

Two-Phase Commit Overview



What is 2phase commit. Kindly explain with diagram



The **Two-Phase Commit (2PC)** is a distributed algorithm used to ensure all participants in a distributed transaction either commit or roll back their changes together. It's commonly used in distributed databases and systems to ensure consistency across multiple systems.

Here's how the **Two-Phase Commit (2PC)** works, broken down into its two phases:

1. Phase 1: Prepare

- **Coordinator** (the transaction manager) sends a **prepare request** to all participants (databases, services, etc.) involved in the transaction, asking if they are ready to commit.
- **Participants** receive the request and each perform their part of the transaction (e.g., locking data, preparing changes), but they don't commit yet. They respond with either:
 - **Ready/Yes**: Indicating they are ready to commit, or
 - **No**: Indicating they encountered an issue and cannot commit.
- If all participants respond **Yes**, the process moves to the second phase. If any participant responds **No**, the transaction is aborted.

2. Phase 2: Commit/Rollback

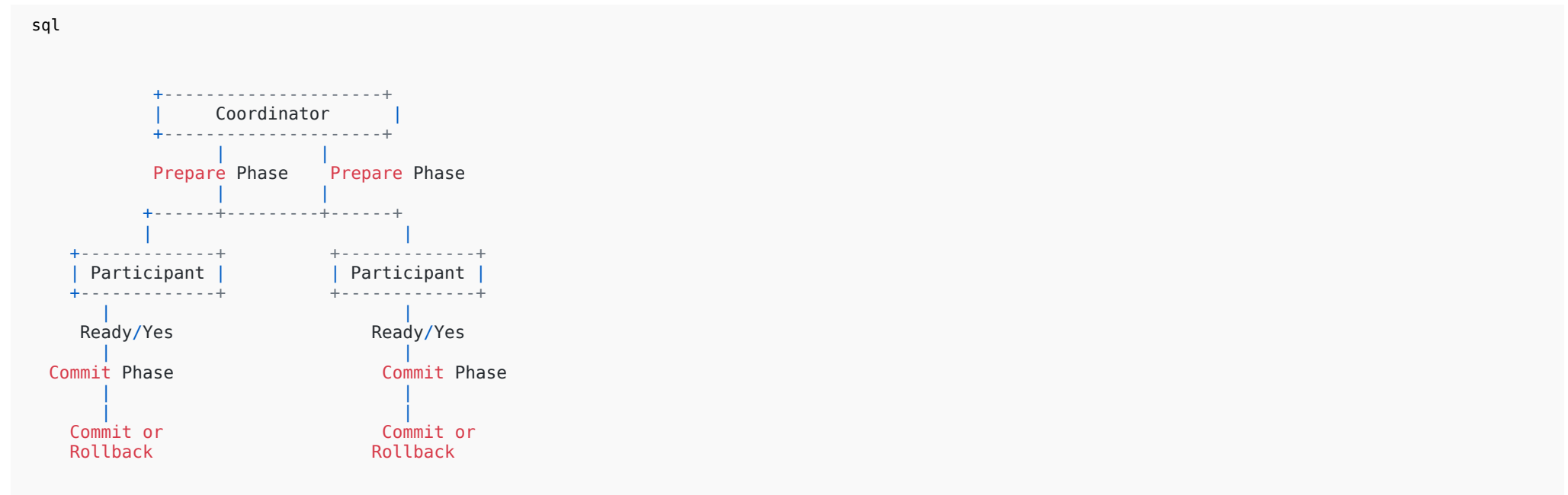
- If all participants are ready, the **Coordinator** sends a **commit request** to all participants, instructing them to finalize and commit the transaction.
- If any participant responded with **No** in the prepare phase, the **Coordinator** sends a **rollback request** to undo any partial changes.

Key Steps of Two-Phase Commit:

1. **Coordinator sends prepare requests.**
2. **Participants respond: Yes or No.**

3. If Yes from all participants, Coordinator sends commit request.
4. If any No, Coordinator sends rollback request.

Diagram Representation:



Explanation of the Diagram:

1. **Coordinator** initiates the process and sends the **prepare** messages to all participants.
2. The **Participants** reply with either **Yes** (ready to commit) or **No** (abort).
3. If all participants say **Yes**, the coordinator sends **commit** messages; otherwise, it sends **rollback** messages to undo changes.

Example Use Case

In a banking system where a user transfers money between two accounts, both the withdrawal from the sender's account and the deposit to the recipient's account must succeed or fail together. Two-phase commit ensures that either both operations happen or neither, maintaining data consistency.

Let me know if you want to dive deeper into any of the phases or the protocol specifics!

What are the drawbacks of 2 phase commit?



The **Two-Phase Commit (2PC)** protocol is widely used to ensure data consistency in distributed systems, but it has several **drawbacks** that can affect performance, scalability, and availability. Here are some of the key limitations:

1. Blocking Nature

- **Coordinator Failure:** If the coordinator crashes during the protocol, participants who have already voted “Yes” in the prepare phase are left in an uncertain state, as they don’t know whether to commit or roll back. They must wait for the coordinator to recover, which can cause delays or system halts.
- **Participant Failure:** If a participant crashes after sending a “Yes” vote, the coordinator waits indefinitely for the participant to recover and commit or roll back. This can cause the entire transaction to be blocked.

2. Single Point of Failure (Coordinator)

- The **Coordinator** is a critical component in the 2PC protocol. If it fails, the whole system may become unresponsive or delayed until the coordinator recovers. This makes the coordinator a **single point of failure**, which impacts the reliability and availability of the system.

3. Synchronous and Slow

- The 2PC protocol requires synchronous communication and coordination between participants and the coordinator. This adds significant **latency** because the participants must wait for each other, and the coordinator must ensure that all participants agree before proceeding. This can become slow, especially in systems with many participants or high network latency.

4. Resource Locking

- During the **Prepare Phase**, participants typically lock resources (e.g., database rows or objects) to ensure the transaction can be committed later. If the transaction is delayed or blocked (due to coordinator failure or network issues), these resources remain locked, preventing other transactions from accessing them, leading to reduced throughput and potential deadlocks.

5. No Fault Tolerance

- The 2PC protocol does not have built-in mechanisms for handling **network partitions** or failures gracefully. If a network partition occurs, participants in one partition will not be able to communicate with the coordinator or other participants, leading to an inconsistent or blocked state until the partition is resolved.

6. Inefficient in Large Systems

- In large, distributed systems with many participants, 2PC becomes inefficient due to the high communication overhead and the need to synchronize all participants. The time required for both phases (prepare and commit) increases with the number of participants, making it unsuitable for highly distributed environments like microservices or large-scale cloud systems.

7. No Automatic Recovery

- While 2PC ensures consistency, it does not handle recovery automatically. If the system crashes or a participant is in an uncertain state, manual intervention or custom recovery logic may be needed to resolve the situation and ensure the system returns to a consistent state.

8. Network Overhead

- Since 2PC involves two phases of communication between the coordinator and participants (prepare and commit), there is **significant network overhead** involved, especially in geographically distributed systems. This can be a bottleneck for systems where low-latency operations are critical.

Summary of Drawbacks:

- **Blocking** due to coordinator or participant failure.
- **Single point of failure** at the coordinator.
- **High latency** and **synchronous communication**.
- **Locks resources** for the duration of the transaction, potentially leading to deadlocks.
- **No fault tolerance** for network failures or partitions.
- **Scalability issues** in large systems.
- **Manual recovery** required in case of failures.
- **High communication and network overhead**.

Modern Alternatives

Because of these drawbacks, alternatives like the **Three-Phase Commit (3PC)**, **Paxos**, or **Raft** consensus algorithms, as well as **eventual consistency** models (used in systems like Amazon DynamoDB or Cassandra), are often preferred in modern distributed systems where fault tolerance and scalability are

more important than strict consistency.

Let me know if you'd like more information on these alternatives!



What is Three-Phase Commit (3PC) and how does it overcome the dependency of 2PC



Three-Phase Commit (3PC) is an improvement over the **Two-Phase Commit (2PC)** protocol that aims to overcome some of the key issues, particularly the **blocking nature** and **single point of failure** that arise in 2PC. 3PC introduces an additional phase to further ensure that participants in a distributed transaction do not get stuck in an uncertain state when there are failures.

Key Goals of 3PC:

1. **Non-blocking**: 3PC tries to avoid situations where participants or the coordinator are left in a waiting or "uncertain" state due to failures or network issues.
2. **Better Fault Tolerance**: 3PC reduces the reliance on a single coordinator, and introduces timeouts and intermediate states to allow participants to recover without blocking the system.

Phases of Three-Phase Commit (3PC)

1. Phase 1: CanCommit (Prepare to Commit)

- The **Coordinator** sends a request to all participants, asking whether they can prepare for the transaction (similar to the Prepare Phase in 2PC).
- Each **Participant** responds with a **Yes (prepared)** or **No** based on whether they are ready to commit.
- If any participant responds with a **No**, the transaction is aborted.

2. Phase 2: PreCommit (Prepared State)

- If all participants respond **Yes**, the coordinator enters a new state called **PreCommit**.
- The **Coordinator** then sends a **PreCommit** message to all participants, signaling that they should prepare to commit but not actually commit the transaction yet.

- Participants acknowledge the **PreCommit** and enter a state where they are prepared to commit the transaction but have not yet made the final commit.
- This phase is crucial because it allows participants to coordinate with each other without blocking, thus avoiding uncertain states in case of failure.

3. Phase 3: DoCommit (Commit or Abort)

- Once the coordinator receives acknowledgment from all participants that they are ready, it sends a **DoCommit** message, instructing participants to commit the transaction.
- If, at any point, a failure occurs (e.g., a timeout happens, or a participant crashes), participants can automatically abort after waiting for a predefined period, preventing a blocking situation.

How 3PC Overcomes the Drawbacks of 2PC

1. Non-Blocking Mechanism

- In 2PC, participants can be left in an uncertain state (especially if the coordinator crashes after receiving the “Yes” votes), and the transaction may hang indefinitely. In 3PC, the **PreCommit** phase introduces an intermediate state that ensures participants are never uncertain. If communication is lost, participants can roll back the transaction or act based on a timeout mechanism, making the protocol **non-blocking**.

2. Timeouts and Fail-Safes

- 3PC introduces timeouts for participants and the coordinator. If the coordinator does not receive a response within a certain time or a participant does not receive the **DoCommit** message, the system can abort the transaction and release resources. This prevents **infinite waiting** and resource locking, which are common problems in 2PC.

3. PreCommit as a Safety Mechanism

- The **PreCommit** phase ensures that all participants are aware of each other's state before proceeding to the final commit. If any participant fails or the coordinator crashes after the PreCommit, the transaction can be safely aborted. This adds a layer of safety, reducing the dependency on a single coordinator.

4. Coordinator Failure Tolerance

- In 3PC, if the **Coordinator** fails after sending the **PreCommit** but before the **DoCommit**, the participants can use the timeout mechanism to abort the transaction. They don't remain blocked because they are in a well-defined **PreCommit** state where they know whether to commit or roll back based on the protocol's rules.

Comparison Between 2PC and 3PC

Aspect	2PC (Two-Phase Commit)	3PC (Three-Phase Commit)
Blocking	Yes, blocking can occur if the coordinator crashes.	No, non-blocking with the PreCommit phase and timeouts.
Single Point of Failure	Yes, if the coordinator fails, the transaction is left in an uncertain state.	Reduced, as participants can recover from certain failures via the PreCommit phase.
Timeout Handling	No built-in timeout handling, can lead to deadlocks.	Timeouts are built in to avoid indefinite waiting and resolve failures.
Communication Overhead	Less overhead (2 phases).	Slightly more overhead (3 phases), but ensures better fault tolerance.
Resource Locking	Locks resources until the coordinator commits or aborts.	Participants don't hold locks indefinitely due to timeouts and safe abort mechanisms.
Recovery	Manual intervention may be required.	Participants can independently decide to commit or abort based on timeouts.

Drawbacks of 3PC

Although 3PC solves some of the issues with 2PC, it is not without its own limitations:

- **Communication Overhead:** Since 3PC introduces an additional phase, it adds more communication steps between the coordinator and participants, which can lead to higher latency, especially in large systems.
- **Network Partitioning:** 3PC does not fully solve the issue of network partitions. If the network gets partitioned, 3PC can still fail, although it provides more safety than 2PC.

Conclusion

3PC offers improvements over 2PC by introducing the **PreCommit** phase, which ensures participants are never in an uncertain state, and by implementing **timeouts** to handle failures. While 3PC adds some complexity and communication overhead, it is generally more fault-tolerant and non-blocking compared to 2PC, making it more suitable for certain distributed systems where fault tolerance is critical.

Would you like more details on the specific use cases or implementation of 3PC?



What is Saga Design Pattern. Kindly explain it in details?



