.
- This feature is especially important for modern, cloud-based microservices architectures where workload can vary significantly.

### 3. Data Consistency and Durability:

- CockroachDB offers strong **data consistency** and **durability**, supported by features like **multi-active availability** and **distributed transactions**.
- This ensures that data is always available, even in case of node or network failures, and guarantees that distributed transactions are handled atomically across the cluster.

### 4. Built-in Change Data Capture (CDC):

- CockroachDB has a built-in **Change Data Capture (CDC)** feature that allows it to track and propagate changes (such as inserts, updates, and deletes) from the database to other systems (e.g., Kafka).
- Using CDC with CockroachDB allows the integration of the **Transactional Outbox Pattern** into an application, which ensures that changes made in the microservice are reliably captured and forwarded as events in an event-driven architecture.

### 5. How CockroachDB Works with CDC:

- Each **node** in the CockroachDB cluster is responsible for maintaining **range checkpoints**, which monitor changes made to specific data ranges in the database.
- When changes are detected, the **changefeed coordinator** sends the data to a **Kafka** instance (or other systems) via **changefeed encoders**.
- This allows for efficient **change data propagation** across distributed systems, ensuring near real-time synchronization between microservices.

## Image 2: Azure Cosmos DB for CDC and Outbox Pattern

**Azure Cosmos DB** is a globally distributed, multi-model database service provided by Microsoft Azure. Here's how it works with **CDC and the Outbox Pattern**:

### 1. Azure Cosmos DB Overview:

- **Azure Cosmos DB** is a **globally distributed** and **multi-model database service** that can support various database models, including:
  - Document-based (e.g., MongoDB)
  - Key-value
  - Column-family
  - Graph
- It provides a flexible API that allows users to interact with the database using **SQL, MongoDB, Cassandra, and Azure Table Storage** interfaces, making it a versatile choice for different types of applications.

### 2. Change Data Capture (CDC) Support:

- **Azure Cosmos DB** has **built-in support for CDC** (Change Data Capture), which allows it to **track and propagate data changes** made to the database to other systems (e.g., event hubs, analytics systems) in near real-time.
- The built-in CDC capability ensures that the **Outbox Pattern** can be implemented easily within Azure Cosmos DB-based applications, supporting real-time event streaming in microservices architectures.

### 3. How CDC Works in Azure Cosmos DB:

- The CDC in Azure Cosmos DB works by **continuously monitoring the transaction logs** of the database for changes, similar to how other CDC systems work.
- Whenever a change is made in the database (like inserting or updating data), the CDC component **extracts those changes** and propagates them to downstream systems (such as **Kafka** or **Azure Event Hubs**).
- This ensures **real-time synchronization** between different microservices and systems.

### 4. Change Feed API:

- **Azure Cosmos DB** provides a **Change Feed API**, which allows developers to easily access the stream of data changes in their applications.
- This API can be integrated into microservices to enable real-time data processing, ensuring consistency across distributed systems without the need for polling.

### 5. Integration with Microservices:

- The diagram shows how Azure Cosmos DB changes are propagated via the **change feed** to various systems, such as:
    - **Event Computing and Notifications:** Systems like Azure Functions, Notification Hubs, etc., can respond to changes in real-time.
    - **Stream Processing:** Platforms like **Apache Kafka**, **Azure Stream Analytics**, and **Apache Storm** can process real-time data streams coming from Cosmos DB.
    - **Data Movement:** Tools like **Azure Data Lake** and **Storage Blob** can be updated with changes coming from the Cosmos DB change feed, ensuring that the data is available for analytics or storage purposes.
- 

## Key Takeaways:

- **CockroachDB** and **Azure Cosmos DB** are both powerful distributed database systems designed for scalability, resilience, and real-time data processing.
- Both databases have **built-in Change Data Capture (CDC)** capabilities, allowing the integration of the **Outbox Pattern** into microservices applications, ensuring reliable data propagation and synchronization.
- CockroachDB focuses on strong **data consistency**, **horizontal scalability**, and **distributed transactions**, making it a robust choice for globally distributed systems.
- **Azure Cosmos DB** offers a wide range of database models and APIs, as well as seamless CDC integration, making it highly flexible for various applications, including event-driven architectures, stream processing, and data analytics.

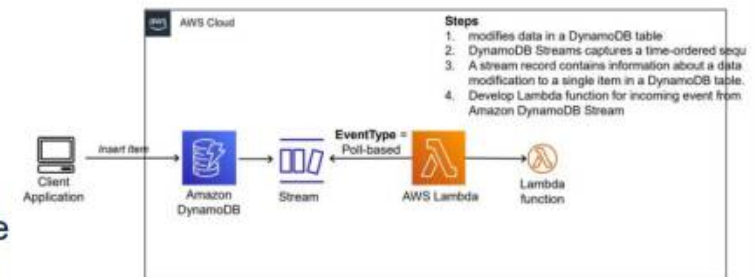
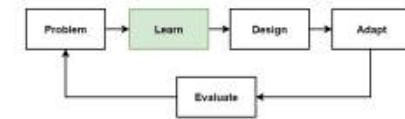
These features make both databases excellent choices for modern microservices architectures where real-time data processing and distributed consistency are critical.





