Distributed Transactions : SAGA

Distributed System Transactions

2 Phase Commit CANNOT be used

SAGA pattern is used for Distributed Transactions

Distributed Transactions : SAGA

SAGA are complicated !!!

Multiple frameworks available for realization

Framework hides complexity

Distributed Transactions : SAGA

Section objective

1. SAGA pattern
2. Types of SAGA (Orchestration Vs. Choreography)
3. Implementation considerations
4. Case Study : Working with SAGA for ACME Sales booking transaction (UML, JAVA, Kafka, PostgreSQL, MongoDB)

Data consistency in Distributed Tx

Distributed transactions and data consistency

Data consistency in microservices

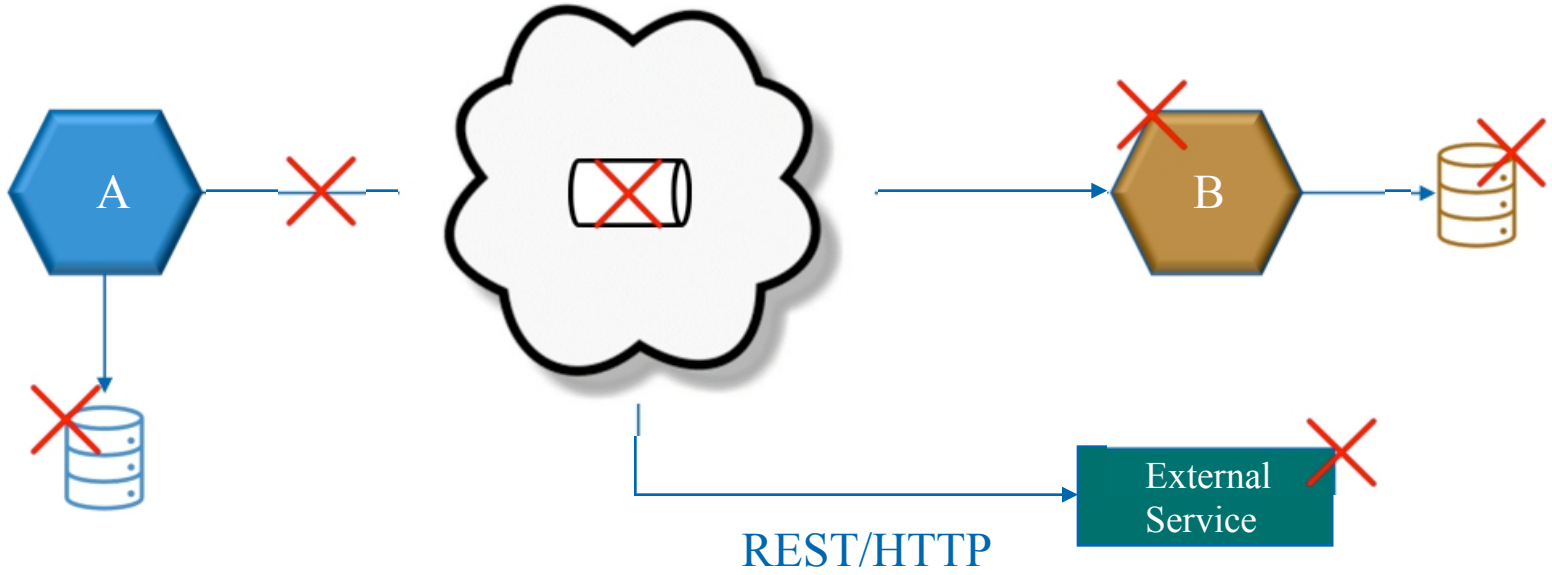
Managing data consistency

Introduction to SAGA pattern

Design for failure

Always anticipate that there will be FAILURES

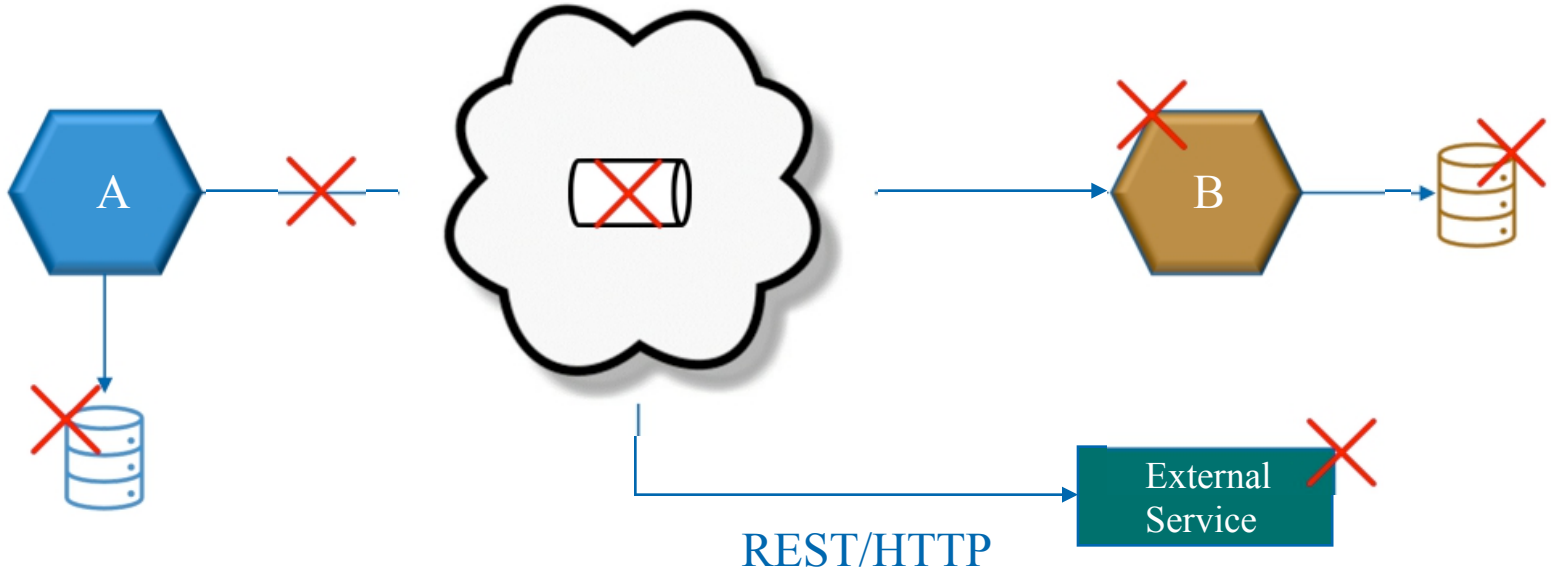
- Identify the "Failure Points" in your architecture



Pro - actively address the "Failure Points"

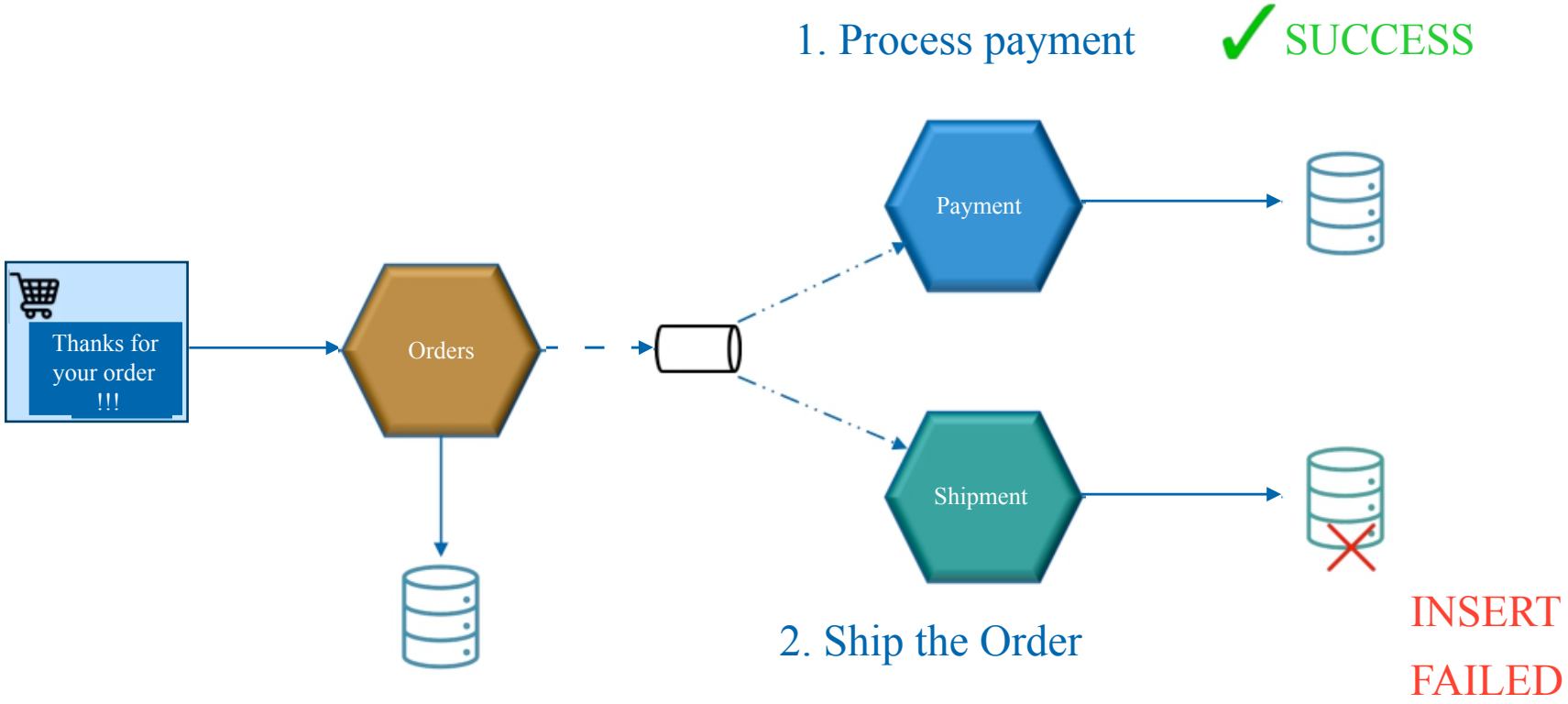
Design for failure

NOT addressing the failures = Inconsistent state of data



NOT addressing the failures = In consistent state of data

Example: Data inconsistency



Customer

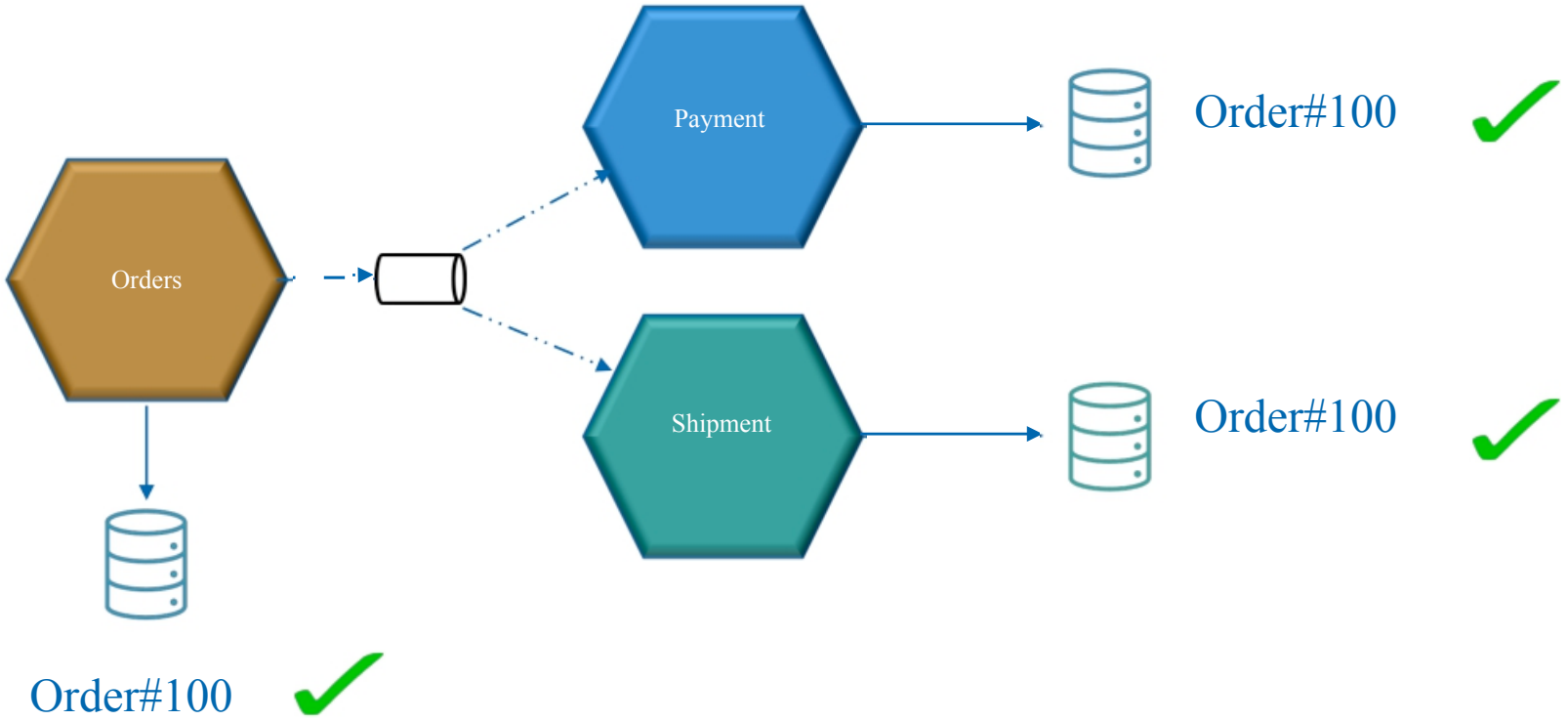
Customer paid

but will not receive the order ☹

Shipment database is not in sync with other databases !!!