The **Saga Design Pattern** is an architectural pattern used to manage distributed transactions in microservices or distributed systems. Unlike traditional ACID transactions that rely on strict consistency and use protocols like **Two-Phase Commit (2PC)**, the Saga pattern embraces the concept of **eventual consistency** and coordinates a long-running sequence of smaller, independent transactions across multiple services.

Each step in the saga is a transaction that updates the data of a single service, and if something goes wrong, the system executes compensating transactions to undo the changes made by preceding steps. This ensures the system remains consistent without the need for locking resources or waiting for a global commit.

How the Saga Pattern Works

The Saga pattern breaks a larger transaction (that spans multiple services) into a sequence of smaller, isolated transactions. Each transaction is executed independently by one service, and the results are shared between services through events. If one step fails, a compensating transaction is triggered to revert or undo the changes of previous steps.

Types of Sagas

There are two main types of Saga patterns:

1. **Choreography-based Saga** (Event-Driven)
2. **Orchestration-based Saga** (Centralized Control)

Let's look at both in detail:

1. Choreography-Based Saga (Event-Driven)

In a **Choreography-based Saga**, each service listens to events from other services and takes action accordingly. There is **no central coordinator**; instead, the services communicate through events to ensure that each step in the saga is executed or undone if necessary.

Example Workflow:

- Service A starts the saga by performing a local transaction and publishing an event (e.g., "Order Created").

- Service B listens for the "Order Created" event and performs its own local transaction, such as reserving inventory, then it publishes another event (e.g., "Inventory Reserved").
- Service C listens to the "Inventory Reserved" event and performs another action, like processing the payment.
- If any service fails, it publishes a compensating event that triggers undo actions in previously completed services (e.g., canceling the order or rolling back inventory).

Advantages:

- **Loose coupling:** Each service independently listens to and reacts to events.
- **Scalability:** The system is highly scalable since there is no central coordinator, and services can handle events asynchronously.

Disadvantages:

- **Complexity in error handling:** Compensating transactions and event handling logic can become complicated, especially in a system with many services.
 - **Lack of visibility:** Without a central coordinator, it can be hard to trace the state of the overall transaction or saga.
-

2. Orchestration-Based Saga (Centralized Control)

In an **Orchestration-based Saga**, there is a **central orchestrator** (a controller or coordinator) that is responsible for controlling the flow of the saga. The orchestrator calls each service in sequence and handles compensating transactions if any step fails.

Example Workflow:

- The **Saga Orchestrator** starts the saga by sending a request to Service A (e.g., "Create Order").
- Once Service A completes its local transaction, the orchestrator then sends a request to Service B (e.g., "Reserve Inventory").
- After Service B completes its task, the orchestrator proceeds to Service C (e.g., "Process Payment").
- If any service fails, the orchestrator triggers compensating actions in reverse order, undoing the completed steps (e.g., rolling back inventory, canceling the order).

Advantages:

- **Central control and visibility:** The orchestrator has a complete view of the transaction's progress and can handle errors and compensations in a more controlled manner.

- **Easier to manage:** With a centralized coordinator, it's easier to implement the saga logic, especially error handling and recovery.

Disadvantages:

- **Single point of failure:** The orchestrator can become a bottleneck or a single point of failure in the system.
 - **Limited scalability:** The orchestrator may struggle to scale when dealing with many transactions simultaneously, especially in large systems.
-

Key Concepts in the Saga Pattern

1. Eventual Consistency

- The Saga pattern embraces the idea that the system may not be immediately consistent, but it will reach a consistent state eventually. This is crucial in distributed systems, where waiting for strict consistency can block performance and scalability.

2. Compensating Transactions

- When a failure occurs, a **compensating transaction** undoes the effect of the previous transactions. For example, if a payment fails, the system may need to trigger compensating actions like releasing reserved inventory or canceling an order.

3. Rollback vs Forward Recovery

- **Forward Recovery** means proceeding with compensating transactions to handle failures, instead of traditional rollback mechanisms that revert the entire transaction. This is more suitable in distributed systems where long-running processes are involved.

4. Asynchronous Communication

- In a choreography-based saga, communication is often **asynchronous**, meaning services operate independently, and each listens for the events it cares about. This allows the system to handle more transactions simultaneously but adds complexity to event coordination.
-

Example: E-commerce Order Processing Saga

Let's consider an example of an e-commerce platform where a customer places an order. The process involves multiple services: Order Service, Inventory Service, and Payment Service.

Workflow:

1. **Order Service** creates a new order and publishes an event "Order Created."
2. **Inventory Service** listens to the "Order Created" event, checks and reserves the requested inventory, and then publishes an event "Inventory Reserved."
3. **Payment Service** listens to the "Inventory Reserved" event, processes the payment, and publishes an event "Payment Completed."
4. **Shipping Service** listens to the "Payment Completed" event and starts shipping the order.

Handling Failure:

- If the **Payment Service** fails, the **Order Service** may cancel the order by triggering compensating actions:
 - **Inventory Service** rolls back the inventory reservation.
 - **Order Service** marks the order as canceled.
-

Advantages of the Saga Design Pattern

1. **Decoupling:** Services in the saga pattern are loosely coupled and communicate through events, allowing them to evolve independently.
2. **Scalability:** By using event-driven architecture (in the case of choreography), services can be scaled independently based on their needs.
3. **Error Recovery:** The pattern introduces the concept of compensating transactions, allowing recovery from partial failures.
4. **Avoidance of Distributed Locks:** Since each service manages its own local transaction, there is no need for distributed locking, which can become a bottleneck in traditional distributed transactions.

Disadvantages of the Saga Design Pattern

1. **Complexity:** Implementing compensating transactions and ensuring eventual consistency can be difficult and requires careful design, especially for large systems with many services.
 2. **Coordination Overhead:** In orchestration-based sagas, the coordinator or orchestrator can become a bottleneck and might reduce the scalability of the system.
 3. **Visibility in Choreography:** Without a centralized orchestrator, it may be difficult to trace the overall progress of the saga in a choreography-based saga, leading to challenges in monitoring and debugging.
-

When to Use the Saga Pattern

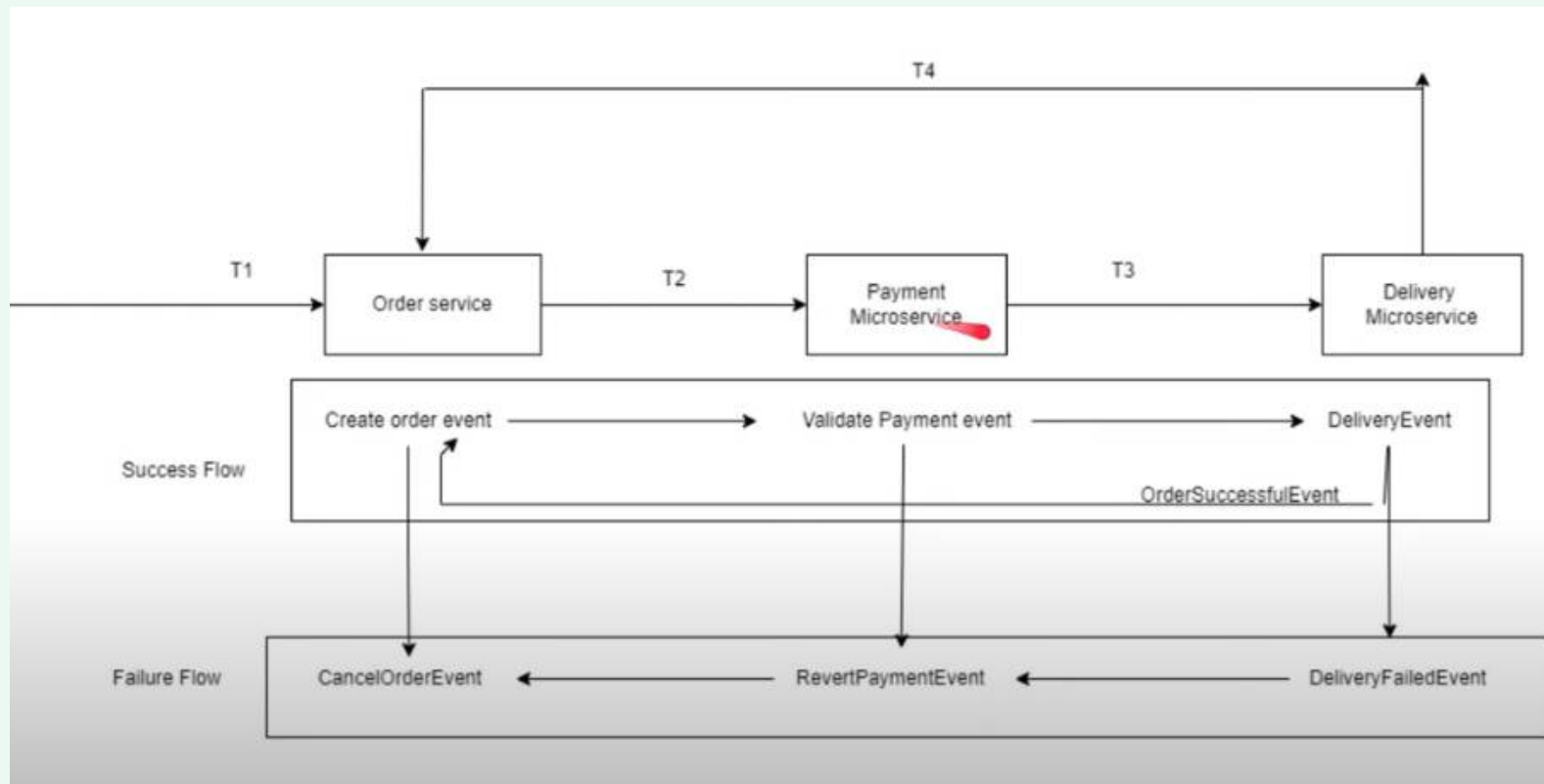
- **Microservices Architecture:** When you're building a system with many independently deployable services, the Saga pattern is often preferred over distributed transactions that require strong consistency.
 - **Long-Running Transactions:** Sagas work well in scenarios where transactions are long-running and can be broken down into a sequence of smaller steps, with compensating actions for failure.
 - **Eventual Consistency:** If strict ACID properties are not required, and eventual consistency is acceptable, the Saga pattern is a good fit.
-

Conclusion

The **Saga Design Pattern** offers an elegant way to manage distributed transactions in microservices, providing flexibility, scalability, and resilience. By breaking a large transaction into smaller, independent steps with compensating actions for failures, the Saga pattern avoids the drawbacks of traditional distributed transaction protocols like 2PC, while enabling eventual consistency in complex systems.

Let me know if you'd like more details on any specific aspect of the Saga pattern!





Kindly explain the above diagram in details



The diagram you shared appears to depict a **Saga Design Pattern** implementation in a microservices architecture, specifically in the context of order processing. Here's a detailed explanation of the flow:

Microservices Involved:

1. **Order Service:** Handles order creation and coordination of the transaction.
2. **Payment Microservice:** Manages payment validation and processing.
3. **Delivery Microservice:** Handles delivery and shipping.

Events and Flows

Success Flow (Top section of the diagram):

The success flow depicts the normal sequence of operations when everything works as expected. The process involves four key steps (T1 to T4):

1. T1 - Order Creation:

- The **Order Service** receives a request to create an order.
- Once the order is created, the **Order Service** emits a ``CreateOrderEvent`` to notify the system that an order has been successfully created.

2. T2 - Payment Validation:

- After the order is created, the **Payment Microservice** is triggered by the ``CreateOrderEvent`` to validate and process the payment.
- The ``ValidatePaymentEvent`` signifies that the payment process has begun.

3. T3 - Delivery Initiation:

- Once the payment is validated, the **Payment Microservice** triggers the **Delivery Microservice** by sending a ``DeliveryEvent`` to initiate the delivery process.
- The **Delivery Microservice** handles the logistics of shipping the product to the customer.

4. T4 - Order Success:

- After the delivery is successfully initiated, the **Delivery Microservice** sends an ``OrderSuccessfulEvent`` back to the **Order Service**, indicating that the entire transaction (order, payment, and delivery) has been completed successfully.

Failure Flow (Bottom section of the diagram):

The failure flow handles what happens when one of the steps in the process fails. The system triggers compensating transactions to undo the previously completed actions. Here's how it works:

1. Payment Failure:

- If the payment fails, the **Payment Microservice** triggers a ``RevertPaymentEvent`` to undo any reservation or payment processing it had done.
- Simultaneously, the **Order Service** listens to this event and triggers a ``CancelOrderEvent`` to cancel the order.

