**Micro services Design Pattern**

Microservices architecture has gained popularity for its ability to improve the flexibility, testability, and scalability of software systems.

By breaking down a monolithic application into smaller, independently deployable services, micro services enable teams to develop, deploy, and scale each service independently.

However, the implementation of microservices architecture comes with its own set of challenges such as:

* **Data Consistency and Eventual Consistency:** In a microservices architecture, data is often distributed across multiple nodes, which can be located in different data centers or even different geographic regions. At any given point in time, there can be discrepancies in the state of data between various nodes. This phenomenon is known as eventual consistency.
* **Security:**Microservices architecture introduces a larger attack surface for malicious actors compared to monolithic systems. It’s crucial to establish appropriate security mechanisms while building microservices. Design patterns such as the API Gateway pattern can help.
* **Scalability and Database Performance:**Microservices are known for their scalability. However, while it is relatively easy to scale the application layer by adding more instances, databases can become performance bottlenecks if not designed for scalability. Patterns such as Database per Service and CQRS help solve this challenge.

Design patterns provide proven solutions to common problems encountered in a micro services architecture. By applying an appropriate design pattern, these problems can be effectively addressed.

There are different Design Patterns

1. Decomposition Design Pattern
2. Decompose by Business needs
3. Decompose by SubDomain
4. Strangler Design Pattern

Suppose we have monolithic application and we are breaking the monolithic application into microservices.

but 80% of the work we do is with brownfield applications, which are big, monolithic applications. Applying all the above design patterns to them will be difficult — breaking them into smaller pieces, at the same time it's being used, is a big task.

The Strangler pattern is a way to gradually replace an old application with a new one. It’s like how a vine slowly wraps around and overtakes a tree. In this approach, instead of replacing the whole application at once, you do it step-by-step.

You start by dividing the application into smaller sections, or "domains." Each domain is moved to its own service that can work alongside the original application. So, both the old and the new versions exist together, sharing the same web address or "URI space."

As you continue, more of the application’s functionality gets transferred to the new services. Over time, the new application effectively "strangles" the old one by taking over all its responsibilities. Eventually, the old application is no longer needed, and you can shut it down.

**Micro services Data Management**

1. **Database per services:-**
2. There is challenge like if we need to adopt the ACID principle for all the distributed DB. It means order is cancelled in one services it should be cancelled for all the DB of different microservices.

How To rollback the databases- This can be done using SAGA principle

1. Let us suppose if I want to join the tables of 2 different services. This can be done by CQRS.
2. Let us suppose if We want to scale the database of order service, then I don’t need to cale the other services database

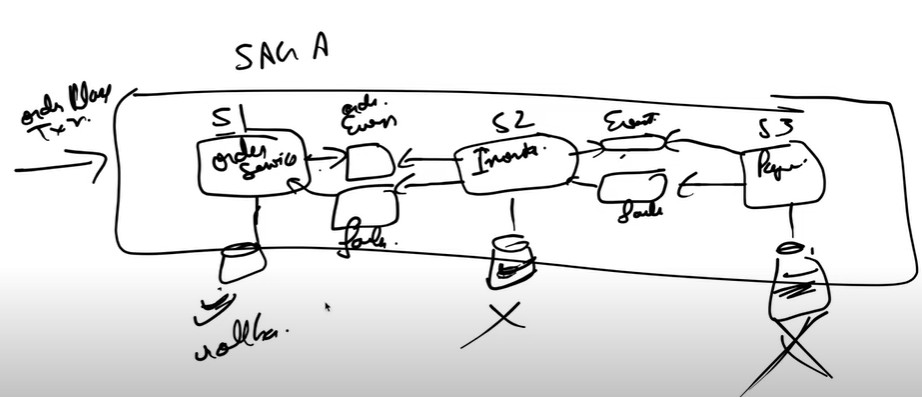
**Why we need SAGA micro services DP?**

**(Sequence of Local Transaction)**

Let us suppose we have 3 services like order, inventory and payment. When we create order, it updates the DB of order and inventory but while payment, it got failed.

Now as per ACID principle, either it should be complete success of complete failed.

Now SAGA with rollback the complete order and DB.

