**Modelling Saga as a State Machine : An orchestrator driven approach for managing distributed and long-running transactions**

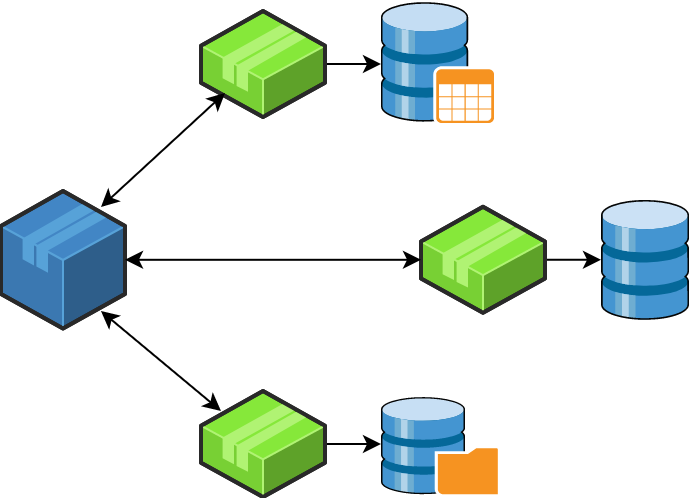
[](https://rohsin47.medium.com/?source=post_page-----cec381acc3ef--------------------------------)

[Rohit Singh](https://rohsin47.medium.com/?source=post_page-----cec381acc3ef--------------------------------)

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fec06d6c075f9%2Fcec381acc3ef&operation=register&redirect=https%3A%2F%2Flevelup.gitconnected.com%2Fmodelling-saga-as-a-state-machine-cec381acc3ef&user=Rohit%20Singh&userId=ec06d6c075f9&source=post_page-ec06d6c075f9----cec381acc3ef---------------------follow_byline-----------)

[Jul 20](https://levelup.gitconnected.com/modelling-saga-as-a-state-machine-cec381acc3ef?source=post_page-----cec381acc3ef--------------------------------) · 11 min read

*This post describes the architecture and conceptual framework to manage distributed and long-running transactions in the microservices ecosystem. The article focus on the challenges of distributed transactions and how to solve the problem in an effective way.*



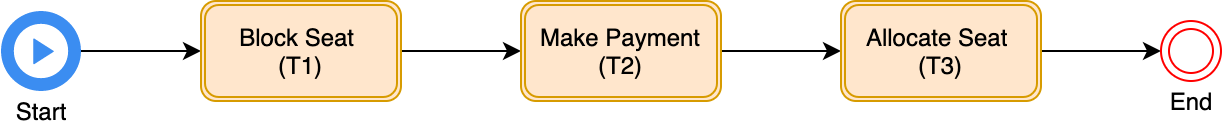
Overview

A microservice, in its true context, is a distributed system. A transaction is distributed to multiple services that are called sequentially or in parallel to complete the entire transaction. In a microservices architecture, transactions that are within a single service use ACID transactions to provide data consistency. The challenge, however, lies in handling a transaction that spans across multiple services, and in some cases needs a long period of time to finish. In this case, the application must use an elaborate mechanism to manage transactions.

Scenario

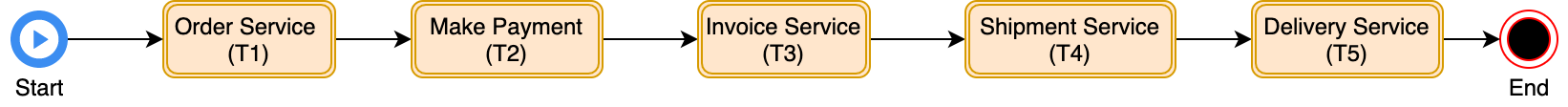
Consider a simple airline flight booking scenario implemented using a microservices architecture. There would be one microservice to block a seat, another to accept payments, and finally, another microservice to allocate the blocked seat, each implementing a local transaction. To successfully complete the flight booking process for a traveler, all three steps must be completed. If any of the steps fails, all of the completed preceding steps must roll back. Since the overall transaction boundary crosses multiple services and databases, it is considered to be a distributed transaction.