2. Delivery Failure:

- If the payment is successful but the delivery fails (perhaps due to an invalid address or unavailable delivery options), the **Delivery Microservice** emits a ``DeliveryFailedEvent``.
- The **Payment Microservice** listens to this event and issues a ``RevertPaymentEvent`` to reverse the payment.
- Finally, the **Order Service** listens to the ``CancelOrderEvent`` to undo the order creation.

Detailed Event Flow Breakdown:

1. **CreateOrderEvent**: Initiated by the **Order Service** after creating the order.
 - Triggers the **Payment Microservice** to start payment validation.
2. **ValidatePaymentEvent**: Indicates the payment process has started.
 - On success, triggers the **Delivery Microservice** to start delivery.
3. **DeliveryEvent**: Signals the start of delivery after payment is validated.
 - On success, triggers the final ``OrderSuccessfulEvent``.
4. **RevertPaymentEvent** (Compensation): This event is triggered if any failure occurs.
 - If payment fails, it reverses any payment action taken earlier.
 - If delivery fails, it also undoes the payment.
5. **CancelOrderEvent** (Compensation): If payment or delivery fails, this event cancels the order in the **Order Service** to ensure data consistency.
6. **OrderSuccessfulEvent**: Signals that all steps (order creation, payment, and delivery) have been completed successfully.

Summary:

- **Success Flow**: The process starts from order creation, followed by payment validation, and then delivery initiation, all culminating in a successful transaction.
- **Failure Flow**: If payment or delivery fails, compensating transactions (like reverting the payment and canceling the order) ensure that the system remains consistent, embracing eventual consistency while handling failures gracefully.

This diagram is a clear representation of the **Saga Pattern** where multiple services (Order, Payment, Delivery) are involved, and the system reacts to events by either completing the transaction or rolling back (via compensating actions) when failures occur.

Let me know if you'd like more details on any specific part!



How SAGA DP handles failure of any individual SAGA?

- The saga pattern provides **transaction management** with using a sequence of local transactions of microservices. **Every microservices** has its own database and it can able to **manage local transaction in atomic way** with strict consistency.
- So saga pattern **grouping these local transactions and sequentially** invoking one by one. Each local transaction updates the database and **publishes an event to trigger the next local transaction**.
- If one of the step is failed, than saga patterns trigger to **rollback transactions** that are a set of **compensating transactions that rollback the changes on previous microservices** and restore data consistency.

Kindly explain the above diagram in details



The image you shared explains how the **Saga Design Pattern** (Saga DP) handles failures of individual sagas in a microservices architecture. Here's a detailed breakdown of the key points outlined in the image:

1. Transaction Management with Local Transactions

- The **Saga pattern** provides **transaction management** by breaking down a global transaction (which spans multiple microservices) into a series of **local transactions**.

- Each microservice involved in the saga is responsible for managing its own local transaction **independently** and can do so in an **atomic way** (meaning each local transaction is either fully completed or fully rolled back).
- **Strict consistency** is maintained within each microservice's local transaction. This means that while the overall system may embrace **eventual consistency**, each service ensures that its changes are committed fully or not at all.

2. Grouping and Sequential Invocation of Local Transactions

- The Saga pattern groups these **local transactions** into a sequence and **invokes them one by one**. After one local transaction completes, it triggers the next one in the sequence.
- Each local transaction **updates its own database** and, upon completion, **publishes an event** that signals the next step in the saga. This means that the flow from one transaction to another is event-driven.
- For example, in an order-processing saga, the creation of an order by the **Order Service** will publish an event to trigger the **Payment Service** to validate the payment. This sequential, event-driven flow continues through all the services involved.

3. Handling Failures and Compensating Transactions

- If any one of the local transactions in the saga **fails**, the Saga pattern provides a mechanism for recovery by triggering **compensating transactions**.
- A **compensating transaction** is designed to **rollback** the changes made by the previous microservices, ensuring that the system can revert to a consistent state.
- This allows the system to maintain **data consistency** even in the event of failures. For instance, if the payment fails after the order is created, a compensating transaction may cancel the order to maintain consistency between the different microservices.

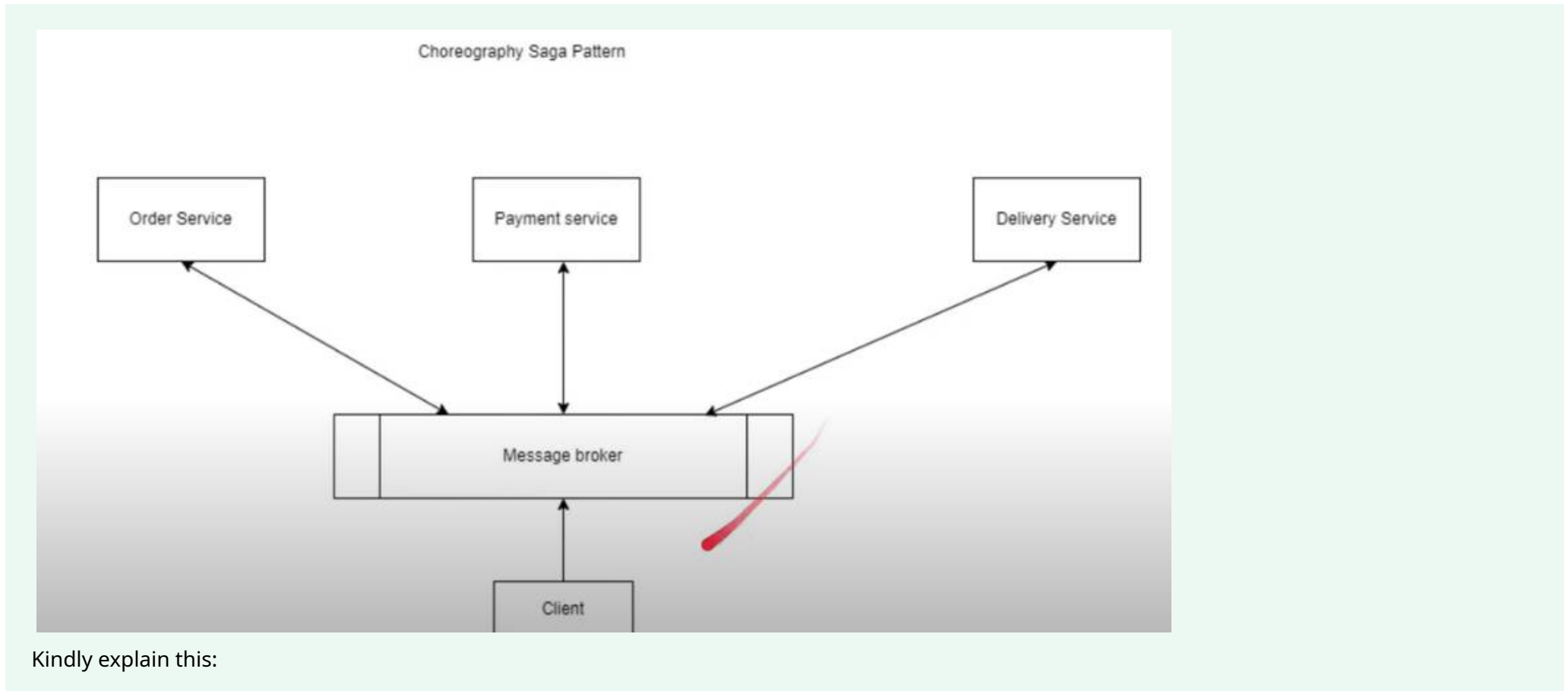
Summary of Key Concepts in the Diagram:

- **Local Transactions:** Each microservice handles its own transaction locally, ensuring atomicity within its own scope.
- **Event-Driven Flow:** Once a microservice completes its transaction, it publishes an event that triggers the next service in the saga.
- **Compensating Transactions:** If a failure occurs, compensating transactions are triggered to roll back previous actions and restore consistency across all microservices.

Practical Implications:

- **Eventual Consistency:** The overall system achieves consistency not through global locking or a centralized transaction manager (as in Two-Phase Commit), but by managing a sequence of independent local transactions and compensating for failures if necessary.
- **Fault Tolerance:** By allowing each microservice to undo its changes through compensating actions, the Saga pattern provides resilience in distributed systems, ensuring that even if one service fails, the system can recover.

This is a high-level summary of how the Saga pattern handles failures. Let me know if you'd like more specific examples or further clarification on any part of the pattern!



The diagram you shared represents a **Choreography-based Saga Pattern** in a microservices architecture using a **message broker** to facilitate communication between services. Here's a detailed explanation of the key components and how the pattern works:

Components in the Diagram:

1. **Order Service:** Handles the order creation process.
2. **Payment Service:** Manages the payment transaction and validation.
3. **Delivery Service:** Handles the delivery or shipping of the order.
4. **Message Broker:** Acts as an intermediary for communication between microservices.
5. **Client:** The entity that initiates the workflow (e.g., a customer placing an order).

How the Choreography Saga Pattern Works:

In a **Choreography-based Saga**, there is **no centralized coordinator** or orchestrator. Instead, each microservice is responsible for executing its part of the transaction and **publishing events** to trigger the next step in the workflow. The message broker is used to relay these events and messages between services.

Workflow Overview:

1. Client Request:

- The client sends a request to the **Order Service** to place an order (e.g., ordering a product in an e-commerce application).

2. Order Service:

- Upon receiving the request, the **Order Service** processes the order and creates an entry for it.
- After successfully creating the order, the **Order Service** sends an event (e.g., ``OrderCreatedEvent``) to the **Message Broker**.
- The event informs the system that the order has been created and the next step, payment processing, should begin.

3. Payment Service:

- The **Payment Service** listens for the ``OrderCreatedEvent`` on the **Message Broker**.
- When it receives the event, it processes the payment for the order (e.g., charging the customer's card).
- After successfully processing the payment, the **Payment Service** sends another event (e.g., ``PaymentSuccessfulEvent``) to the **Message Broker**, signaling the next step in the workflow: initiating delivery.

4. Delivery Service:

- The **Delivery Service** listens for the ``PaymentSuccessfulEvent`` on the **Message Broker**.
- Upon receiving the event, the **Delivery Service** initiates the delivery of the product.

- After successfully starting the delivery, the **Delivery Service** could send an event (e.g., ``DeliveryStartedEvent`` or ``OrderCompletedEvent``), indicating that the entire saga is complete.

Message Broker:

- The **Message Broker** is a crucial part of this pattern. It acts as an intermediary between the services, allowing them to communicate through **events** instead of direct calls.
- The services publish their events to the message broker, which then routes these events to the appropriate services that are listening for them.
- Common message brokers used in this pattern include **Kafka**, **RabbitMQ**, and **AWS SQS**.

Event-Driven Choreography:

In this choreography-based approach, the flow of the saga is **event-driven**, meaning each microservice acts independently, performing its task based on the events it receives from the message broker. Each service listens for specific events, executes its local transaction, and then publishes new events to trigger the next steps in the saga.

Failure Handling:

If a failure occurs in any service, the system would need to trigger compensating transactions to undo the changes made by previous services. For example:

- If the **Payment Service** fails to process the payment, it might publish a ``PaymentFailedEvent``, which could trigger the **Order Service** to cancel the order.
- Similarly, if delivery fails, the **Delivery Service** might trigger a compensating event to roll back the payment or mark the order as failed.

Advantages of Choreography:

1. **Loose Coupling**: Services do not need to know about each other directly; they only need to listen for and publish events. This reduces tight coupling between services.
2. **Scalability**: Each service can be scaled independently, as they only interact through asynchronous messages.
3. **No Central Coordinator**: Since there is no central orchestration, this pattern avoids a single point of failure and reduces complexity in the coordination logic.

Disadvantages of Choreography:

1. **Complex Event Management**: In large systems with many services, event management can become complex and hard to track.

2. **Lack of Central Control:** Without a central orchestrator, it can be challenging to have full visibility into the progress of the overall transaction, which can make debugging and monitoring more difficult.