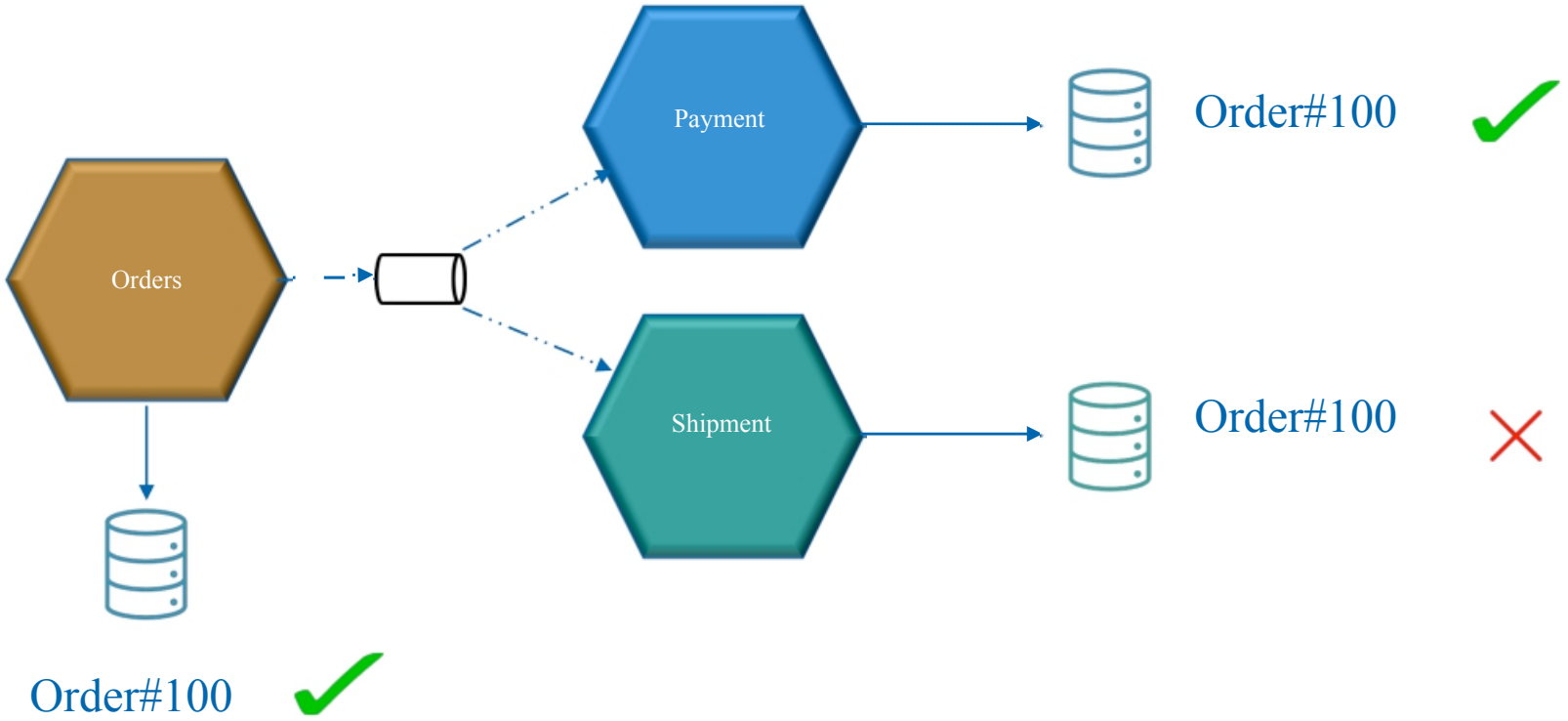
Objective

Data MUST be consistent across all distributed services



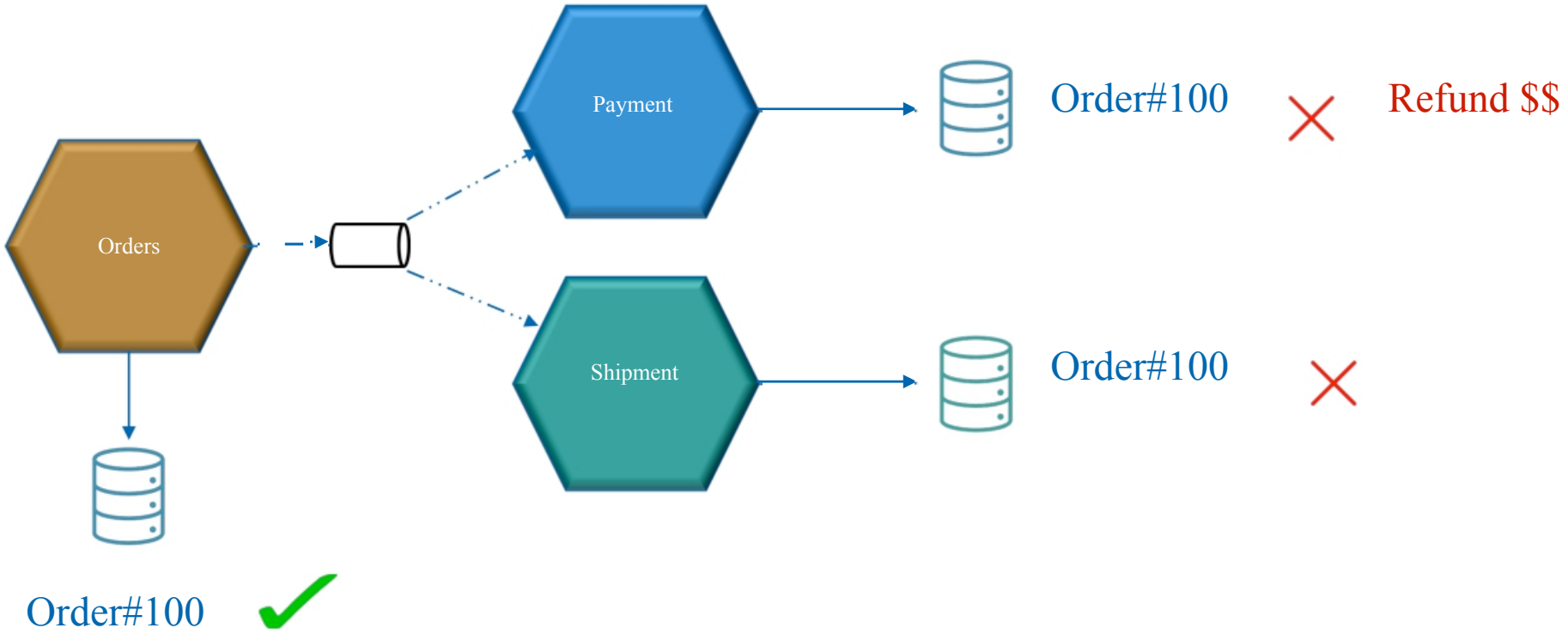
Objective

Data MUST be consistent across all distributed services



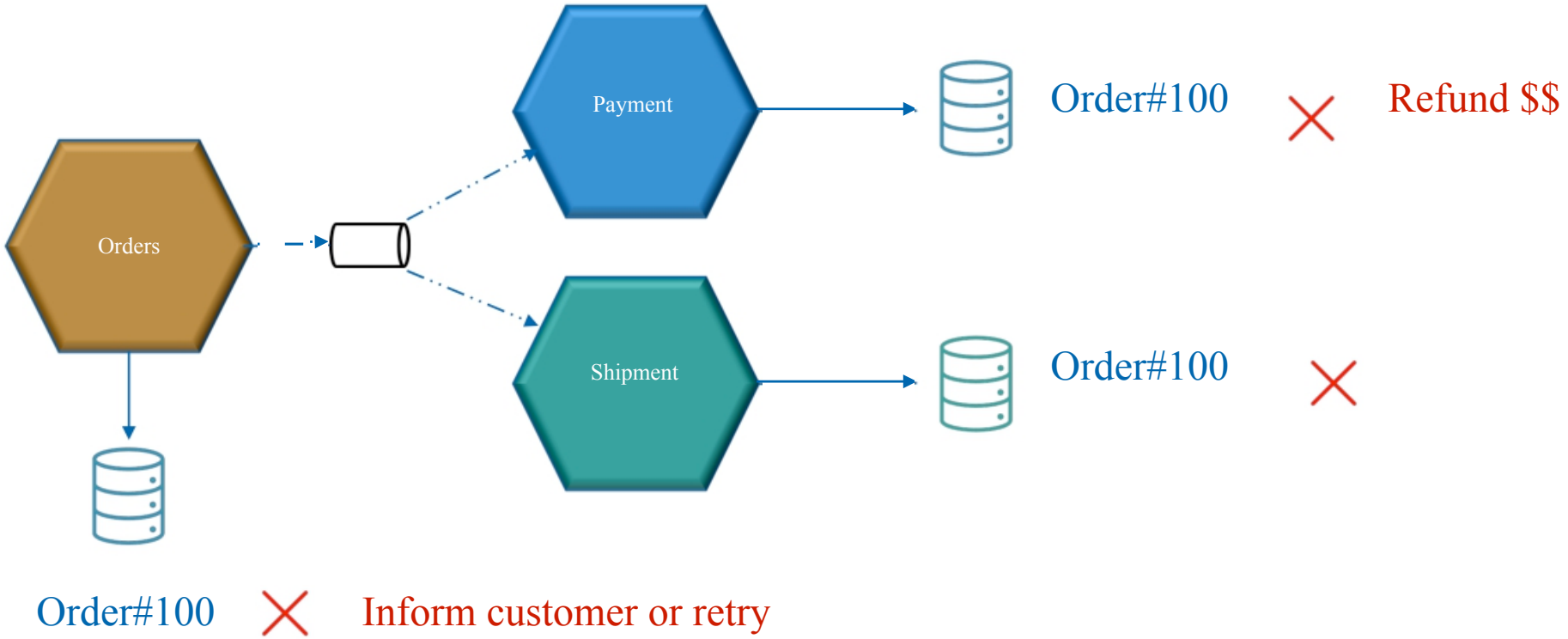
Objective

Data MUST be consistent across all distributed services



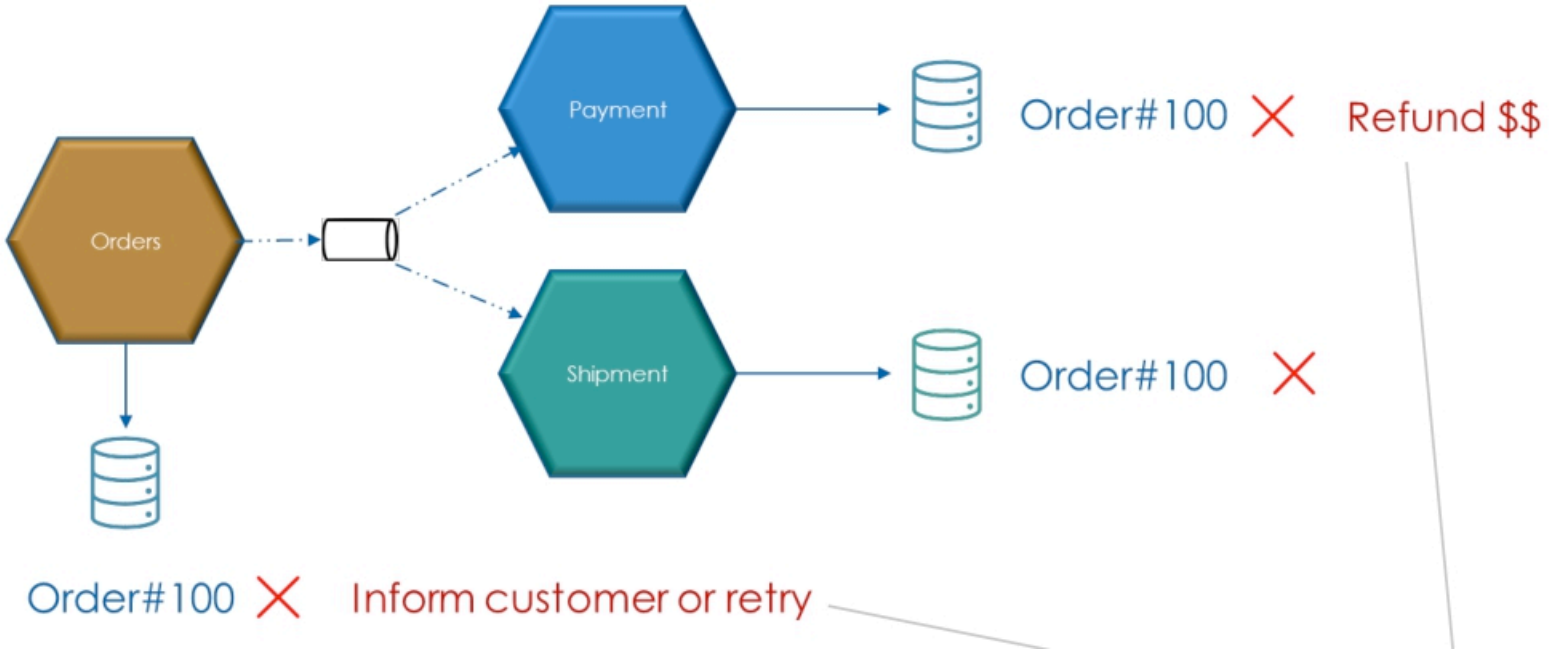
Objective

Data MUST be consistent across all distributed services



Local Transactions

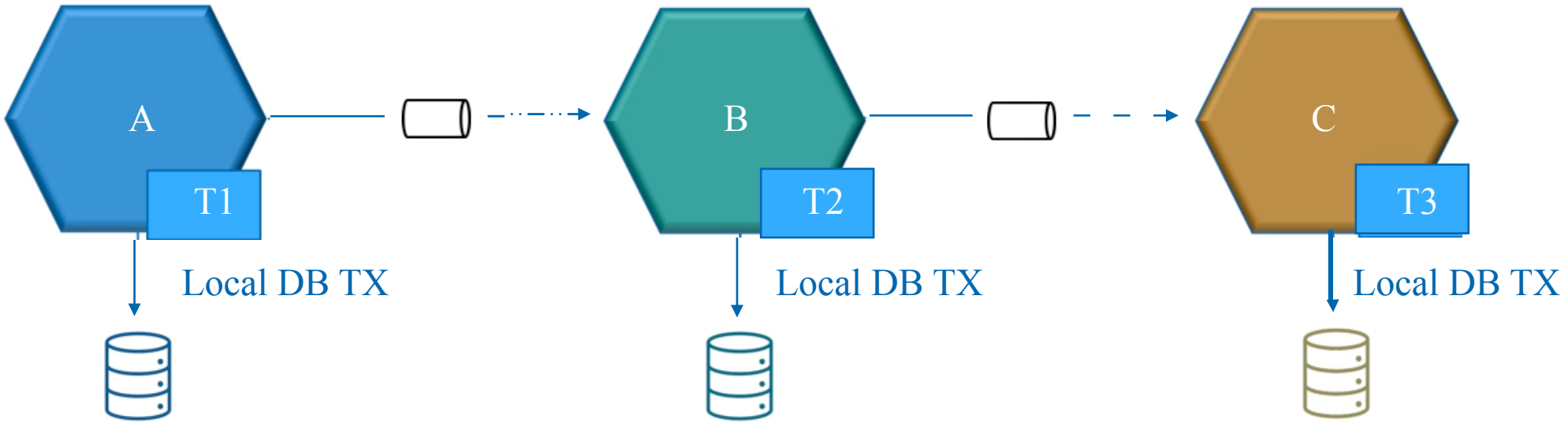
CANNOT be used for reverting the DB changes



CANNOT use Database Tx Rollback

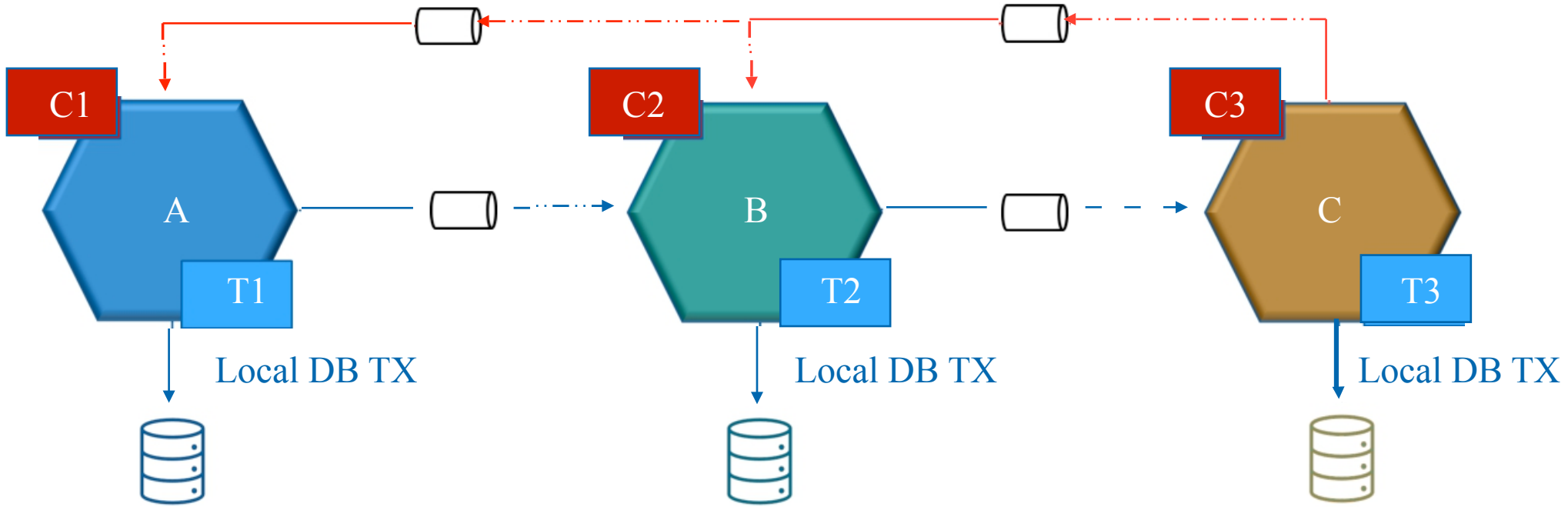
SAGA pattern

Use Local Transactions coupled with compensating transactions



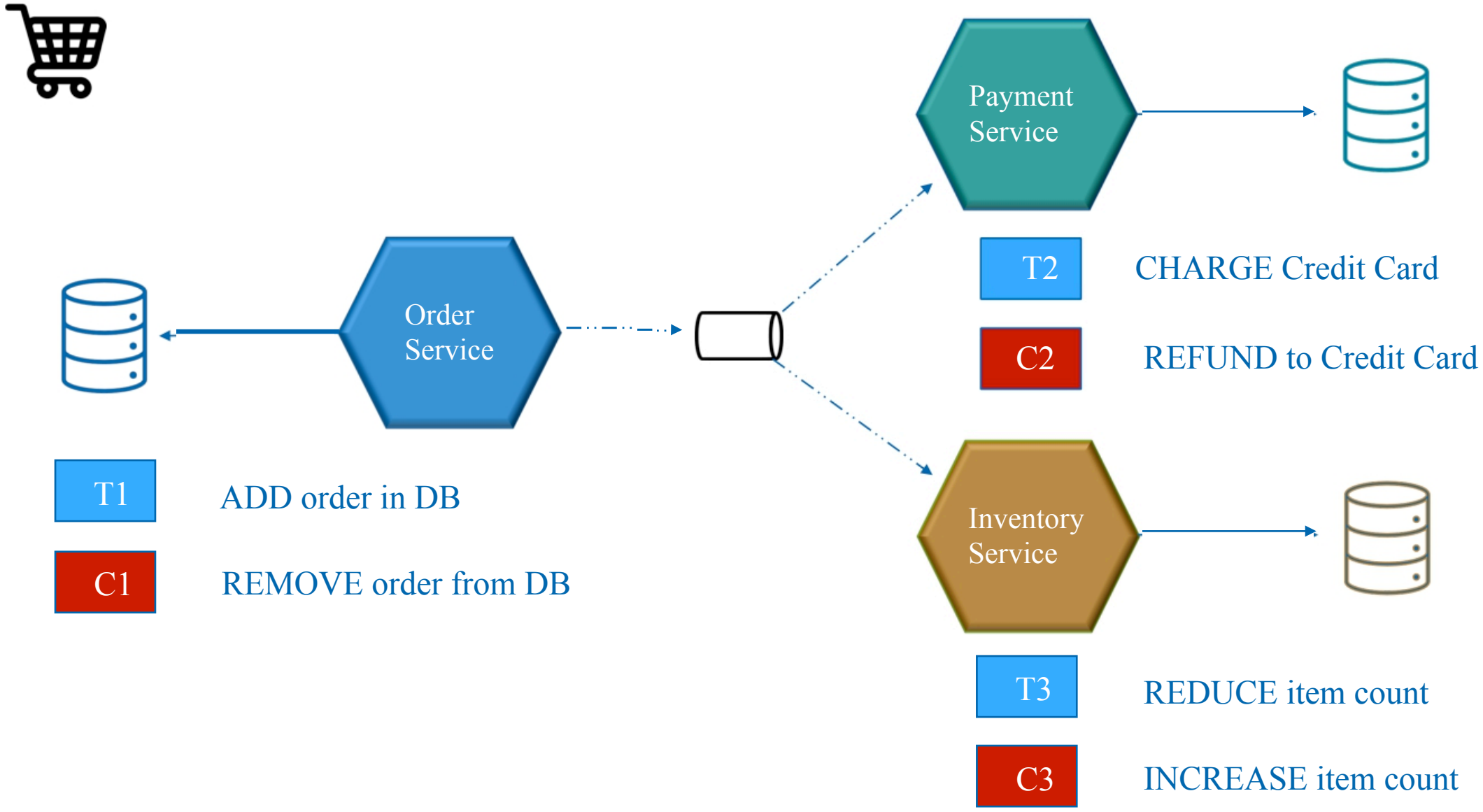
SAGA Pattern

Compensating transactions revert the database changes



Requires each Tx to have corresponding Compensating Tx

Example: Compensating transactions



History

Introduced in a paper published in 1987 !!

