Transactions and Queries in Microservices

# Intro

* Many monolithic applications rely on **transactions to guarantee consistency and isolation** when changing application state. Obtaining these properties is straightforward: an application typically interacts with a single database, with strong consistency guarantees, using frameworks that provide support for starting, committing, or rolling back transactional operations. Each logical transaction might involve several distinct entities; for example, placing an order will update transactions, reserve stock positions, and charge fees.
* You’re not so lucky in a microservice application. As you learned earlier, each independent service is responsible for a specific capability. Data ownership is decentralized, **ensuring a single owner for each “source of truth.”** This level of decoupling helps you gain autonomy, but you sacrifice some of **the safety you were previously afforded, making consistency an application-level problem**. Decentralized data ownership also makes retrieving data more complex. Queries that previously used database-level joins now require calls to multiple services. This is acceptable for some use cases but painful for large data sets.
* **Availability also impacts your application design. Interactions between services might fail, causing business processes to halt, leaving your system in an inconsistent state.**

In this document, we’ll learn how to use ***sagas* to coordinate complex transactions** across multiple services and explore best practices for efficiently querying data. Along the way, we’ll examine different types of **event-based architectures**, **such as** **event sourcing**, and their applicability to microservice applications.

# *Consistent Transactions in Distributed Applications*

Imagine you want to place an order in some stock selling app. From your perspective as a customer, this operation appears to be atomic: charging a fee, reserving stock, and creating an order happen at the same time, and you can’t sell stock that you don’t have or sell a stock you do have more than once.

In many monolithic applications,1 those requirements are easy to meet: you can wrap your database operations in an ACID transaction and rest easy in the knowledge that errors will cause an invalid state to be rolled back.

By contrast, in your microservice application, each of the actions is performed by a distinct service responsible for a subset of application state. Decentralized data ownership helps ensure services are independent and loosely coupled, but it forces you to build application-level mechanisms to maintain overall data consistency.

Let’s say an orders service is responsible for coordinating the process of selling a stock. It calls account transactions to reserve stock and then the fees service to charge the customer. But that transaction fails.

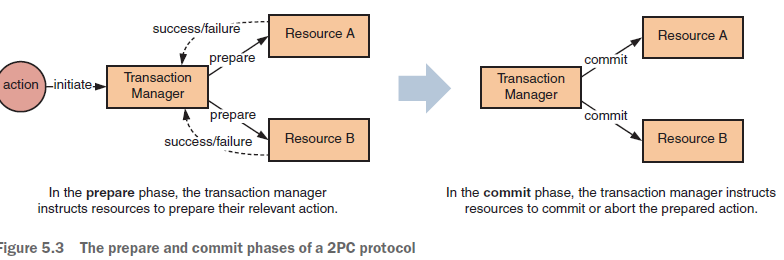
At this stage, your system is in an inconsistent state: stock is reserved, an order is created, but you haven’t charged the customer. You can’t leave it like this — so the implementation of orders needs to initiate corrective action, instructing the account transactions service to compensate and remove the stock reservation. This might look simple, but it becomes increasingly complex when many services are involved, transactions are long-running, or an action triggers further interleaved downstream transactions.

## *Why Can’t You Use Distributed Transactions?*

Faced with this problem, your first impulse might be to design a system that achieves

transactional guarantees across multiple services. A common approach is to use the **two-phase commit (2PC)** protocol. In this approach, you use a **transaction manager** to split

operations across multiple resources into two phases: **prepare** and **commit.**



This sounds great — like what you’re used to. Unfortunately, this approach is flawed. First, 2PC implies synchronicity of communication between the transaction manager and resources. If a resource is unavailable, the transaction can’t be committed and must roll back. **This in turn increases the volume of retries** and decreases the availability of the overall system. To support asynchronous service interactions, you would need to support 2PC with services *and* the messaging layer between them, limiting your technical choices.

* In a microservice application, availability is the product of all microservices involved in processing a given action. Because no service is 100% reliable, involving more services lessens overall reliability, increasing the probability of failure. We’ll explore this in detail in the next chapter.

Handing off significant orchestration responsibility to a transaction manager also violates one of the core principles of microservices: service autonomy. At worst, you’d end up with dumb services representing CRUD operations against data, with transaction managers wholly encapsulating the interesting behavior of your system. Finally, a distributed transaction places a lock on the resources under transaction to ensure isolation. This makes it inappropriate for long-running operations, as it increases the risk of contention and deadlock. What should you do instead?