Consider another scenario of order fulfillment which is implemented through a microservices approach. The workflow transaction starts at the order service by creating an order first, moving to the next service for making payment, then creating an invoice for the transaction, after that sends for shipping, and finally order delivery and completing the workflow, again each implementing local transactions. The order processing here is distributed in nature and it might take days to weeks to complete the workflow. Such a transaction can be termed as a long-running transaction as all steps cannot be executed in one go and using traditional ACID transaction semantics.





Challenges

With the advent of microservices architecture, there are two key problems with respect to distributed transaction management:

* Atomicity : Atomicity implies that all of the steps in the transaction must be successful or if a step fails, then all of the previously completed steps should be rolled back. However, in a microservices architecture, a transaction can consist of multiple local transactions handled by different microservices. Therefore, if one of the local transactions fails, how to roll back the successful transactions that were previously completed?
* Isolation : The transaction isolation level specifies the amount of data that is visible to a statement in a transaction, specifically when the same data source is accessed by multiple service calls simultaneously. If an object from any one of the microservices is persisted to the database while another request reads the same object at the same time, should the service return the old data or new?

In order to address these problems and provide an effective transaction management capability, two approaches can be taken:

* Two-phase commit (2PC)
* Saga

2PC

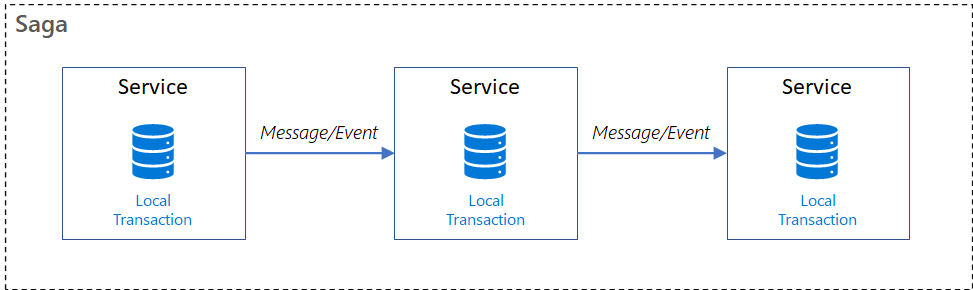
The traditional approach to maintain data consistency across multiple services is to use distributed transactions. The de facto standard for this is 2PC (2 phase commit). 2PC ensures all participants in a transaction are either commit or rollback. It works in two phases, phase 1 is called the prepare phase, where the controlling node asks all of the participating nodes if they are ready to commit, and phase 2, called the commit phase, where if all of the nodes replied in the affirmative, then the controlling node asks them to commit, or else rollback. Even though 2PC can help provide transaction management in a distributed system, it also becomes the single point of failure as the onus of a transaction falls onto the coordinator, and typical implementations of such a coordinator are synchronous in nature, which can lead to a reduced throughput in the future. Thus, 2PC still has the following shortcomings:

* Modern NoSQL databases like MongoDB and Cassandra don’t support them.
* Modern message brokers like Apache Kafka don’t support them.
* Synchronous IPC reduces availability.
* All the participants must be available.

Saga

To solve the more complex problem of maintaining data consistency in a microservices architecture, an application must use a different mechanism that builds on the concept of loosely coupled, asynchronous services. This is where sagas come in. Saga is an architectural pattern that provides an elegant approach to implement a transaction that spans multiple services, is asynchronous and reactive in nature. So, a saga can be defined as an event-driven sequence of local transactions, where each local transaction updates the database and publishes a command or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions. The saga implementation ensures that all transactions are executed or all changes are undone, and thus provides an atomicity guarantee. Designing a saga as a state machine model will provide countermeasures to handle isolation.

