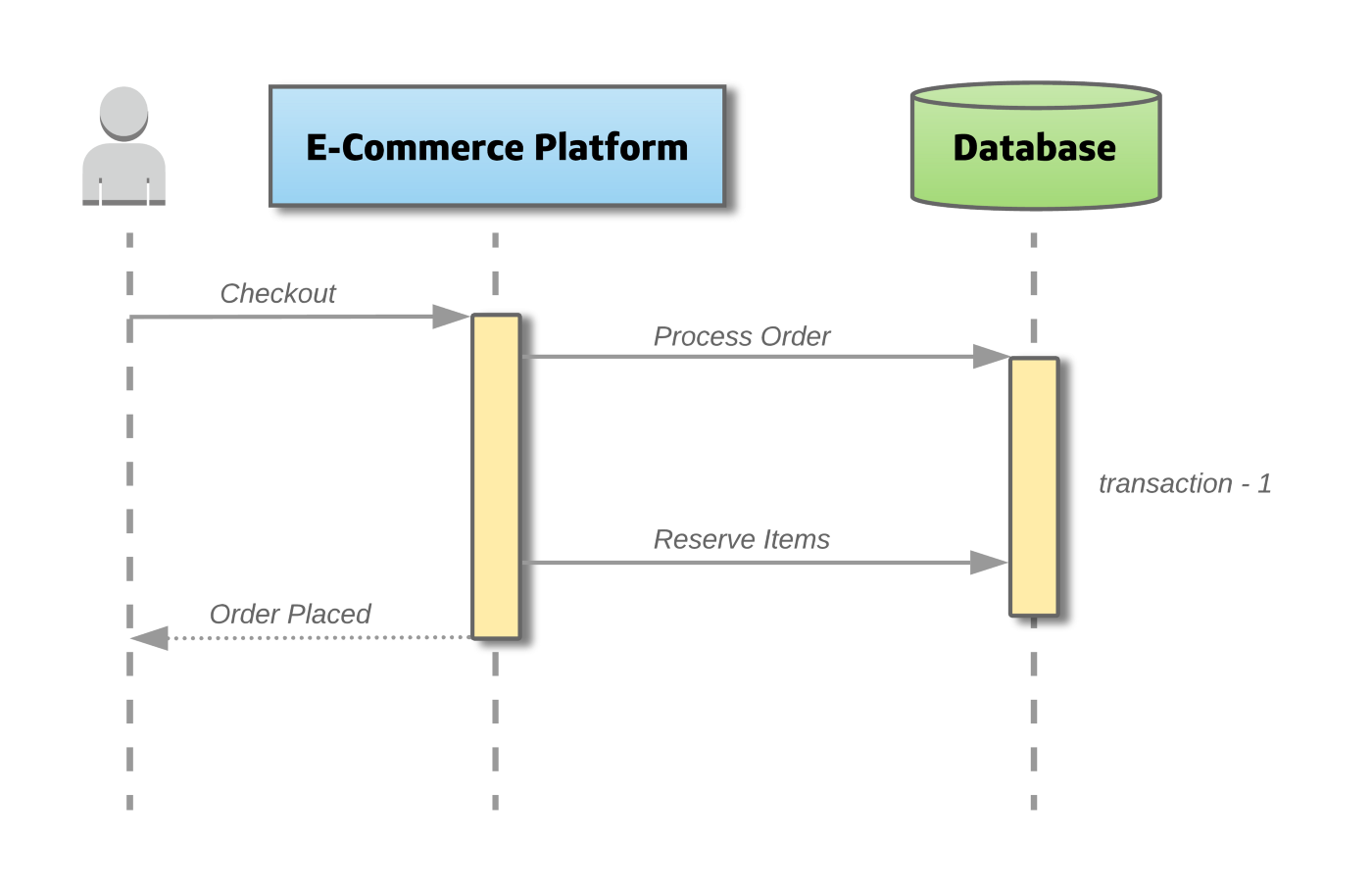
**Distributed Transaction Management for Microservices**