Sagas

Hector Garcia-Molina, Kenneth Salem
Princeton University 1987

https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

SAGA pattern

May be applied to monolithic & distributed systems

- Distributed SAGA = Services are distributed

Quick Review

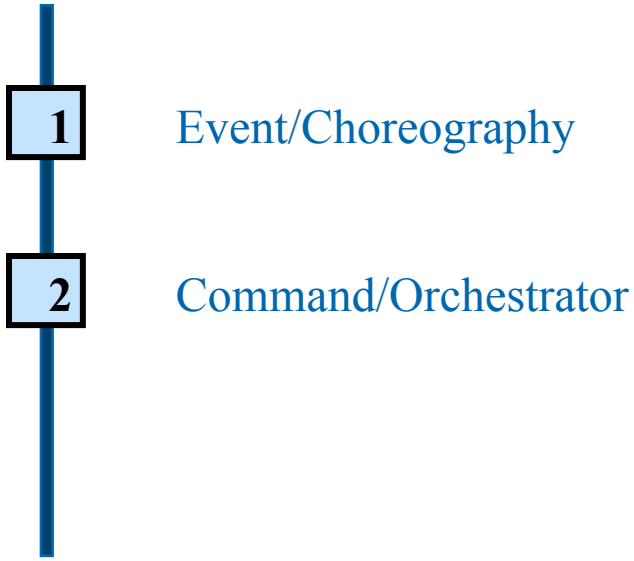
SAGA pattern - Managing data consistency across microservices

Uses Local Transactions for persisting Txns to the database

Uses compensating transactions for reverting database changes

SAGA Call Flow

De - Centralized versus Centralized



Flavors of SAGA

SAGA may be implemented in two ways

Event/Choreography

- NO Central component to manage transactions

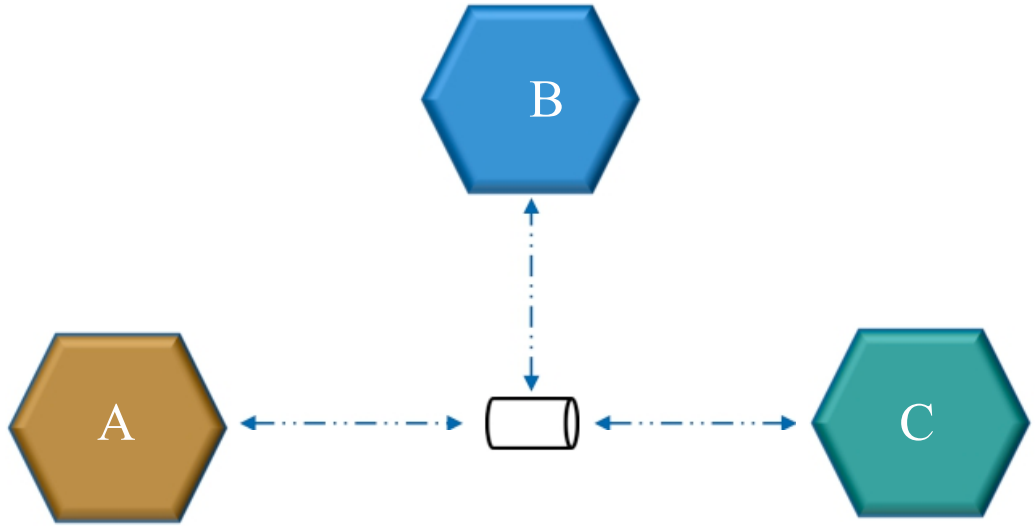
Command/Orchestration

- Central component for managing the flow
- Central component = SAGA Execution Coordinator (SEC)

Event/Choreography SAGA

NO central component to manage the SAGA

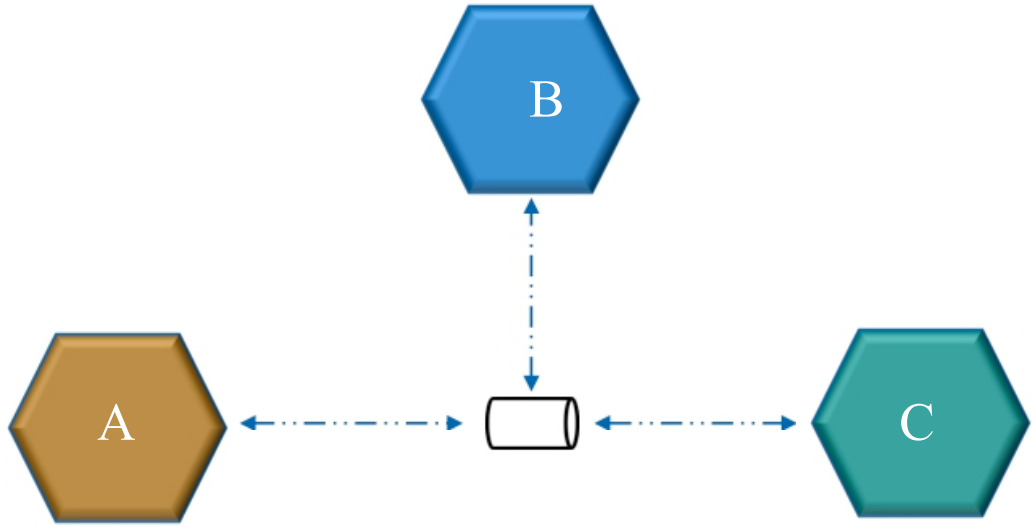
- Services emit & receive domain events
- Services decide on actions independently



Challenges with Event Choreography

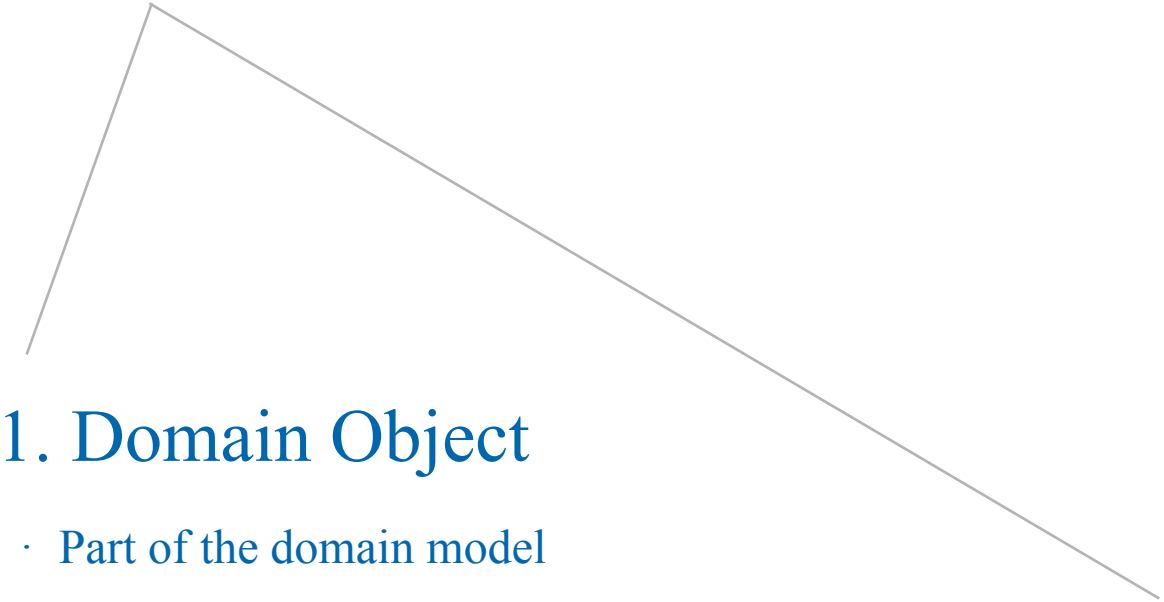
Leads to HIGHLY decoupled services but there are challenges

- Difficult to implement, test and debug
- Out of sequence events
- Coordinating the failure scenarios
- Cyclic dependencies



Command/Orchestration

Central component manages the calls to services in SAGA



1. Domain Object

- Part of the domain model

2. Dedicated orchestrator

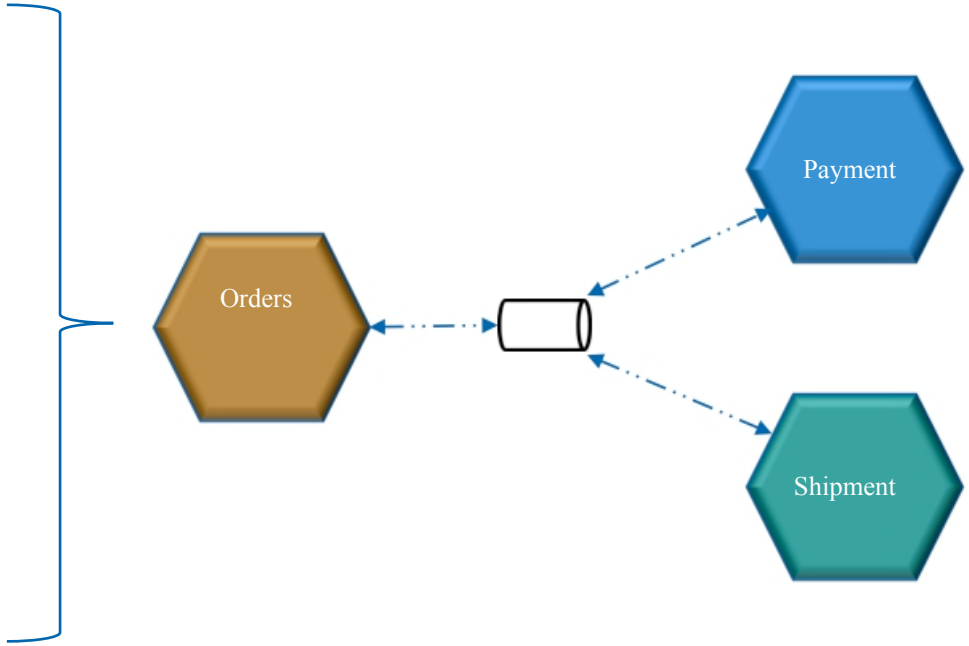
- Generic i.e., outside of the domain model

Central Component = SAGA Execution Coordinator

SAGA Command/Orchestrator

1. Key domain object plays the role of SEC

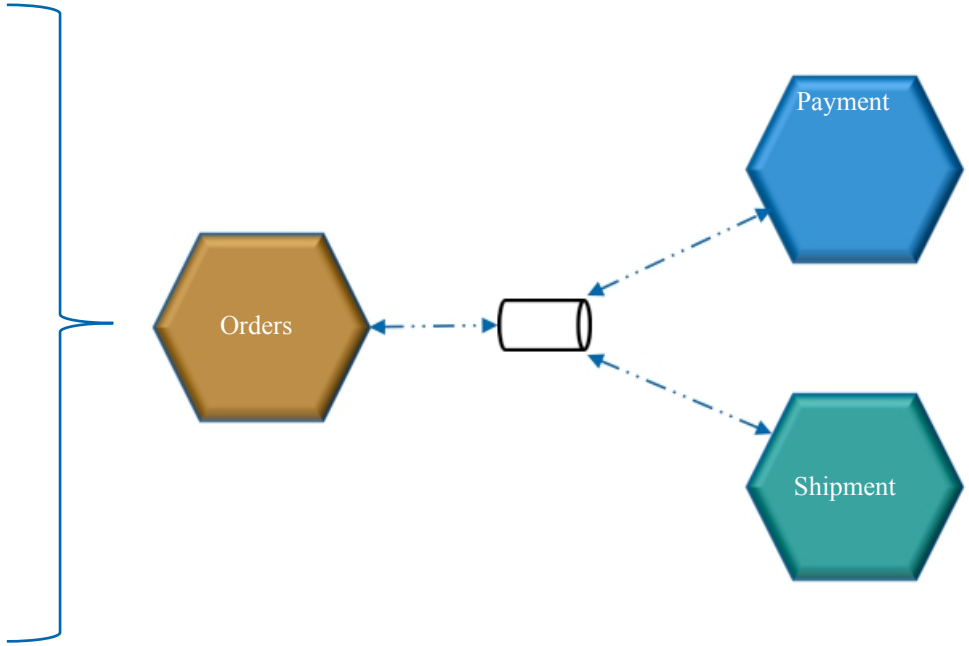
- Manages state of orders
- Initiates the transactions on services
- Manages failures



SAGA Command/Orchestrator

1. Key domain object plays the role of SEC

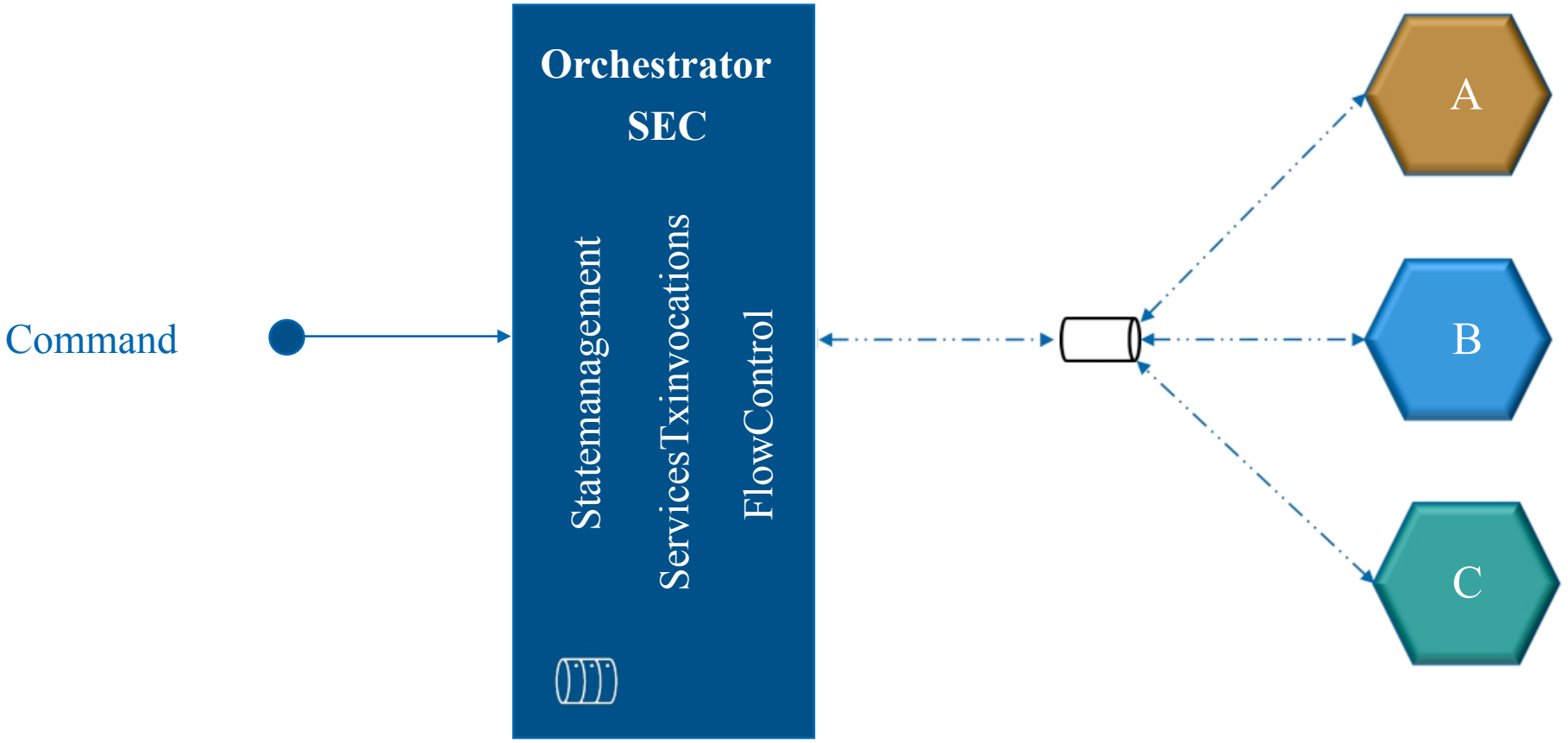
- Uses Command/Reply pattern
- Reliable messaging pattern



State Management will introduce COMPLEXITY in the domain object !!!