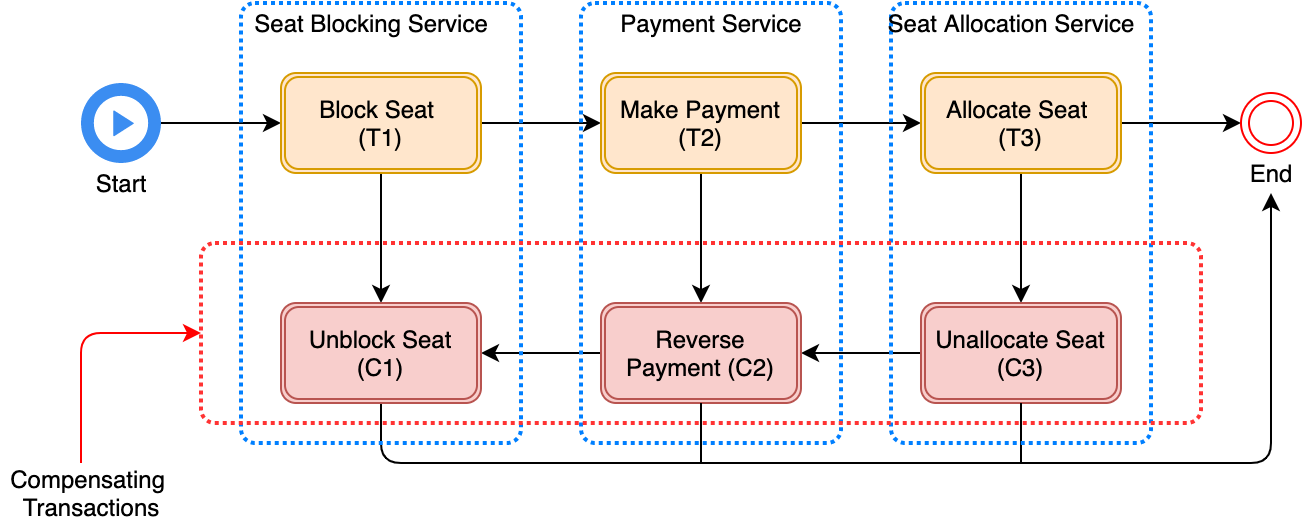




How Saga pattern helps

With a microservices architecture, a single business process brings multiple microservices together to provide an overall solution. It is very difficult to implement **ACID** (Atomicity, Consistency, Isolation, Durability) transactions using a microservices architecture and it’s impossible in some cases. For example, in the aforementioned flight booking scenario, a micro-service with the block seat functionality can’t acquire a lock on the payment database, since it could be an external service in most cases. But some form of transaction management is still required, and such transactions are referred to as ***BASE*** transactions: **B**asically **A**vailable, **S**oft state, and **E**ventually consistent. Compensating actions must be taken to revert anything that occurred as part of the transaction. Below, it can be seen how a saga can be visualised for the aforementioned flight booking scenario.

