Distributed Sagas for Microservices

Da Huo, Yaxi Lei

Overview

- Challenges of microservices architectures
- What is “Saga” and where does the term come from
- Intro to distributed sagas
    - Components
    - Characteristics
- Implementation Approaches
- AWS Step Functions as a Saga Execution Coordinator
- Detailed case study of distributed sagas

The Problem

Benefit of Microservices Architecture:

- Scalability
- Flexibility
- Productivity
- ...

Challenges with Microservices Architecture:

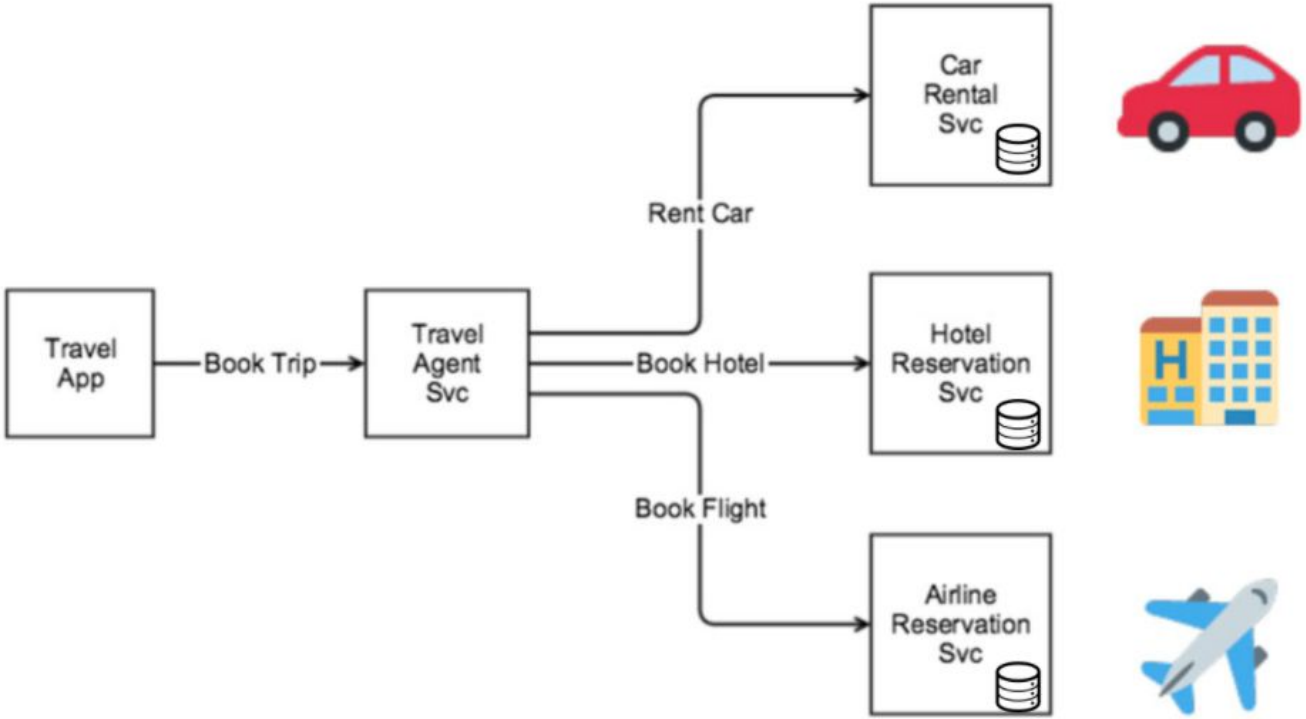
It is hard to maintain the correctness & consistency in a distributed transaction

What is a transaction?

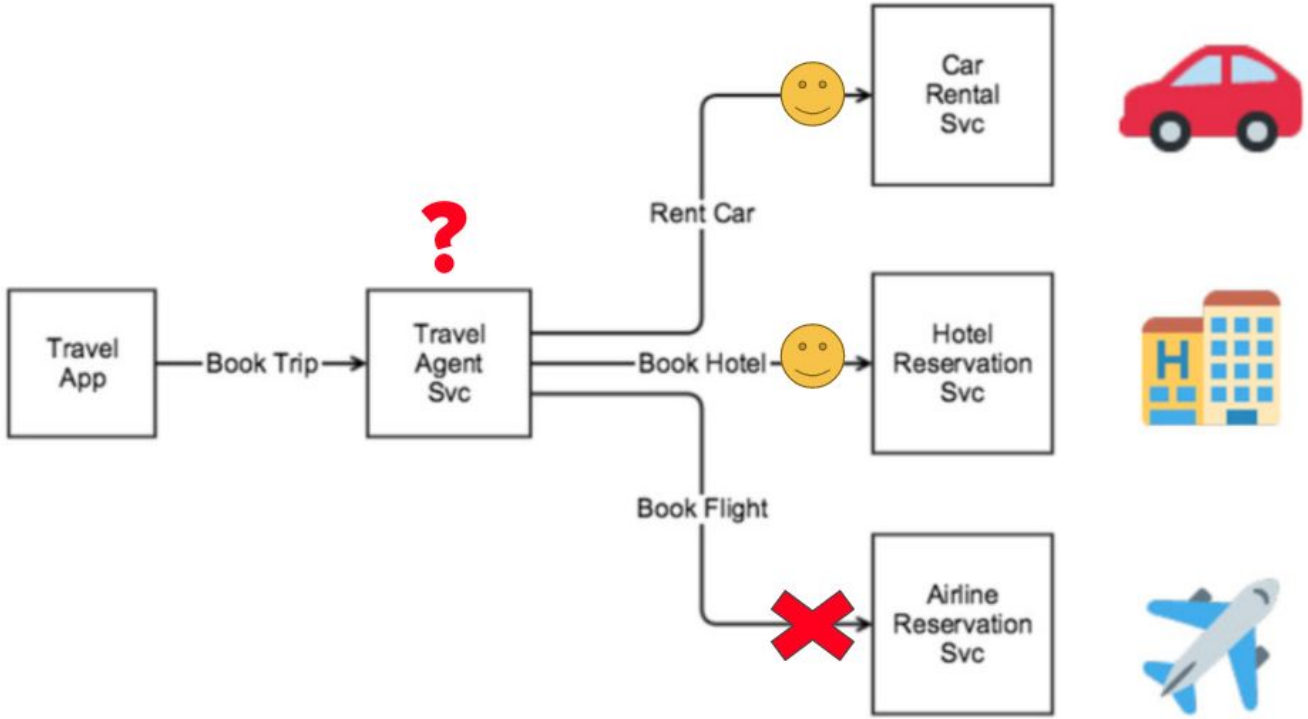
A set of operations that need to be performed together

- In monolithic systems:
  - Use a single relational database to maintain ACID semantics
  - Failure is all-or-nothing
- In microservices based systems:
  - Different components/services can fail
  - Hard to maintain correctness and consistency

Example of Distributed Transaction



What if one of the services has failed?



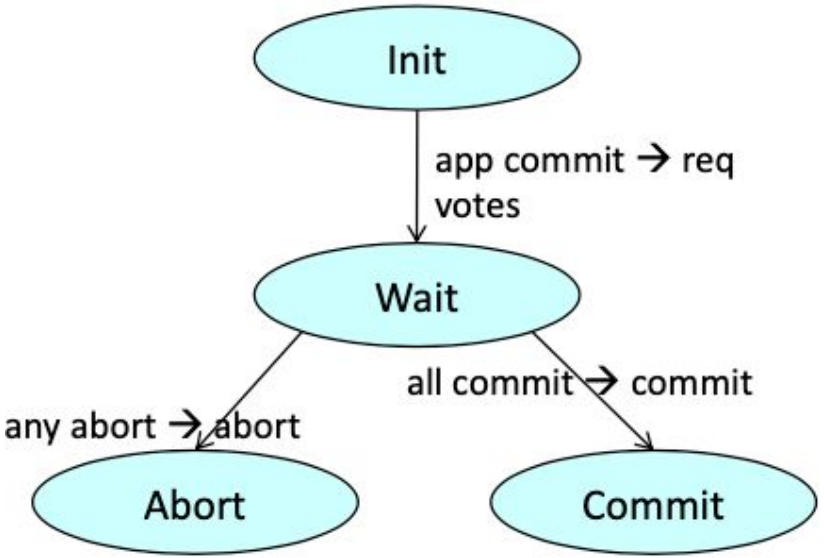
2 Phase Commit

Phase1:

- Coordinator ask all participants to vote if ready to commit
- Participants vote

Phase2:

- If all participants voted yes → All participants commit
- If any voted no → All participants abort



Problems with 2PC

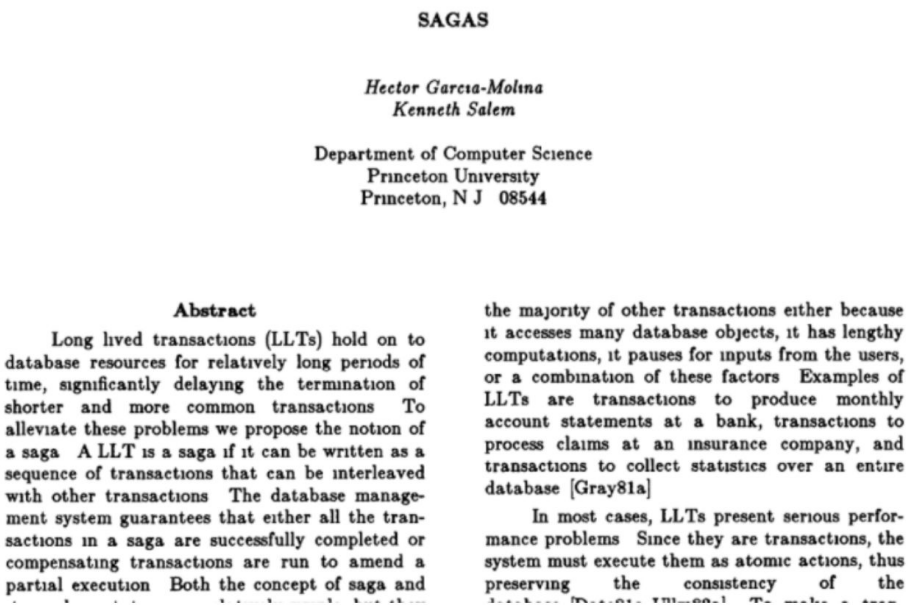
- 2PC does not scale
- O(n^2) messages required in worst case
- Throughput is limited by the slowest node in the cluster
- Coordinator is a single point of failure

Distributed Sagas

- A protocol for coordinating microservices
- A way to ensure data consistency in a distributed architecture without having a single ACID transaction

Where does the term “Saga” come from?

- The term Saga was first used in a database systems research paper in 1987



SAGAS

Hector Garcia-Molina
Kenneth Salem

Department of Computer Science
Princeton University
Princeton, N J 08544

Abstract

Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions To alleviate these problems we propose the notion of a saga A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions The database management system guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to amend a partial execution Both the concept of saga and

the majority of other transactions either because it accesses many database objects, it has lengthy computations, it pauses for inputs from the users, or a combination of these factors Examples of LLTs are transactions to produce monthly account statements at a bank, transactions to process claims at an insurance company, and transactions to collect statistics over an entire database [Gray81a]

In most cases, LLTs present serious performance problems Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the

Challenges with DBMS in 1987

- Long lived transactions hold on to database resources for a long period of time causing the delay of other lighter and more common transactions
  - Lock resources for the entire duration of transaction
  - Other transactions have to wait until the long lived transactions to finished
- Example of long lived transactions:
  - Produce monthly bank statements at a bank
  - Collect statistics over the entire database

Solution

- Find and break sagas into a set of sub-transactions
- Execute sub-transactions and lock resources separately
- If any sub-transaction failed, execute compensating transactions for their corresponding completed sub-transactions
  - A compensating transaction semantically undoes its corresponding transactions
  - Covered in detail in later slides

What is a Saga?

- In DBMS:

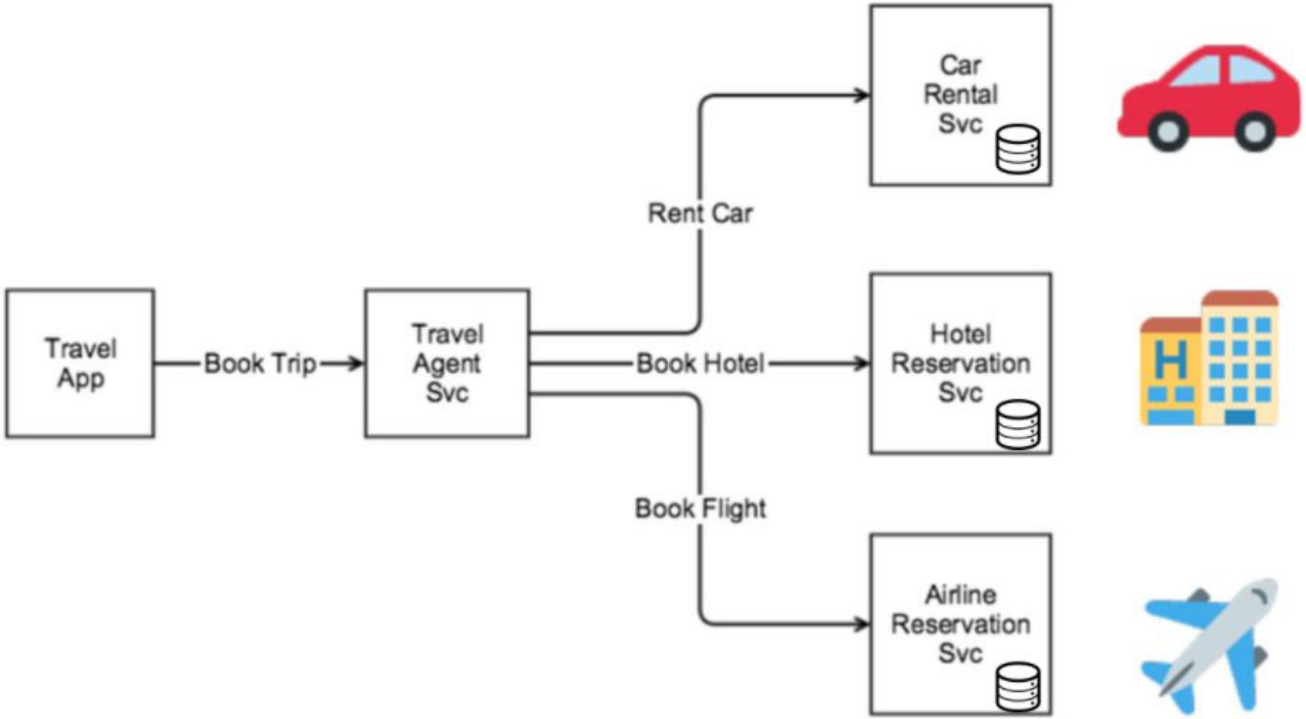
  A saga is a long lived transaction that can be broken up into a collection of sub-transactions that can be interleaved in any way with other transactions

- In Distributed Systems / Microservices:

  A Saga represents a high-level business process that consists of several

  low-level Requests that each update data within a single service

Book Trip is a Saga consists of Book car, Book hotel, and Book flight



Distributed Sagas

A distributed saga contains 2 parts:

- A collection of requests
  - Example: Book hotel, Book car, Book flight
- Compensating requests for each request
  - Semantically undoes it’s corresponding request
  - Cancel hotel, Cancel car, Cancel flight

Characteristics of distributed sagas requests

- Requests can abort (service can reject a request at any time)
- Requests must be idempotent

Characteristics of compensating requests

- Compensating requests CANNOT abort
- Compensating requests must be idempotent
- Compensating requests must be commutative
    - Book hotel
Cancel hotel **=** Cancel hotel
Book hotel

Guarantees of Distributed Sagas

With distributed sagas, one of the follow two outcomes will happen:

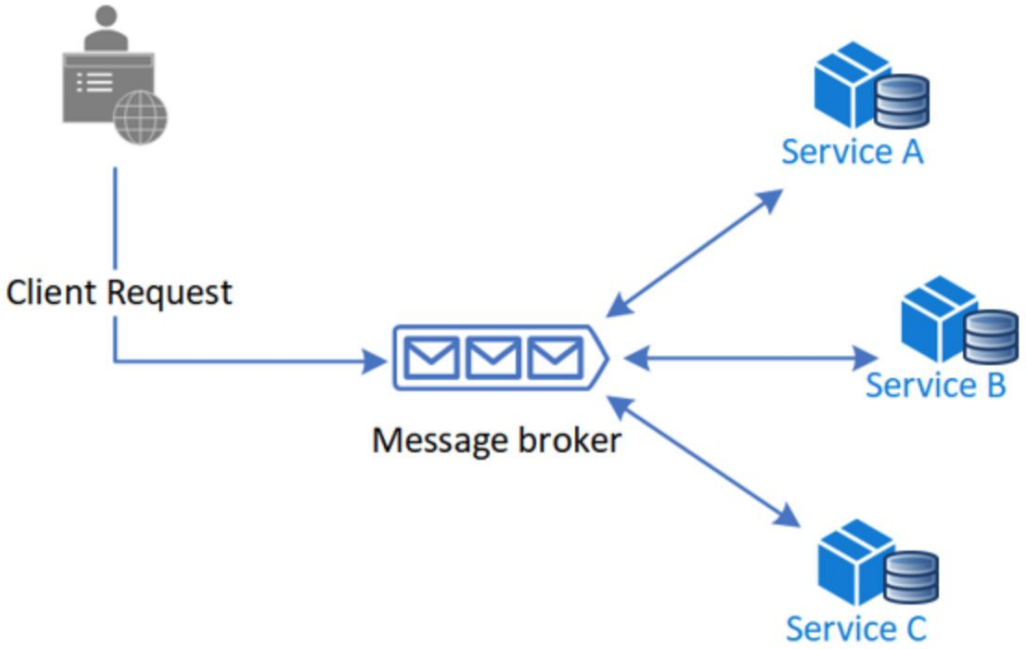
1. All requests are successfully completed
2. A subset of requests and their compensating requests are executed

Distributed Sagas Implementation Approaches

1. Event-driven choreography
2. Orchestration

Event-driven choreography

- No central coordination
- Each service will produce and consume to events of other services and decide what actions to take



Benefit of Event-driven choreography

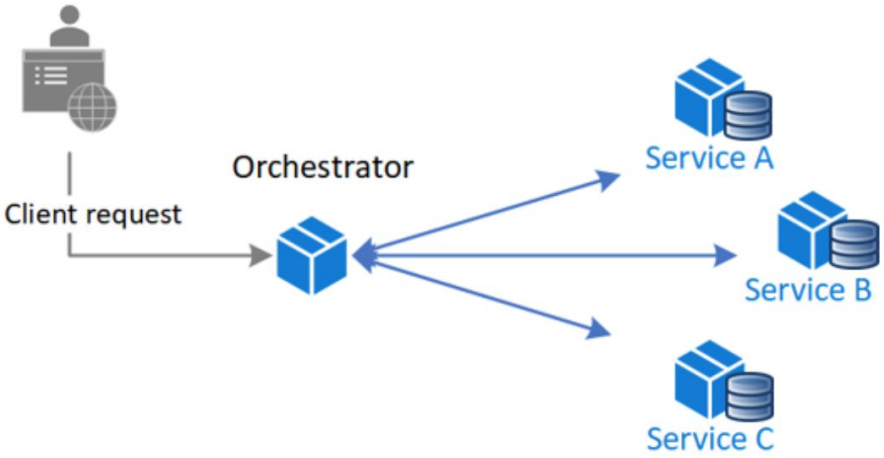
- Does not require additional coordinator logic implementation and maintenance
- No single point of failure

Drawbacks of Event-driven choreography

- Workflow can be confusing as the microservice architecture gets increasingly complex
- Risk of cyclic dependency between services (A consumes events from B, B consumes events from A)

Orchestration

- A centralized coordinator service is responsible for decision making
- Coordinator stores and interprets Saga’s current state
- Coordinator tells services what requests to execute
- Coordinator handles failure recovery by executing compensating requests



Benefit of Orchestration

- Clear workflow for complex systems with many participants
- Does not introduce cyclic dependencies

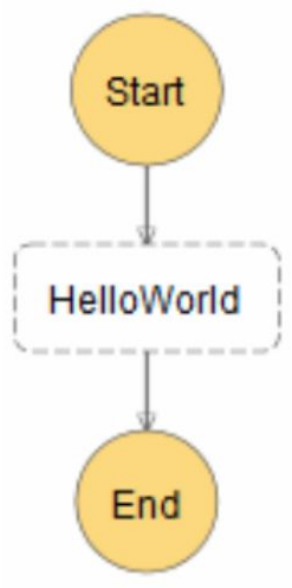
Drawbacks of Orchestration

- Additional logic implementation and maintenance for the coordinator
- Coordinator is an additional point of failure

Define a Distributed Saga - AWS States Language

AWS States Language

```
{
  "Comment": "Hello World example",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",the
      "Resource": "<lambdaARN>",
      "End": true
    }
  }
}
```



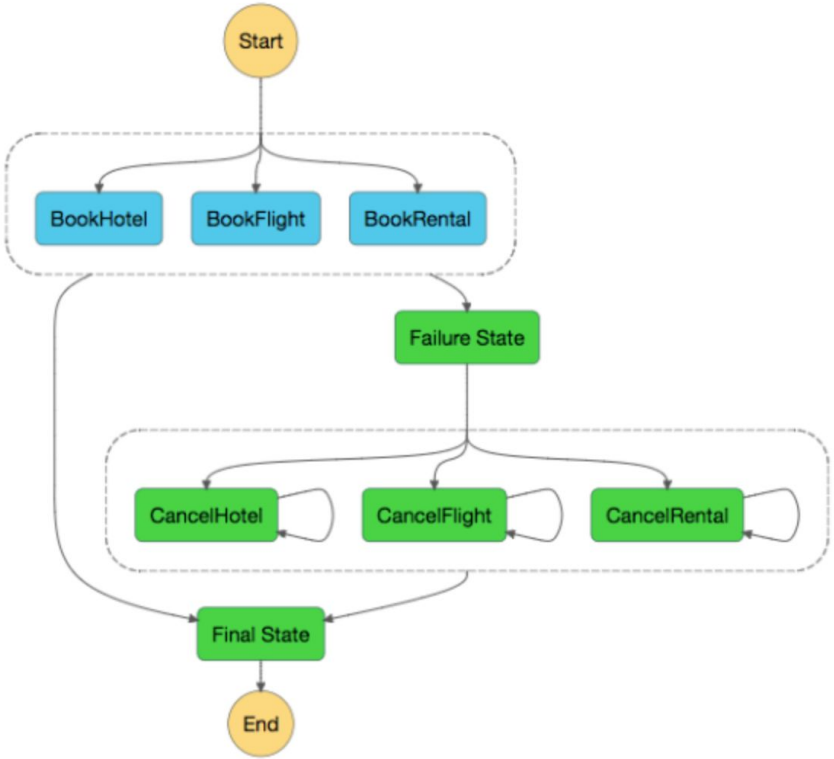
Define a Distributed Saga - Saga Execution Coordinator

- Saga Execution Coordinator
  - Store & Interprets the Saga's state machine
  - Execute the steps of Saga
    - Interact with services
  - Handles failure recovery
    - Executes compensating actions

- AWS Step function
  - Serverless orchestration services
  - Based on state machines and tasks
  - Could performs other AWS Service

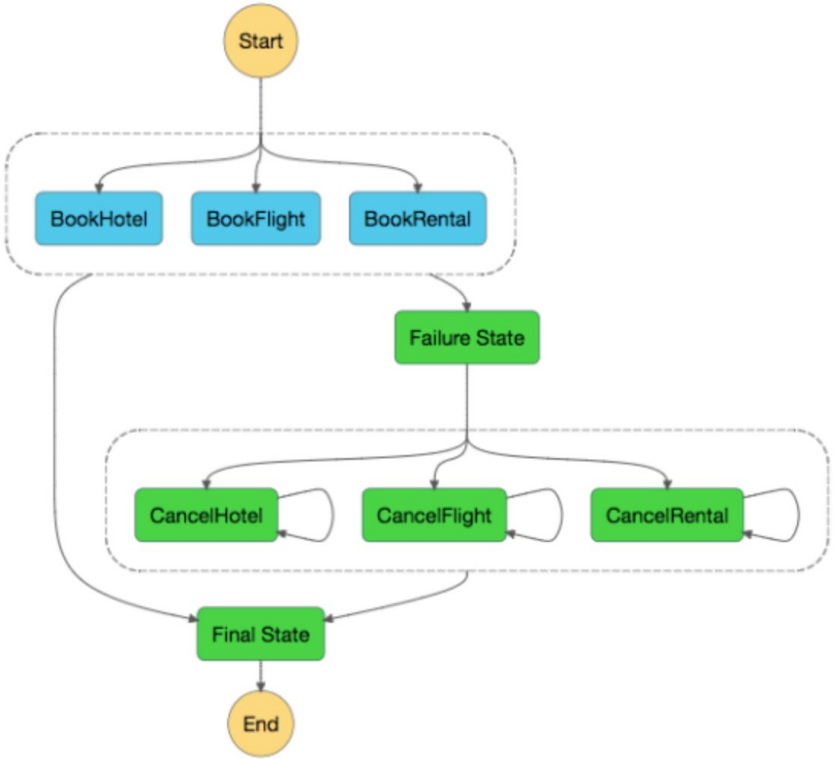
Define a Distributed Saga - Case Study

```
{
  "Comment": "A distributed saga example.",
  "StartAt": "BookTrip",
  "States": {
    "BookTrip": {
      "Type": "Parallel",
      "Next": "Trip Booking Successful",
      "Branches": [
         {
          "StartAt": "BookHotel",
          "States": {
             "BookHotel": {
                "Type": "Task",
                "Resource":
"arn:aws:lambda:{YOUR_AWS_REGION}:{YOUR_AWS_ACCOUNT_ID}:fu
nction:serverless-sagas-dev-bookHotel",
                "ResultPath": "$.BookHotelResult",
                "End": true
             }
          }
        },
        {
          "StartAt": "BookFlight",
          ……,
```