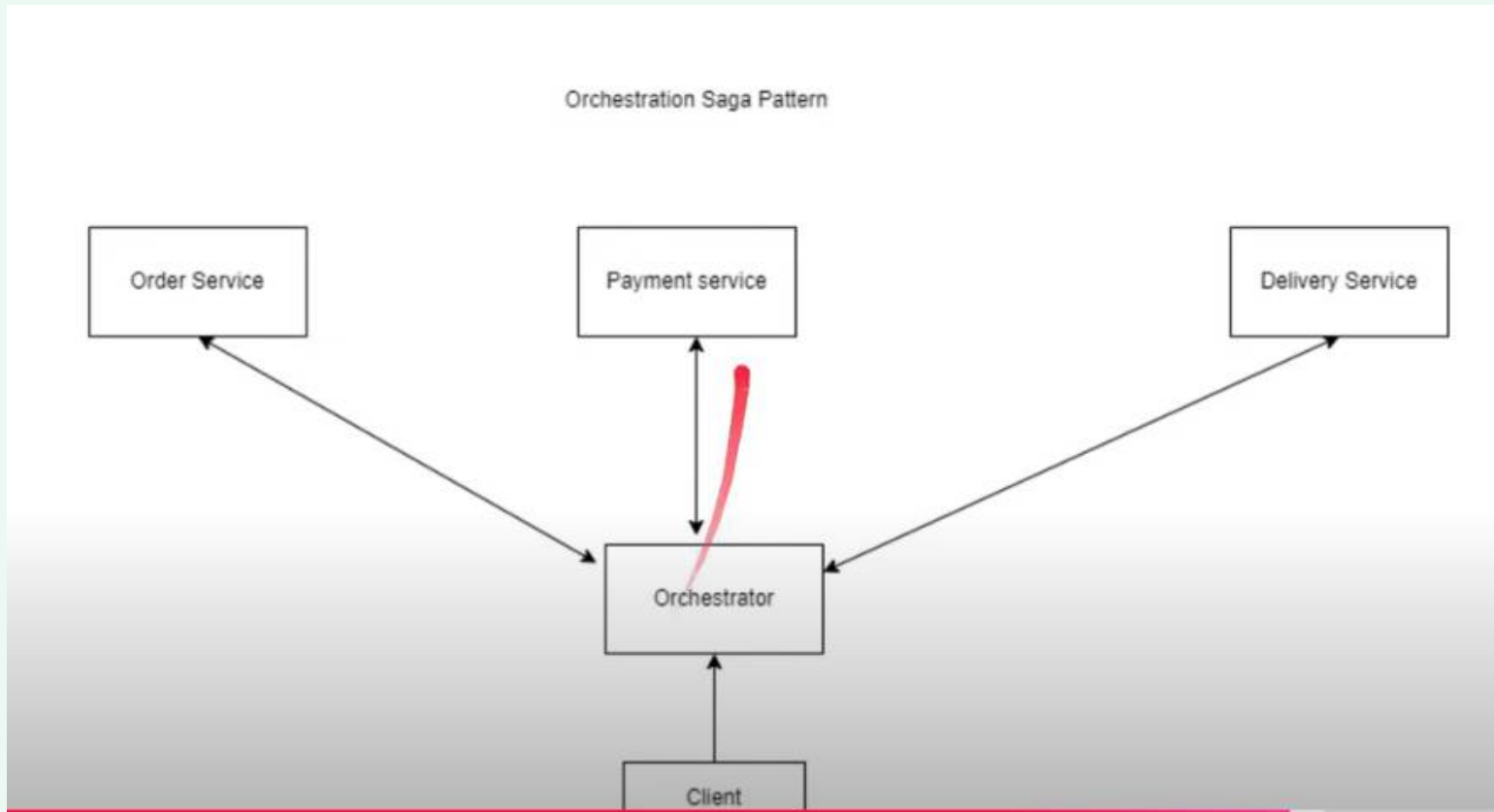
Summary:

- In a **Choreography-based Saga**, services communicate indirectly through events published to a **Message Broker**.
- The **Order Service**, **Payment Service**, and **Delivery Service** handle their respective parts of the transaction based on the events they receive.
- The **Message Broker** enables asynchronous, event-driven communication between the microservices, ensuring that the transaction flows smoothly from one service to the next.

This approach provides a loosely coupled, scalable system but requires careful handling of events and compensating actions for failure scenarios.

Let me know if you'd like further clarification on any part of this!





Kindly explain above.



The diagram you shared illustrates the **Orchestration Saga Pattern**, which is a different approach to managing distributed transactions in a microservices architecture compared to the Choreography Saga Pattern. Here's an explanation of the components and how the **Orchestration Saga Pattern** works:

Components in the Diagram:

1. **Order Service:** Handles the creation and management of orders.
2. **Payment Service:** Manages the payment process for the order.

3. **Delivery Service:** Handles the delivery or shipping of the order.
4. **Orchestrator:** This is the central component that coordinates and controls the flow of the saga by explicitly invoking each service in the correct order.
5. **Client:** Initiates the saga (e.g., a customer placing an order).

How the Orchestration Saga Pattern Works:

In the **Orchestration Saga Pattern**, there is a **central orchestrator** that is responsible for coordinating the saga across multiple services. Unlike the **Choreography-based Saga**, where services interact via events, in this pattern, the orchestrator **directly manages the execution** of each step in the saga by calling the appropriate services and handling compensating transactions if any failures occur.

Workflow Overview:

1. Client Request:

- The **Client** (e.g., the front-end or an external system) sends a request to the **Orchestrator** to initiate the process (for example, placing an order).

2. Orchestrator and Order Service:

- The **Orchestrator** first calls the **Order Service** to create the order.
- Once the order is successfully created, the **Orchestrator** moves on to the next step, which is payment processing.

3. Orchestrator and Payment Service:

- The **Orchestrator** calls the **Payment Service** to process the payment for the order.
- After the payment is successfully processed, the orchestrator proceeds to the final step: arranging delivery.

4. Orchestrator and Delivery Service:

- The **Orchestrator** calls the **Delivery Service** to handle the shipping or delivery of the order.
- Once the delivery is successfully initiated, the orchestrator can signal that the saga is complete.

5. Completion:

- The **Orchestrator** ensures that each service completes its task in the correct order. Once all steps (order, payment, delivery) are successfully executed, the entire saga is considered successful.

Handling Failures and Compensations:

One of the key responsibilities of the **Orchestrator** is to handle failures and trigger **compensating transactions** when needed. Here's how failures are managed:

1. Failure in Payment:

- If the **Payment Service** fails to process the payment, the **Orchestrator** will trigger a compensating action to **cancel the order** by calling the **Order Service** to undo the order creation.

2. Failure in Delivery:

- If the **Delivery Service** fails after the payment is successful, the **Orchestrator** will handle compensations by calling the **Payment Service** to **refund** the payment and the **Order Service** to **cancel** the order.

Key Features of Orchestration:

- **Central Control:** The **Orchestrator** centrally manages and coordinates the entire saga. It knows the status of each service and explicitly decides which service to call next.
- **Direct Communication:** Unlike the event-driven communication in a choreography-based saga, the orchestrator uses direct calls to each service to execute transactions, making the flow more predictable and easier to manage.
- **Compensating Actions:** If a failure occurs at any step, the orchestrator triggers **compensating transactions** to undo the previously completed steps. For example, if the payment succeeds but the delivery fails, the orchestrator could refund the payment and cancel the order.

Advantages of Orchestration:

1. **Centralized Control:** The **Orchestrator** has complete control over the saga, which makes it easier to manage and monitor the flow of the transaction.
2. **Clear Visibility:** Since the orchestrator manages every step, there's a clear view of the current state of the saga, making debugging and tracking easier.
3. **Simple Error Handling:** The orchestrator is responsible for handling failures and triggering compensating transactions, reducing the complexity for individual services.

Disadvantages of Orchestration:

1. **Single Point of Failure:** The **Orchestrator** can become a bottleneck or a single point of failure in the system. If it fails, the entire transaction process can halt.
2. **Scalability Concerns:** The orchestrator handles the entire flow, which may lead to performance and scalability issues as the number of transactions increases.

Comparison with Choreography:

- In **Choreography**, each service autonomously reacts to events and manages the flow. There is no central controller, but it can lead to complex event management.
- In **Orchestration**, the flow is centrally controlled by the **Orchestrator**, which provides better control but can create a bottleneck.

Summary:

The **Orchestration Saga Pattern** uses a **centralized orchestrator** to manage distributed transactions across microservices, ensuring the correct execution of each step. The orchestrator directly calls each service and is responsible for handling failures by invoking compensating transactions when needed. This pattern provides more control and visibility but introduces the orchestrator as a potential bottleneck or single point of failure.

Let me know if you need more details or clarification on any aspect!



what are the advantages and disadvantages of Choreography and Orchestration saga pattern



The **Choreography** and **Orchestration** Saga Patterns are two different approaches to managing distributed transactions in a microservices architecture. Each has its own set of advantages and disadvantages depending on the complexity of the system, scalability requirements, and how much control and visibility you need. Here's a detailed comparison of both patterns:

Choreography Saga Pattern

Advantages:

1. Loose Coupling:

- In the Choreography pattern, services are **loosely coupled** since they communicate indirectly through events rather than direct service-to-service calls. Each service only knows about the events it needs to react to, making them more independent.

2. No Single Point of Failure:

- There's **no central orchestrator**, meaning there is no single point of failure. Each service operates autonomously, so the system can continue to function even if one service has issues.

3. Asynchronous and Scalable:

- Choreography is usually **asynchronous**, which makes it highly **scalable**. Services can work in parallel and handle a large number of transactions at the same time, which is particularly important in distributed systems.

4. Natural Event-Driven Architecture:

- Choreography fits well into an **event-driven architecture**, where services react to events without the need for explicit control by a central system. This makes it easier to extend the system by adding more services that listen to the same events.

5. Simplicity in Small Systems:

- In small systems with fewer services, the Choreography pattern can be simpler because services are just reacting to events rather than coordinating through a central orchestrator.

Disadvantages:

1. Complex Event Management:

- As the number of microservices grows, managing events becomes **complex**. It can be difficult to trace how an event flows through the system, and services may become tightly coupled to specific event flows over time.

2. Lack of Centralized Control:

- There's no single place where the entire saga can be monitored or controlled. This makes it harder to have **visibility** into the progress of the saga or transaction, making debugging and monitoring more difficult.

3. Difficult to Handle Compensations:

- Handling **compensating transactions** (when something goes wrong) can be challenging since there's no central place where the saga's state is maintained. Services have to independently decide when and how to roll back, making the flow of compensations more complex.

4. Cascading Failures:

- If one service fails or sends out incorrect events, it can result in **cascading failures** across multiple services, and the lack of centralized error handling makes it difficult to manage these failures.

5. Increased Latency:

- Due to the **asynchronous nature** of event processing, there may be increased latency because services do not execute immediately but rather wait for events to be published and processed. This might not be ideal for time-sensitive transactions.

Orchestration Saga Pattern

Advantages:

1. Centralized Control:

- With the **Orchestrator**, there is **centralized control** over the entire workflow. The orchestrator explicitly directs each service to perform its part of the saga, making the flow of the transaction more predictable and **easier to manage**.

2. Better Visibility and Monitoring:

- The **Orchestrator** has a complete view of the entire saga, so it's easier to track the status of the transaction. This provides better **traceability**, **monitoring**, and **debugging**, as the orchestrator knows the current state of each step in the saga.

3. Simplified Compensating Transactions:

- The **Orchestrator** can handle compensating transactions explicitly. If any step fails, it can issue commands to trigger compensating actions to roll back the previous steps, making error handling and recovery more **straightforward**.

4. Predictable Flow:

- Since the **Orchestrator** directs the flow, it's easier to predict the order of execution and the dependencies between services. This is especially useful when the saga has complex workflows and dependencies.

5. Easier to Implement Complex Flows:

- Complex workflows, such as conditional branches (e.g., if payment fails, try again or take a different path), are **easier to implement** in an orchestrated saga because the orchestrator explicitly defines the flow.

Disadvantages:

1. Single Point of Failure:

- The **Orchestrator** can become a **single point of failure**. If the orchestrator crashes or is overloaded, the entire saga may fail. This adds a potential bottleneck in the system and reduces resilience.

2. Increased Latency Due to Centralization:

- The orchestrator introduces additional hops between services since it must coordinate every step. This can add **latency**, particularly in systems with high transaction volumes or complex workflows.

3. Limited Scalability:

- The orchestrator can become a **bottleneck** if there are many sagas happening simultaneously. It must handle all coordination, which can lead to performance issues as the system grows. Scaling the orchestrator itself might become a challenge.

4. Tight Coupling to the Orchestrator:

- Services become **tightly coupled** to the orchestrator since they depend on it to trigger their transactions. If the orchestrator needs to be changed or updated, this can affect all the services it manages.

5. Complexity in Orchestrator Logic:

- While services are decoupled, the **orchestrator's logic** can become complex as it handles the coordination of the entire saga, including error handling, compensations, and retries. This can lead to a large amount of logic residing in one place, which somewhat contradicts the microservices principle of distributed logic.

Comparison Summary

Aspect	Choreography Saga	Orchestration Saga
Control	Decentralized (event-driven, no central control)	Centralized (orchestrator manages the flow)
Scalability	High scalability due to event-driven, async nature	Less scalable due to central orchestration bottleneck
Visibility	Limited visibility (hard to track saga progress)	Full visibility (orchestrator tracks the entire saga)
Error Handling	Complex, compensations are harder to coordinate	Simplified, orchestrator explicitly handles compensations
Coupling	Loose coupling (services react to events)	Tight coupling to orchestrator
Single Point of Failure	No single point of failure	Orchestrator is a single point of failure

Aspect	Choreography Saga	Orchestration Saga
Complexity	Services handle their own logic (can get complex)	Orchestrator can become complex with logic handling
Latency	Lower (but can be high due to async nature)	Potentially higher due to orchestrator coordination
Ideal Use Case	Simple, independent services with fewer steps	Complex workflows with many interdependencies

When to Use Choreography:

- You have a **simple** microservices system with relatively **independent services**.
- You want high scalability and loose coupling between services.
- You prefer a **fully event-driven architecture**.
- Monitoring and debugging complexity is acceptable.