SAGA Command/Orchestrator

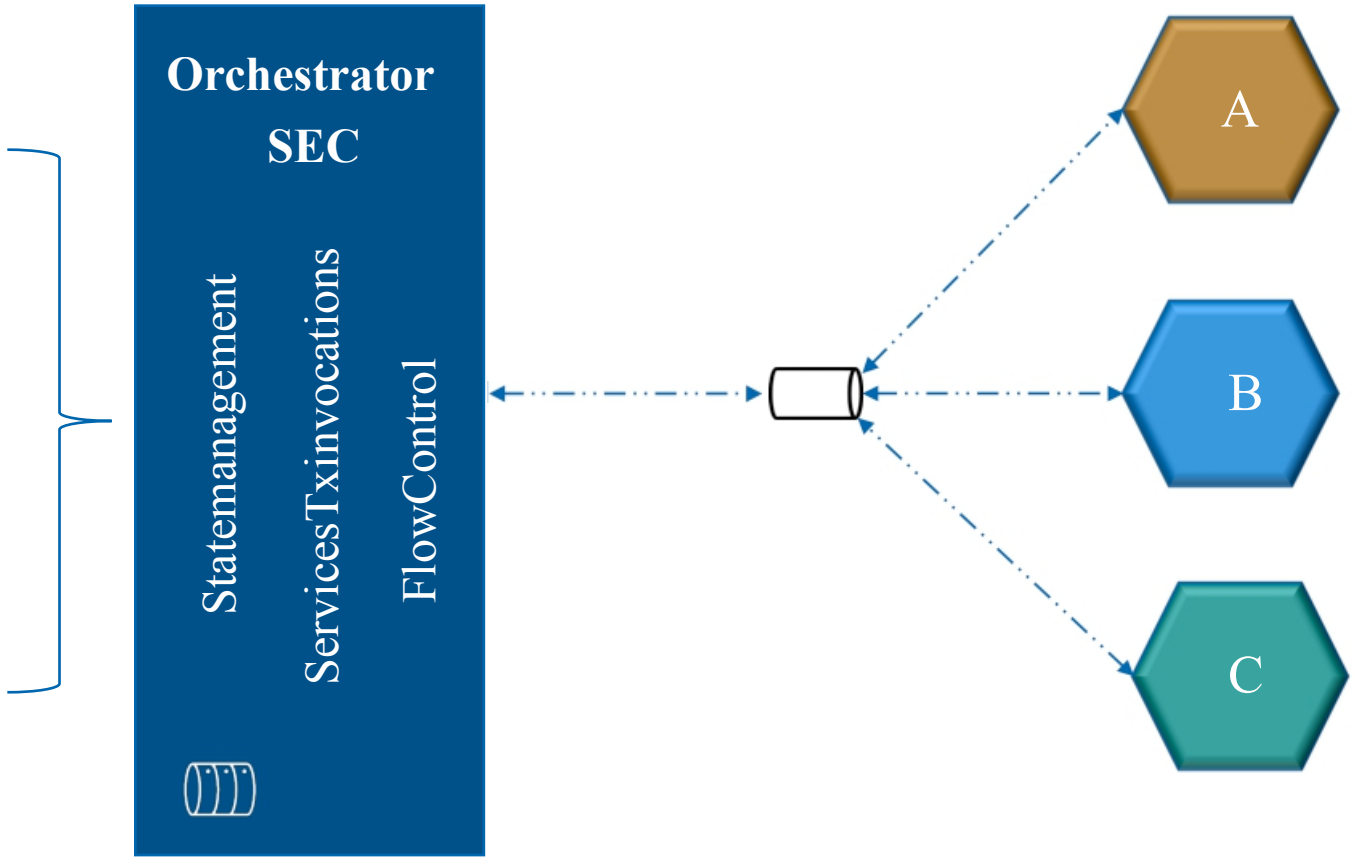
2. A dedicated SEC outside of the domain model



SAGA Command/Orchestrator

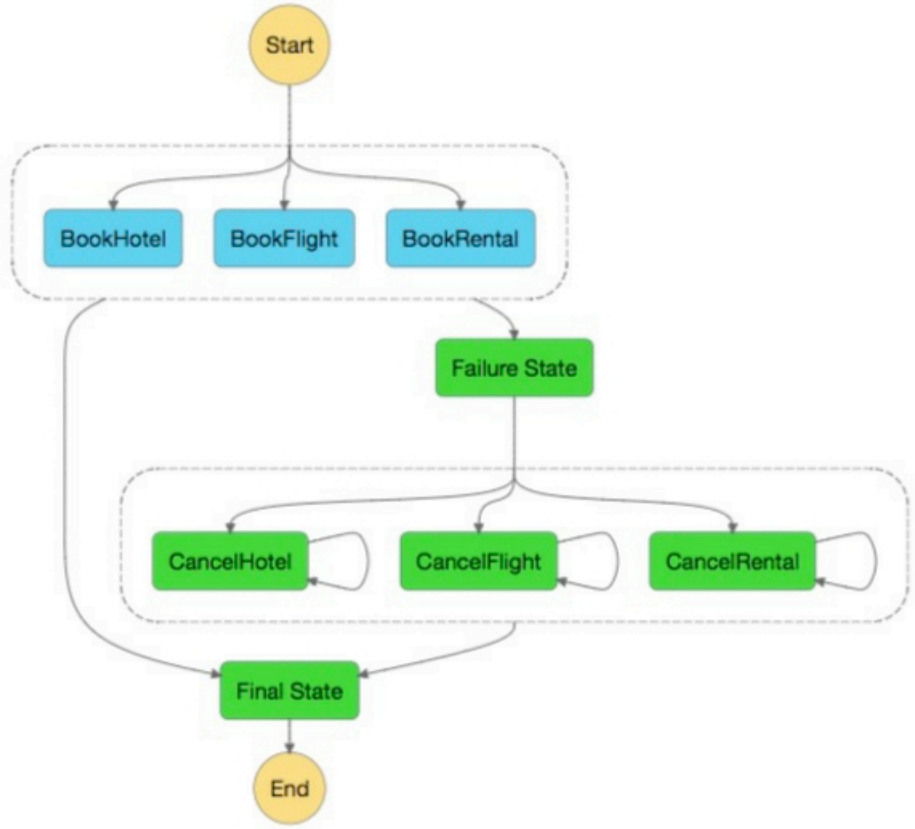
2. A dedicated SEC outside of the domain model

- E.g., use of BPM tool
- E.g., AWS step function
- E.g., Spring framework



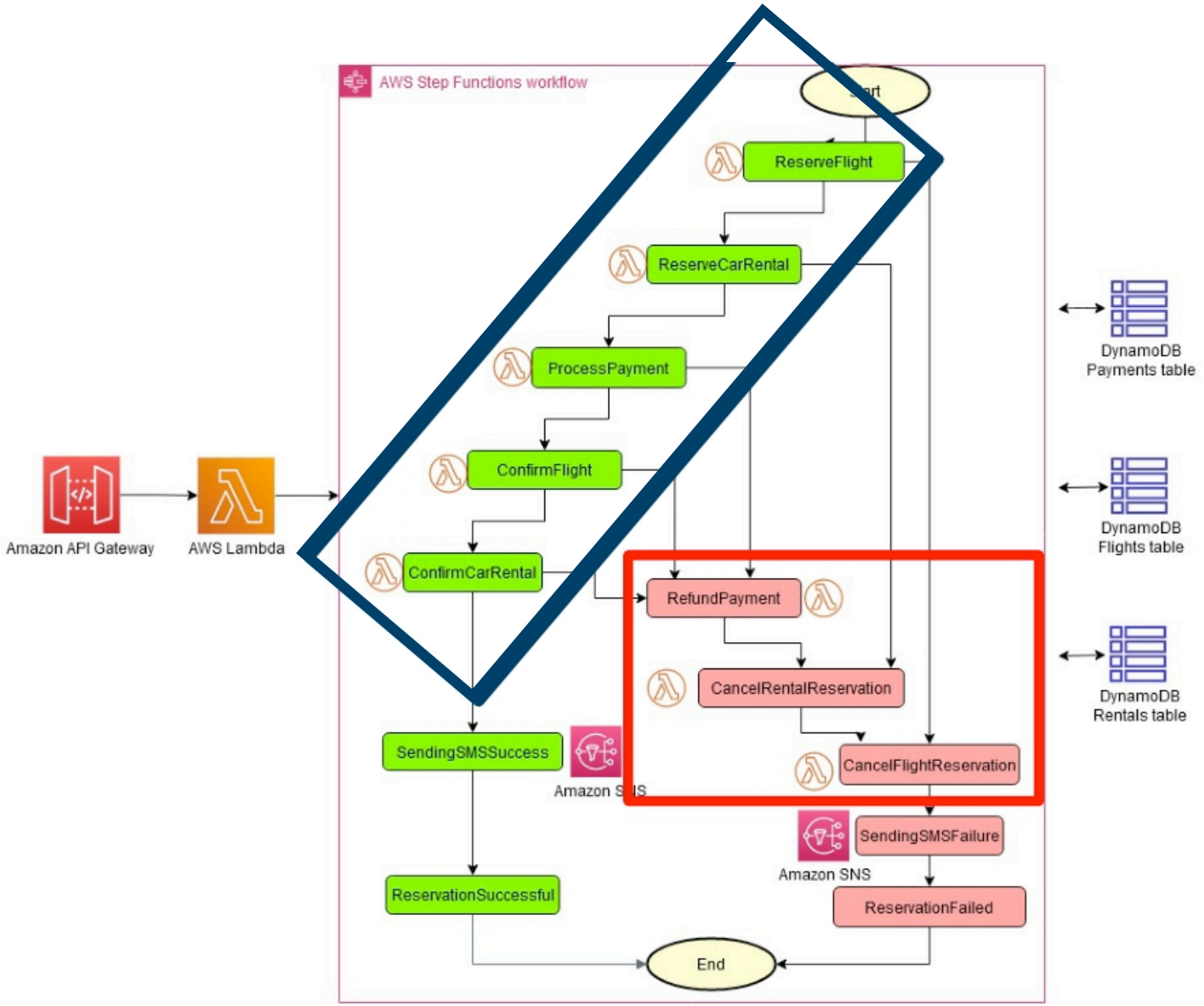
Example SEC: AWS Steps functions

Define the business process



- Declarative State machine in JSON format
- Business logic "AWS Lambda functions"
- "Step Function" coordinates the execution

Example: AWS Steps function workflow



Benefits : Command/Orchestrator

Less decoupled compared & introduces Single point of failure BUT

- Simplicity
- Easier to implement, test and manage
- Rollbacks are easier to manage
- Centralized way to check out the state

Quick Review

SAGA

- Event/Choreography = NO central coordinator
- Command/Orchestrator= SAGA Execution Coordinator
  - External component
  - Domain object

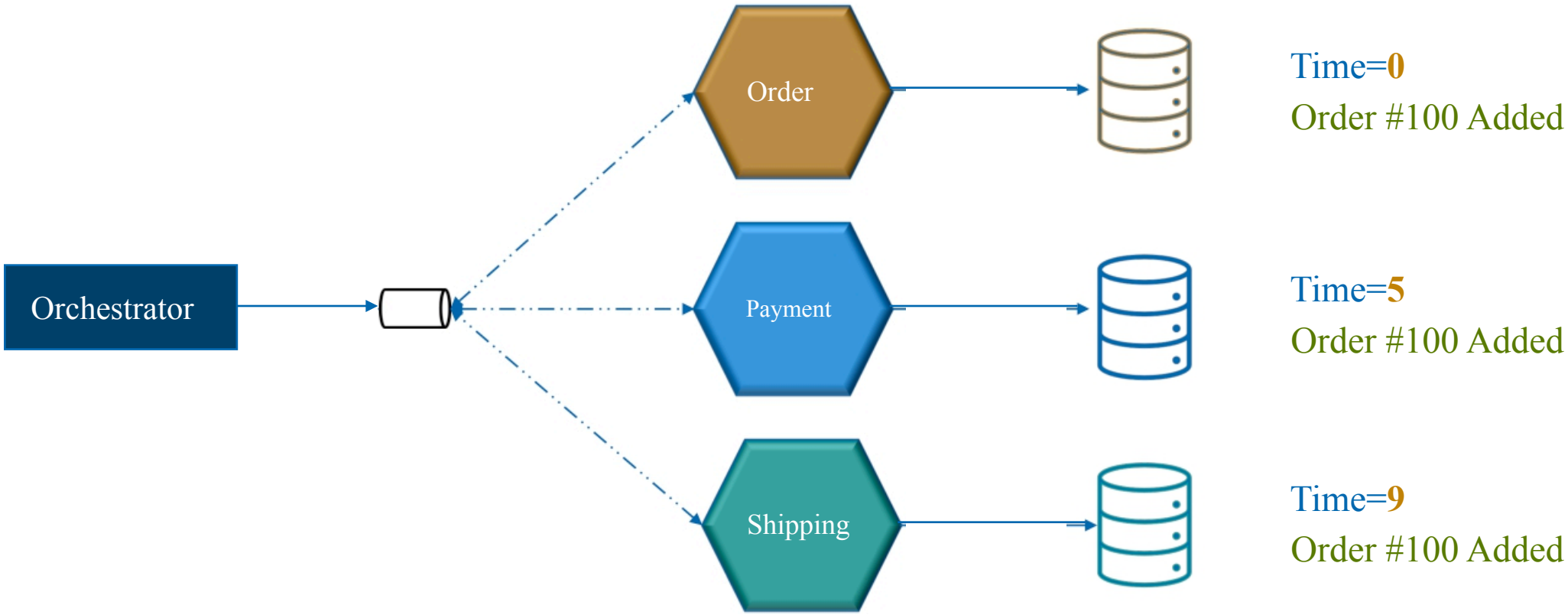
SAGA Design Considerations

Managing data consistency in Microservices

1. Data consistency challenge
2. Pros of shared database
3. Cons of shared database

Eventually consistent

Microservices may reflect different states while the Tx is in progress



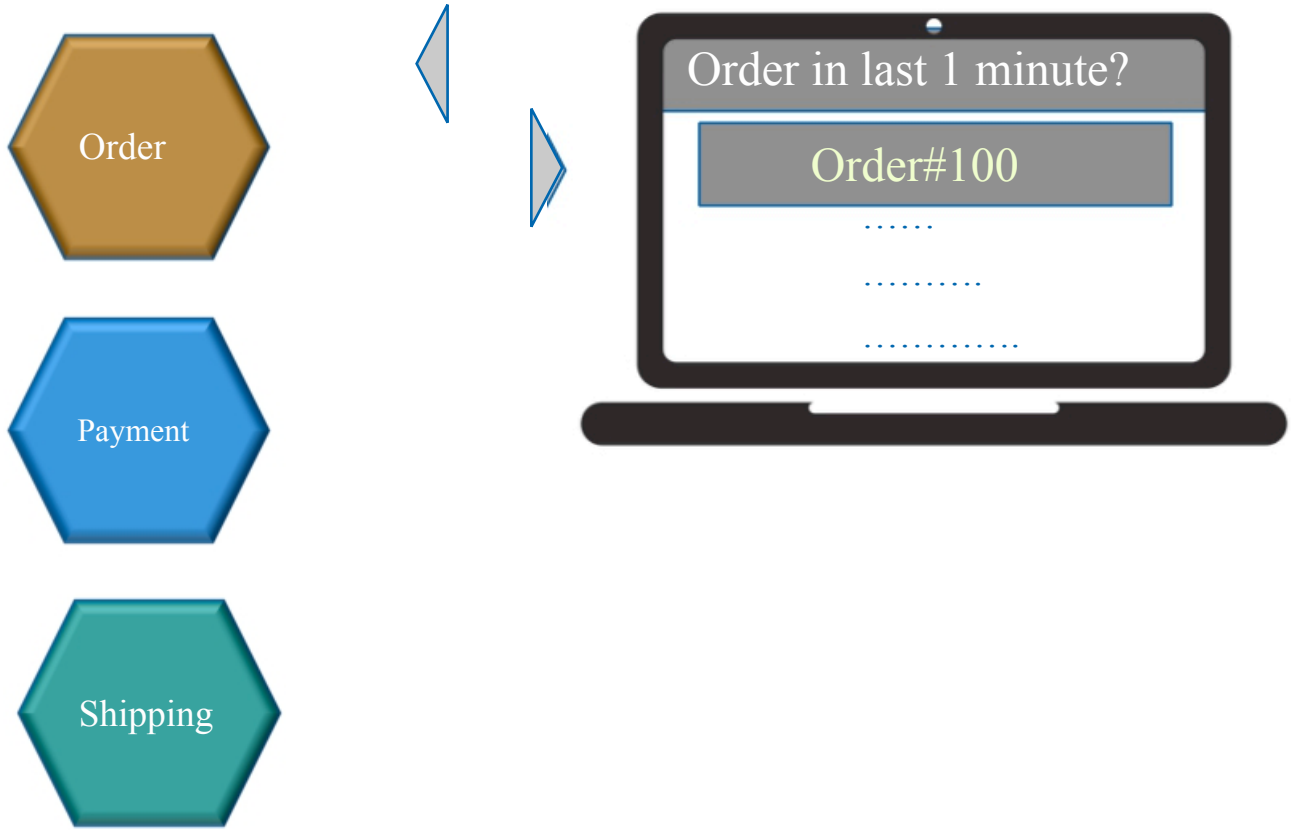
Isolation Level

SAGA provides isolation level = "*READ Uncommitted*"