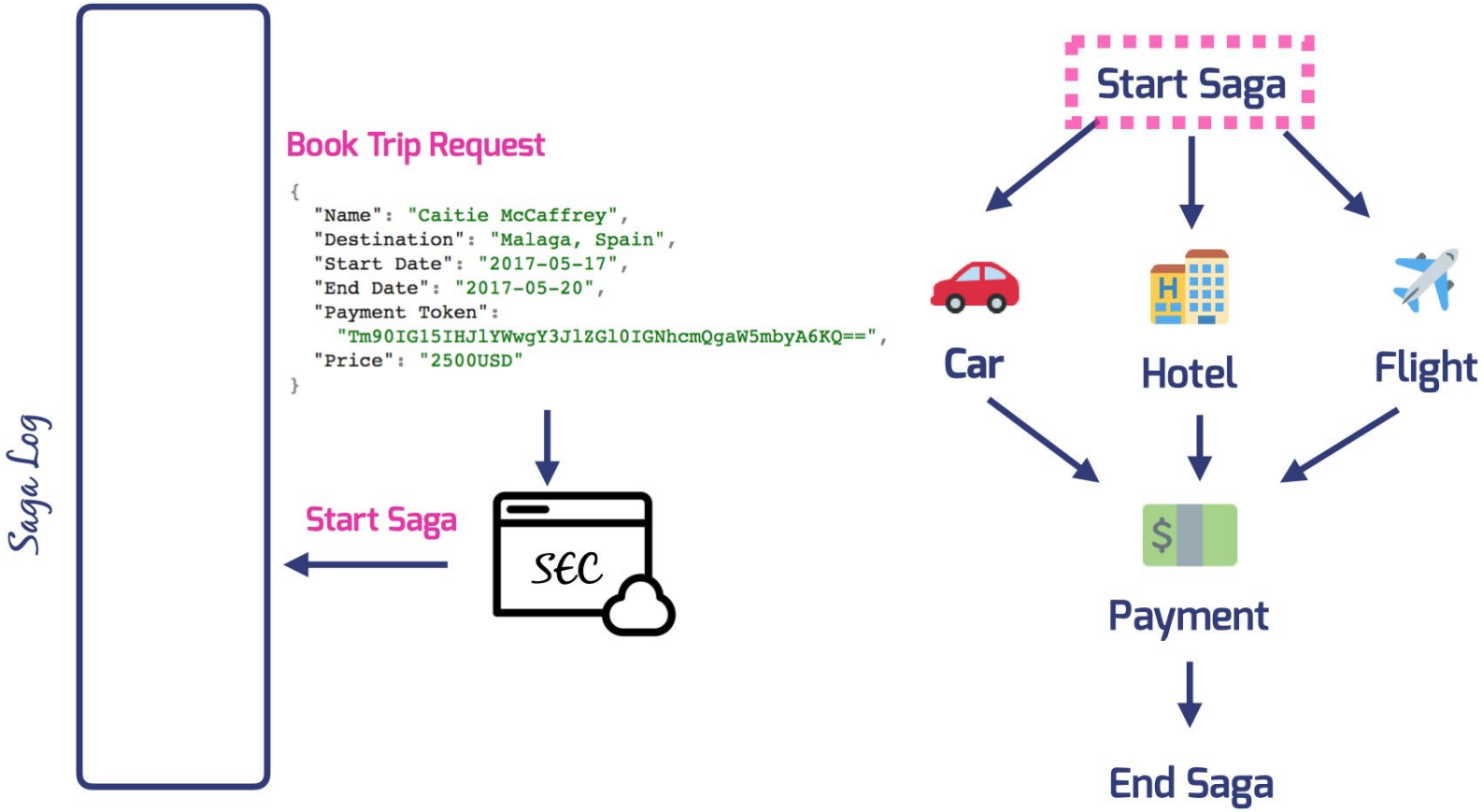


Define a Distributed Saga - Case Study

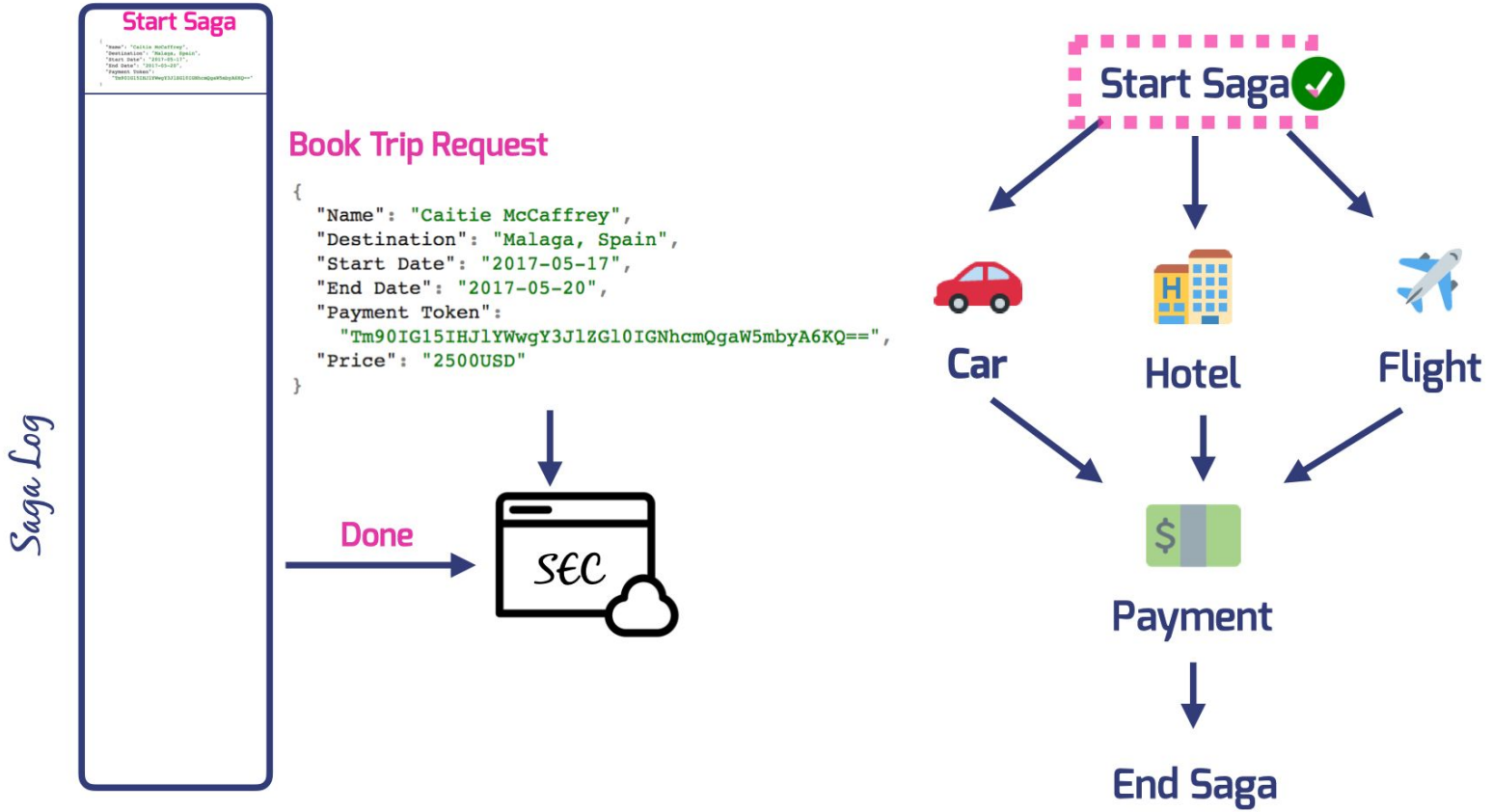
```
"Catch": [
        {
          "ErrorEquals": ["States.ALL"],
          "ResultPath": "$.BookTripError",
          "Next": "Trip Booking Failed"
        }
      ]
    },
    "Trip Booking Failed": {
      "Type": "Pass",
      "Next": "CancelTrip"
    },
    "CancelTrip": {
      "Type": "Parallel",
      "Next": "Trip Booking Cancelled",
      "Branches": [
        {
          "StartAt": "CancelHotel",
          "States": {
            "CancelHotel": {
              "Type": "Task",
              "Resource":
"arn:aws:lambda:{YOUR_AWS_REGION}:{YOUR_AWS_ACCOUNT_ID}:fu
nction:serverless-sagas-dev-cancelHotel",
```



