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.

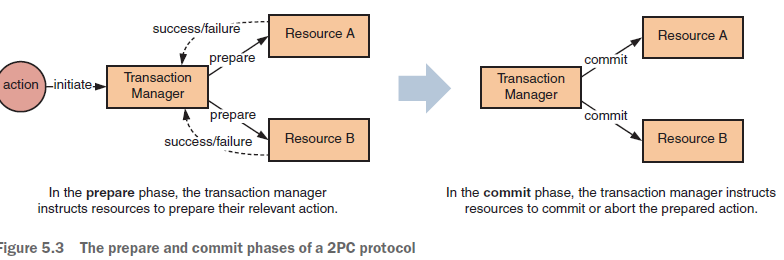
At this stage, your system is in an inconsistent state: stock is reserved, an order is created, but you haven’t charged the customer. You can’t leave it like this — so the implementation of orders needs to initiate corrective action, instructing the account transactions service to compensate and remove 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*