# *Event-Based Communication*

**Asynchronous events aid in decoupling services** from each other **and increase overall system availability**, but they also encourage service authors to think in terms of ***eventual consistency***. In an eventually consistent system, you design complex outcomes to result from several independent local transactions over time, which leads you to explicitly design **underlying resources to represent tentative states**. From the perspective of Eric Brewer’s CAP theorem, this design approach **prioritizes the availability of underlying data**.

In a **synchronous** approach, the **orders service orchestrates the behavior of other services**, invoking a sequence of steps until the order is placed to the market. If any steps fail, the orders service is responsible for initiating rollback action with other services, such as reversing the charge. In this approach, the orders service takes on substantial responsibility:

* **It knows which services it needs to call, as well as their order.**
* **It needs to know what to do in case any downstream service produces an error or can’t proceed due to business rules.**

Although this type of interaction is easy to reason through — as the call graph is logical

and sequential — **this level of responsibility tightly couples the orders service to**

**other services**, **limiting its independence and increasing the difficulty of making future changes.**

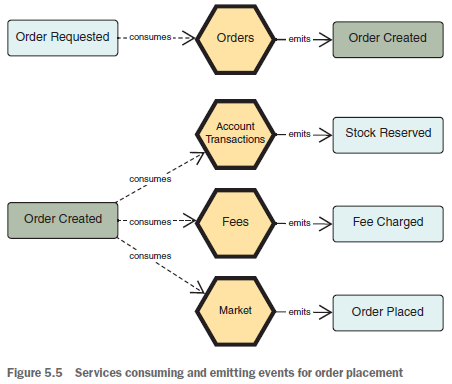
## *Events and Choreography*

You can redesign this scenario to use events. **Each service subscribes to events that interest it** to know when it must perform some work:

1. When the user issues a sell request via the UI, the application publishes an OrderRequested event.
2. The orders service picks up this event, processes it, and publishes back to the event queue an OrderCreated event.
3. Both the transaction and fees services then pick up this event. Each one of them performs its work and publishes back events to notify about the completion.
4. The market service in turn is waiting for a pair of events notifying it of the charging of fees and the reservation of stocks. When both arrive, it knows it can place the order against the stock exchange. Once that’s finished, the market service publishes a final event back to the queue.

Events allow you to take **an optimistic approach to availability**. For example, if the fees service were down, **the orders service would still be able to create orders**. When the fees service came back online, it could continue processing a backlog of events. **You can extend this to rollback: if the fees service fails to charge because of insufficient funds, it could emit a ChargeFailed event, which other services would then consume to cancel order placement.**

This interaction is ***choreographed***: each service reacts to events, acting independently without knowledge of the overall outcome of the process. These services are like dancers: they know the steps and what to do in each section of a musical piece, and they react accordingly without you needing to explicitly invoke or command them. In turn, this design decouples services from each other, increasing their independence and making it easier to deploy changes independently.



# Sagas

The choreographed approach is a basic example of the *saga* pattern**. A saga is a coordinated series of local transactions; a previous step triggers each step in the saga.**

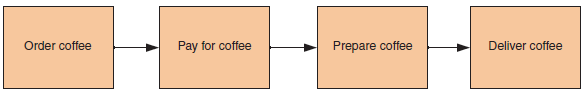
As with distributed transactions, locking in long-lived transactions reduces availability — a saga solves this as a sequence of interleaved, individual transactions.

As **each local transaction is atomic** — **but not the saga as a whole** — a developer must write their code to ensure that the system ultimately reaches a consistent state, even if individual transactions fail.

**In a distributed transaction**, you **manage uncertainty using locks on data**; without transactions, you manage uncertainty through semantically appropriate workflows that confirm, cancel, or compensate for actions as they occur.

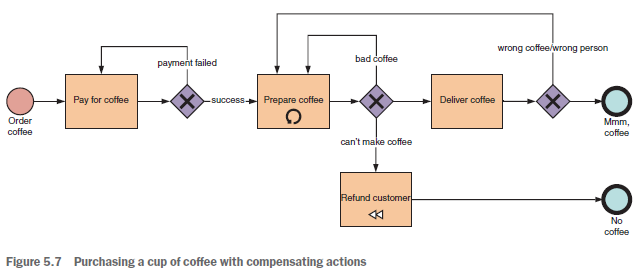
let’s look at a simple real-world saga:

purchasing a cup of coffee. Typically, this might involve four steps: **ordering**, **payment**, **preparation**, and **delivery**. In the normal outcome, the customer pays for and receives the coffee they ordered.