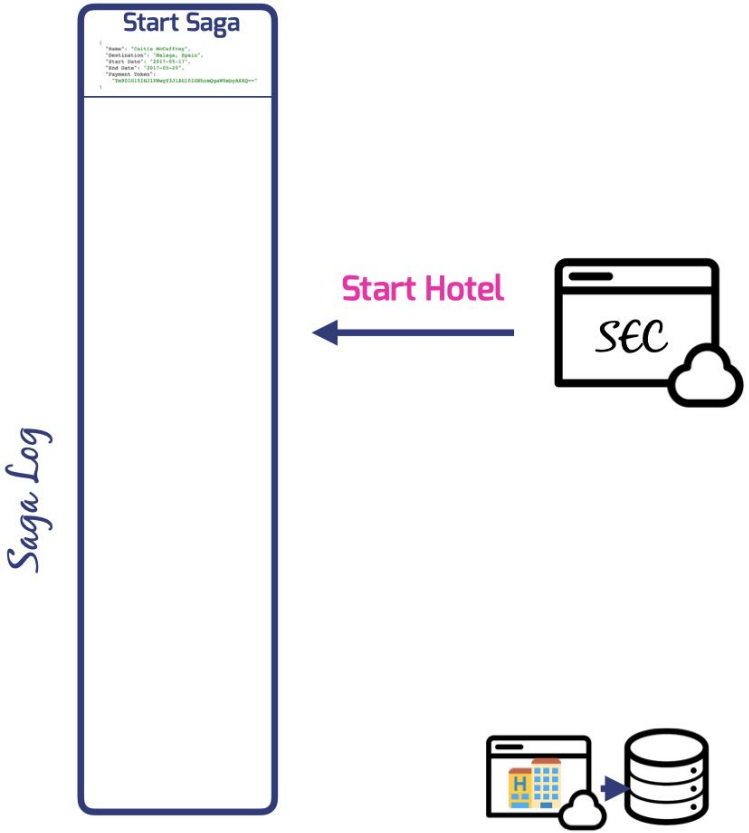
Executing a Distributed Sagas - Case Study

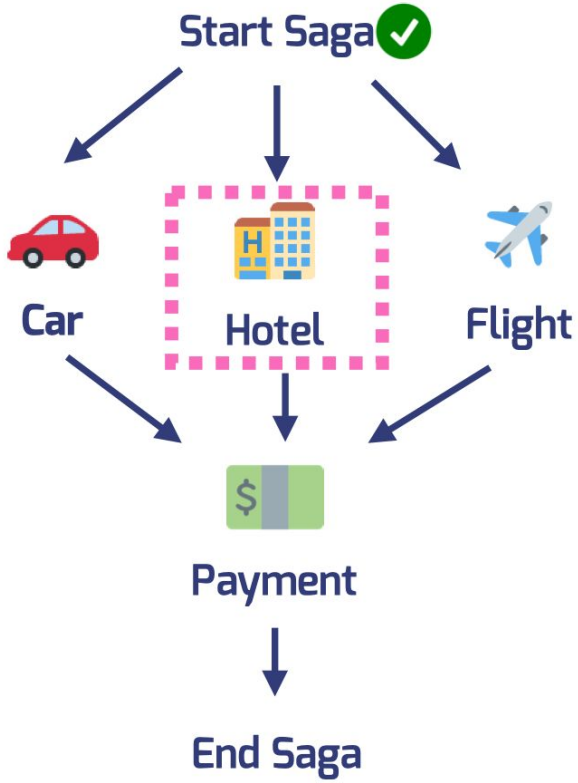


Executing a Distributed Sagas - Case Study

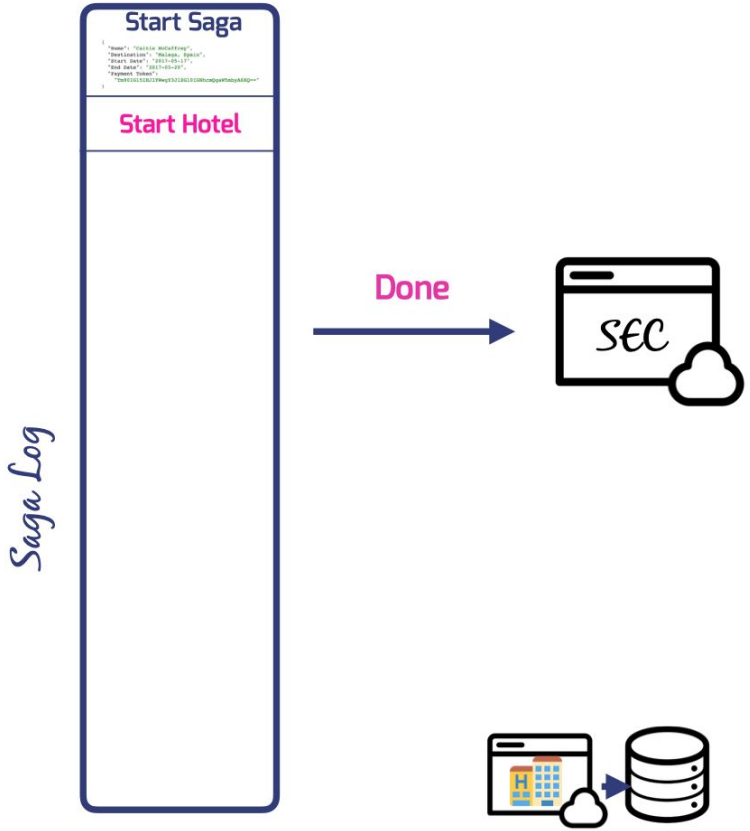


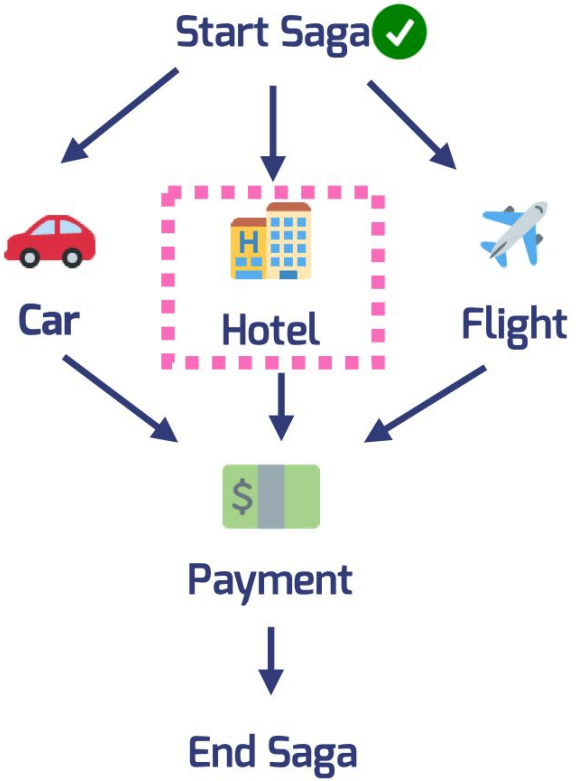
Executing a Distributed Sagas - Case Study



