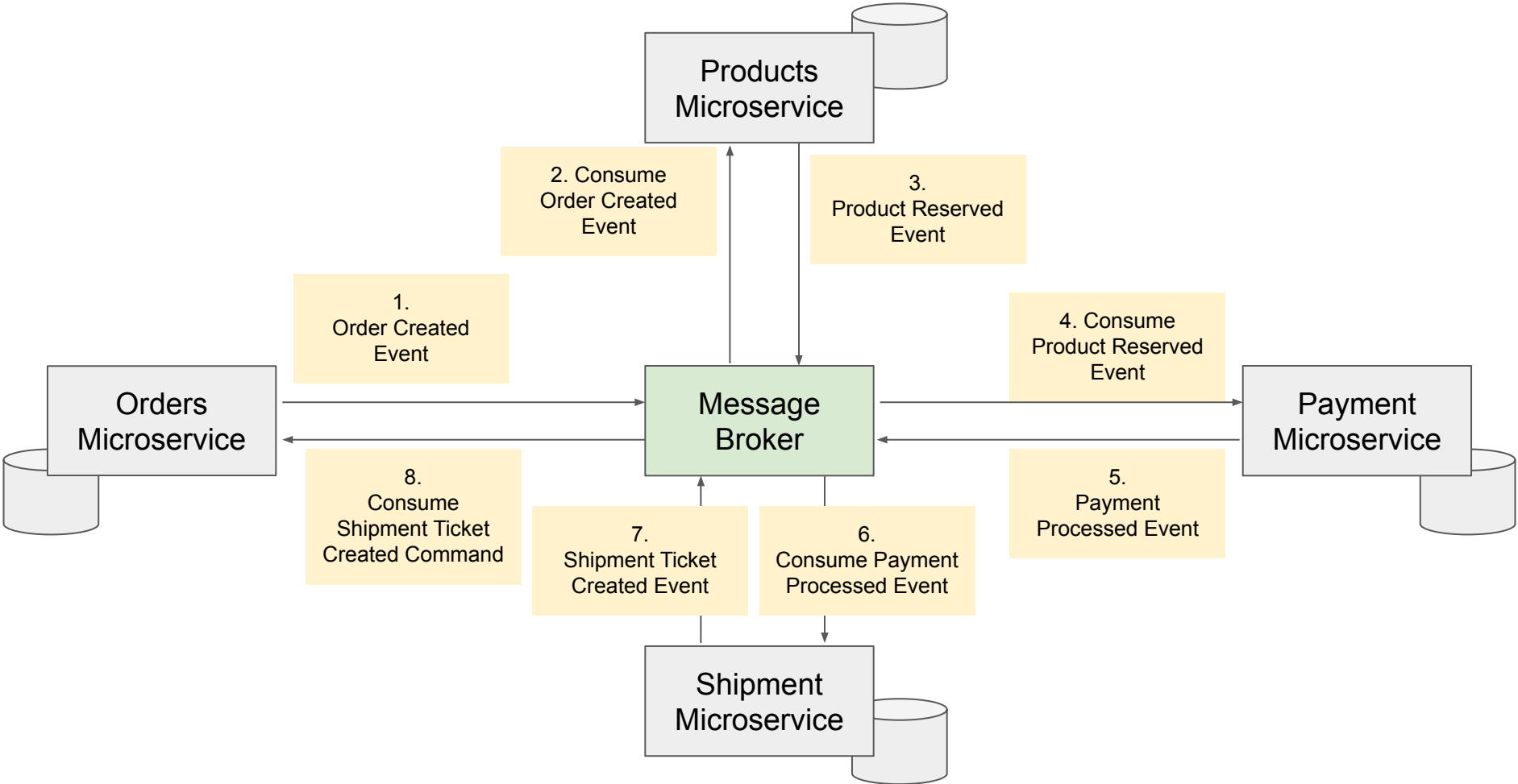


SAGA

Design Pattern

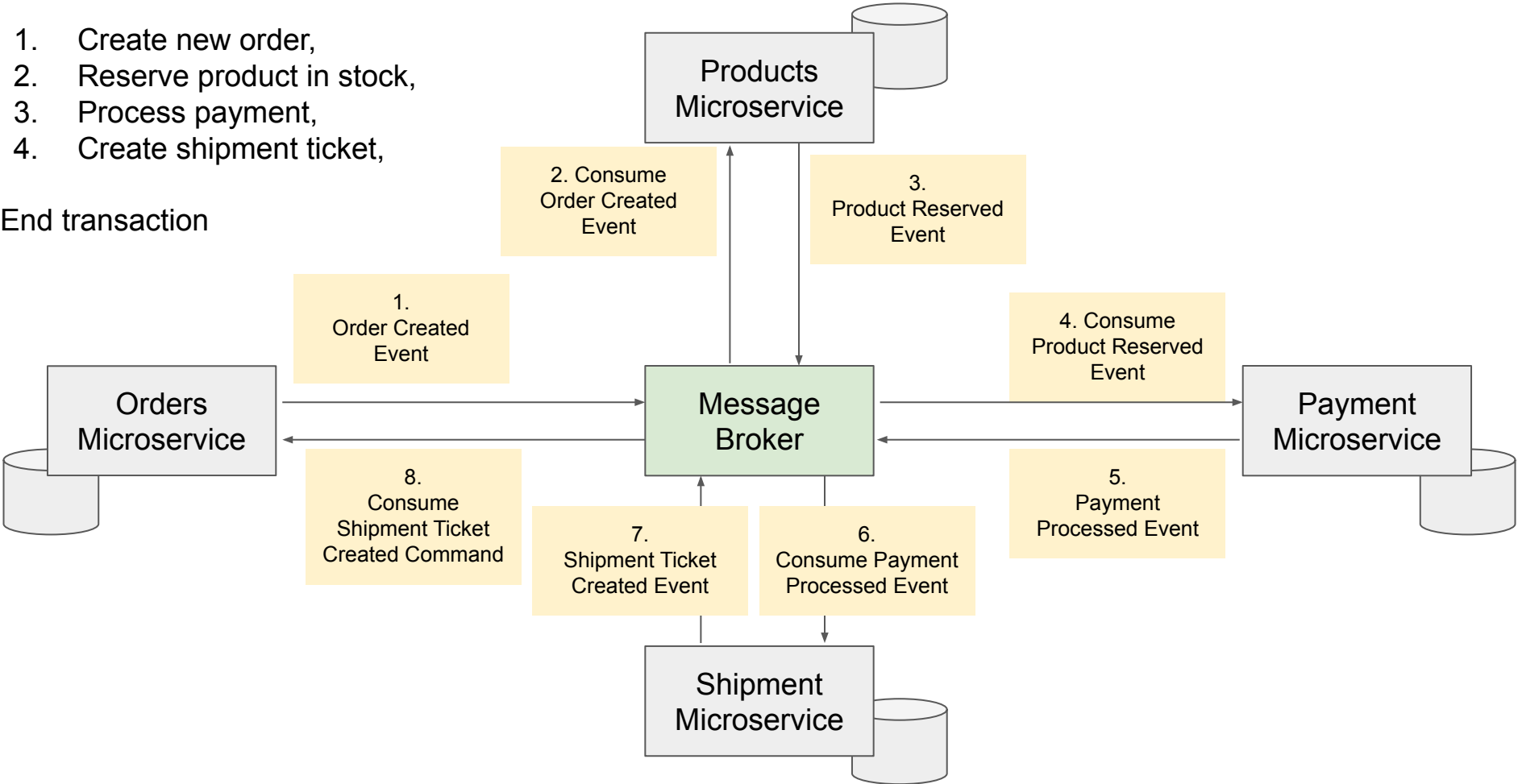
Choreography-Based Saga



Begin Transaction

- 1. Create new order,
- 2. Reserve product in stock,
- 3. Process payment,
- 4. Create shipment ticket,

End transaction



Initial Flow:

- 1. Create Order operation,
- 2. Reserve Product operation.

Compensating Transaction:

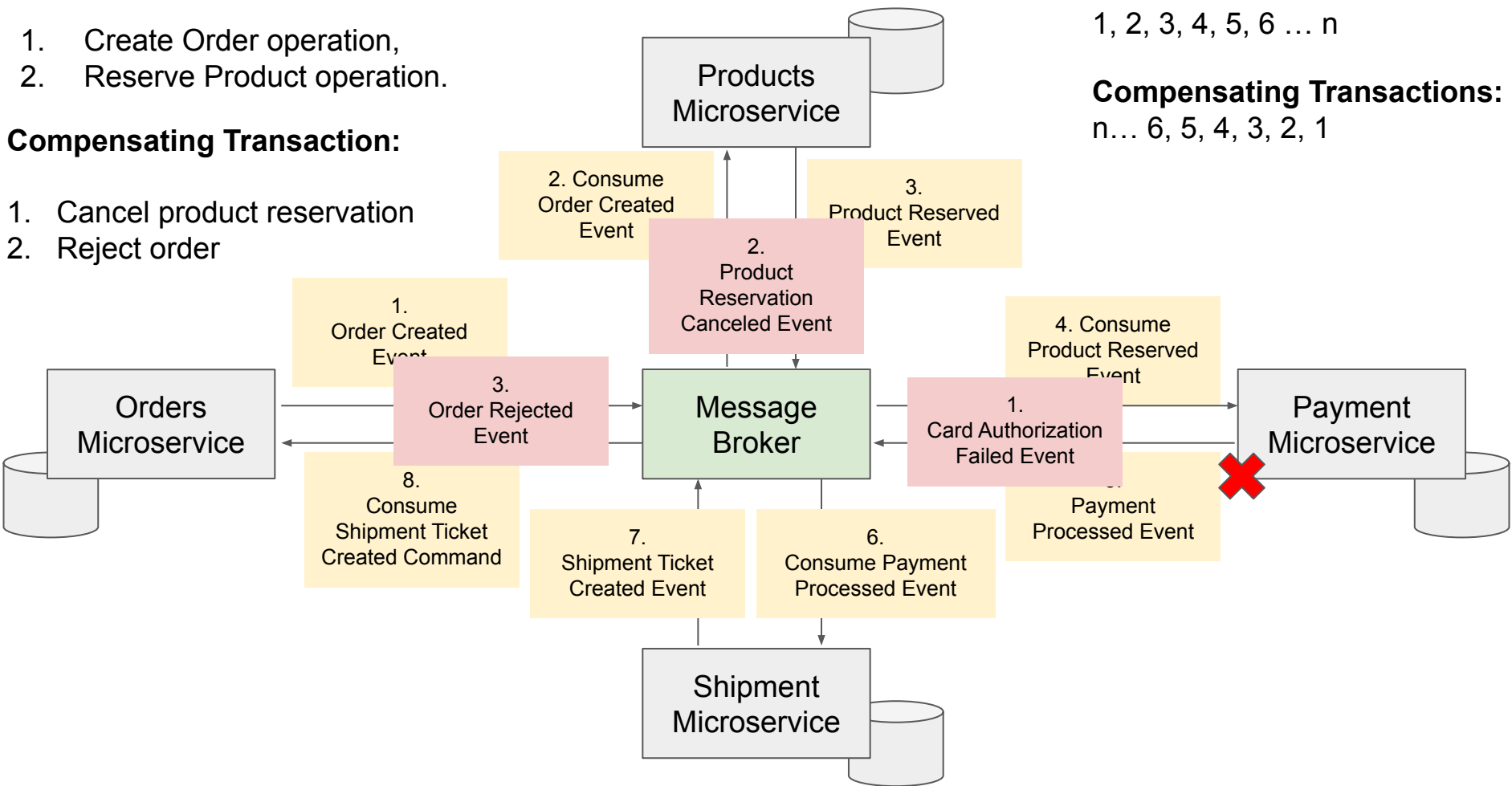
- 1. Cancel product reservation
- 2. Reject order

Initial Flow:

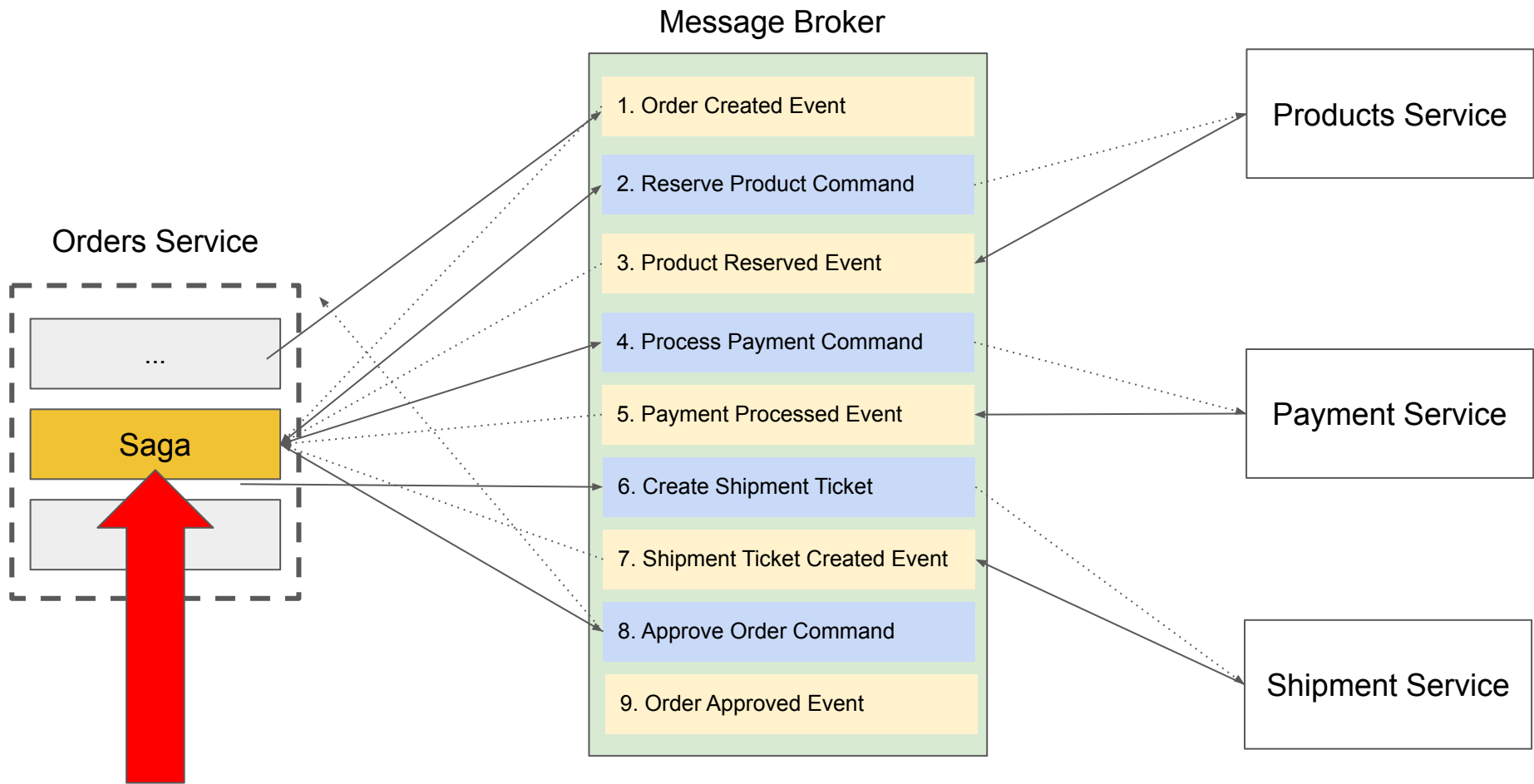
1, 2, 3, 4, 5, 6 ... n

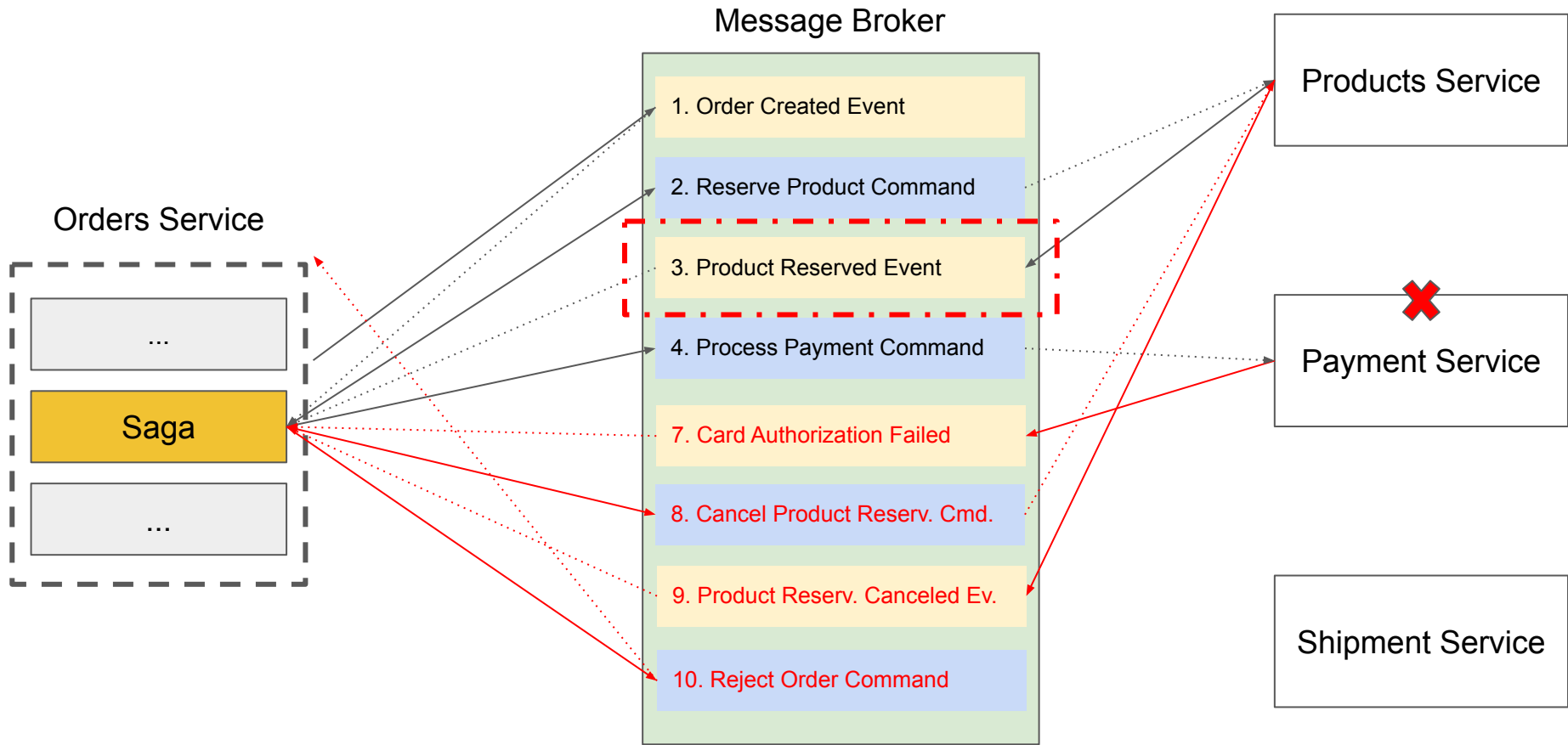
Compensating Transactions:

n... 6, 5, 4, 3, 2, 1

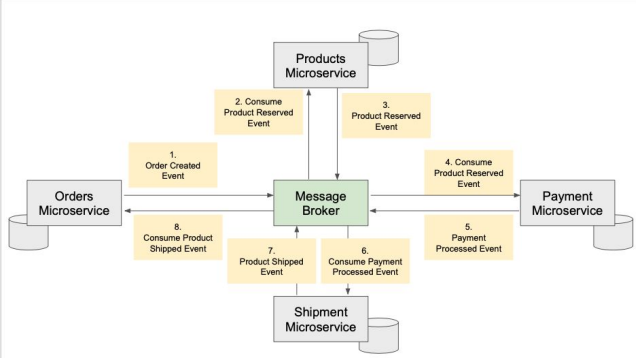


Orchestration-Based Saga





Which Saga pattern to use?



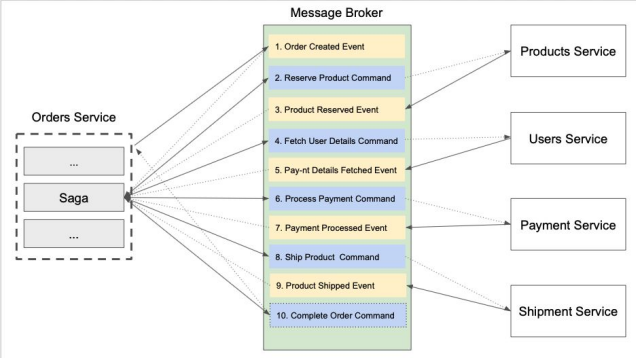
Choreography-based Saga

Benefits:

- Good for simple workflows that require few microservices and don't need a coordination logic,
- Doesn't require additional service implementation and maintenance to coordinate transactions,
- Doesn't introduce a single point of failure, since the responsibilities are distributed across the saga participants.

Drawbacks:

- Workflow can become confusing when adding new steps and Microservices. The more steps in a transaction, the more difficult it is to track which saga participants listen to which commands.
- There's a risk of cyclic dependency between saga participants because they have to consume each other's commands.
- Integration testing is difficult because all services must be running to simulate a transaction.



Orchestration-based Saga

Benefits:

- Good for complex workflows involving many Microservices,
- Easier to control the flow of activities,
- Doesn't introduce cyclic dependencies, because the orchestrator unilaterally depends on the saga participants,
- Saga participants don't need to know about commands for other participants. Clear separation of concerns simplifies business logic.

Drawbacks:

- Additional design complexity requires an implementation of a coordination logic.
- There's an additional point of failure, because the orchestrator manages the complete workflow.