Time=0
Order #100 Added

Time=1
Order #100 processing

Time=..
pending



Isolation Level

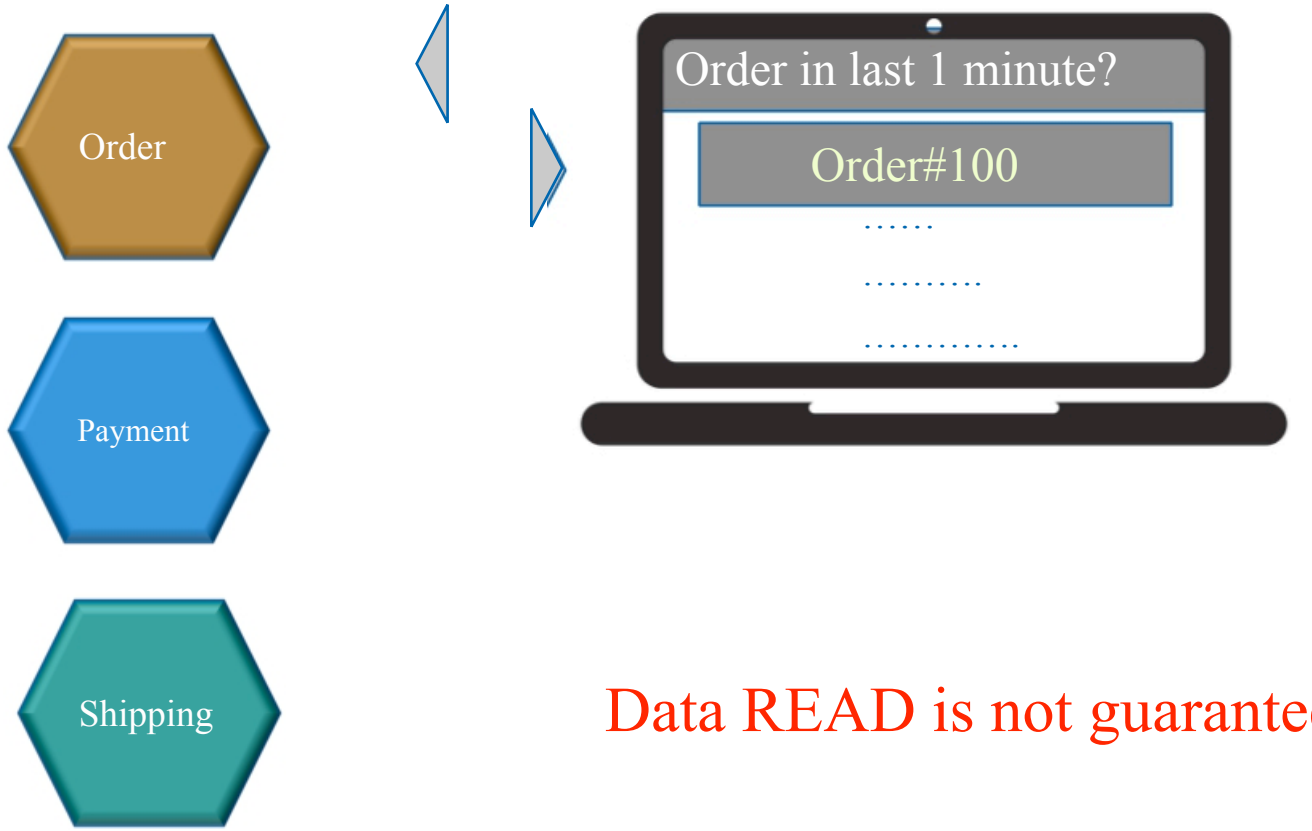
SAGA provides isolation level = "*READ Uncommitted*"

Time=0
Rollback Order#100

Time=0
Order #100 Added

Time=12
Order #100 processingfailed

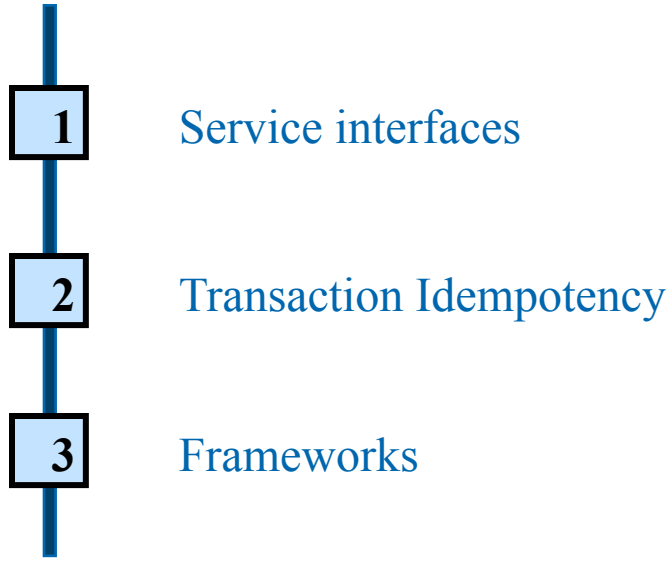
Time=..
pending



Data READ is not guaranteed !!

Implementation considerations

Distributed SAGA implementation approach



Frameworks

Consider using Frameworks

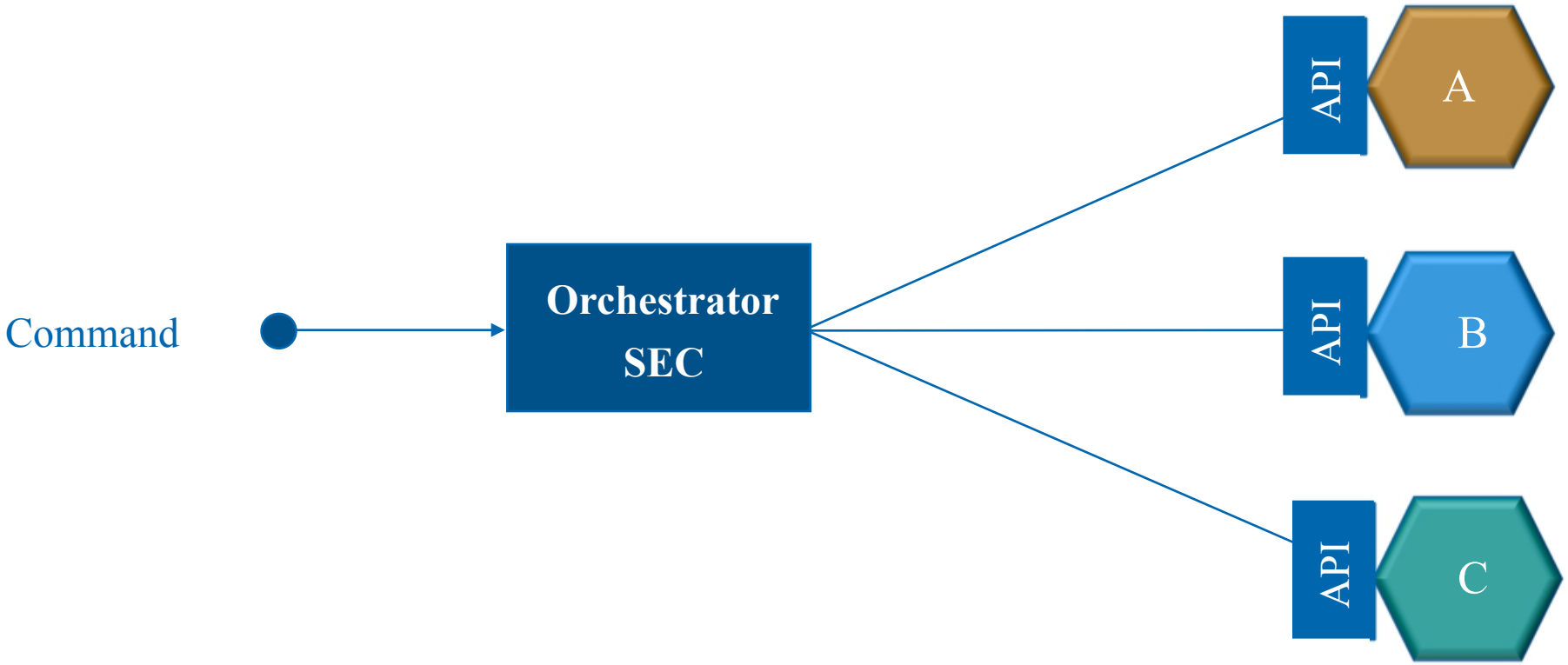
https://eventuate.io/

https://axoniq.io/

http://seata.io

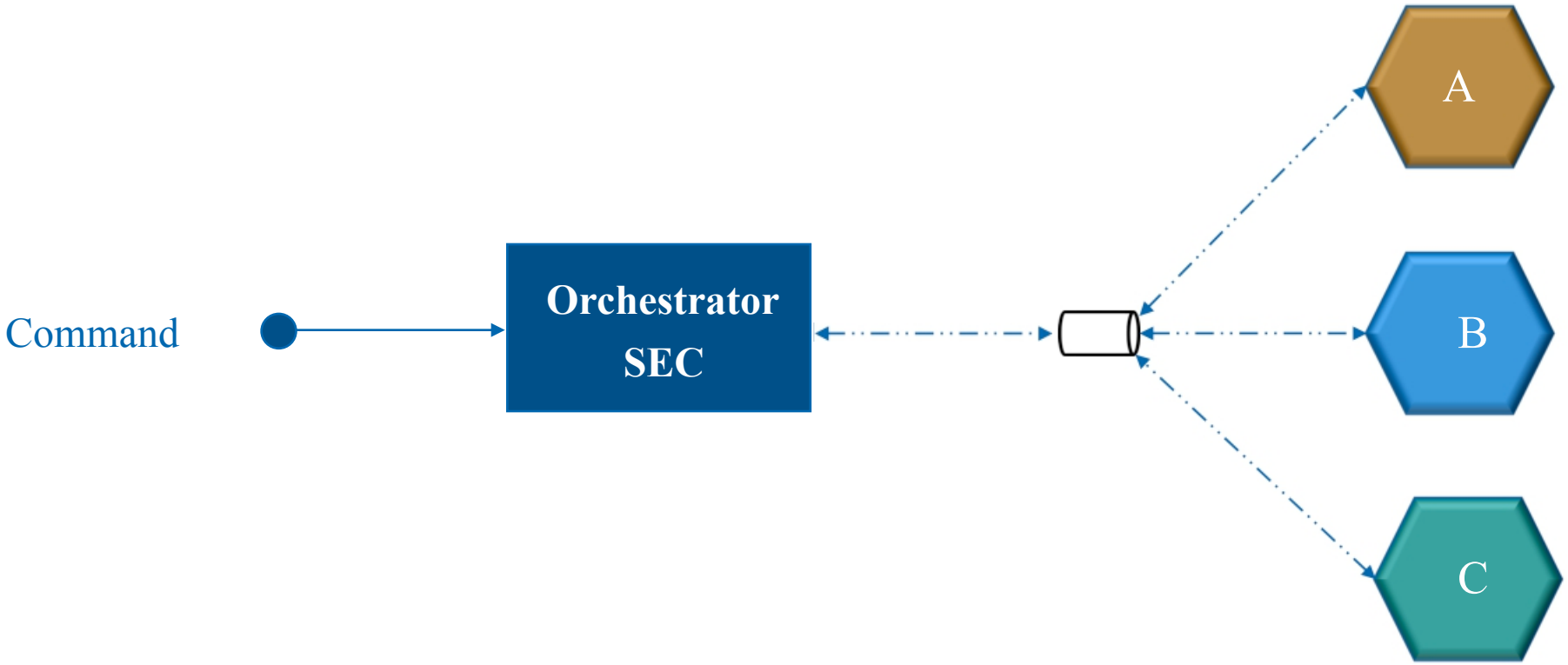
Synchronous Vs Asynchronous calls

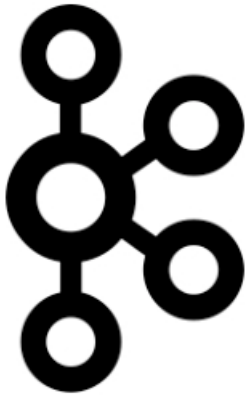
Synchronous calls may be used but messaging is preferred



Synchronous Vs Asynchronous calls

Synchronous calls may be used but messaging is preferred





OR

What would you use for implementing SAGA?

Unique ID per transaction

Each transaction | event has a UNIQUE identity

- Helps with Event Sourcing
- Consumers can identify duplicate transactions

Idempotent

Service transactions MUST be idempotent

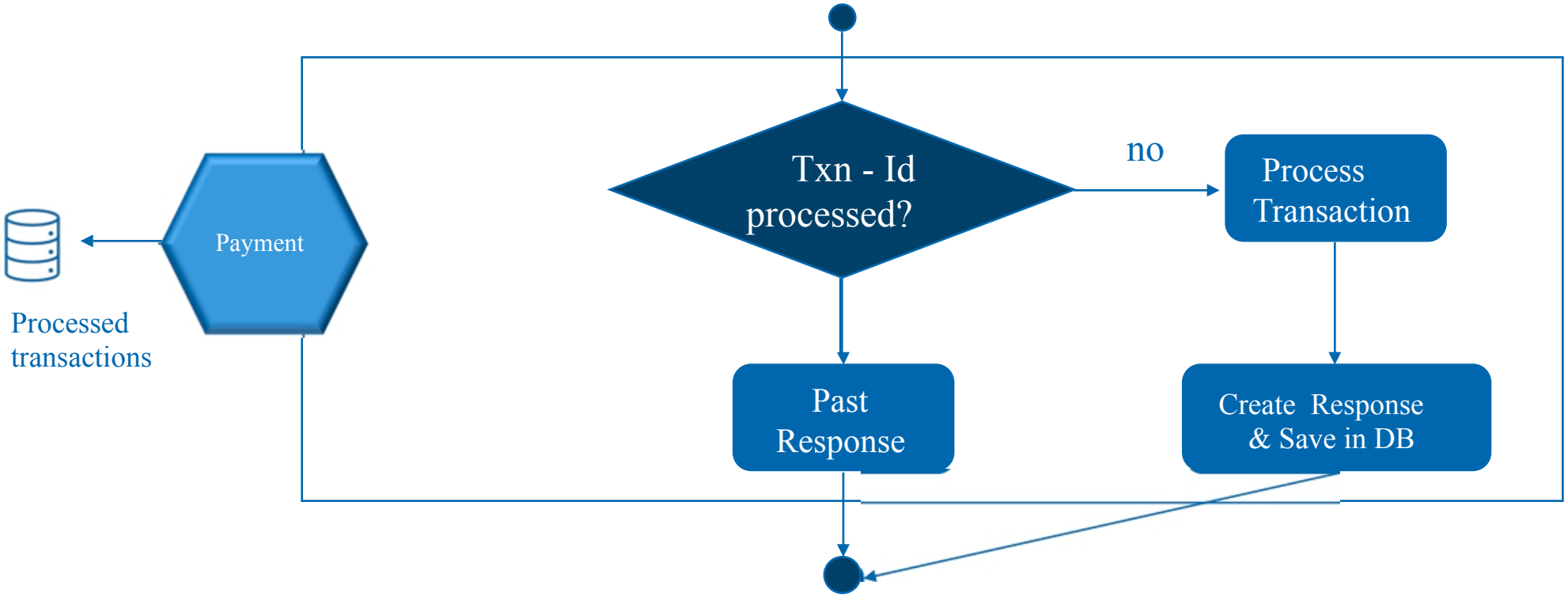
Idempotence is the property of certain operations in mathematics and computer science whereby they can be applied multiple times without changing the result beyond the initial application. — Wikipedia