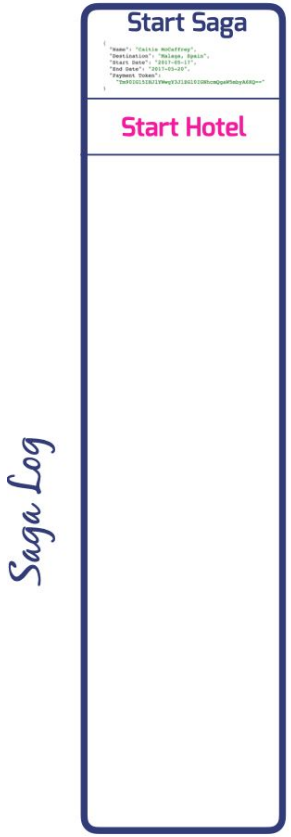


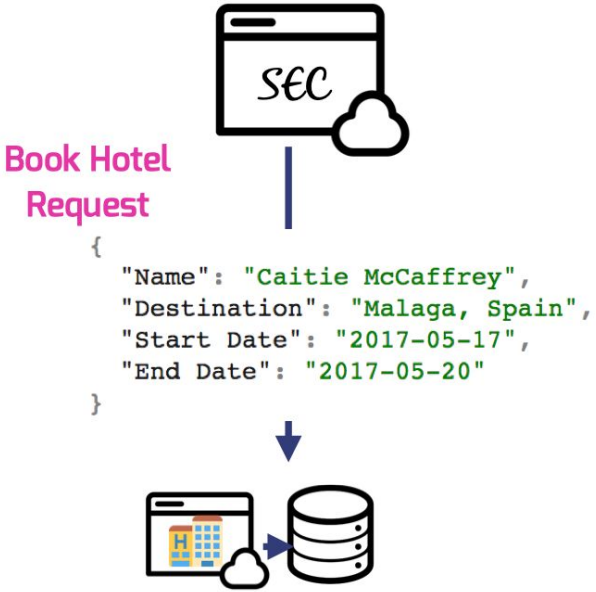
Executing a Distributed Sagas - Case Study

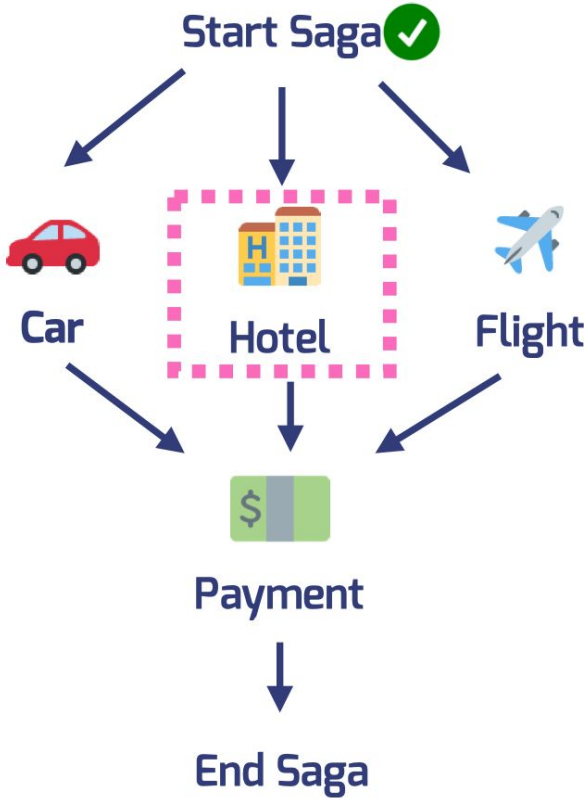




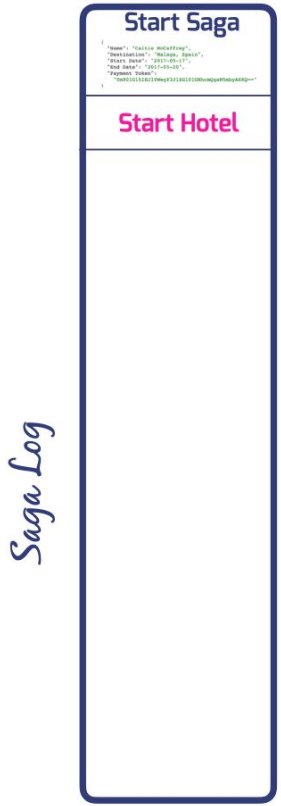
Executing a Distributed Sagas - Case Study

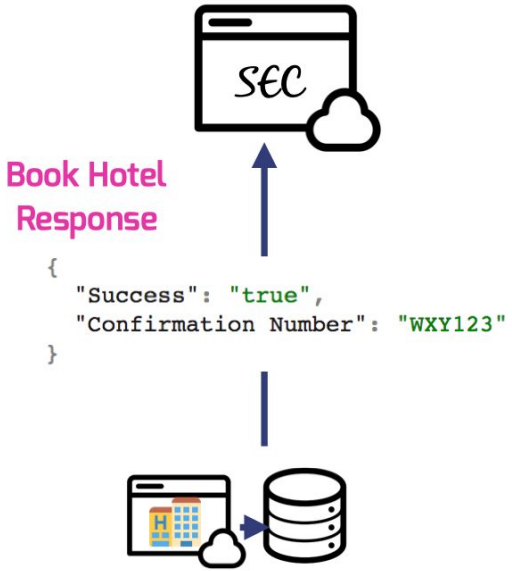


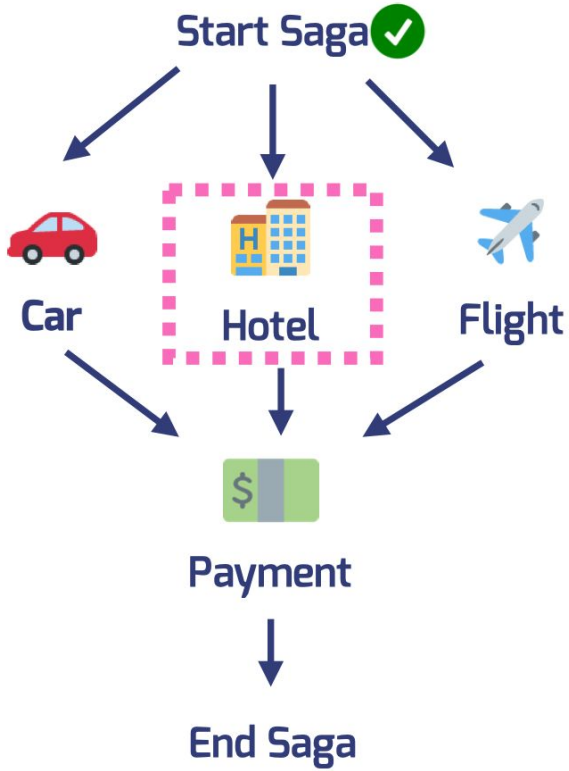




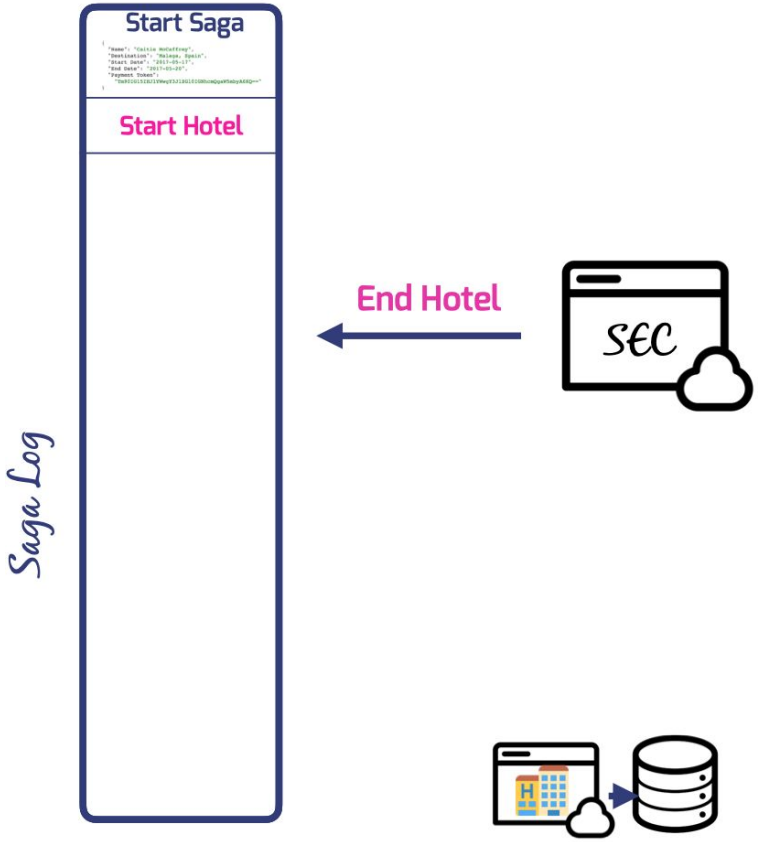
Executing a Distributed Sagas - Case Study