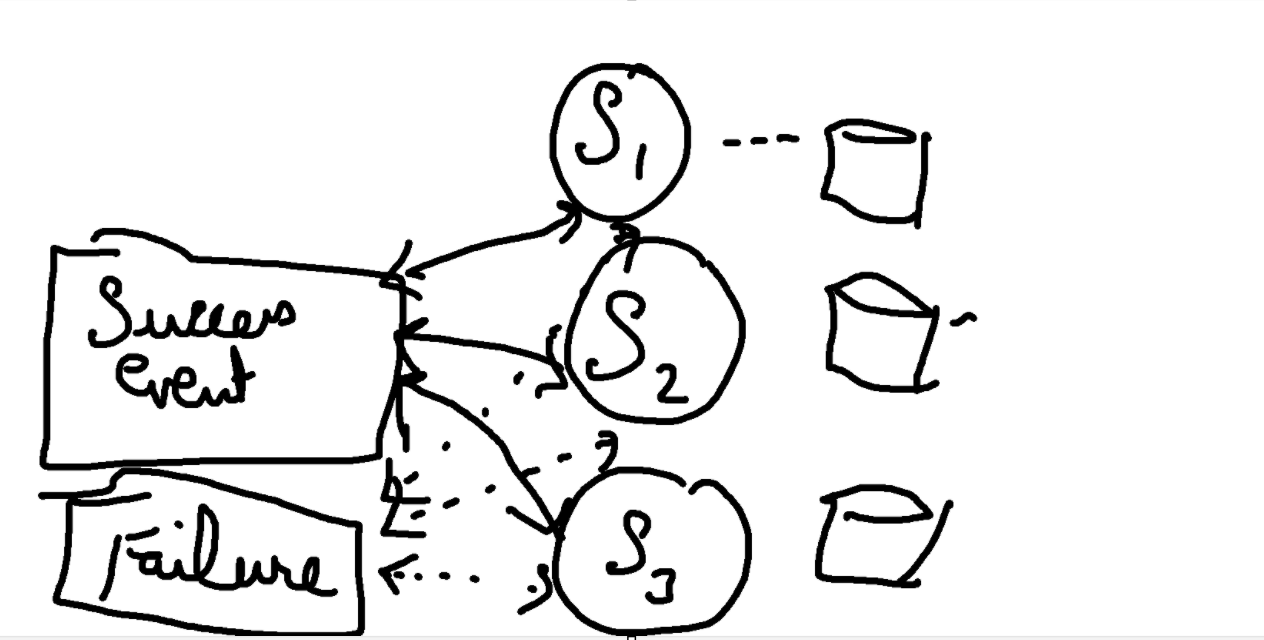
****

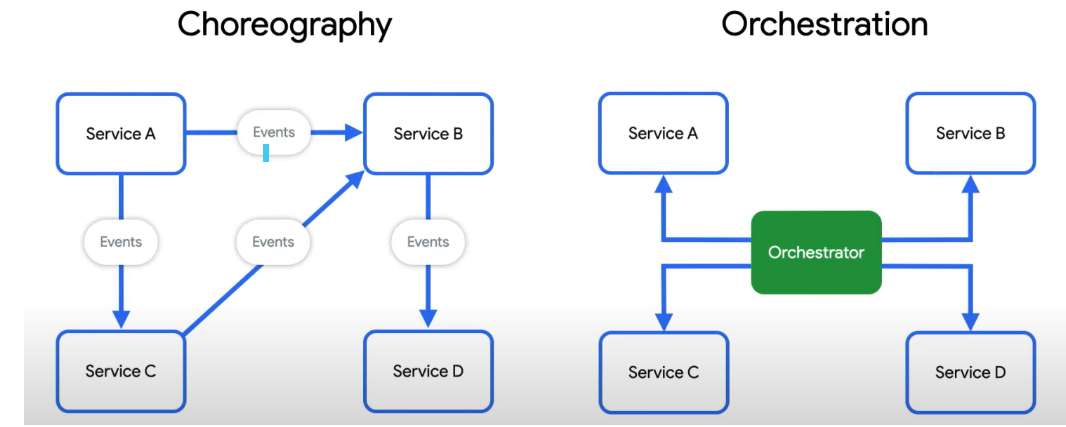
When order placed then order services update the DB and generate the event to Inventory. Inventory will consume the event and update the DB and send event to Payment.

Let us suppose Payment is failed to update the DB, then Payment will send the failure event to which will consume by Inventory and rollback the DB. Now Inventory will send the failure event to Order and then order will rollback the DB.

There are 2 types of SAGA design pattern

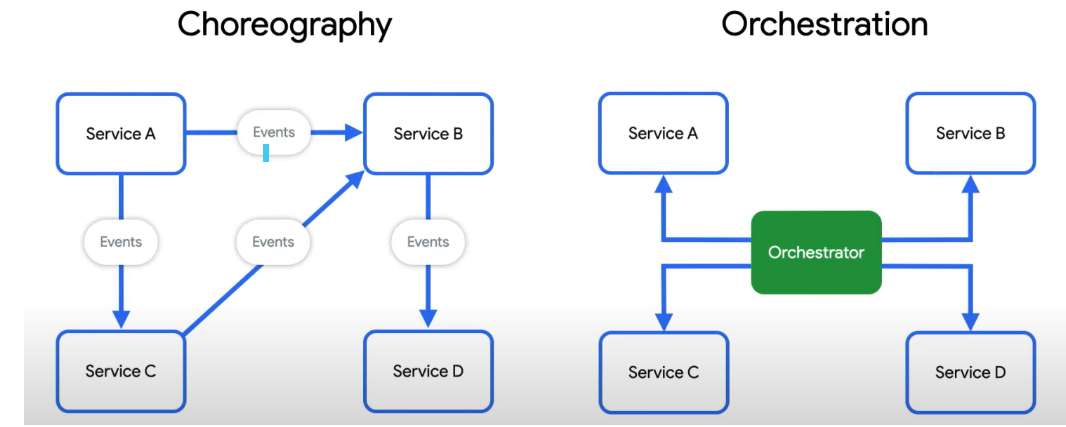
1. **Choreography---**

****

****

There might be a cyclic dependency for choreography SAGA.

1. **Orchestration Design Pattern**

****

CQRS🡪

1. Command Query Request Segregation

#### ****Problem****

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services — it's not possible. So, how do we implement queries in a microservices architecture?

#### ****Solution****

CQRS suggests splitting the application into two parts — the command side and the query side. The command side handles the Create, Update, and Delete requests. The query side handles the query component by using materialized views. The **event sourcing design pattern** is generally used along with it to create events for any data change. As such, materialized views are kept up to date by subscribing to the stream of events.