---

Idempotent

Services transactions MUST be idempotent

- Using a Unique ID for each message can help



Service Failures

Request failure OK if followed by compensating transactions

- Compensating Transactions CANNOT fail

- Consider event sourcing and state management for services

Quick Review

Services

Asynchronous/Messaging is preferred

Each transactions have a unique identity

Operations are idempotent

Compensating transactions cannot fail

SAGA based booking

Simulation to demonstrate working of SAGA

1. Booking SAGA Flow
2. Success & Failure scenarios
3. Booking SAGA flow in action

IT Lead

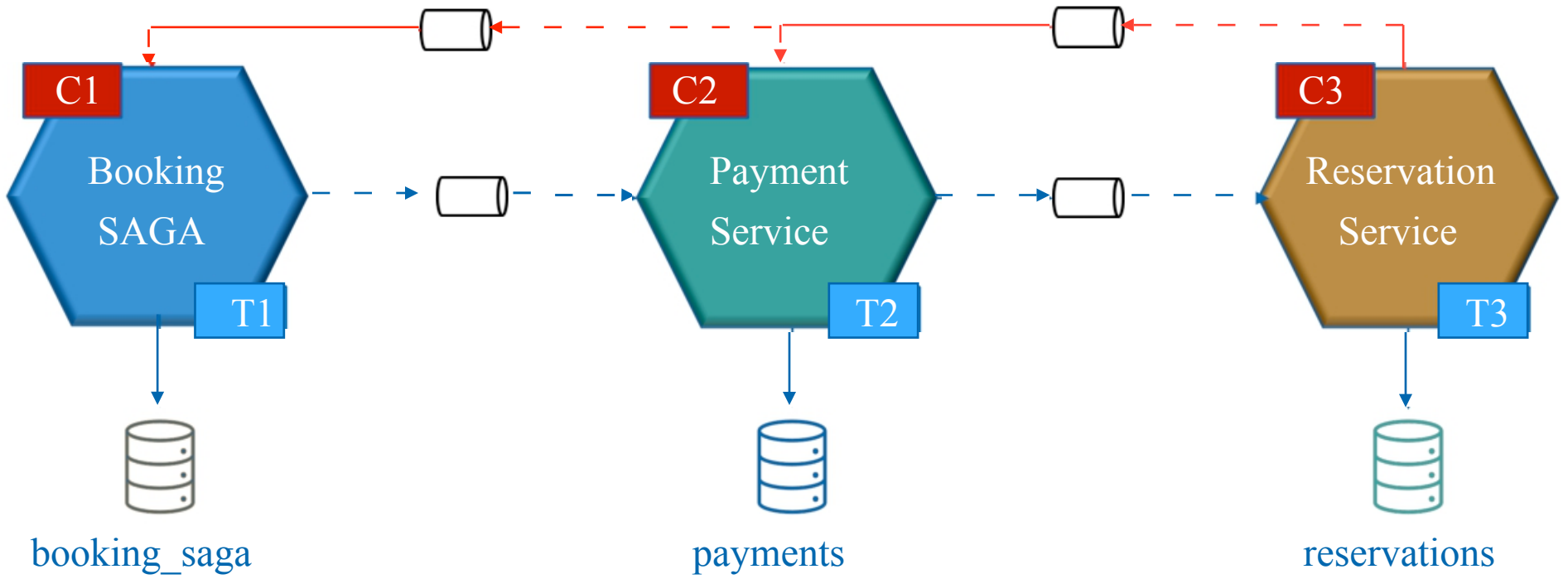
Design the booking process as a SAGA

SAGA pattern is difficult to implement without a framework

We will build version 1 of Booking SAGA without framework

This will help us understand the flow of SAGA

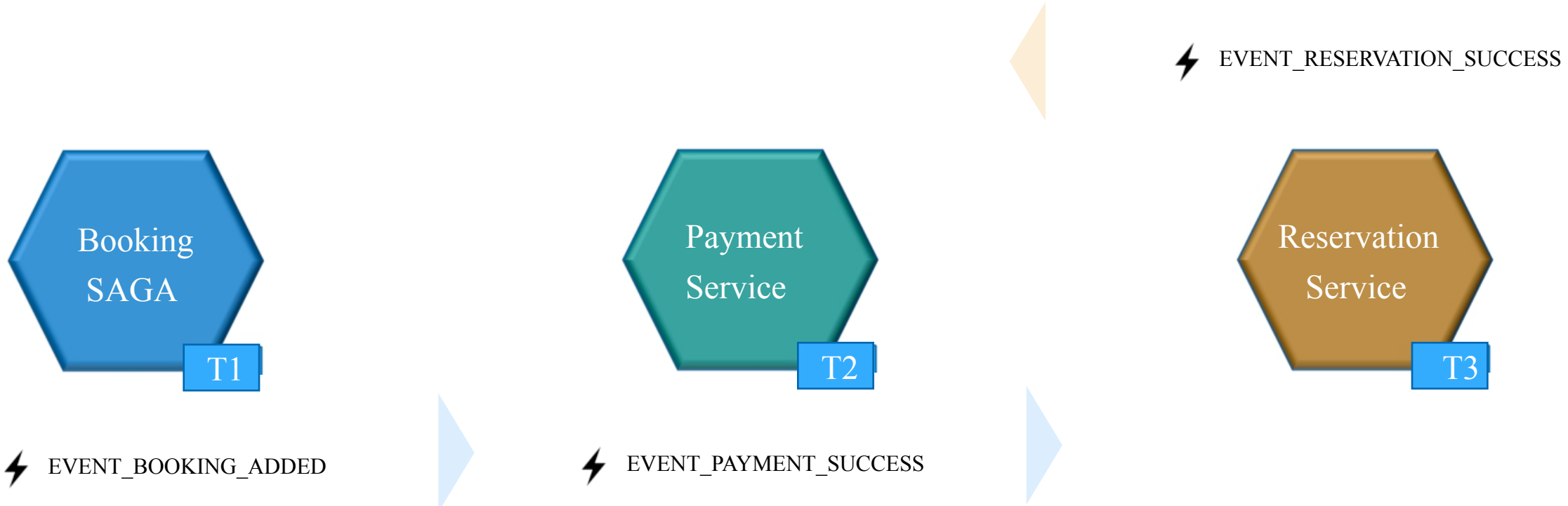
Booking Flow as a SAGA



T1	add_booking	T2	process_payment	T3	process_reservations
	mark_booking_failed	C2	refund_payment	C3	cancel_reservations