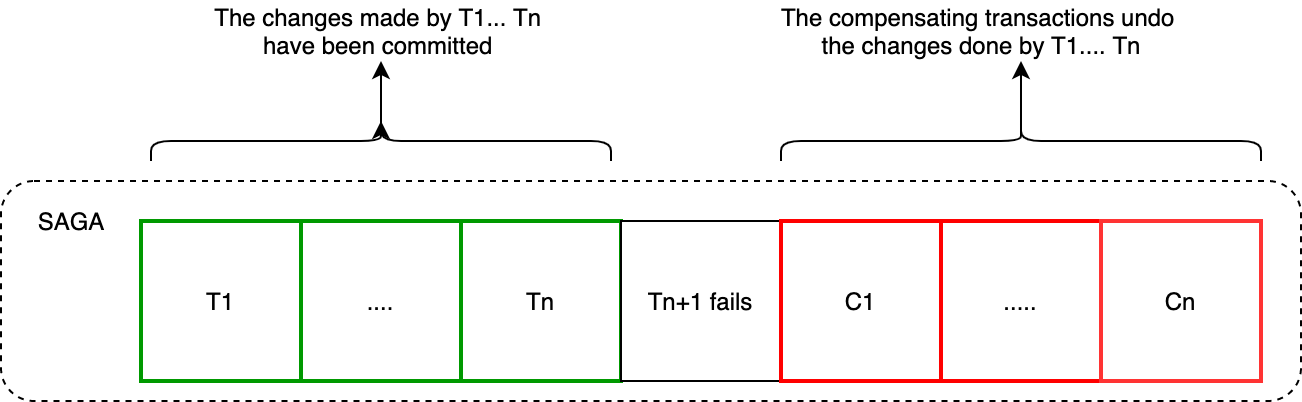




Compensating Transactions

When a step of a saga fails because of a business rule violation, the saga must explicitly undo the updates made by previous steps by executing compensating transactions. Suppose that the (*n* + 1)th transaction of a saga fails. The effects of the previous *n* transactions must be undone. Conceptually, each of those steps, Ti, has a corresponding compensating transaction, Ci, which undoes the effects of the Ti. To undo the effects of those first *n* steps, the saga must execute each Ci in reverse order. The sequence of steps is T1 … Tn, Cn … C1, as shown. In this example, Tn+1 fails, which requires steps T1 … Tn to be undone. The saga executes the compensation transactions in reverse order of the forward transactions: Cn … C1. The mechanics of sequencing the Cis aren’t any different than sequencing the Tis. The completion of Ci must trigger the execution of Ci-1.

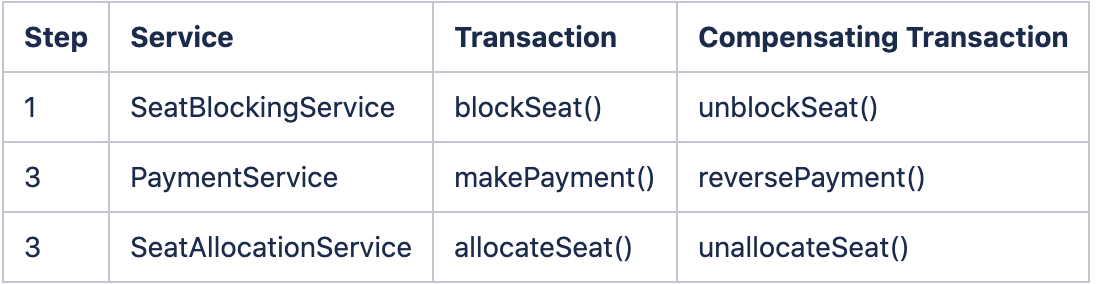




Pivot and Retryable Transactions

The below table shows the compensating transactions for each step of the flight booking saga. The three steps of the flight booking saga are termed as compensating*transactions* because they’re followed by steps that can fail. It is also important to note that not all steps need compensating transactions. There are two other transaction types in saga pattern, one is Pivot Transaction, it is like a go/no-go point in saga. If the pivot transaction commits, the saga runs until completion. The other is Retryable transactions*,*transactions that follow the pivot transaction and are guaranteed to succeed.





Saga Guarantee

A distributed saga guarantees one of the following two outcomes. Either all requests in the saga are successfully completed, or a subset of requests and their compensating requests are executed. Both requests and compensating requests need to obey certain principles:

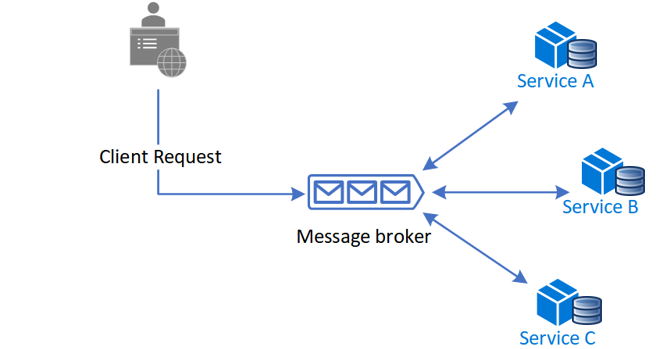
* individual transaction can abort, and must be idempotent.
* compensating transactions must be idempotent, commutative, and they cannot abort (they must be retried indefinitely or resolved through manual intervention when necessary).