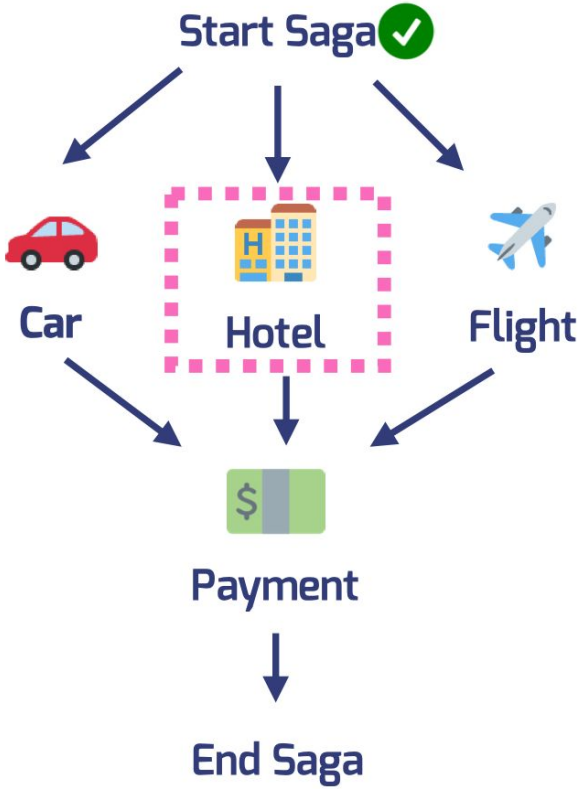




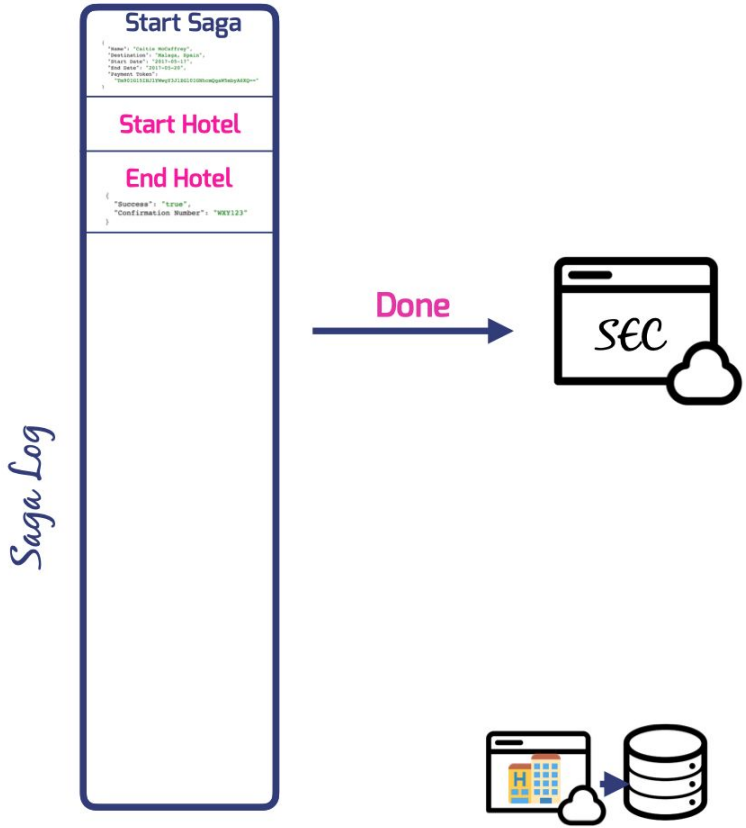


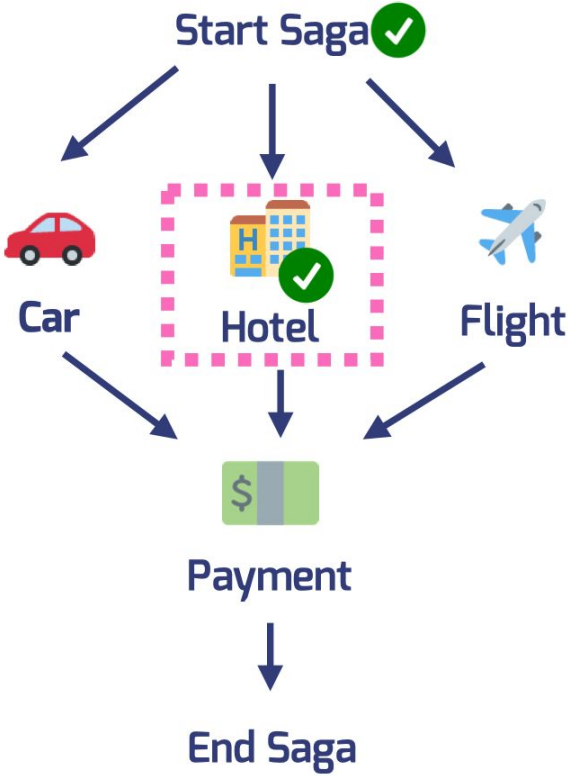
Executing a Distributed Sagas - Case Study



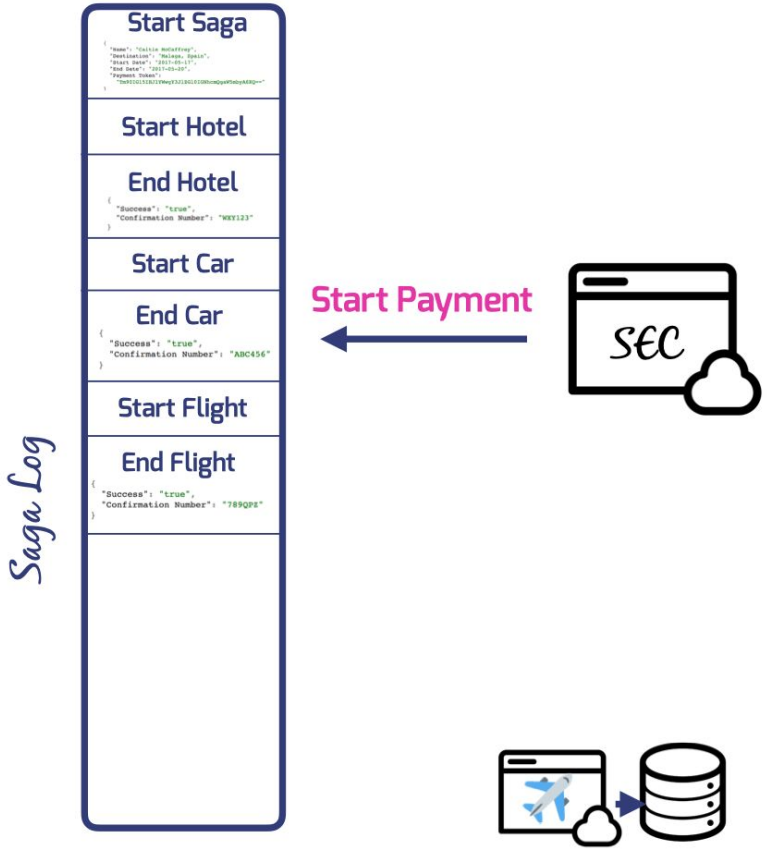


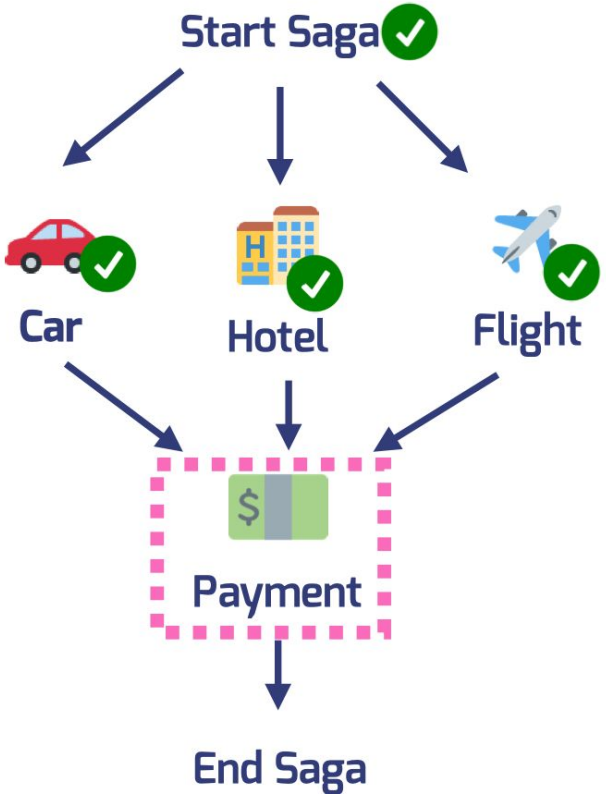
Executing a Distributed Sagas - Case Study