# Amazon DynamoDB Streams for CDC and Outbox

- **DynamoDB** supports **streaming** of **item-level change data capture** records in **the near-real time**.
- **DynamoDB stream** is an **ordered flow of information** about changes to items in a **DynamoDB table**.
- Whenever an application **creates, updates, or deletes items** in the **table**, **DynamoDB Streams** writes a **stream record** with the **primary key attributes** of the items that were modified.
- A **stream record** contains information about a **data modification** to a **single item** in a **DynamoDB table**. That can includes capture additional information, such as the **“before”** and **“after”** images of modified items.
- **DynamoDB Streams** writes **stream records** in near-real time so that you can build applications that **consume these streams** and **take action** based on the contents.
- Most of the applications can **benefit** from **data capturing changes** into DynamoDB table; **Notifications, Mobile Apps, Financial Apps**.

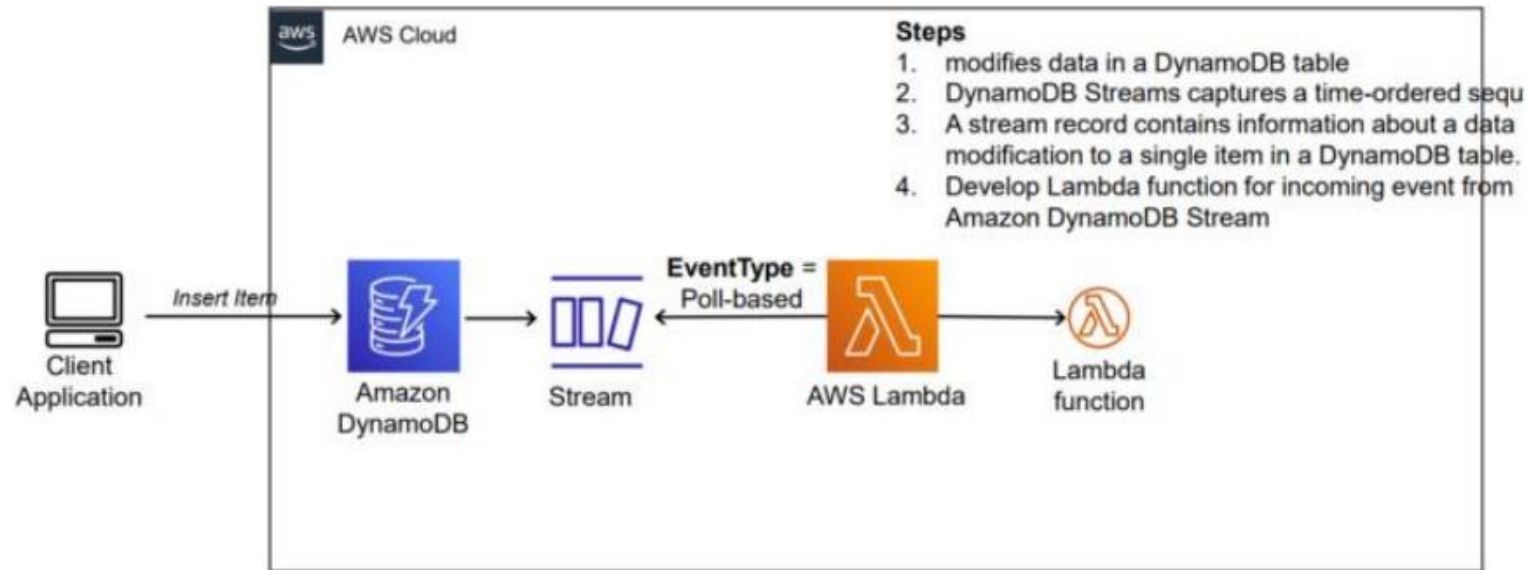
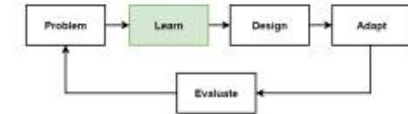


<https://medium.com/aws-lambda-serverless-developer-guide-with-hands/dynamodb-streams-using-aws-lambda-to-process-dynamodb-streams-for-change-data-capture-2e3ab8df27ca>

Mehmet Ozkaya 536



# Amazon DynamoDB Streams for CDC and Outbox



<https://medium.com/aws-lambda-serverless-developer-guide-with-hands/dynamodb-streams-using-aws-lambda-to-process-dynamodb-streams-for-change-data-capture-2e3ab8df27ca>

Mehmet Ozkaya 537

Kindly explain the above in details



The two images provide an explanation of how **Amazon DynamoDB Streams** can be used for **Change Data Capture (CDC)** and integrated with the **Outbox Pattern** for real-time data processing. Here's a detailed breakdown:

## Image 1: Amazon DynamoDB Streams for CDC and Outbox

**Amazon DynamoDB Streams** is a feature of **DynamoDB**, Amazon's NoSQL database service, that allows streaming of real-time changes to items in a DynamoDB table. It provides a way to capture and propagate **data modification events** (such as inserts, updates, and deletes) in near real-time.