This can go wrong! The coffee shop machine might break; the barista might make a cappuccino, but I wanted a flat white; they might give my coffee to the wrong customer; and so on. If one of these events occurs, the barista will naturally compensate: they might make my coffee again or refund my payment.

In most cases, I’ll eventually get my coffee. **You use compensating actions in sagas to undo previous operations and return your system to a more consistent state**. The system isn’t guaranteed to be returned to the *original* state; the appropriate actions depend on business semantics. This design approach makes writing business logic more complex — because you need to consider a wide range of potential scenarios — but is a great tool for building reliable interactions between distributed services.



## Choreographed Sagas

Let’s return to the earlier example — sell orders.

The actions in this **saga are choreographed**:

**each action, TX, is performed in response to another**, but without an overall conductor or orchestrator. You can break this task into five subtasks:

¡ T1 — Create the order.

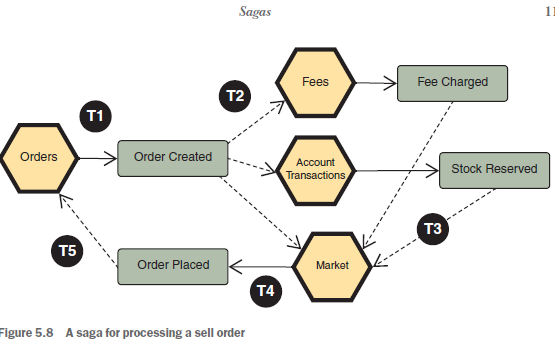
¡ T2 — Reserve the stock position, which the account transaction service implements.

¡ T3 — Calculate and charge the fee, which the fees service implements.

¡ T4 — Place the purchase order to the market, which the market service implements.

¡ T5 — Update the status of the order to be placed.

The Figure below, illustrates **the optimistic — most likely — path** of this interaction:



Each of these tasks might fail — in which case, your application should roll back to a sane, consistent state. Each of your tasks has a compensating action:

¡ C1 — Cancel the order that the customer created.

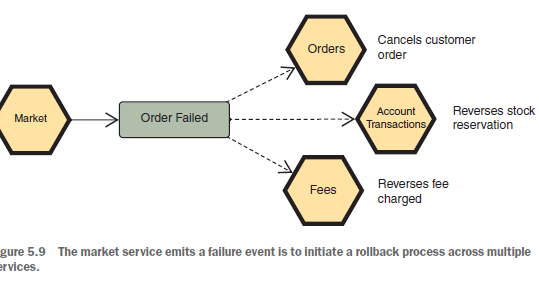
¡ C2 — Reverse the reservation of stock positions.

¡ C3 — Revert the fee charge, refunding the customer.

¡ C4 — Cancel the order placed to market.

¡ C5 — Reverse the state of the order.

What triggers these actions? You guessed it — events! For example, imagine that placing the order to market fails. **The market service will cancel the order by emitting an event — OrderFailed** — that each other service involved in this saga consumes. When receiving the event, each service will act appropriately: the orders service will cancel the customer’s order; the transaction service will cancel the stock reservation; and the fees service will reverse the fee charged, executing actions C1, C2, and C3, respectively. This is shown in figure below:



This form of rollback is intended to make the system ***semantically***, not mathematically **consistent**. Your system on rollback of an operation may not be able to return to the exact same initial state. Imagine one of the tasks executed on calculating the fees was sending out an email. You can’t unsend an email, so you’d instead send another one acknowledging the error and saying the amount that the fees service had charged was deposited back to the account.

Every action involved in a process might have one or more appropriate compensating actions. This approach adds to system complexity — both in anticipating scenarios and in coding for them and testing them — especially because the more services involved in an interaction, the greater the possible intricacy of rolling back. **Anticipating failure scenarios is a crucial part of building services that reflect real-world circumstance, rather than operating in isolation**. When designing microservices, you need to take compensation into account to ensure that the wider application is resilient.