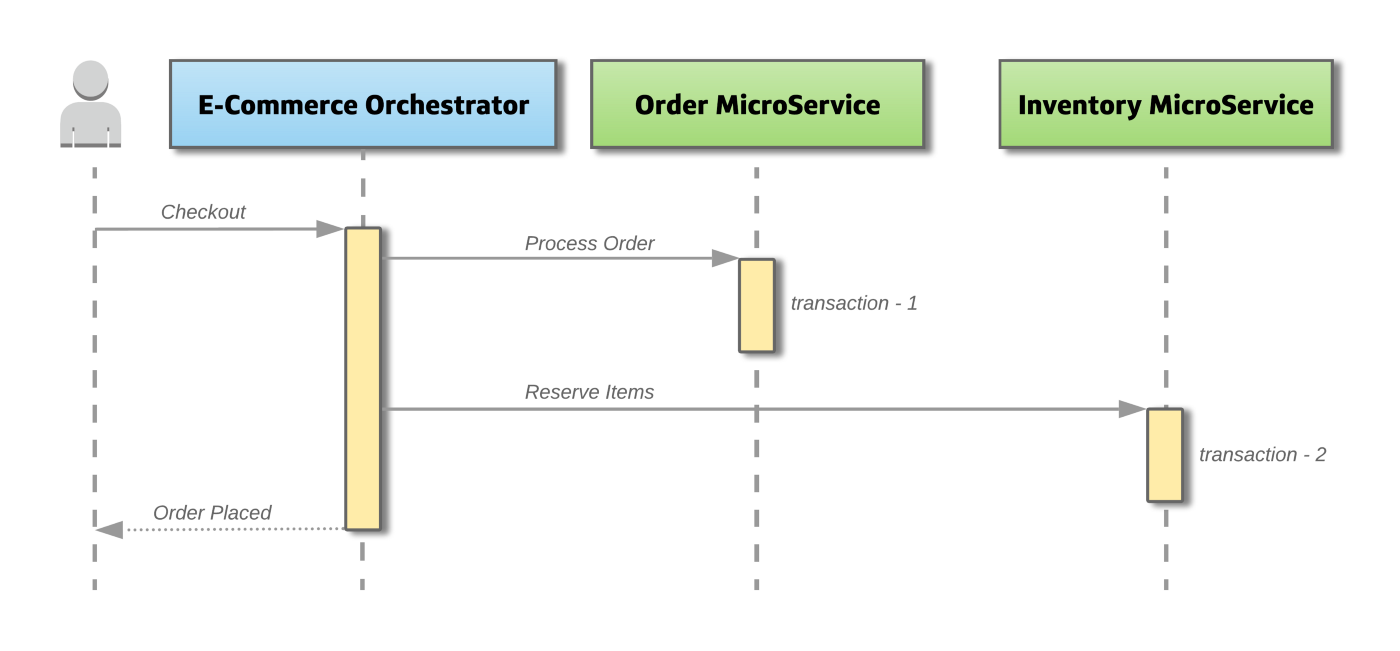
* **Transactions that span over multiple physical systems over the network**, are simply **termed Distributed Transactions**. In the world of microservices, a transaction is now distributed to multiple services that are called in a sequence or parallel to complete the entire transaction.

Here is a **monolithic e-commerce system** using transactions:



In the system above, if a **user sends a Checkout request** to the platform, the **platform will create a local database transaction that works over multiple database tables**, **to Process the order and Reserve items from the inventory**. If any step fails, the transaction can roll back, both the order and items reserved. This is known as the **ACID principle (Atomicity, Consistency, Isolation, Durability),** which is guaranteed by the database system.

Here is the **e-commerce system decomposed as microservices**:



When we decompose this system, we created the microservices **OrderMicroservice** and **InventoryMicroservice**, which have **separate databases**. **When a Checkout request comes from the user**, **both these microservices will be invoked to apply changes into their own database**. Because the transaction is now across multiple databases via multiple systems, it is now considered a distributed transaction.

**Problems with distributed transactions**

**How do we keep the transaction atomic?**

* **Atomicity means that in a transaction either all steps are completed or no step is completed**. In the example above, if the ‘reserve items’ in the InventoryMicroservice method fails, how do we roll back the ‘process order’ changes that were applied by the OrderMicroservice?