When to Use Orchestration:

- Your system requires **complex workflows** with **multiple steps** and conditional logic.
- You need **centralized control** and full visibility over the saga's execution.
- **Error handling** and compensation need to be centrally managed.
- You are comfortable with the potential bottleneck and complexity of a central orchestrator.

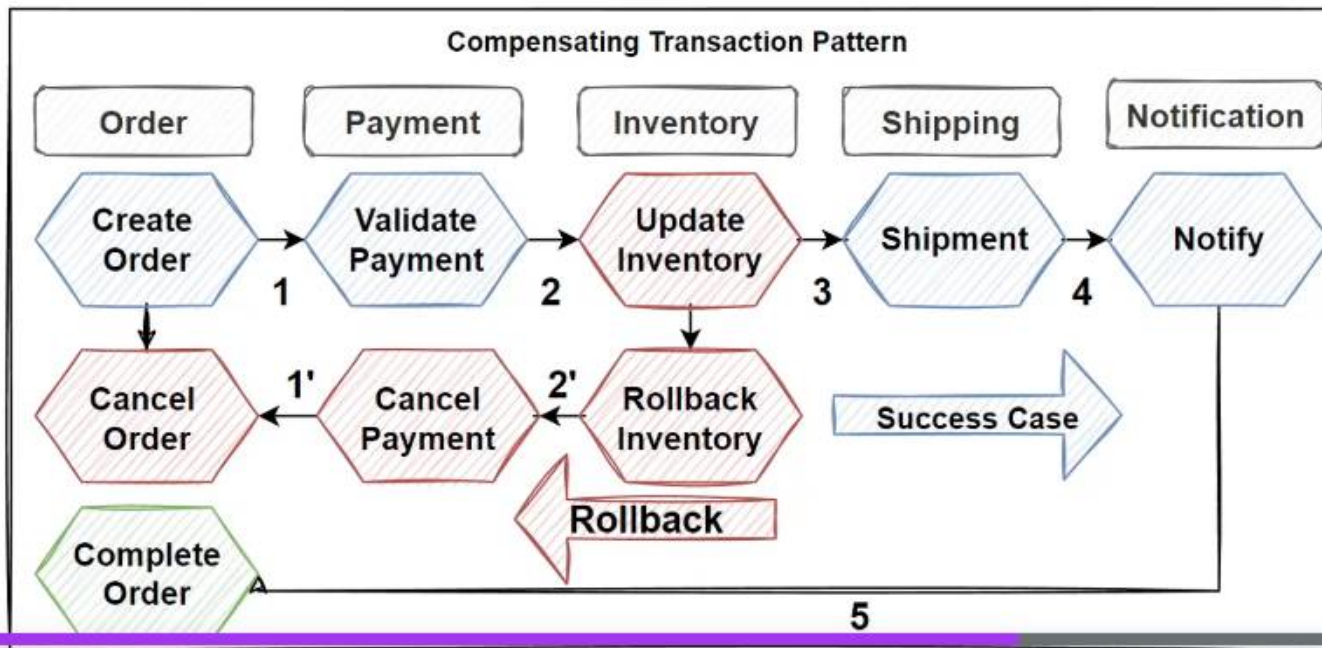
Both patterns have their merits, and the decision between Choreography and Orchestration depends on your system's specific requirements, such as complexity, scalability, and visibility. You might even use a hybrid approach where certain parts of the system use Choreography and others use Orchestration.

Let me know if you need more details on any aspect!



Ecommerce Order Fullfilment Compensating Transaction Pattern

- Customer places an order on the e-commerce application



Please explain above related to compensating transaction pattern



The diagram you've shared represents an **E-commerce Order Fulfillment** process using the **Compensating Transaction Pattern**. This pattern is commonly used in distributed systems and microservices to handle failures gracefully by rolling back previously completed actions. The key concept here is that when a failure occurs, compensating transactions are triggered to undo previous actions and maintain system consistency.

Here's a detailed explanation of each step in the diagram, along with the compensating actions involved:

Workflow Breakdown

The flow represents an e-commerce order fulfillment process involving multiple services, including **Order**, **Payment**, **Inventory**, **Shipping**, and **Notification**. Each service is responsible for a specific step in the transaction. If any step fails, compensating transactions are triggered to roll back the earlier actions.

Success Case:

1. Order Creation (Step 1):

- The process starts when the **Order Service** creates a new order. This is the first step in the transaction.
- Once the order is successfully created, the system moves to the payment validation step.

2. Payment Validation (Step 2):

- The **Payment Service** validates and processes the payment. If the payment is successfully validated, the system proceeds to the inventory update.
- **Success Path:** If everything is successful up to this point, the process moves forward to update the inventory.

3. Inventory Update (Step 3):

- The **Inventory Service** updates the stock, deducting the quantity of the items ordered from the available inventory.
- Once the inventory is updated, the system proceeds to initiate the shipment.

4. Shipment Initiation (Step 4):

- The **Shipping Service** is triggered to initiate the delivery of the order.
- If the shipment is successful, the process moves to notifying the customer.

5. Notification (Step 5):

- The **Notification Service** sends a confirmation message or email to the customer, informing them that the order has been successfully placed and is on its way.

6. Order Completion:

- After all services successfully complete their tasks (order, payment, inventory, shipping, notification), the order is marked as **Complete**.

Failure and Compensation Case:

When a failure occurs at any point in the workflow, **compensating transactions** are triggered to roll back the actions taken by previous services, ensuring consistency in the system.

1. Payment Failure:

- If payment validation fails (between Step 1 and 2), the **Payment Service** sends a ``Cancel Payment`` event (Step 1').
- The system triggers a compensating transaction to **cancel the order** (via the ``CancelOrderEvent``).
- No further steps, such as inventory or shipping, are taken since the order is already canceled.

2. Inventory Failure:

- If the inventory update fails (between Step 2 and 3), the **Inventory Service** triggers a rollback by sending a ``RollbackInventoryEvent`` (Step 2').
- This compensating transaction undoes the inventory changes, rolling back the stock that was deducted.
- The system also compensates by **canceled the payment** (Step 1'), ensuring that the customer is not charged for the failed order.
- The entire transaction is rolled back, and no shipping or notification actions are taken.

3. Shipping Failure:

- If the shipping process fails after the inventory is updated (Step 3), compensating actions are triggered, but the diagram does not explicitly show the shipping rollback here. However, typically:
 - The inventory may be adjusted back (rollback), and the payment might be refunded.

Compensating Transactions (Rollback):

- The compensating actions are triggered by failures, and each service in the flow knows how to undo its work in case of a failure.
- For example:
 - If the **Payment Service** fails, the system cancels the order (Step 1').
 - If the **Inventory Service** fails, the system rolls back both the inventory and the payment (Step 2').
- The goal is to ensure the system remains consistent even when some services fail.

Key Concepts in the Diagram:

- **Compensating Transaction Pattern:**
 - This pattern ensures that when a failure occurs in a distributed system, earlier successful operations are undone (compensated) to keep the system consistent.
 - Compensating transactions are actions that reverse the effects of previously completed transactions in the workflow.

- **Rollback:**

- Whenever a failure is detected (in payment, inventory, or shipping), the system initiates a rollback to reverse the previous actions taken by the services. This rollback ensures that no partial or inconsistent state remains in the system.

- **Red Hexagons:**

- The red hexagons in the diagram represent the **compensating transactions** or rollback actions. These are triggered when the main transaction fails to complete successfully.

- **Success Case:**

- The process completes successfully when all services (order, payment, inventory, shipping, and notification) complete their respective tasks without failures. The success flow leads to the final "Complete Order" step.

Summary:

- The diagram showcases the **Compensating Transaction Pattern** in an e-commerce system where multiple services (Order, Payment, Inventory, Shipping, Notification) are involved in fulfilling an order.
- In case of failure at any step, compensating transactions are triggered to roll back the earlier steps and ensure that the system remains consistent.
- If all steps are successful, the process proceeds to completion, and the order is fulfilled.

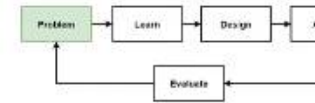
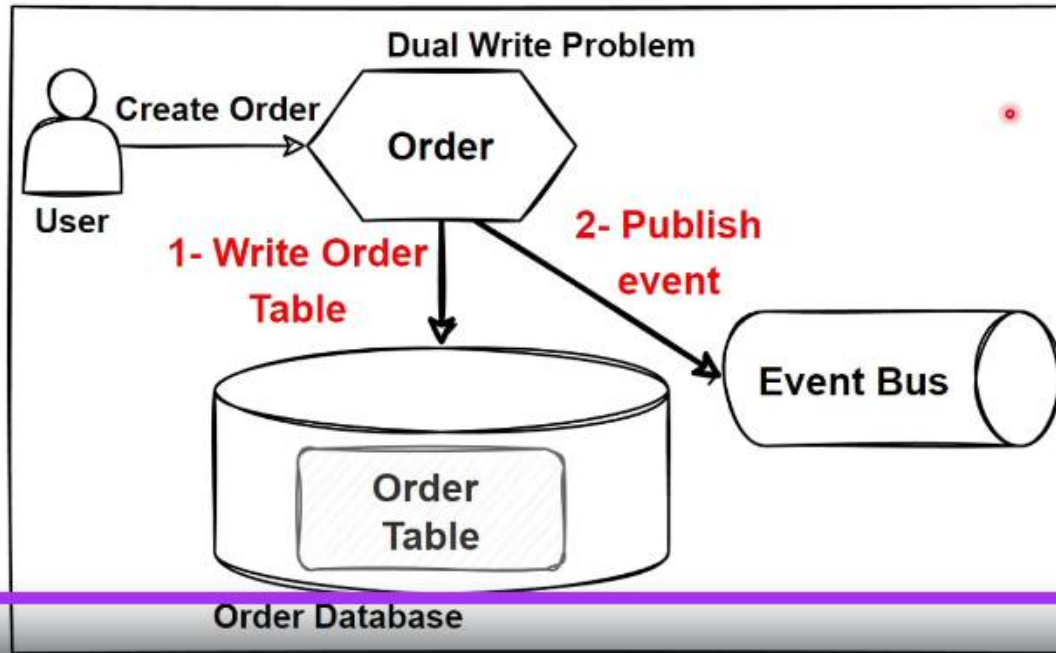
This pattern ensures that even in distributed systems, where failures are common, the system can gracefully handle errors and maintain data consistency through compensation.

Let me know if you need more details or clarification on any part!



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

2 / 5:06

Mehmet Ozkaya 522

What is Dual Write problemn. Kindly explain above with reference to given diagram



The diagram you shared illustrates the **Dual-Write Problem**, which is a common issue in distributed systems, especially when writing to multiple systems (such as a database and an event bus) simultaneously but without a unified transaction. This problem arises because the two operations are not atomic, meaning there's no guarantee that both will either succeed or fail together.

What is the Dual-Write Problem?

The **Dual-Write Problem** occurs when a system performs two independent writes to different systems (like a database and an event bus) as part of a single logical operation, but the writes are not executed in a single atomic transaction. If something goes wrong in the middle of these operations, there could be an **inconsistent state** between the two systems. Specifically, one write may succeed, while the other may fail, leading to data inconsistency.

Breakdown of the Diagram:

In the given example, when a user places an order, two main operations need to happen:

1. **Write to the Order Table:** The new order is written into the **Order Table** in the **Order Database**.
2. **Publish an Event:** The system also needs to publish an event (e.g., ``order_created``) to the **Event Bus** (such as **Apache Kafka**) so that other services in the system are notified that a new order has been placed.

Problem Scenario:

In a **dual-write scenario**, both of these operations—writing to the database and publishing the event—are **independent transactions**. Since they are not part of a single atomic operation, two failure cases can happen:

1. Order Written, Event Not Published:

- The order is successfully written to the **Order Table**, but due to a failure (e.g., network issue, Kafka service failure), the event is **not published** to the **Event Bus**.
- As a result, downstream systems (such as the payment service or inventory service) that rely on the ``order_created`` event won't be notified that a new order was created. This causes the system to be in an **inconsistent state** because the database indicates that an order exists, but no event was sent to trigger the rest of the workflow.