### Advantages and Draw-Backs

* The choreographed style of interaction is helpful because participating services don’t need to explicitly know about each other, which ensures they’re loosely coupled. In turn, this increases the autonomy of each service.

Unfortunately, It’s not perfect:

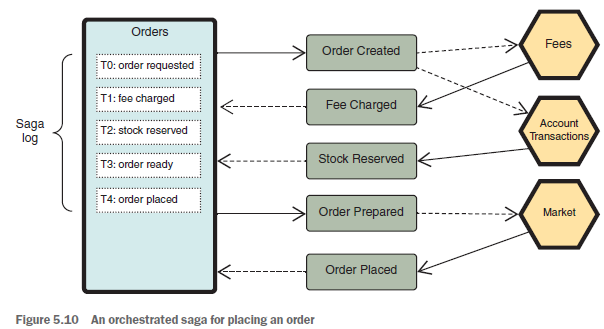
* No single piece of your code knows how to execute a sell order. This can **make validation challenging,** spreading those rules across multiple distinct services.
* It also increases the complexity of state management: each service needs to reflect distinct states in the processing of an order. For example, the orders service must track whether an order has been created, placed, canceled, rejected, and so on. This additional complexity increases the difficulty of reasoning about your system.
* Choreography also introduces **cyclic dependencies between services**: the orders service emits events that the market service consumes, but, in turn, it also consumes events that the market service emits. **These types of dependencies can lead to release time coupling between services**.
* Generally, when opting for an asynchronous communication style, you must invest in monitoring and tracing to be able to follow the execution flow of your system. In case of an error, or if you need to debug a distributed system, the monitoring and tracing capabilities act as a flight recorder. **You should have all that happens stored there so you can later investigate every single event to make sense of what happened in a multitude of systems**. This capability is crucial for choreographed interactions.
* **A choreographed approach makes it difficult to know how far along a process is**. Likewise, **the order of rollback might be important**; this isn’t guaranteed by choreography, which has looser time guarantees than an orchestrated or synchronous approach. For simple, near-instant workflows, knowing where you’re at is often irrelevant, but many business processes aren’t instant — they might take multiple days and involve disparate systems, people, and organizations.

## Orchestrated Sagas

Instead of choreography, you can use *orchestration* to implement sagas. In an orchestrated saga, **a service** takes on the role of **orchestrator** (or coordinator): **a process that executes and tracks the outcome of a saga across multiple services.** An orchestrator might be an independent service — recall the verb-oriented services from chapter 4 — or a capability of an existing service.

The sole responsibility of the orchestrator is to manage the execution of the saga. It may interact with participants in the saga via asynchronous events or request/response messages. Most importantly, it should track **the state of execution for each stage** in the process; this is sometimes called the *saga log*.

Let’s make the orders service a saga coordinator. **the orders service tracks** the execution of each sub-step in the process of placing an order. It’s useful to think of the **coordinator as a state machine**: a series of states and transitions between those states. **Each response from a collaborator triggers a state change**, moving the orchestrator toward the saga outcome. Figure below, illustrates the happy path where a customer places an order successfully:



a saga won’t always be successful. In an orchestrated saga, **the coordinator is responsible** **for initiating appropriate reconciliation actions** to return the entities affected by the failed transaction to a valid, consistent state.

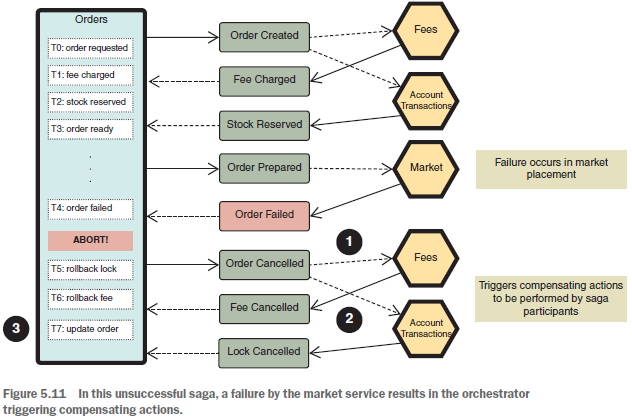
imagine the market service can’t place the order to market. The orchestrating service will initiate compensating actions:

**1** It’ll issue a request to the account transaction service to reverse the lock placed on the holdings to be sold.

**2** It’ll issue a request to cancel the fee that was charged to the customer.

**3** It may change the state of the order to reflect the outcome of the saga — for example, to rejected or failed. This depends on the business logic (and whether failed orders should be shown to the customer or retried).