**How do we handle concurrent requests?**

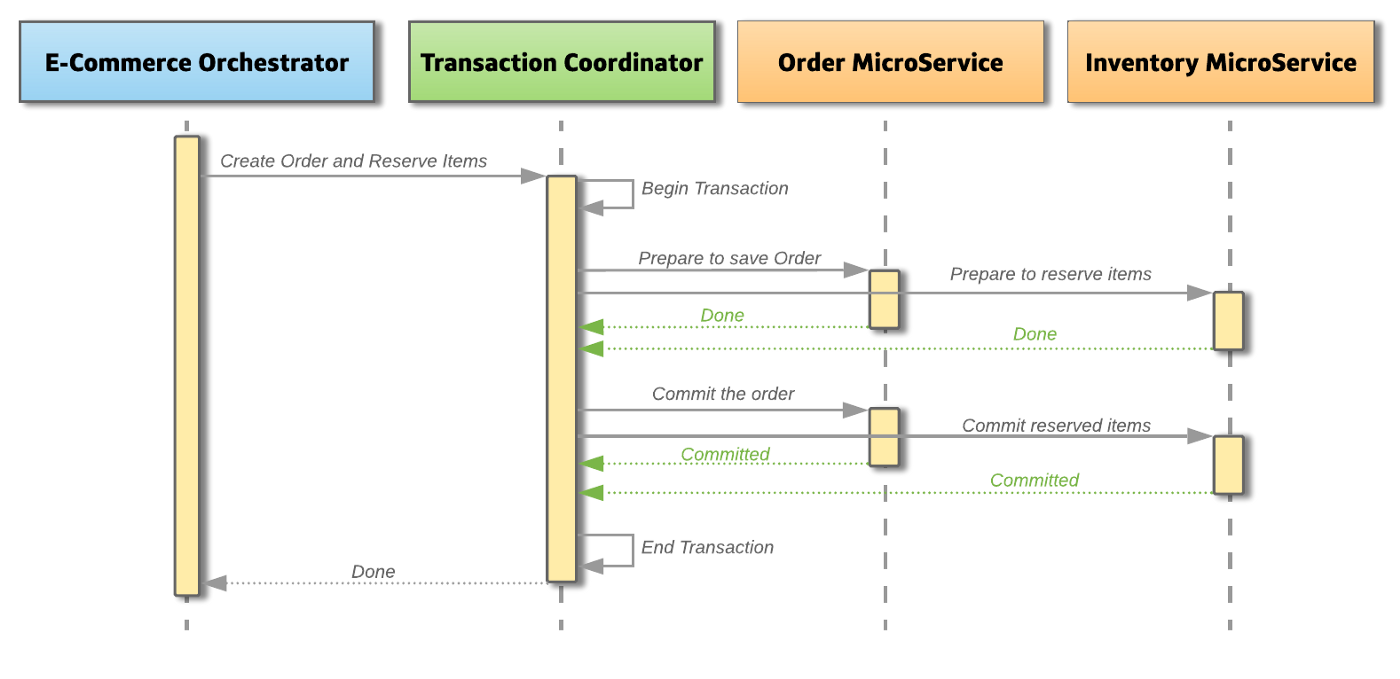
* **If an object from any one of the microservice is being persisted to the database and at the same time, another request reads the same object**. Should the service return the old data or new? In the example above, once OrderMicroservice is complete and the InventoryMicroservice is now performing its update, should the request for the number of orders placed by the customer include the current order?