2. Event Published, Order Not Written:

- The event is successfully published to the **Event Bus**, but the order fails to be written to the **Order Table** (e.g., due to a database failure).
- In this case, the rest of the system (e.g., payment and inventory services) will receive the event and proceed with payment and inventory updates, but the **Order Table** in the database does not reflect the new order. This again leads to an inconsistent state where the downstream systems process an order that doesn't actually exist in the database.

Order of Execution (Run Transactions):

- The diagram shows that you could run the transactions in two different orders:
 - **1 then 2:** Write to the order table first, then publish the event.

- **2 then 1:** Publish the event first, then write to the order table.
- In either case, you are performing **dual writes** across two different systems, and failures in the middle of the operation lead to inconsistent states.

Why is this a Problem?

- **No Transactional Guarantees:** There's no **transaction** that guarantees both the database write and the event publication happen together. If one of these operations fails and the other succeeds, the system enters an inconsistent state.
- **Data Inconsistency:** The order service (or other microservices) may be in an inconsistent state if the database and the event bus do not reflect the same state of the transaction.
- **Hard to Detect:** Often, detecting these issues can be difficult because there is no immediate feedback if one of the writes fails, leading to undetected discrepancies in the system.

Solutions to the Dual-Write Problem

To resolve or mitigate the dual-write problem, several approaches can be used:

1. Distributed Transactions (2PC or 3PC):

- Using a protocol like **Two-Phase Commit (2PC)** can ensure that both the database write and event publishing are atomic. However, this approach is often slow and not widely recommended in distributed microservices architectures due to scalability issues.

2. Transactional Outbox Pattern:

- In this pattern, the application first writes the event to the **same database** as the business data (e.g., writing both the order data and the event data to the same **Order Table** or a dedicated outbox table).
- A separate process (e.g., an event relay or worker) later reads the outbox table and publishes the event to the **Event Bus**. This ensures that the event and the order data are written in the same transaction, avoiding inconsistencies.

3. Change Data Capture (CDC):

- In a **Change Data Capture (CDC)** pattern, the system monitors changes to the database and automatically publishes corresponding events to the event bus when data changes. This way, any insert, update, or delete operation in the database is translated into an event without requiring a dual write.

4. Idempotency and Retry Mechanism:

- Implementing **idempotency** in services and ensuring that the system can retry failed operations can help mitigate the effects of the dual-write problem. However, this doesn't completely eliminate the issue.

5. Outbox with Saga Pattern:

- Combining the **Outbox Pattern** with the **Saga Pattern** can ensure that compensating actions are taken in case of failure, further improving consistency.

Summary:

The **Dual-Write Problem** arises when two operations (like writing to a database and publishing an event) are performed independently and not atomically. If either operation fails, the system can end up in an inconsistent state. This is especially problematic in microservices architectures where different services depend on consistent data and event flows.

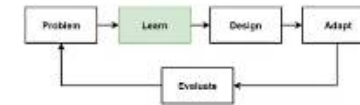
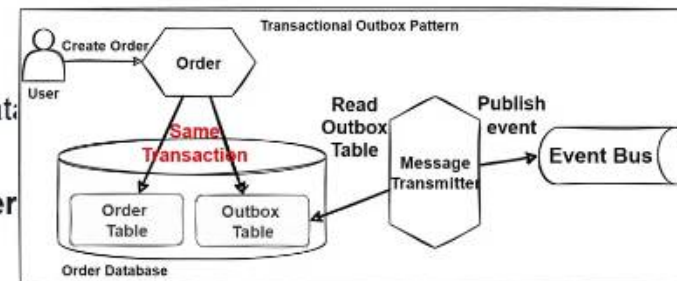
The diagram illustrates the flow where an **Order Service** writes to the **Order Table** and publishes an event to an **Event Bus**, highlighting the risk of **dual writes**. In order to avoid the problems associated with dual writes, patterns like **Transactional Outbox**, **CDC**, or **distributed transactions** can be used to ensure consistency between the database and the event bus.

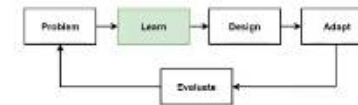
Let me know if you'd like more details on any specific solution to the dual-write problem!



Transactional Outbox Pattern

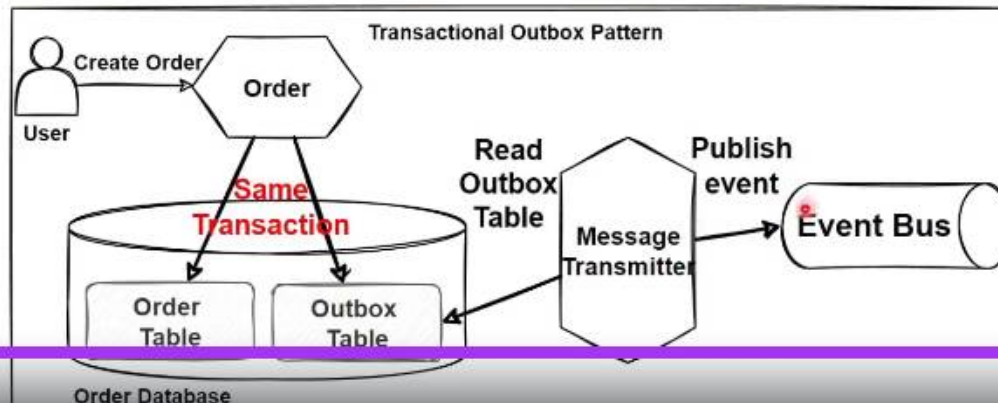
- The idea is to have an **"Outbox" table** in the microservice's database. It provides to publish events reliably.
- **Dual write problem** happens when application needs to change data in two different systems.
- Instead of sending the data to two separate locations, **send a single transaction** that 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





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.



What is Transactional Outbox pattern. Kindly explain it in details



The **Transactional Outbox Pattern** is a design pattern used in distributed systems, particularly in microservices, to address the **dual-write problem**. This pattern ensures that data is written to both a **database** and an **event bus** in a reliable, atomic fashion, maintaining consistency between the two systems.

Here's a detailed explanation of the pattern and how it works based on the diagrams you shared:

The Problem: Dual-Write Issue

In a distributed microservices architecture, when an application needs to modify data and publish events (e.g., for other services to react), two writes typically occur:

1. **Write data** to the service's local database (e.g., creating an order in the **Order Table**).
2. **Publish an event** (e.g., ``order_created``) to an **Event Bus** (like Apache Kafka).

If these two writes are independent (i.e., not in the same transaction), you encounter the **dual-write problem** where one write may succeed, and the other may fail, leading to **data inconsistency**.

For example:

- The order might be created in the database, but the event fails to publish, leaving the rest of the system unaware of the new order.
- Alternatively, the event might be published, but the order fails to be written to the database.

The Solution: Transactional Outbox Pattern

Overview:

The **Transactional Outbox Pattern** solves the dual-write problem by ensuring both the database write and the event publishing are part of a **single transaction**. This guarantees atomicity, meaning that either both operations succeed or both are rolled back.

Steps in the Transactional Outbox Pattern:

1. Outbox Table:

- The idea is to introduce an **"Outbox Table"** inside the service's own database.
- Instead of directly publishing events to the event bus after updating the database, you write the event to the **Outbox Table** as part of the same **database transaction** where the business data is written (e.g., the order).

2. Single Transaction:

- Both the **business operation** (e.g., creating the order) and the **event** (e.g., ``order_created``) are written to the database in a single **ACID-compliant transaction**.
- The order is stored in the **Order Table**, and the event (to be published) is stored in the **Outbox Table**.

If either the data write or the event write fails, the whole transaction is rolled back, ensuring that the system remains consistent.

3. Event Publishing via Message Relay:

- A separate **background process** or job periodically polls the **Outbox Table** for any events that need to be published to the event bus.
- This process reads the events from the **Outbox Table** and publishes them to the **Event Bus** (such as Kafka or RabbitMQ).
- Once the event is successfully published, the corresponding record is removed from the **Outbox Table**.

4. Atomicity and Reliability:

- Since the event and the business data are written in the same transaction, the system is **reliable**—even in the face of system crashes or failures, there is no risk of inconsistency between the database and the event bus.
- The background job ensures **eventual consistency** by delivering the event reliably, even if there is a temporary failure in publishing.

Key Concepts:

- **Outbox Table:** This is an additional table in the service's database where events (such as ``order_created``) are stored. The outbox ensures that events are stored reliably before they are published.
- **Single Transaction:** Both the business data (e.g., the order) and the event are written as part of a single transaction, ensuring atomicity.
- **Polling and Message Relay:** A background worker or job polls the **Outbox Table** for new events and publishes them to the **Event Bus** asynchronously, ensuring reliable delivery.

Benefits of the Transactional Outbox Pattern:

1. No Dual-Write Problem:

- Since both the database operation and the event writing are done in the same transaction, the **dual-write problem** is avoided. Either both succeed, or both are rolled back.

2. Data Consistency:

- The system remains consistent because both the event and the business data are written atomically. There is no risk of having an order in the database without a corresponding event, or vice versa.

3. Asynchronous Processing:

- The pattern decouples the actual publishing of the event from the main transaction. The background worker can publish events asynchronously without blocking the main application workflow.

4. Reliability and Fault Tolerance:

- Even if the event bus (e.g., Kafka) is temporarily down or unavailable, the events are safely stored in the **Outbox Table**, and the background worker can retry publishing them when the event bus becomes available.

5. **Guaranteed Event Delivery:**

- The use of the outbox ensures that events are reliably published to the event bus, minimizing the chances of event loss due to failure or inconsistency.

Example Flow in the Diagram:

1. The **User** triggers the **Order Service** to create a new order.
2. The **Order Service** writes both the **Order Data** to the **Order Table** and an **order_created** event to the **Outbox Table** as part of a **single transaction**.
3. A **Message Transmitter** (background job) periodically checks the **Outbox Table** for any pending events that need to be published.
4. The **Message Transmitter** reads the event from the **Outbox Table** and publishes it to the **Event Bus**.
5. Once the event is successfully published, the record is removed from the **Outbox Table**.

Summary:

The **Transactional Outbox Pattern** is a robust solution to the dual-write problem in distributed systems. By using the **Outbox Table** and a single transaction to handle both data changes and event creation, it ensures atomicity and consistency. The background job further ensures that events are reliably published to the event bus without risking data loss, providing fault tolerance and asynchronous event processing.

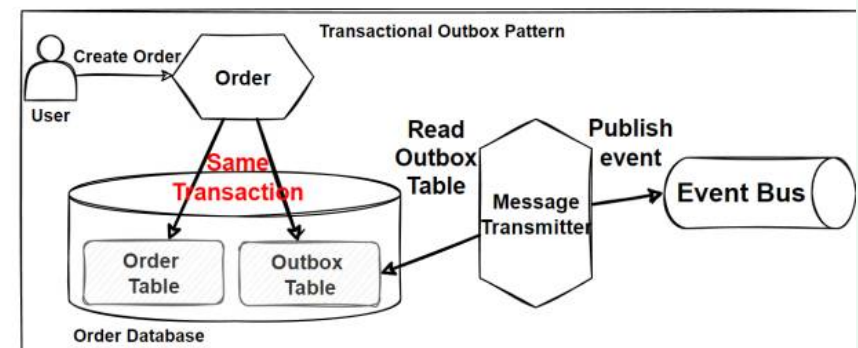
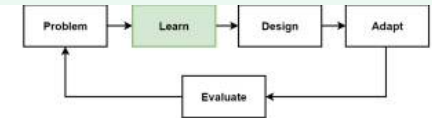
This pattern is commonly used in microservices architectures where consistent data and reliable event-based communication are critical.

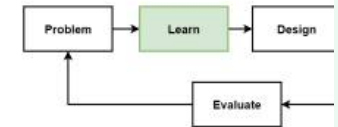
Let me know if you'd like further clarification on any part!