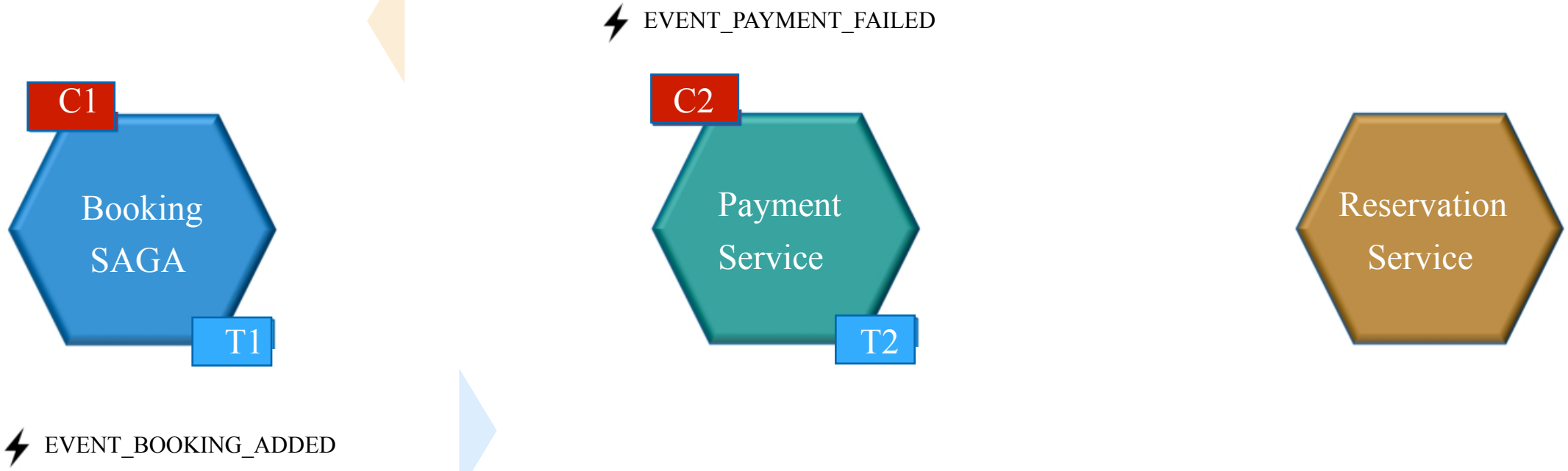
Events flow

Successful reservations scenario



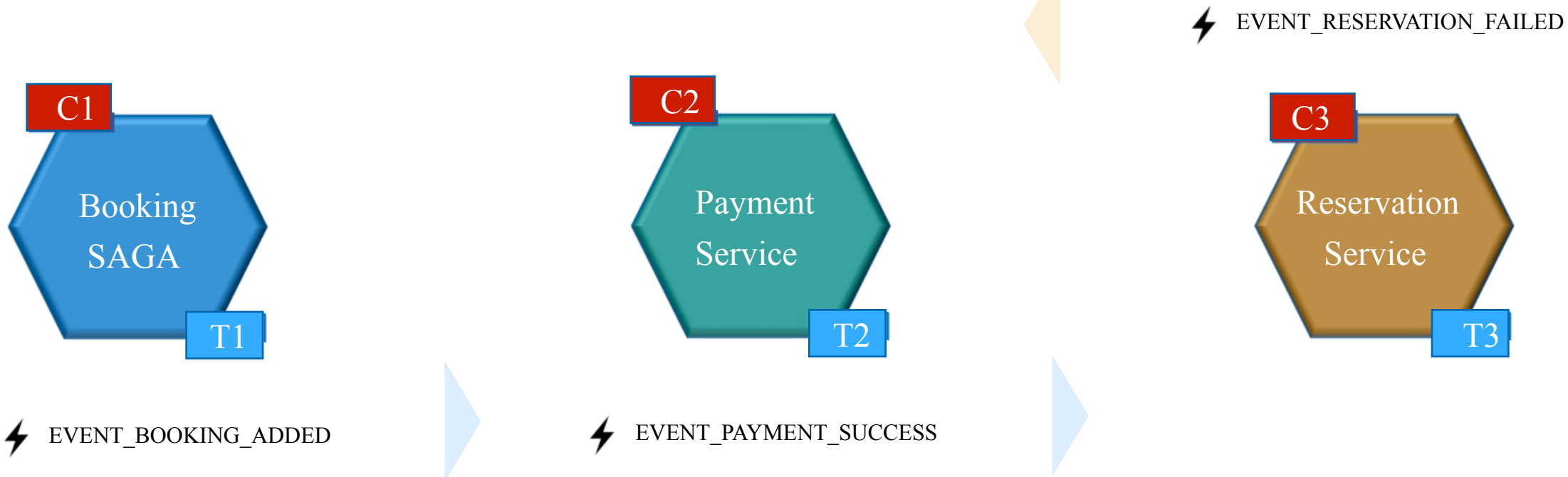
Events flow

Payment processing failure scenario



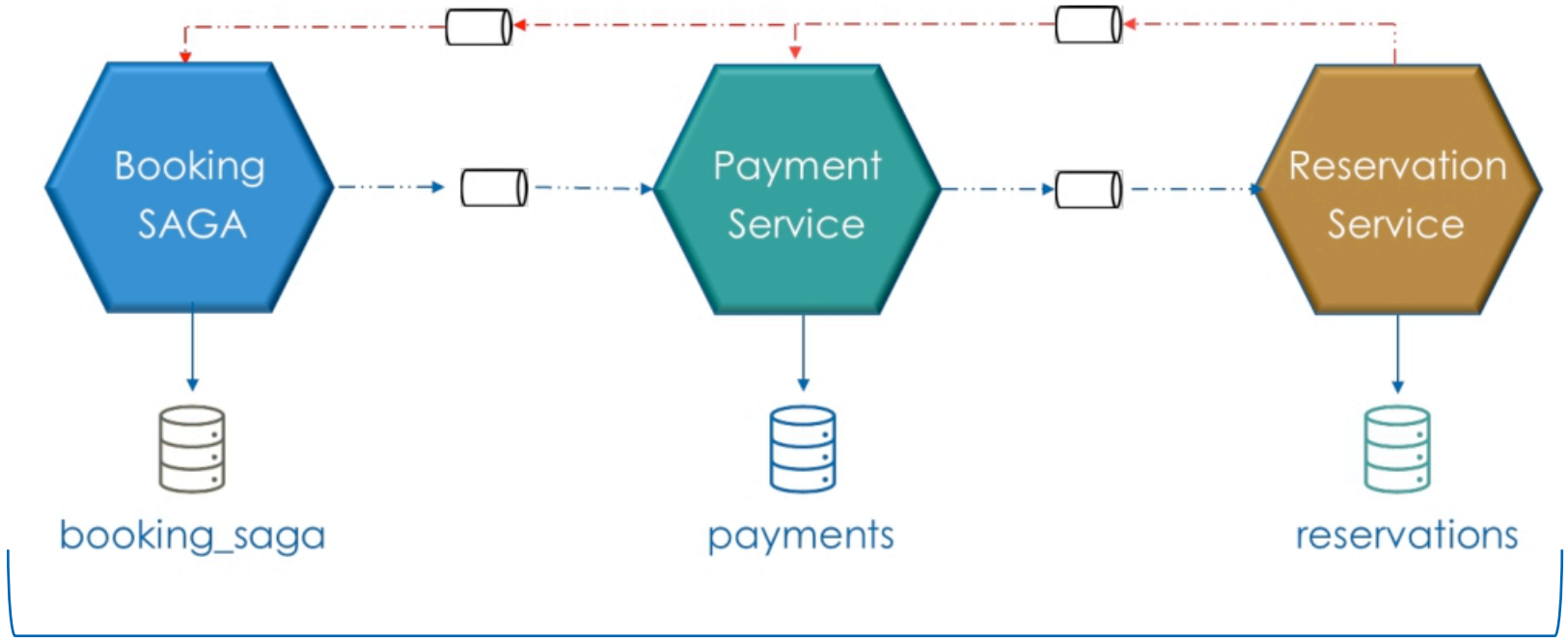
Events flow

Reservations failure scenario



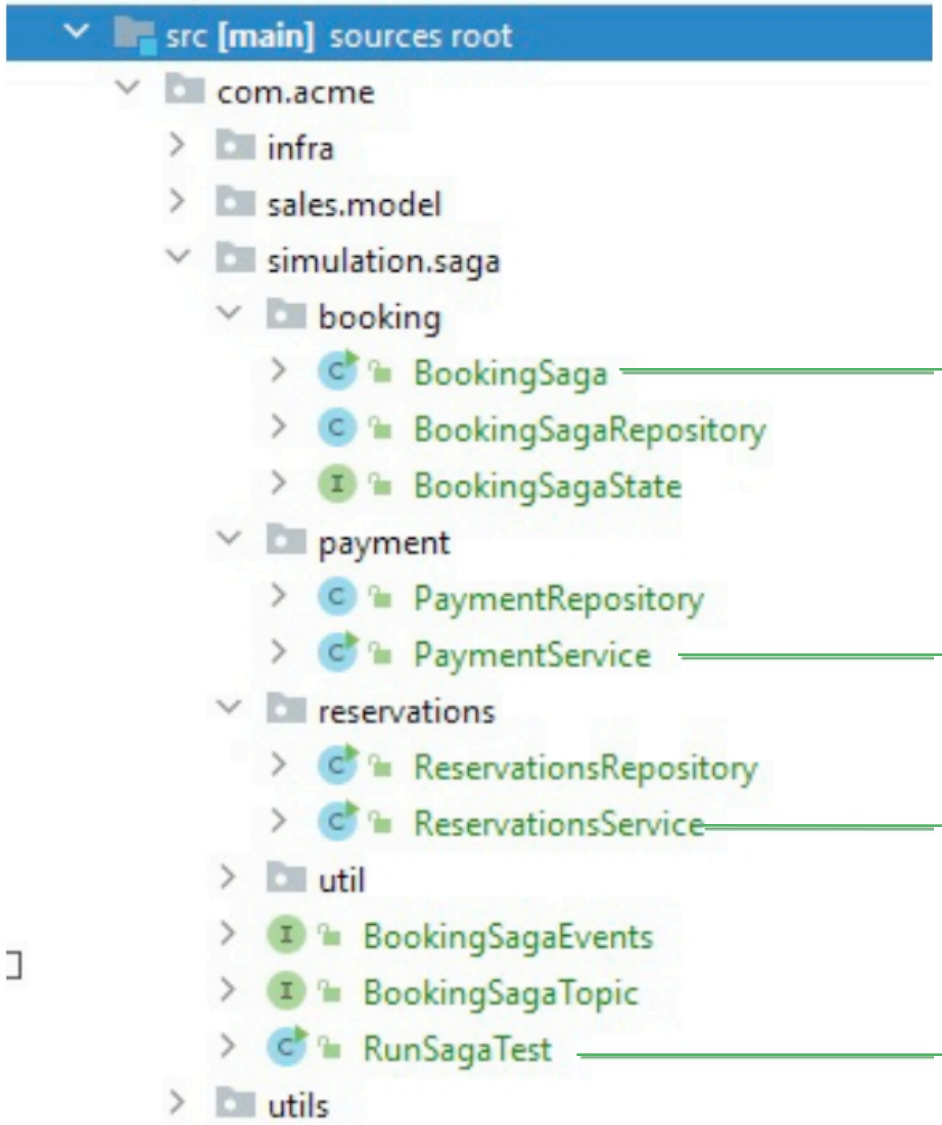
Implementation stack

Common topic on Kafka · Topic *[user name]* - bookingsaga · Key *booking_id*



mongoDB. As collections in a single database

Demo



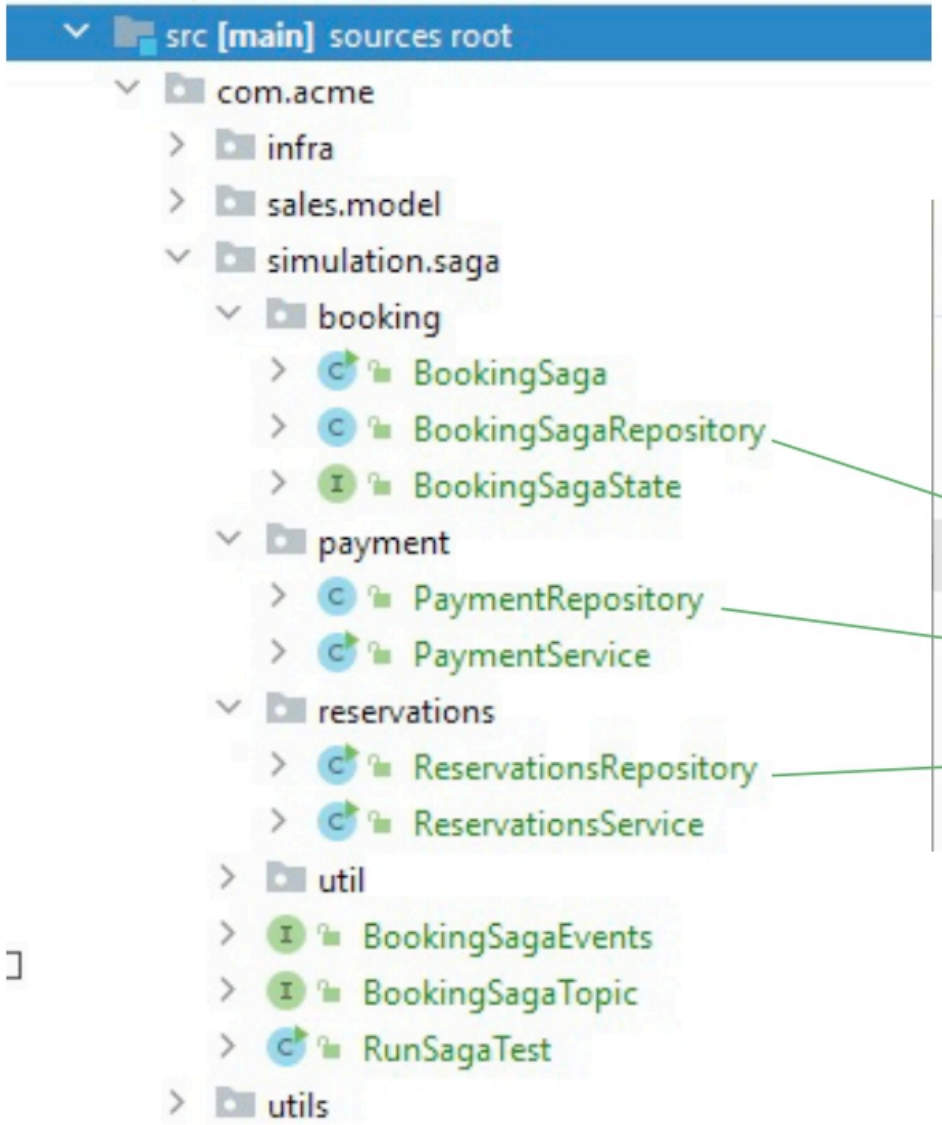
Manages state of booking SAGA

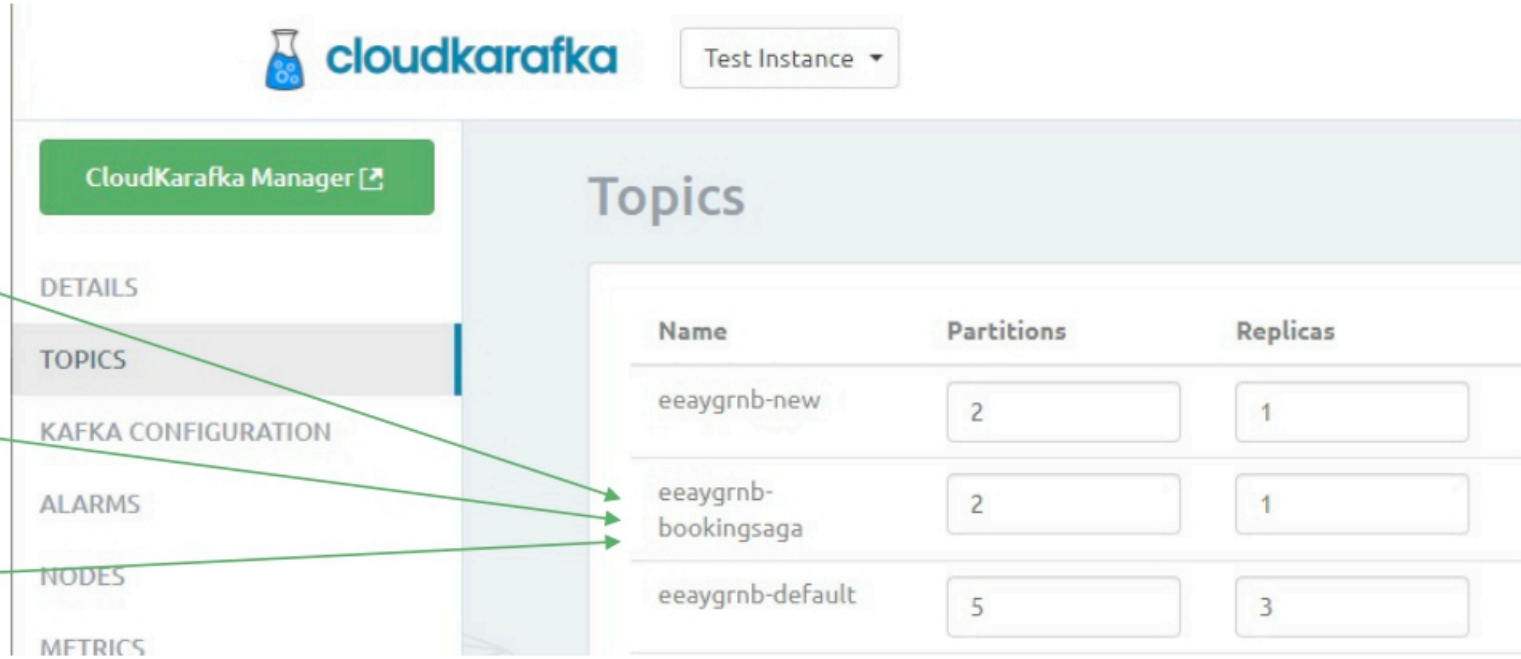
Simulates processing of payments

Simulates processing of reservations

Starts the Booking SAGA for booking_id =?