**Possible Solutions**

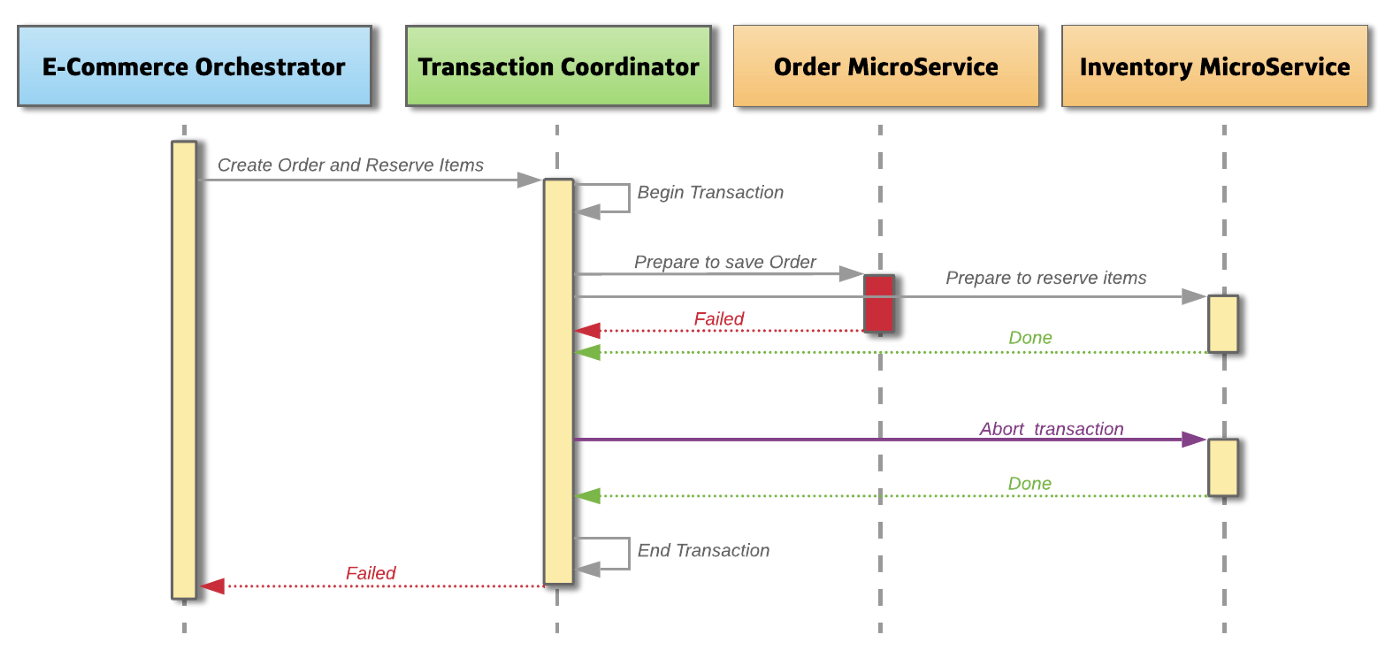
**1. Two-Phase Commit**

As the name suggests, this way of **handling transactions has two stages**, **a prepare phase**, and **a commit phase**. One important participant is the **Transaction Coordinator** which **maintains the lifecycle of the transaction**.

In the **preparation phase**, **all microservices involved prepare for the commit** and **notify the coordinator that they are ready to complete the transaction**. Then in the **commit phase**, **either a commit or a rollback command is issued** by the transaction coordinator to all microservices.



In the example above, when a **user sends a checkout request** the **TransactionCoordinator** will first **begin a global transaction with all the context information**. First, it will send out a **prepare command to the OrderMicroservice**, **to create an order**. Then it will send out a **prepare command to the InventoryMicroservice**, **to reserve the items**. When both the **services are OK to perform the change**, **they lock down the objects from further changes and notify the TransactionCoordinator**. Once the TransactionCoordinator has confirmed that all microservices are ready to apply their changes, it will then ask them to **persist their changes by requesting a commit with the transaction**. At this point, all objects will be unlocked.



In a failure scenario, if at any point **a single microservice fails to prepare**, the **TransactionCoordinator** will **abort the transaction and begin the rollback process**. In the diagram, the **OrderMicroservice failed to create an order** for some reason, but the **InventoryMicroservice** has replied that **it is prepared to create the order**. The **TransactionCoordinator** will **request an abort on the InventoryMicroservice** and the **service will then roll back any changes made** and unlock the database objects.