Saga Coordination Strategies

The Saga Execution Coordinator (SEC) is the core component for implementing a successful saga flow. Saga coordination can be implemented in :

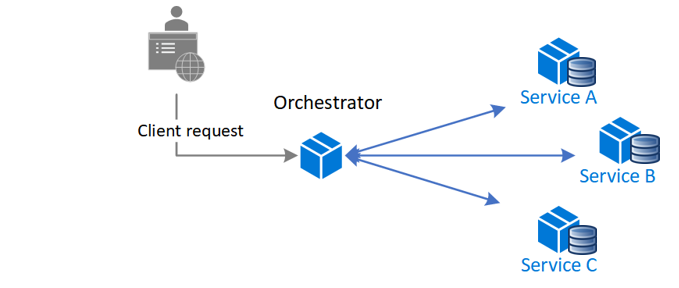
* **Choreography** — Distribute the decision making and sequencing among the saga participants. In other words, participants exchange events without a centralized point of control and each local transaction publishes domain events that trigger local transactions in other services. Though, saga choreography is simple and reliable event based communication, but it is ideal for simple use cases and has some limitations which will not make it an ideal candidate for managing distributed transactions. Choreography-based sagas are difficult to understand, often create cyclic dependencies, and there is a risk of tight coupling between saga participants.





* **Orchestration —**Centralize a saga’s coordination logic in a saga orchestrator class. A saga orchestrator sends commands to saga participants and acts on event outcomes. The orchestrator executes saga requests, stores and interprets the states of each task, and handles failure recovery with compensating transactions. Orchestrator-based sagas are more suitable for complex event processing and make them a good candidate for managing distributed transactions.