Demo





mongoDB. AcmeTravel database

Next steps

- Design | Code walkthrough of JAVA implementation

- Setup & try out the flow on your local machine

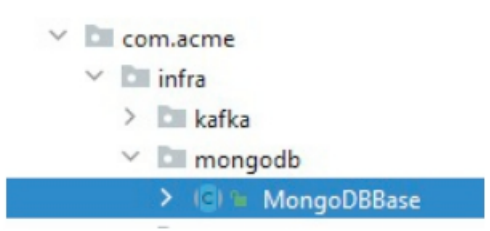
SAGA Implementation walkthrough

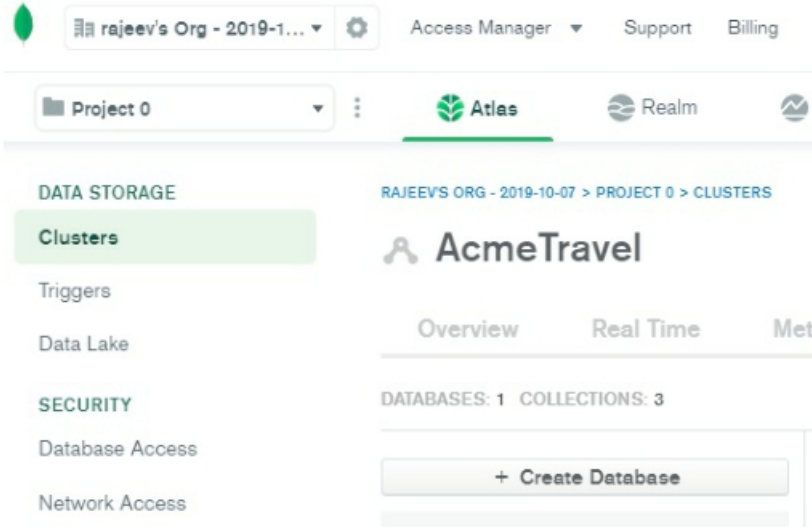
Distributed Booking SAGA implementation

1. Walkthrough of Class & State diagrams
2. Quick code walkthrough
3. SAGA testing steps

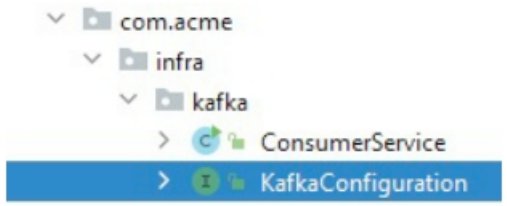
Pre - Requisites

- MongoDB setup is in place





- Cloud Karafka setup is in place

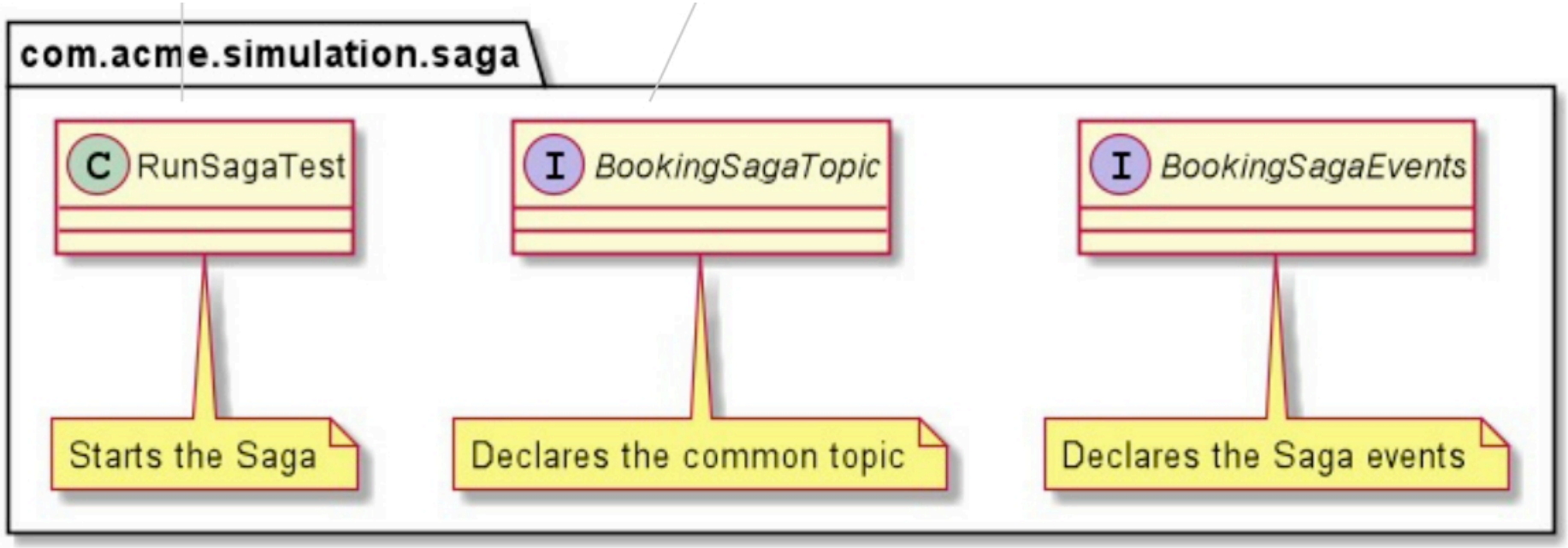


Class Diagram

/ uml /saga/ booking.test.classes.puml

- Execute this class for testing; MUST use different booking_id each time

- Set the topic name in this interface



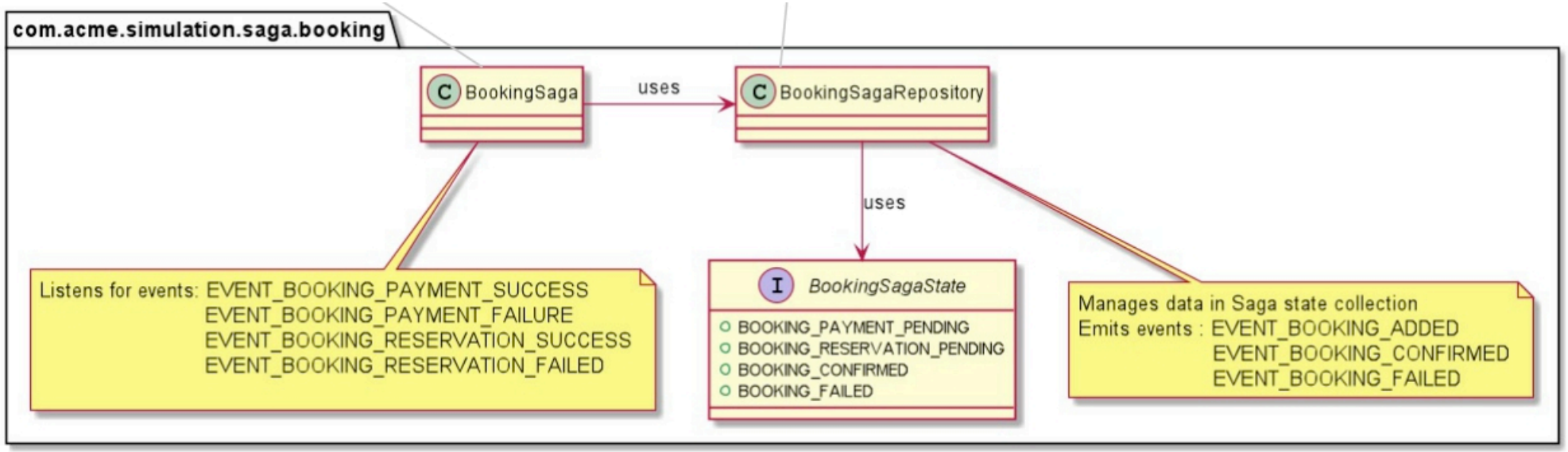
Class Diagram

/ uml /saga/ booking.simuation.classes.puml

- Handles the events

- Repository extends *MongoDBBase*

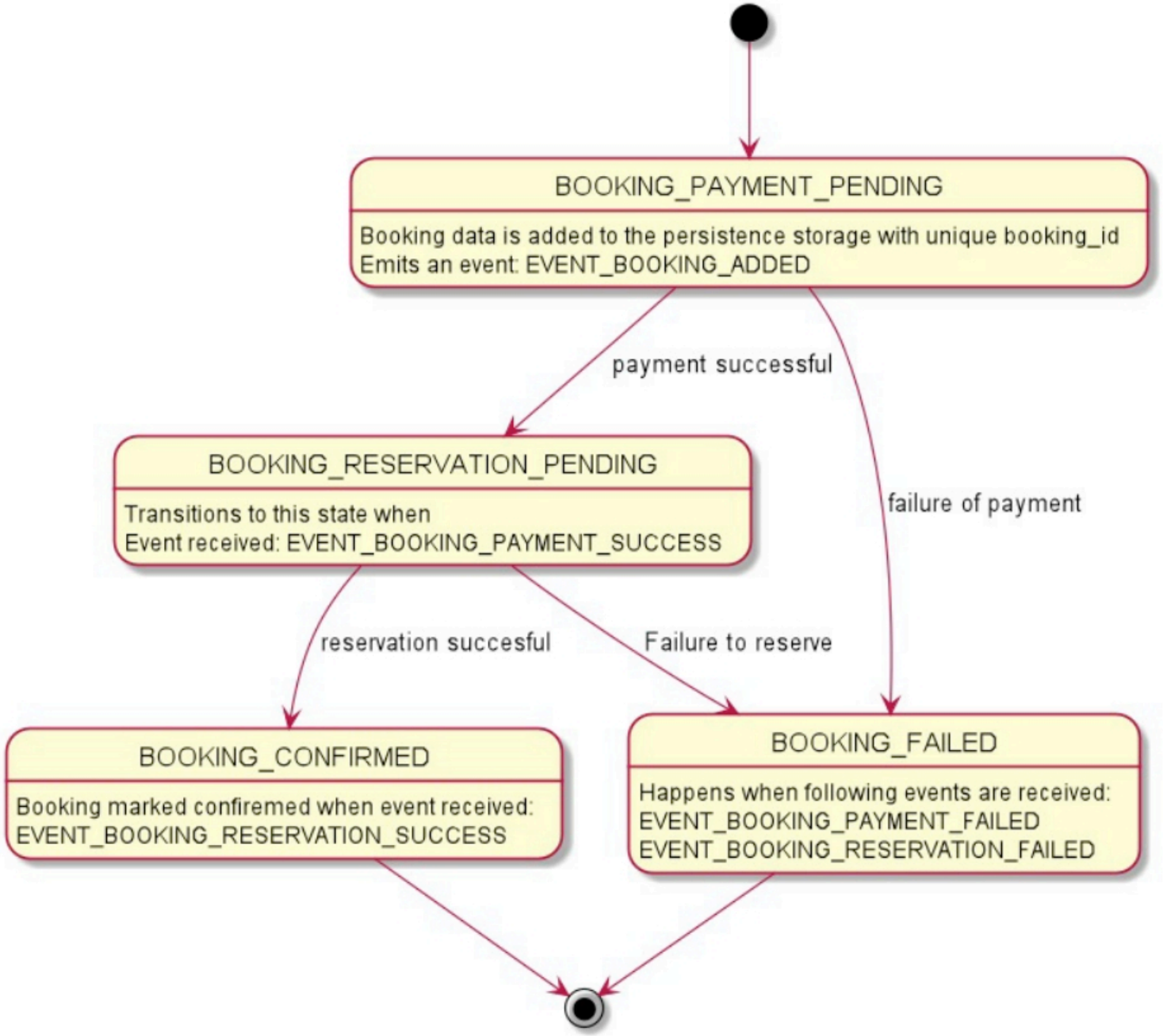
- Events are published from the Repository



State Diagram

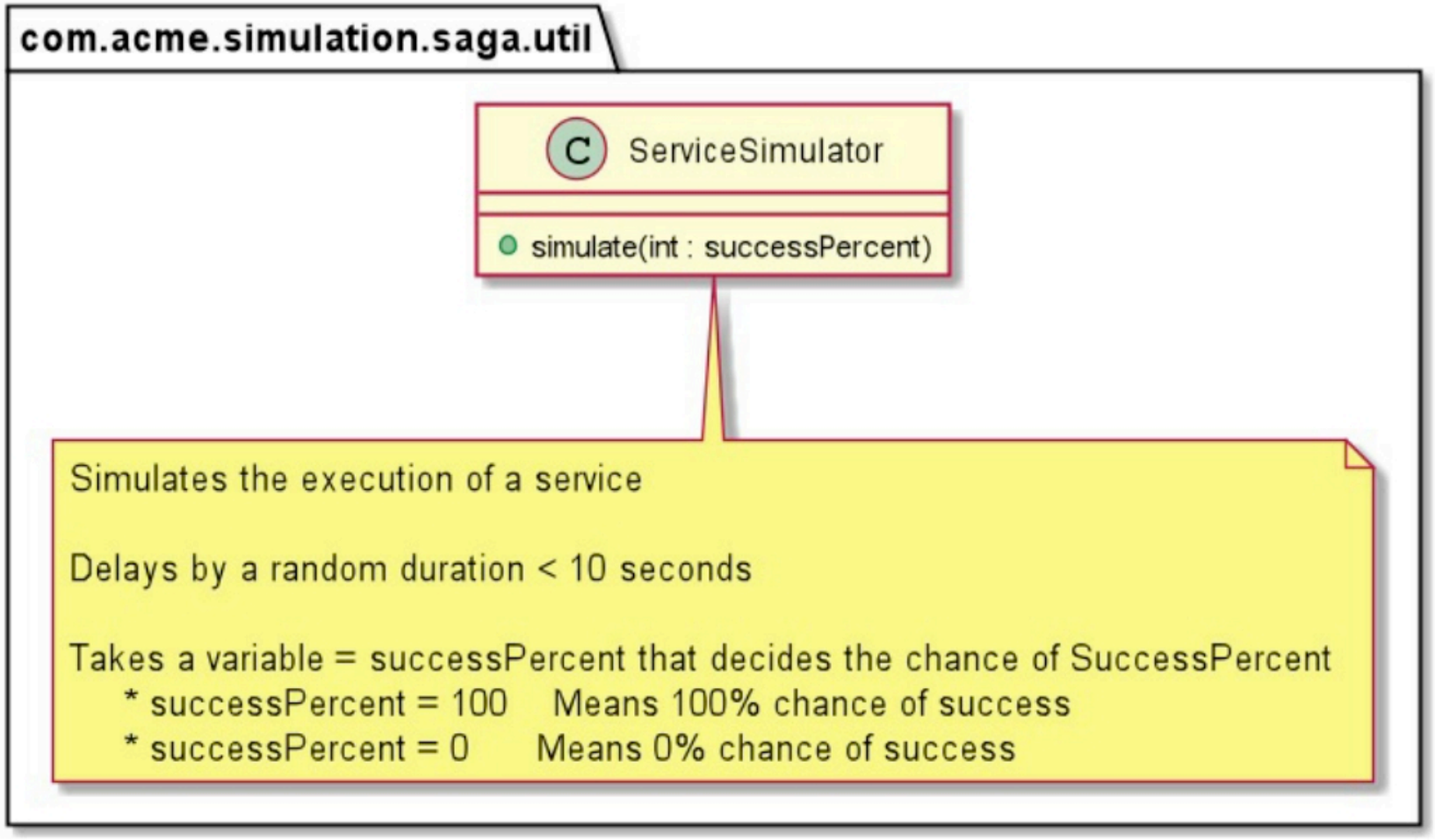
/ uml/saga/booking.saga.state.puml

SAGA Execution Coordinator



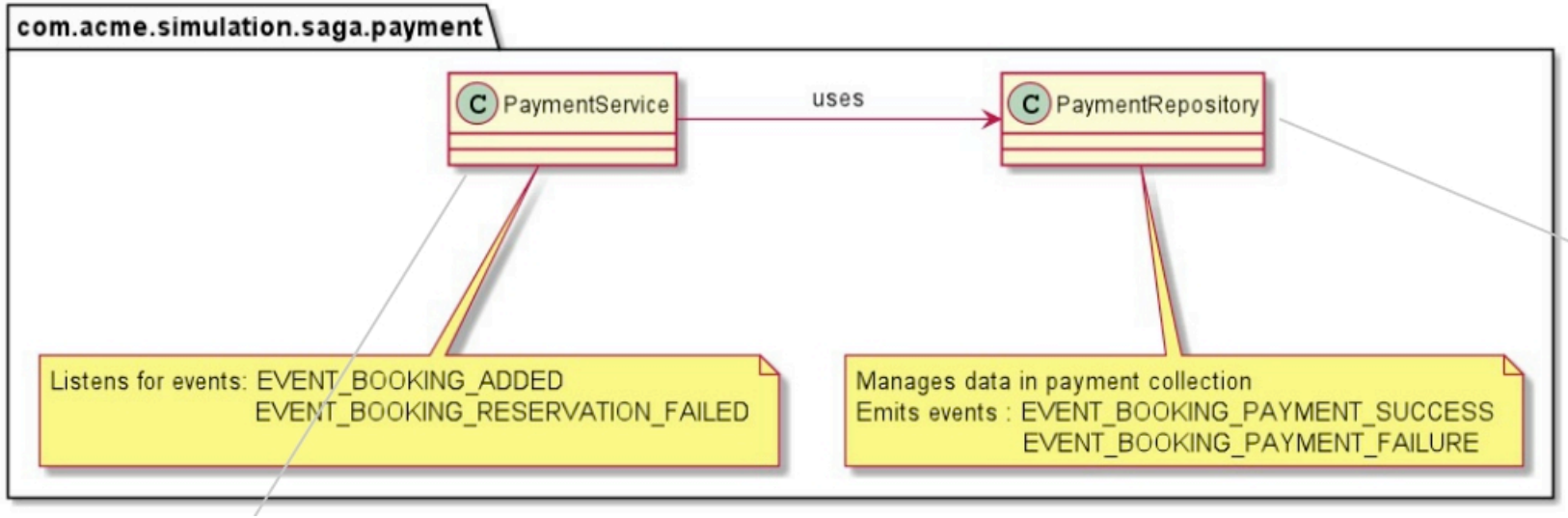
Service simulation clss

/ uml /saga/ simulator.class.puml



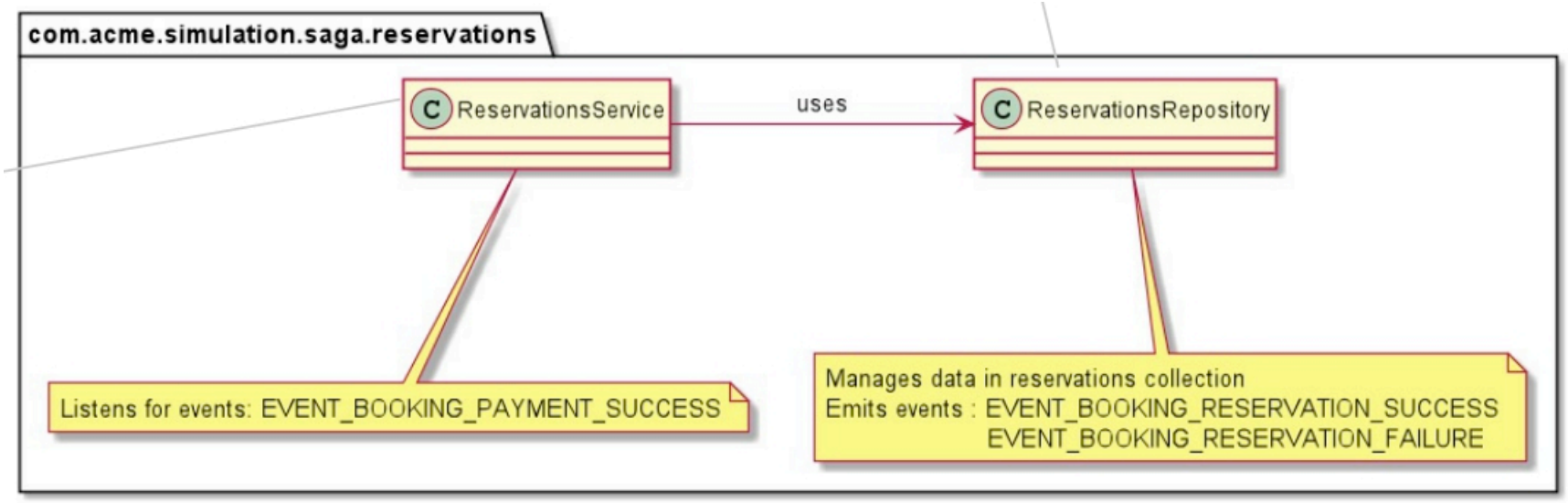
Class Diagram

/ uml /saga/ booking.simuation.classes.puml



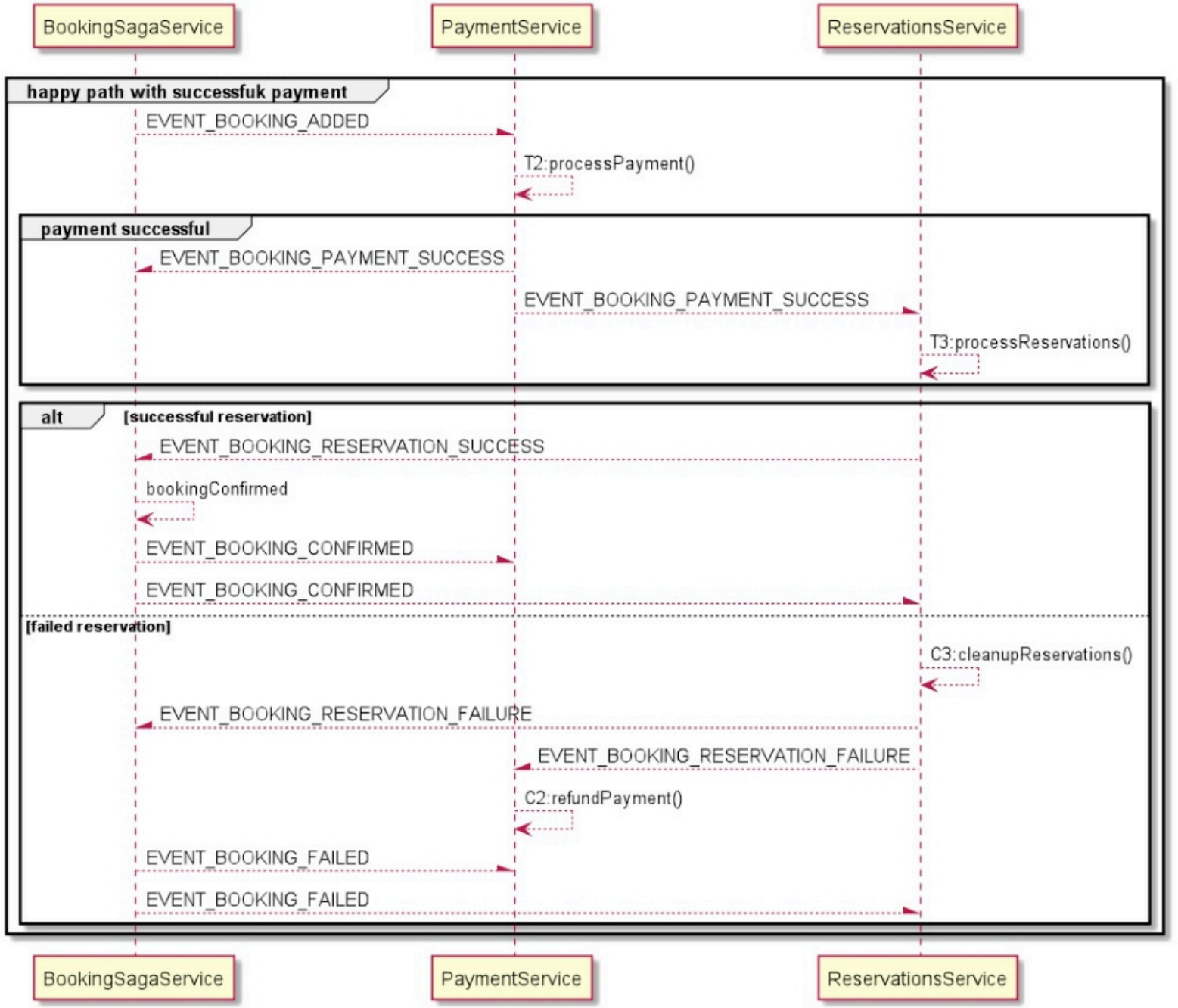
- Extends *MongoDbBase*

- Handles the events



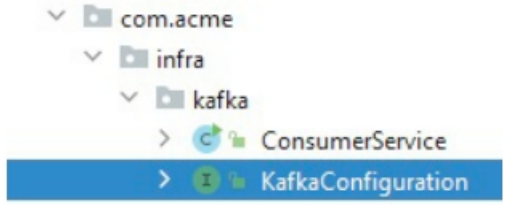
Sequence

/ uml /saga/ booking.event.sequence.puml

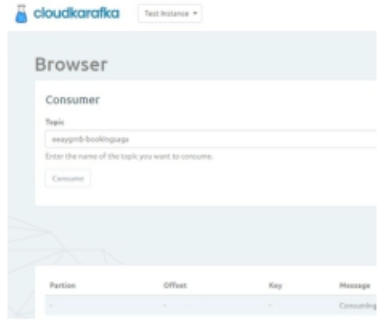


Test Steps

1. Setup kafka topic : booksaga



2. Use Kafka consumer to observe messages



3. Launch the BookingSaga, PaymentService & ReservationsService

4. Execute RunSagaTest with different booking_id