**Advantages**

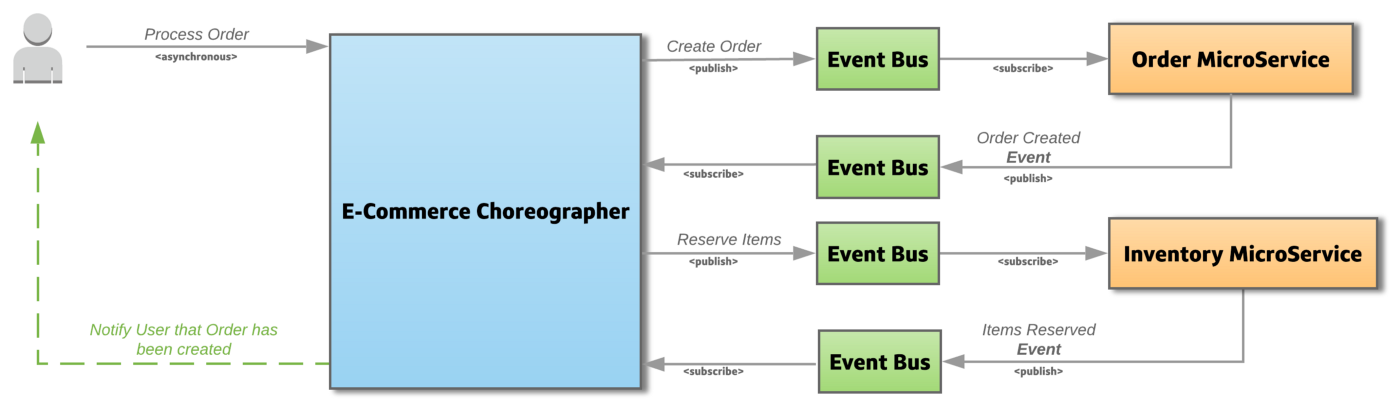
* The **approach guarantees that the transaction is atomic**. The transaction will end with either all microservices being successful or all microservices have nothing changed.
* Secondly, it **allows read-write isolation**, the **changes on objects are not visible until the transaction coordinator commits the changes**.
* The approach is a **synchronous call**, where the **client would be notified of success or failure**.

**Dis-Advantages**

* Everything isn’t perfect, **two-phase commits are quite slow compared to the time for operation of a single microservice**. They are **highly dependent on the transaction coordinator**, which can really slow down the system during high loads.
* The other main drawback is the **locking of database rows**. The **lock could become a performance bottleneck** and it is **possible to have a Deadlock**, where **two transactions mutually lock each other**.

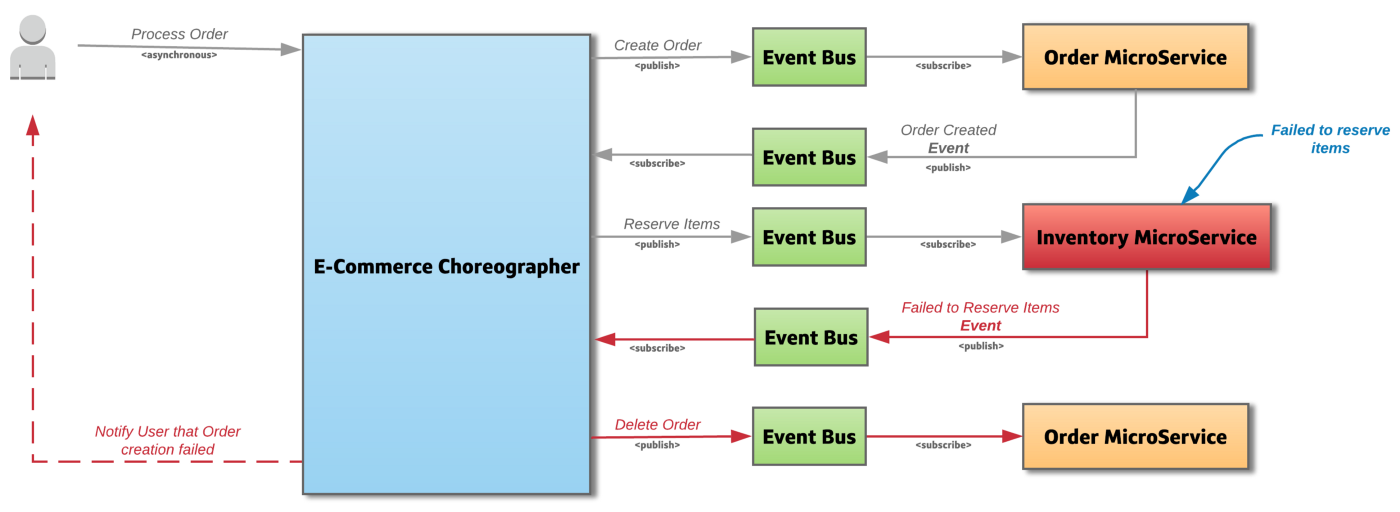
**2. Eventual Consistency and Compensation / SAGA**