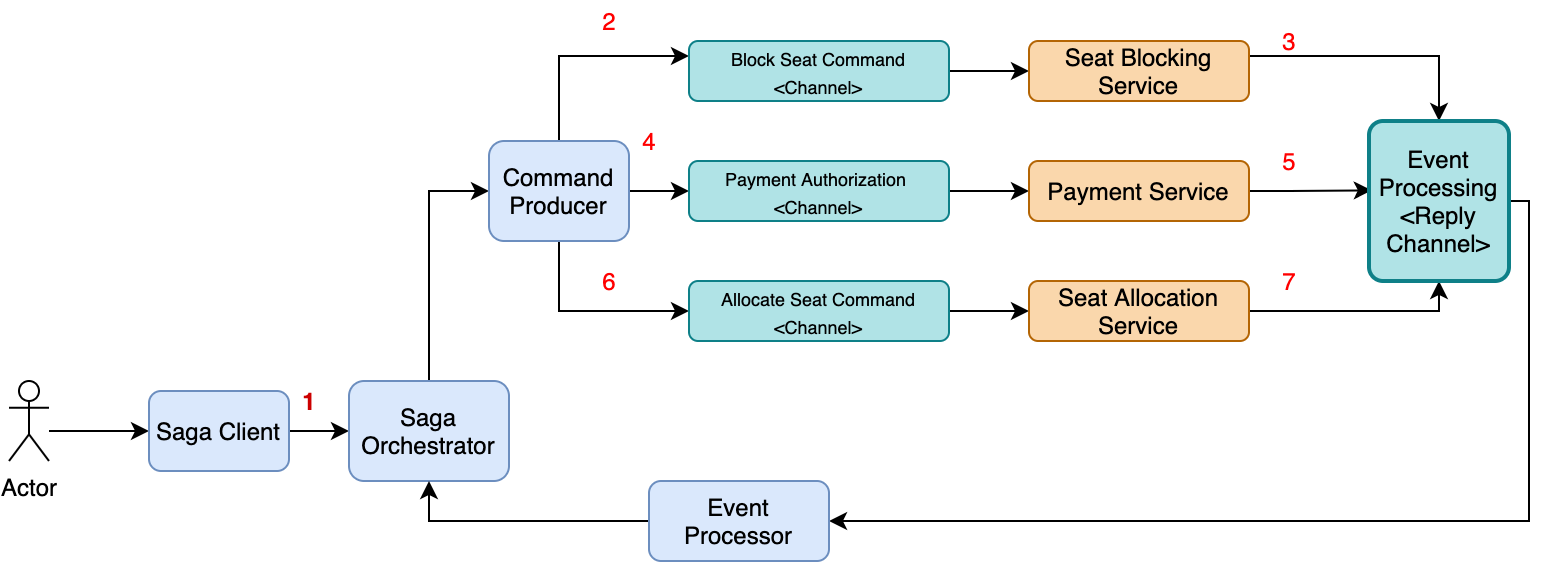




Saga Orchestrator

As Saga “orchestration” pattern suggests, there is a single orchestrator component that is responsible for managing the overall process workflow. When using orchestration, you define an orchestrator class whose sole responsibility is to tell the saga participants what to do. The saga orchestrator communicates with the participants using command/async reply-style interaction. To execute a saga step, it sends a command message to a participant telling it what operation to perform. After the saga participant has performed the operation, it sends a reply message to the orchestrator. The orchestrator then processes the message and determines which saga step to perform next.





The above diagram shows the design of the orchestration-based version of the flight booking saga. The saga is orchestrated by the SagaOrchestrator component, which invokes the saga participants using asynchronous request/response. Saga orchestrator keeps track of the process and sends command operations to saga participants, such as SeatBlockingService and PaymentService, through a command producer component, and reads reply messages from its reply channel through an event processor, and then determines the next step, if any, in the saga. The steps for a happy day saga path would be as:

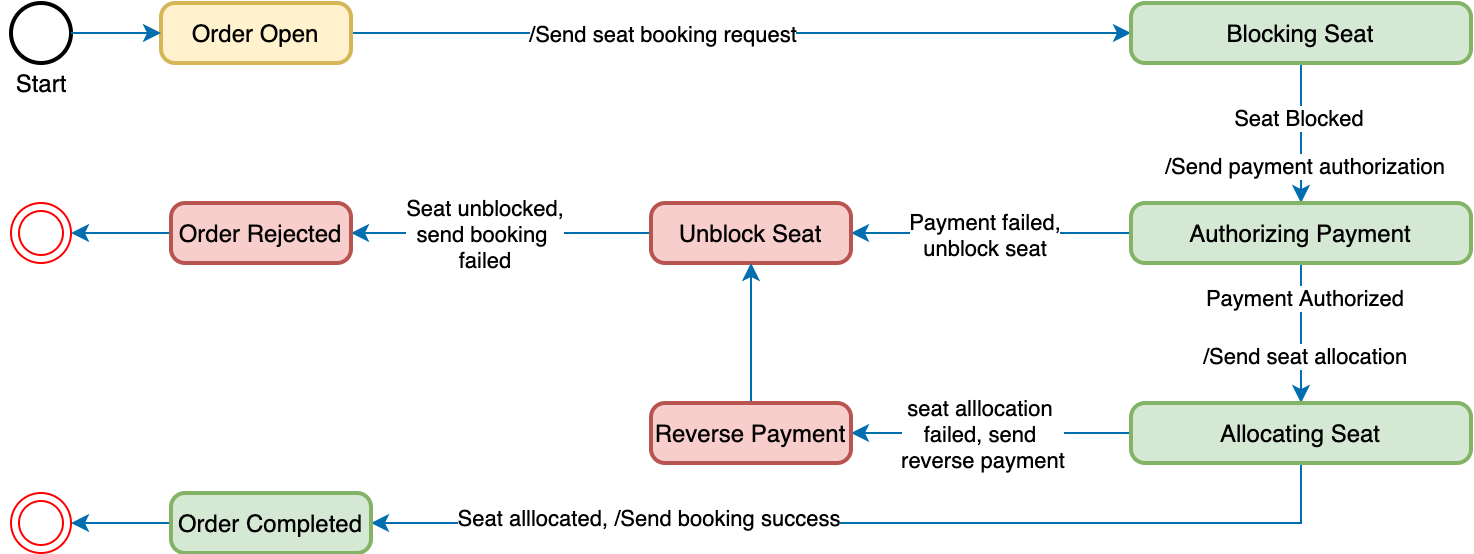
1. The FrontEnd UI sends a seat booking request to saga orchestrator.
2. The saga orchestrator starts a new workflow and sends a SeatBlockingCommand to SeatBlockingService.
3. SeatBlockingService process the command and reply with a SeatBlockedEvent.
4. The saga orchestrator triggers the next action in workflow and sends a PaymentRequestCommand to PaymentService.
5. PaymentService replies with a PaymentSuccessEvent.
6. The saga orchestrator then sends an SeatAllocationCommand to SeatAllocationService.
7. SeatAllocationService replies with a SeatAllocatedEvent.
8. The saga orchestrator ends the transaction and completes the workflow.