Transactional Outbox Pattern in Microservices

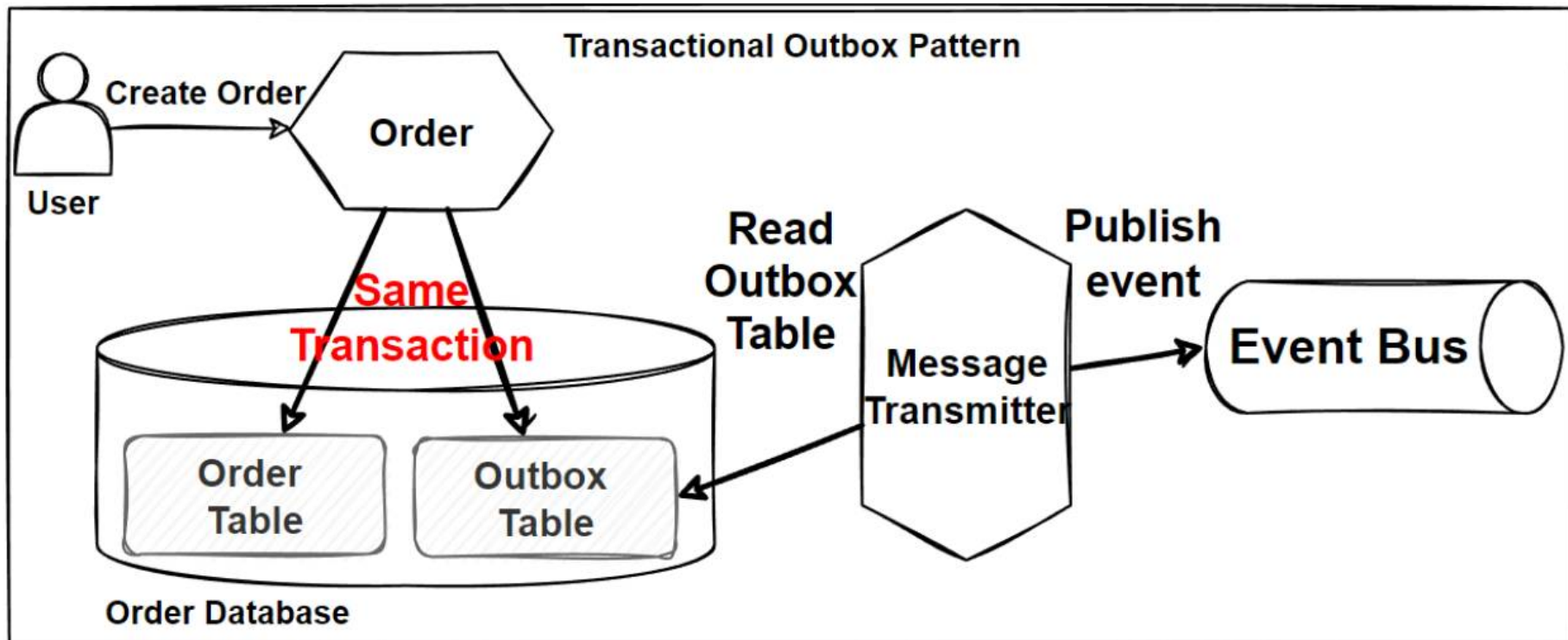
- **Microservice** provides an **outbox table** within **its database**. Outbox table will **include** all the **events**.
- There will be a **CDC (change data capture) plugin** that **reads** the **commit log** of the **outbox table** and **publish the events** to the relevant queue.
- It provides that **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.





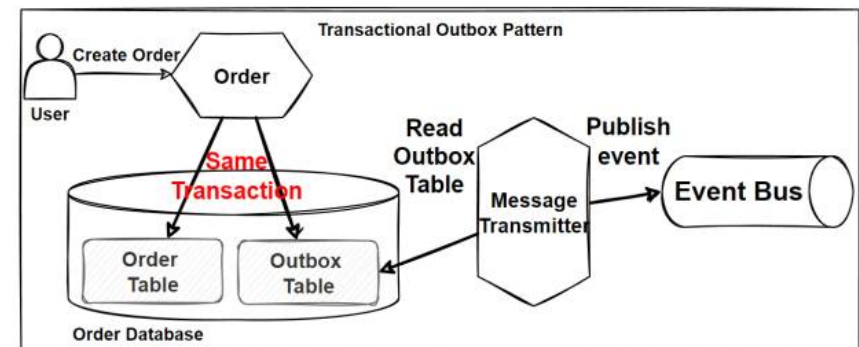
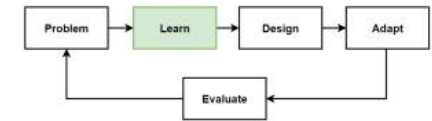
Customer Places an Order Use Case

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 it in details all the images in sequence



The images you've shared provide a detailed explanation of the **Transactional Outbox Pattern** in microservices, specifically focusing on how it handles data consistency and message reliability across distributed systems.

Image 1: Overview of the Transactional Outbox Pattern in Microservices

1. Outbox Table:

- The **outbox table** is part of the microservice's database and stores all the events that need to be published. Rather than sending events directly to the event bus, the events are first written to the outbox table.

2. CDC (Change Data Capture):

- A **CDC (Change Data Capture)** plugin reads the **commit log** of the outbox table and publishes events to the event bus. This ensures that only committed transactions (successful ones) are picked up for further processing.

3. Reliability:

- The pattern ensures that **messages are reliably delivered** between microservices, even if the transaction that triggered the message fails. Events are persisted in the outbox table before they are processed and published.

4. Atomic Transactions:

- The message/event is stored in the **local outbox table** within the same transaction that modifies the business data (e.g., creating an order). This guarantees that the event is committed along with the data change, preventing partial failures.

5. Ensures Consistency:

- The **outbox pattern** guarantees consistency by ensuring that **messages are delivered even if the system is unavailable** or experiences failures.

6. Event Delivery:

- The system stores messages in the outbox table, and a message transmitter picks them up later to publish them to the **event bus**. This decouples event publishing from the main transaction, ensuring reliability.

Image 2: Customer Places an Order Use Case

This image explains the **Customer Order Process** using the **Transactional Outbox Pattern**:

1. Create Order:

- The user triggers the **Order Service** to create a new order.

2. Single Transaction:

- The **Order Service** writes the new order data to the **Order Table** and the corresponding event (e.g., ``OrderCreatedEvent``) to the **Outbox Table** in a **single atomic transaction**. This ensures that either both the data and the event are saved or neither is (following ACID properties).

3. Polling the Outbox:

- A **message transmitter** or background worker later polls the **Outbox Table** to pick up events that need to be published.

4. Publish to Event Bus:

- The **message transmitter** publishes the events to the **Event Bus** (e.g., Kafka, RabbitMQ). This step happens asynchronously and ensures that the event is propagated to other services.

Image 3: Why Use the Outbox Pattern?

This image explains **why the outbox pattern** is essential, especially in cases where data consistency is critical:

1. Critical Data Consistency:

- The pattern is ideal for handling **critical data** where consistency is crucial. For example, in financial or order management systems, the data must be consistent and accurate to avoid financial discrepancies.

2. Atomicity:

- Ensuring that both the **database update** and the **event sending** are atomic is key to maintaining **data consistency**. If either operation fails, the other will not proceed, avoiding partial updates.

3. Order Sales and Financial Transactions:

- This pattern is especially useful in scenarios like **order sales** and **financial business transactions**, where calculations must be 100% accurate, and any missing events could lead to inconsistencies in the system.

4. Avoiding Event Loss:

- Using this pattern ensures that the system does not **lose any event messages**. This is important in cases where missing events could lead to inconsistencies in the downstream services.

5. When to Use:

- The **outbox pattern** should be used in situations where **accuracy, consistency, and reliability** of data and events are paramount, especially in distributed systems where multiple services depend on event-driven communication.

Summary of the Transactional Outbox Pattern:

The **Transactional Outbox Pattern** solves the **dual-write problem** in microservices by ensuring that both the database transaction and the event publishing are atomic and consistent. The steps are:

1. The business data (e.g., an order) and the event (e.g., ``OrderCreatedEvent``) are written to the same database in a single atomic transaction.
2. The event is written to an **outbox table** in the same database transaction.
3. A **message transmitter** or a background worker periodically checks the **outbox table** and publishes the events to the **event bus**.
4. This pattern ensures **data consistency**, **reliable event delivery**, and prevents **data loss** in the event of system failures.

The **Outbox Pattern** is especially useful in systems where data consistency is critical, such as financial transactions or order processing systems.

Let me know if you'd like more details or have any specific questions about this pattern!



Problem: Listen and Polling Outbox Table

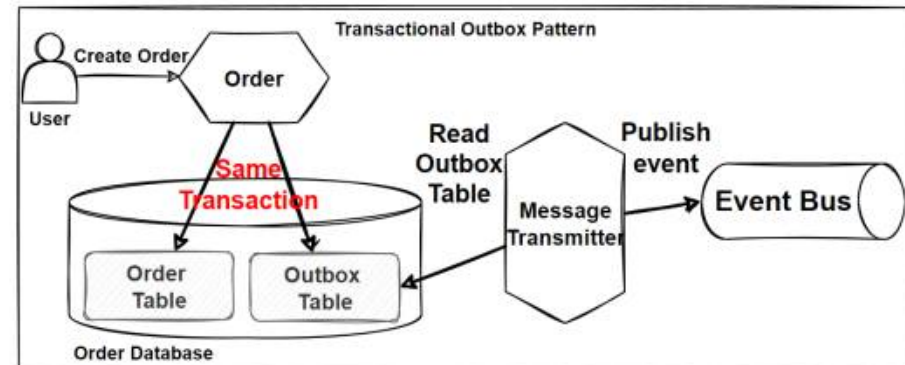
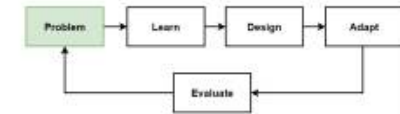
- **Creating new order record** into order table and **creating new order_created event** into outbox table **are in the same transactions**.
- Another service **listen** and **polls** the **outbox table records** and publish events into separate processes.
- The current architecture **required additional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



Kindly explain the above image in details



The image you've shared explains the **challenges** associated with the **Transactional Outbox Pattern**—specifically focusing on the problems caused by **listening and polling the outbox table**. Here's a detailed explanation of the key points and how these issues can be mitigated.

Problem: Listen and Polling the Outbox Table

The **Transactional Outbox Pattern** ensures that business data (e.g., an order) and its associated events (e.g., ``OrderCreatedEvent``) are written to the database within a **single transaction**. After the transaction is committed, another service (or process) is responsible for **polling the outbox table** to publish these events to the **event bus**.

Steps in the Process:

1. Order Creation:

- When a user creates an order, the **Order Service** stores the order in the **Order Table** and an ``OrderCreatedEvent`` in the **Outbox Table** in the same transaction. This ensures that both operations are atomic.

2. Polling Service:

- Another microservice (or component) **polls the outbox table** periodically to check for new events that need to be published. Once it finds new events, it **publishes them** to the event bus (e.g., Kafka, RabbitMQ).

3. Architecture Requirement:

- The architecture requires additional microservices or processes to **listen** and **poll** the outbox table records, which introduces complexity to the system.

Problems with Polling and Listening:

1. Performance and Error-Prone Operations:

- Continuously polling the outbox table to check for new events can reduce the overall **performance** of the system. It also increases the likelihood of **errors** because it introduces more components and dependencies that can fail.

2. Wasted Time and Resources:

- Polling typically occurs at regular intervals, meaning there's often a delay between when an event is written to the outbox table and when it's actually published. This can lead to **resource waste**, as the system constantly checks for new events even if none are present, wasting CPU and database resources.

3. Unnecessary Pull Requests:

- Since the system continuously polls the outbox table even when no new events exist, it generates **unnecessary queries** (pull requests) to the database, leading to performance degradation, especially in high-traffic environments.

4. Latency Overhead:

- Polling introduces **latency** because the event is not published immediately after it's written to the outbox table. The event will only be published after the polling service detects it, which may take time depending on the polling interval.
- This delay may lead to a **slower overall event propagation**, which can impact real-time or time-sensitive applications.

5. Potential for Additional Dual-Write Problems:

- The polling architecture can introduce its own set of dual-write problems. If the polling service fails after pulling an event from the outbox table but before it is published to the event bus, the system can still end up in an inconsistent state, with the event missing from the event bus.

How to Solve These Problems When Using the Outbox Pattern?

The image suggests using **CDC (Change Data Capture)** to improve the performance and reliability of the outbox pattern:

1. CDC (Change Data Capture) with the Outbox Pattern:

- Instead of using a polling mechanism to detect new events, you can use **CDC** technology, which tracks changes made to the outbox table by monitoring the **database commit log**. This ensures that only committed changes are published.
- With CDC, the system **reacts to changes** in the outbox table (as they happen) rather than continuously polling, significantly improving performance and eliminating unnecessary resource consumption.

2. More Efficient and Real-Time Event Publishing:

- With **CDC**, the events in the outbox table can be published to the event bus almost immediately after they are written. This eliminates the latency and overhead introduced by polling and allows for near real-time event processing.

3. Reliability:

- CDC tools are more reliable than manual polling processes, as they are integrated with the database transaction logs and ensure that **only committed data** is captured and forwarded to the event bus. This reduces the risk of missing or duplicate events.

Summary of Key Concepts:

1. Transactional Outbox Pattern:

- Ensures that business operations (like creating an order) and event publishing are atomic by storing events in an **outbox table** as part of the same transaction.

2. Polling and Listening Challenges:

- **Performance overhead**, wasted resources, **latency**, and increased complexity due to unnecessary polling of the outbox table.
- Polling can introduce **dual-write problems** if the event isn't published after being pulled from the outbox table.