In this approach, the **distributed transaction is fulfilled by asynchronous local transactions on related microservices**. The **microservices communicate with each other through an event bus**.



In the example above, the client requests the system to Process The Order. On this request the **Choreographer emits an event Create Order**, **marking the start of the transaction**. The **OrderMicroservice** **listens to this event** and **creates an order**, **if it was successful it emits an Order Created event**. The **Choreographer listens for this event** and **proceeds to reserve the items**, by **emitting the Reserve Items event**. The **InventoryMicroservice** **listens for this event** and then **reserve’s the items**, **if it was successful it emits an Items Reserved event**. Which in this example means the end of the transaction.

All the **event-based communication** between microservices **happens via the Event Bus** and is **Choreographed by another system to address the complexity issue**.



If for any reason the **InventoryMicroservice** **failed to reserve the items**, it **emits a Failed to Reserve Items event**. The **Choreographer listens for this event** and **starts a Compensating Transaction**, by **emitting a Delete Order event**. The **OrderMicroservice listens to this event** and **deletes the order that was created**.

**Advantages**

* One **big advantage** of this approach is that **each microservice focuses only on its own atomic transaction**.
* **Microservices are not blocked if another service is taking a long time**. This also means that there is **no database lock required**.
* Using this approach **makes the system highly scalable under heavy load**, due to its **asynchronous event-based solution**.

**Dis-Advantages**

* The **main disadvantage** is the approach **does not have read isolation**. This means, in the above example the **client could see the order was created, but in the next second, the order is removed due to a compensating transaction**.
* Also, **when the number of microservices increases** it **becomes harder to debug and maintain**.

When there is **a need to update data in two places as a result of one event**, the **Eventual Consistency / SAGA approach is a preferable way of handling distributed transactions** as compared to the two-phase commit. The main reason being the **two-phase commit does not scale in a distributed environment**.

**Different ways to implement the SAGA pattern**

**1. Choreography-based SAGA**

**No central coordinator exists in this case**. **Each service produces an event after completion of its task and each service listens to events to take an action**. This pattern **requires a mature event-driven architecture**.

* **Event Sourcing** is an **approach to store the state of event changes** **by using an Event Store**. **Event Store is a message broker acting as an event database**. States are reconstructed by replaying the events from the Event Store.
* **Choreography-based SAGA pattern** can **work well for a small number of steps in a transaction** (e.g. 2 to 4 steps). When the number of steps in a transaction is increasing, it is **difficult to track which services listen to which events**.

The **Saga choreography pattern is ideal when you start your microservices journey** **and understand that it is necessary to introduce microservices in due course**.

**2. Orchestration-based SAGA**

**A coordinator service** (**Saga Execution Orchestrator**) is **responsible for sequencing transactions according to business logic**. **Orchestrator** **decides which operation should be performed**. If an **operation fails**, **Orchestrator undoes the previous steps**. It is called a **compensation operation**. **Compensations are the actions to apply when a failure happens, to keep the system in a consistent state**.

* **Undoing changes may not be possible** already when **data has been changed by a different transaction**.
* **Compensations must be idempotent** because they might be **called more than once within the retry mechanism**.

The **Saga orchestration pattern** is **ideal for scenarios where you already built your microservices and now want to create a process flow using these microservices**. There **must be a compensating service available** and **adding another piece of code is not necessary**. This is unlike the **Saga choreography pattern**, where you **must align the code with a framework** that you chose to implement the Saga.