But, the overall flight booking saga scenario can fail due to a failure in either SeatBlockingService, PaymentService, or SeatAllocationService. In order to manage the workflow effectively and handle faults, it is recommended to model a saga as a state machine, because it describes all possible scenarios and lets the orchestrator determine what action needs to be performed.

Saga as a State Machine

Modelling a saga orchestrator as a state machine is an effective way to not only manage distributed transactions, but also support long running business transactions . A state machine consists of a set of states and a set of transitions between states that are triggered by events. Each transition can have an action, which for a saga is the invocation of a saga participant. The transitions between states are triggered by the completion of a local transaction performed by a saga participant. The current state and the specific outcome of the local transaction determine the state transition and what action, if any, to perform. As a result, using a state machine model makes designing, implementing, and testing sagas easier.

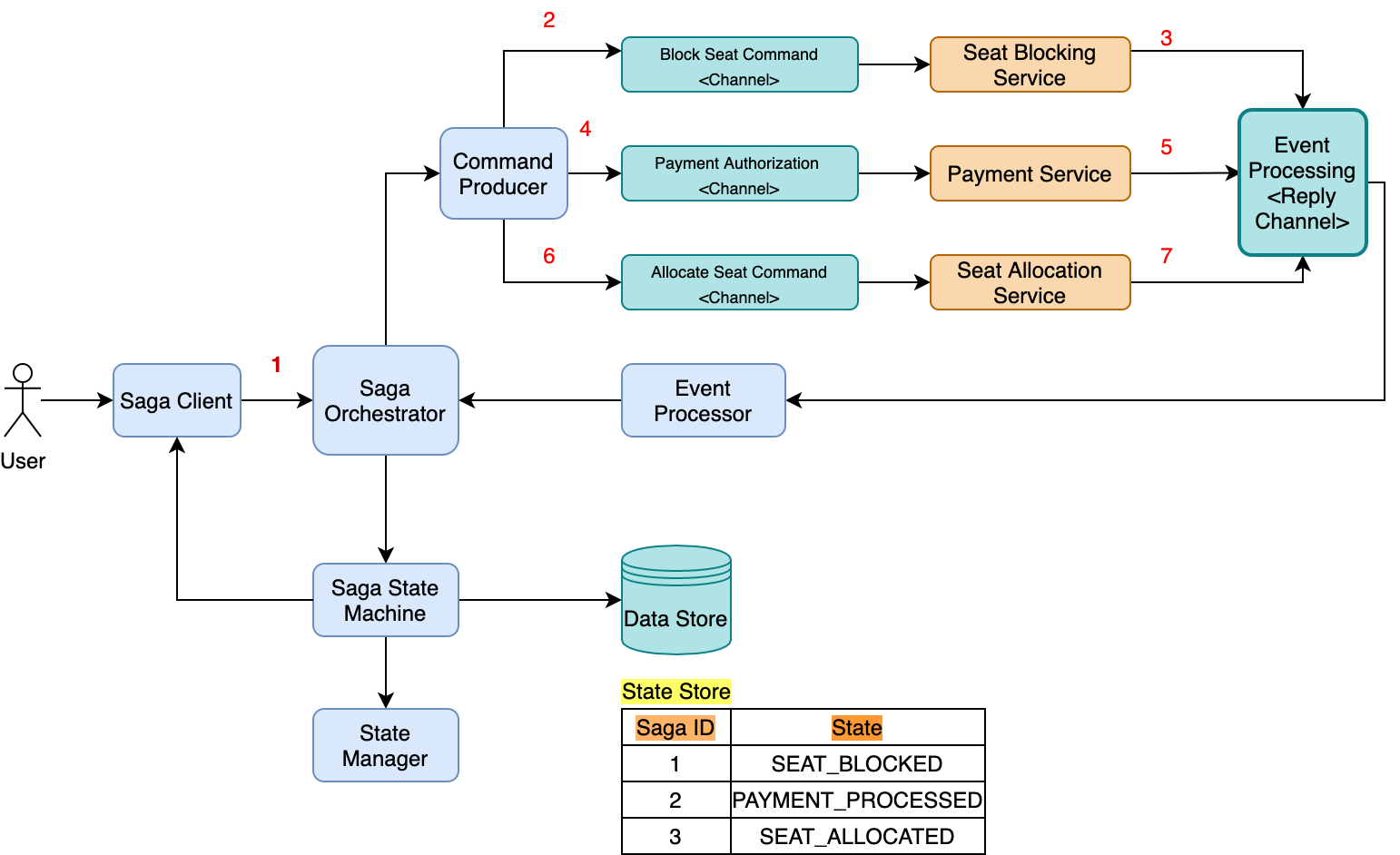




The above diagram highlights the state machine model for the flight booking saga. This state machine consists of numerous states and transitions, including the following:

* Order Open — The initial state. Saga set this state on start of workflow.
* Blocking Seat — When in this state, the saga is waiting for the SeatBlockingService to block the seat for booking.
* Authorizing Payment — The saga is waiting for a reply to the payment authorization command from PaymentService.
* Allocating Seat — Waiting for SeatAllocationService to allocate the seat after payment success.
* Reverse Payment — If Seat allocation fails, saga would send a request for payment refund.
* Unblock Seat — If payment authorization fails, saga would send fail event to unblock seat.
* Order Completed — A final state indicating that the saga completed successfully.
* Order Rejected — A final state indicating that the Order was rejected by one of the participants.