In turn, the orchestrator also could track the outcome of actions 1 and 2. Figure below illustrates this failure scenario:



* Don’t forget that compensating actions might not all happen instantaneously or at the same time. For example, if the fee was charged to a customer’s debit card, it might take a week for their bank to reverse the charge.

But if the desired actions you want to happen can fail, **the compensating actions — or the orchestrator itself — also could fail**. You should **design** **compensating actions** **to be safe to retry without unintentional side effects (for example, double refunds).** At worst, repeated failure during rollback might require manual intervention. Thorough error monitoring should catch these scenarios.

### Advantages and Draw-Backs

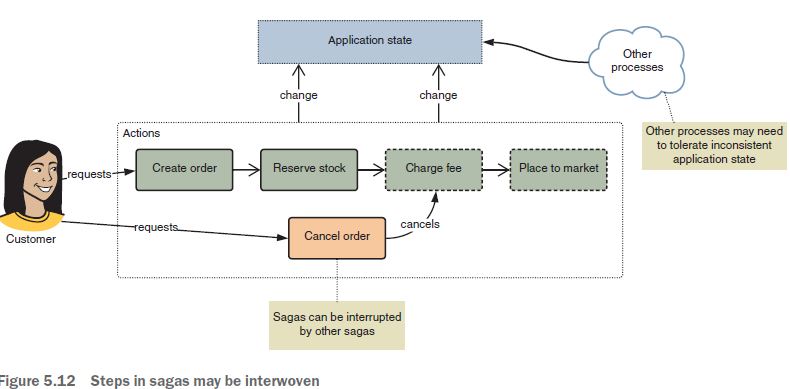
* Centralizing the saga’s sequencing logic in a single service makes it significantly easier to reason about the outcome and progress of that saga, as well as change the sequencing in one place. In turn, this can simplify individual services, reducing the complexity of states they need to manage, because that logic moves to the coordinator.
* This approach does run the risk of moving too much logic to the coordinator. At worst, this makes the other services anemic wrappers for data storage, rather than autonomous and independently responsible business capabilities.

**Many microservice practitioners advocate peer-to-peer choreography over orchestration, as they see this approach to reflect the “smart endpoints, dumb pipes” aim of microservice architecture**, in contrast to the heavy workflow tools (such as WS-BPEL) people often used in enterprise SOA. But **orchestrated approaches** are becoming increasingly popular in the community, **especially for building long-running interactions**, as seen by the popularity of projects like Netflix Conductor and AWS Step Workflows.

## Interwoven Sagas

Unlike ACID transactions, sagas aren’t isolated. The result of each local transaction is immediately visible to other transactions affecting that entity. This visibility means that a given entity might get simultaneously involved in multiple, concurrent sagas. As such**, you need to design your business logic to expect and handle intermediate states.** The complexity of the interleaving required primarily depends on the nature of the underlying business logic.

imagine that a customer placed an order by accident and wanted to cancel it. If they issued their request before the order was placed to market, the order placement saga would still be in progress, and this new instruction would potentially need to interrupt it:



Three common strategies for handling interwoven sagas are available: **short-circuiting**, **locking**, **and interruption**.

### Short-Circuiting

You could prevent the new saga from being initiated while the order is still within another saga. For example, the customer couldn't cancel the order until after the market service attempted to place it to the market. This isn’t great for a user but is probably the easiest strategy!

### Locking

You could use locks to control access to an entity. Different sagas that want to change the state of the entity would wait to obtain the lock. You’ve already seen an example of this in action: you place a reservation — or lock — on a stock balance to ensure that a customer can’t sell a holding twice if it’s involved in an active order. This can lead to deadlocks if multiple sagas block each other trying to access the lock, requiring you to implement deadlock monitoring and timeouts to make sure the system doesn’t grind to a halt.

### Interruption

Lastly, you could choose to interrupt the actions taking place. For example, you could update the order status to “failed.” When receiving a message to send an order to market, the market gateway could revalidate the latest order status to ensure the order was still valid to send, and in this case, it would see a “failed” status. This approach increases the complexity of business logic but avoids the risk of deadlocks.