3. CDC as a Solution:

- By using **CDC (Change Data Capture)**, the system can listen for changes to the outbox table and publish events to the event bus in **real-time** without the need for polling.
- This improves performance, reduces latency, and ensures more reliable event delivery.

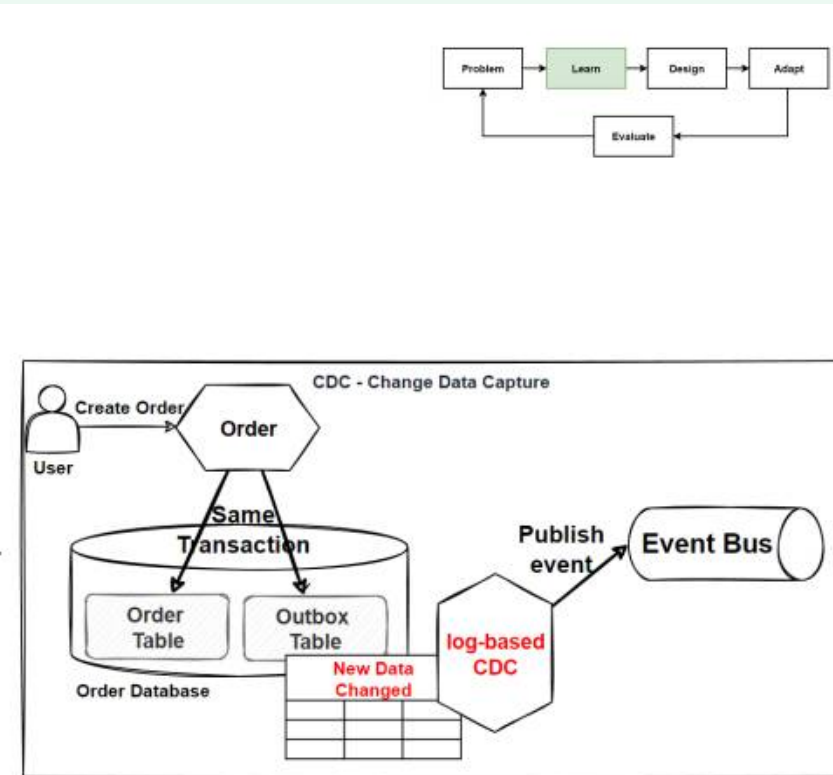
The **Transactional Outbox Pattern** is a powerful solution to the dual-write problem, but it can introduce its own challenges, especially around polling and event publishing. Implementing **CDC** can help eliminate many of these issues, making the system more efficient and reliable.

Let me know if you'd like further clarification on any part!



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

Kindly explain the above diagram



The diagram you shared explains the **Change Data Capture (CDC)** mechanism and how it integrates with the **Transactional Outbox Pattern** to improve performance and reliability by tracking and processing changes in the database in **real-time**.

Here's a detailed explanation of the key concepts in the diagram:

What is Change Data Capture (CDC)?

Change Data Capture (CDC) is a technology that captures changes—**insert**, **update**, or **delete**—that happen on a database. Instead of relying on batch processing or continuous polling (which can be resource-intensive and inefficient), CDC listens for changes in the **transaction log** of the database and then extracts and propagates those changes to the target system, ensuring **near real-time** data synchronization.

How CDC Works:

- **Log-based CDC:** CDC typically operates by monitoring the **transaction logs** of the database for any changes, like newly inserted records, updated records, or deleted records. These transaction logs are part of the database's internal mechanism to keep track of changes.
- **Near Real-Time Updates:** As soon as a change is detected in the transaction log, CDC picks up the change and forwards it to the relevant system (like an event bus or another service) in **near real-time**, ensuring that the changes are propagated without the delay typically seen in batch-based processing systems.

How CDC is Integrated with the Transactional Outbox Pattern:

1. Creating the Order (Same Transaction):

- The user creates an order using the **Order Service**. The new order is written into the **Order Table** as part of a **single transaction**.
- In the same transaction, an event (e.g., ``OrderCreatedEvent``) is written into the **Outbox Table**. This ensures atomicity—both the order and the event are committed together.

2. Log-Based CDC:

- The **CDC** mechanism monitors the **transaction log** of the **Outbox Table** for changes. When new data (like an event) is written to the outbox, CDC detects this and triggers the next step.

3. Publishing the Event to the Event Bus:

- As soon as CDC detects a new change in the **Outbox Table**, it **publishes the event** to the **Event Bus** (e.g., Kafka, RabbitMQ). This allows other services to consume the event and take appropriate actions (like initiating payment, updating inventory, etc.).

Key Benefits of Using CDC:

1. Near Real-Time Event Propagation:

- By monitoring the database's transaction logs, **CDC** ensures that changes (like newly created orders) are detected and propagated to the target system (event bus) almost immediately, eliminating the need for polling or scheduled jobs.

2. Stay Up-to-Date:

- CDC allows other systems (e.g., microservices, data analytics platforms) to stay up-to-date with the source system (the order database) in **real-time** by listening to changes and processing them as they happen.

3. Data Synchronization Across Systems:

- CDC is commonly used in microservices architectures to synchronize data between different services or databases, ensuring that all parts of the system have access to the latest data.

4. Reduces Overhead:

- Compared to manual polling or batch processes, **CDC** is more efficient. It eliminates the need to continuously query the database for new changes, which can significantly reduce the load on the system and improve performance.

5. Reliability:

- Since CDC is based on the database's **transaction log**, it only captures committed changes. This ensures that only valid data is propagated to downstream systems, reducing the risk of errors or inconsistencies.

Key Points from the Diagram:

1. Order Creation:

- The user places an order via the **Order Service**. This transaction writes the new order into the **Order Table** and the corresponding event into the **Outbox Table**.

2. Log-Based CDC:

- **CDC** monitors the **Outbox Table** for new events (via the database's transaction log). When it detects a change, it pulls the new event.

3. Publish Event to Event Bus:

- CDC pushes the event (e.g., ``OrderCreatedEvent``) to the **Event Bus** in near real-time, enabling other microservices to react to the event (e.g., process payment, notify inventory services).

Use Cases for CDC:

- **Data Replication:** Synchronizing data between systems or databases in near real-time.
- **Event-Driven Architectures:** Publishing events when changes occur in the database (like order creation or updates) to notify other services in the system.
- **Real-Time Analytics:** Capturing data changes for real-time data analytics, keeping analytics systems up-to-date with minimal delay.

Conclusion:

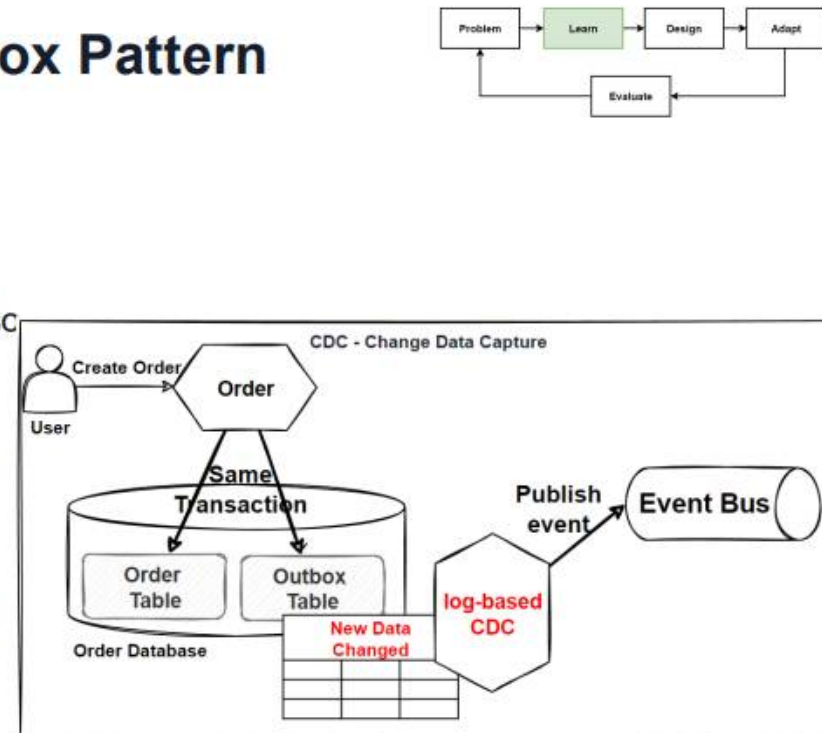
In this diagram, **CDC (Change Data Capture)** is used to monitor changes in the **Outbox Table** and push events to the **Event Bus** as soon as they occur, ensuring **real-time** propagation of changes with minimal latency. This approach solves the problem of resource-intensive polling while ensuring that data remains consistent and up-to-date across systems in distributed architectures like microservices.

Let me know if you have further questions or need additional details!



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 image in details



The image explains how **Change Data Capture (CDC)** works in conjunction with the **Outbox Pattern** to ensure reliable data propagation and synchronization in a **microservices architecture**. Below is a detailed explanation of the key concepts and components of the image.

CDC with the Outbox Pattern:

1. Outbox Pattern Ensuring Data Propagation:

- The **Outbox Pattern** ensures that any **data changes** made by a microservice are **eventually propagated** to other microservices or components.
- Whenever a microservice updates its **business data** (e.g., an order), it also writes an event record (e.g., ``OrderCreatedEvent``) to an **Outbox Table** in the same transaction. This guarantees atomicity—both the data update and the event are committed together.

2. How Data Changes Are Written to the Outbox Table:

- Anytime a microservice **updates data** in its database (for example, writing a new order in the **Order Table**), the microservice also writes a corresponding **event** (like ``OrderCreatedEvent``) to the **Outbox Table**. The outbox table contains details about the change (e.g., the type of event, the data associated with the event, etc.).

3. CDC Monitoring the Outbox Table:

- **CDC** (Change Data Capture) monitors the **Outbox Table** for new records by continuously watching the **transaction logs** of the database.
- When a new event is written to the **Outbox Table**, CDC detects the new entry and extracts it.
- CDC then **propagates these data changes** to other systems (for example, by publishing them to an **Event Bus** such as Kafka or RabbitMQ) to notify other microservices about the change.

4. Decoupling Data Updates and Event Propagation:

- By combining **CDC** with the **Outbox Pattern**, a microservice can **decouple its data updates** from the process of publishing events to notify other services.
- This separation ensures that each microservice focuses on its **own data** (such as order creation or inventory updates), and **CDC** handles the job of propagating the event to other systems.
- This decoupling helps maintain **eventual consistency** in a distributed system, where microservices can operate independently but still share information reliably.

5. Scalability and Maintainability:

- By using CDC with the outbox pattern, it becomes **easier to scale and maintain** a microservices architecture. Each microservice is only concerned with updating its own local data, and CDC takes care of distributing the changes to other services.

- This architecture enables microservices to function independently, making it easier to scale each component without needing direct synchronization between services.

6. Focus on Data Updates, CDC Handles Synchronization:

- Each microservice can focus on **its own database operations** (e.g., processing an order or updating stock levels).
- **CDC** monitors changes in the **Outbox Table** and synchronizes the changes by publishing events to other systems or microservices. This removes the need for the microservice to manage event propagation directly, simplifying the design.

Diagram Explanation:

- The diagram illustrates a typical CDC with the **Outbox Pattern** setup in the context of an order service:
 1. A user triggers the **Order Service** to create a new order.
 2. The order is saved in the **Order Table**, and an associated event (``OrderCreatedEvent``) is written to the **Outbox Table** in the same transaction.
 3. **Log-based CDC** monitors the database transaction logs and detects the new entry in the **Outbox Table**.
 4. CDC extracts this new event and publishes it to the **Event Bus**, making the event available to other services (e.g., Payment Service, Inventory Service) that depend on the new order.

Benefits of CDC with Outbox Pattern:

1. **Atomicity:** The business data (e.g., order) and the event (e.g., ``OrderCreatedEvent``) are written in the same transaction, ensuring that both are committed together.
2. **Decoupling:** Microservices focus only on their own data updates, while CDC ensures that changes are propagated to other systems.
3. **Scalability:** This architecture supports scalable and maintainable microservices by decoupling the process of updating data from the process of synchronizing events.
4. **Eventual Consistency:** Even in distributed systems where real-time consistency may not be possible, CDC ensures that all microservices eventually receive the necessary updates through events.

Key Takeaways:

- The **Outbox Pattern** combined with **CDC** allows reliable and efficient propagation of data changes across microservices.
- **CDC** continuously monitors changes in the **Outbox Table** and propagates them to other systems by publishing the relevant events.
- This approach decouples microservices, enabling them to scale independently and focus on their business logic without worrying about inter-service communication.

Let me know if you'd like further details or examples related to this pattern!

ChatGPT can make mistakes. Check important info.