Finally, the saga workflow can be redesigned as saga state machine, as shown here. The saga orchestrator is linked to a state machine, which is responsible for managing transaction states through a state manager API. In addition to that, it is also responsible for storing transaction states in a persistent data store to ensure recovery when system faults happen. The saga state machine thus has the responsibility to either get the overall business transaction completed or to leave the system in a known state, so that it can determine the order in which to potentially execute the next state of actions, or compensation activities, whether transactions occurring are distributed in nature or, long lived.

Benefits and potential use cases

* **Simpler dependencies —**The saga orchestrator invokes the saga participants, but the participants don’t invoke the orchestrator. As a result, the orchestrator depends on the participants but not vice versa, and so there are no cyclic dependencies.
* **Less coupling —**Each service implements an API that is invoked by the orchestrator, so it does not need to know about the events published by the saga participants.
* **Separation of concerns —**The saga coordination logic is localized in the saga orchestrator. The domain objects are simpler and have no knowledge of the sagas that they participate in.
* **Data Consistency —**Maintain data consistency across multiple microservices without tight coupling.
* **Developer experience —**Design allows developers to focus only on the business logic of the saga participants and simplify the implementation of stateful workflows on the saga orchestrator.

A couple of potential use cases where such implementation can be carried out:

1. Order Management System

* e-commerce
* food delivery
* flight booking
* hotel/taxi booking

2. Settlement transactions.

Guidelines and recommendations

If we are designing and building orchestrator driven saga for supporting distributed and long running transactions, the below guidelines are recommended:

1. Orchestrator should only be responsible for managing transactions and states, and there should not be any business logic added here. Business logic should be defined in individual service participants.
2. All events and commands to and from the orchestrator should be carrying only transaction data, not reference data.
3. Use asynchronous style messaging to communicate between services.
4. Implement idempotency and state checks for resiliency, if using message brokers like Kafka.
5. Suitable for designing command side (write model) in CQRS and Event Sourcing architecture.

**References**

s