### Key Points:

#### 1. Real-Time Streaming:

- **DynamoDB Streams** captures **item-level changes** in DynamoDB in **near real-time**.
- This allows developers to build real-time event-driven applications by capturing the changes and streaming them to other systems or services.

#### 2. Ordered Flow of Changes:

- A **DynamoDB Stream** provides an **ordered flow of information** about changes to items in the DynamoDB table.
- Every time an item is created, updated, or deleted, a **stream record** is generated and contains the **primary key attributes** of the item, along with the change details.

#### 3. Stream Record Details:

- The **stream record** contains information about the **data modification** made to an individual item in the DynamoDB table.
- Additional details such as the **before** and **after** images of modified items can be captured, allowing you to see exactly how the data changed over time.

#### 4. Near Real-Time Streaming:

- DynamoDB Streams writes **stream records** in near real-time, enabling applications to consume these streams and **take action** based on the data changes.

#### 5. Use Cases for CDC with DynamoDB:

- **Notifications:** Trigger notifications when items are modified in the database.
- **Mobile Apps:** Sync real-time data to mobile applications.
- **Financial Apps:** Capture real-time changes to financial transactions or records.

## Image 2: DynamoDB Stream and AWS Lambda Flow

This image depicts how **DynamoDB Streams** integrates with **AWS Lambda** to process data changes in real-time. It provides a clear step-by-step flow of how this integration works.

### Steps Explained:

#### 1. Data Modification in DynamoDB:

- A **client application** (e.g., a mobile app or web app) inserts or modifies an item in the **Amazon DynamoDB table**.

#### 2. Stream Captures Changes:

- **DynamoDB Streams** captures the changes in the **DynamoDB table** in a time-ordered sequence. Each change generates a **stream record** that describes what was changed, when, and the primary key of the modified item.

#### 3. Poll-Based Event Processing:

- The stream is **poll-based**, meaning the system checks for new events at regular intervals. Once new records are available, they are forwarded to the next processing stage.

#### 4. AWS Lambda for Event Handling:

- **AWS Lambda** is used to process the **stream events**. A **Lambda function** is triggered whenever a new stream record is created.
- The Lambda function can perform various actions based on the data changes, such as:
  - Sending notifications
  - Updating another database
  - Triggering workflows or integrating with other AWS services

### Example Workflow:

- A **client application** inserts an item (e.g., a new order or user data) into a **DynamoDB table**.
- **DynamoDB Streams** captures this change in real-time and creates a **stream record**.
- This stream record is **consumed** by a **Lambda function** that processes the change and takes the required action (e.g., send notifications, update a different system, etc.).

---

### Key Benefits of DynamoDB Streams for CDC and Outbox:

#### 1. Real-Time Data Processing:

- DynamoDB Streams provides **near real-time data** capture and propagation, making it ideal for event-driven applications that require immediate responses to data changes.

## 2. Decoupling with Lambda:

- By using **AWS Lambda**, the system can handle changes in a **serverless** manner, allowing for easy scaling and decoupling of processing logic from the database itself.

## 3. Data Capture Flexibility:

- With the ability to capture **before and after** images of modified items, DynamoDB Streams offers flexibility for different use cases, such as audit logging, analytics, or tracking data changes over time.

## 4. Scalability:

- **DynamoDB Streams** is fully managed and can scale automatically with the size of the DynamoDB table, making it highly scalable for large applications with heavy write traffic.

## 5. Integration with AWS Services:

- Seamless integration with **AWS Lambda** enables developers to quickly build **event-driven** architectures that respond to data changes in DynamoDB without managing servers or infrastructure.

---

## Summary:

**Amazon DynamoDB Streams** provides a powerful mechanism for capturing **real-time changes** to items in a DynamoDB table and forwarding these changes to other systems for further processing. It integrates smoothly with **AWS Lambda**, enabling **serverless event handling** and making it easy to build real-time, event-driven architectures. Applications that require **notifications**, **real-time syncing**, or **processing based on data changes** can benefit from this feature, especially when combined with the **Outbox Pattern** for ensuring reliable event delivery.

ChatGPT can make mistakes. Check important info.