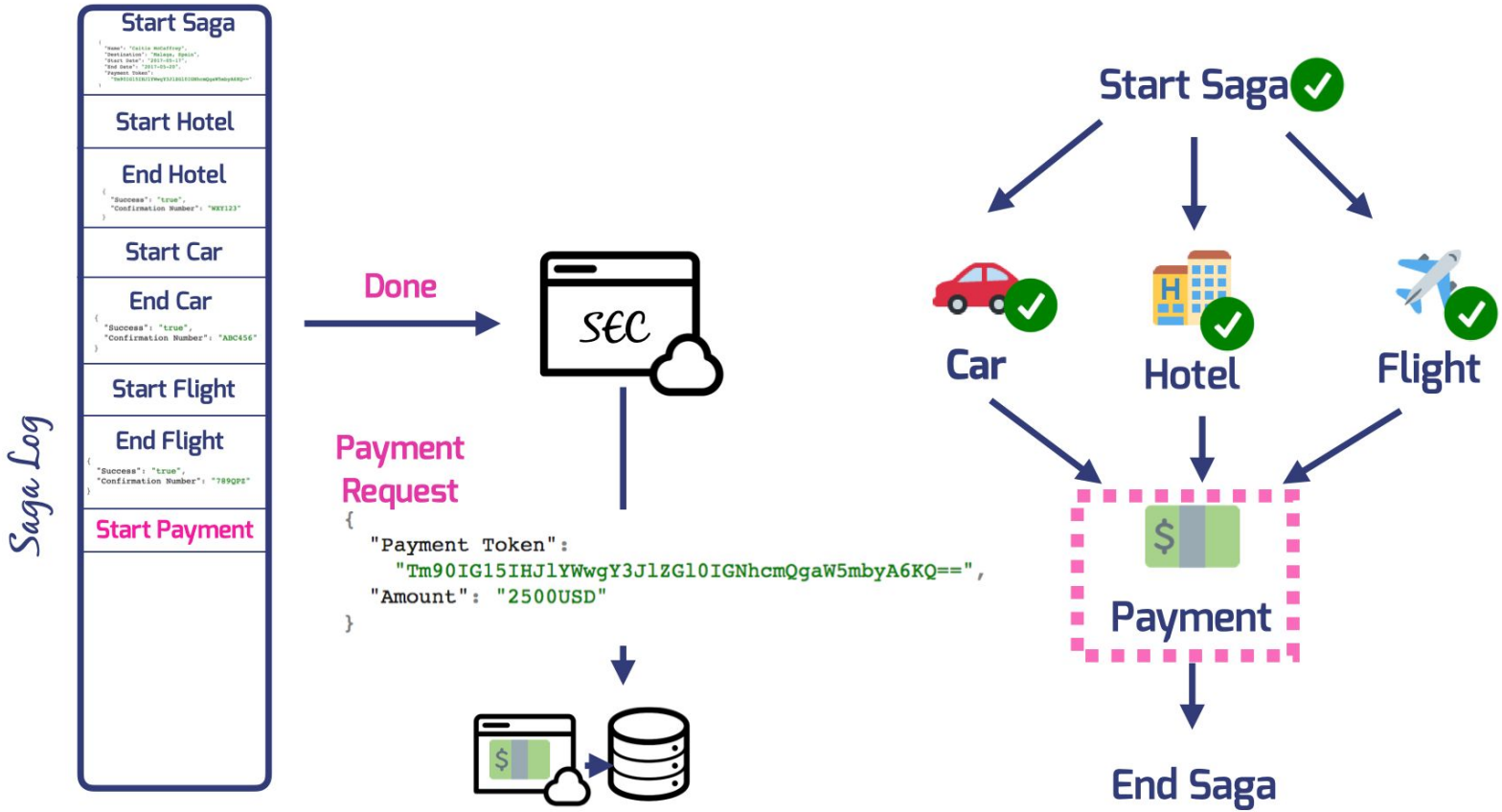


Executing a Distributed Sagas - Case Study

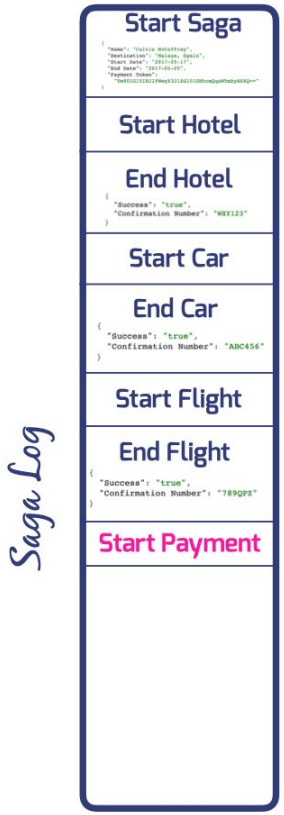


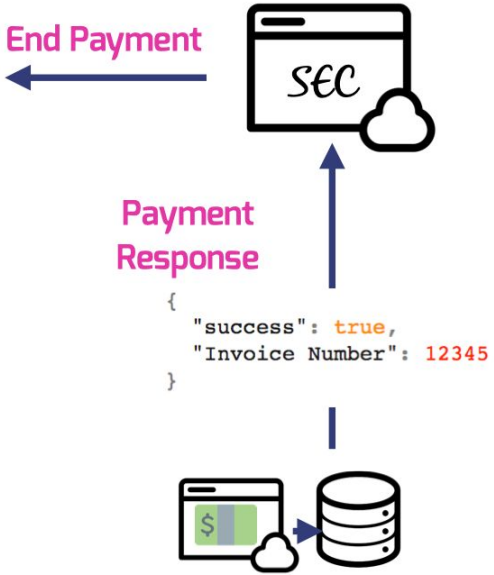


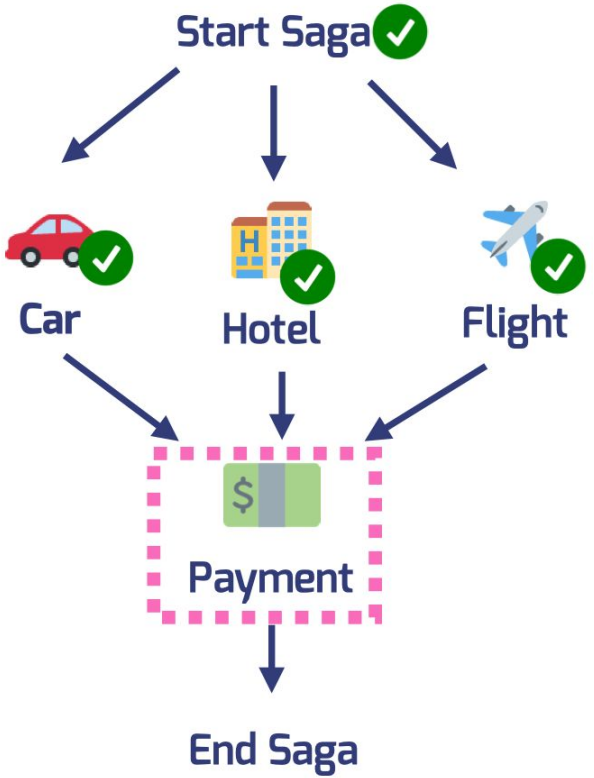
Executing a Distributed Sagas - Case Study



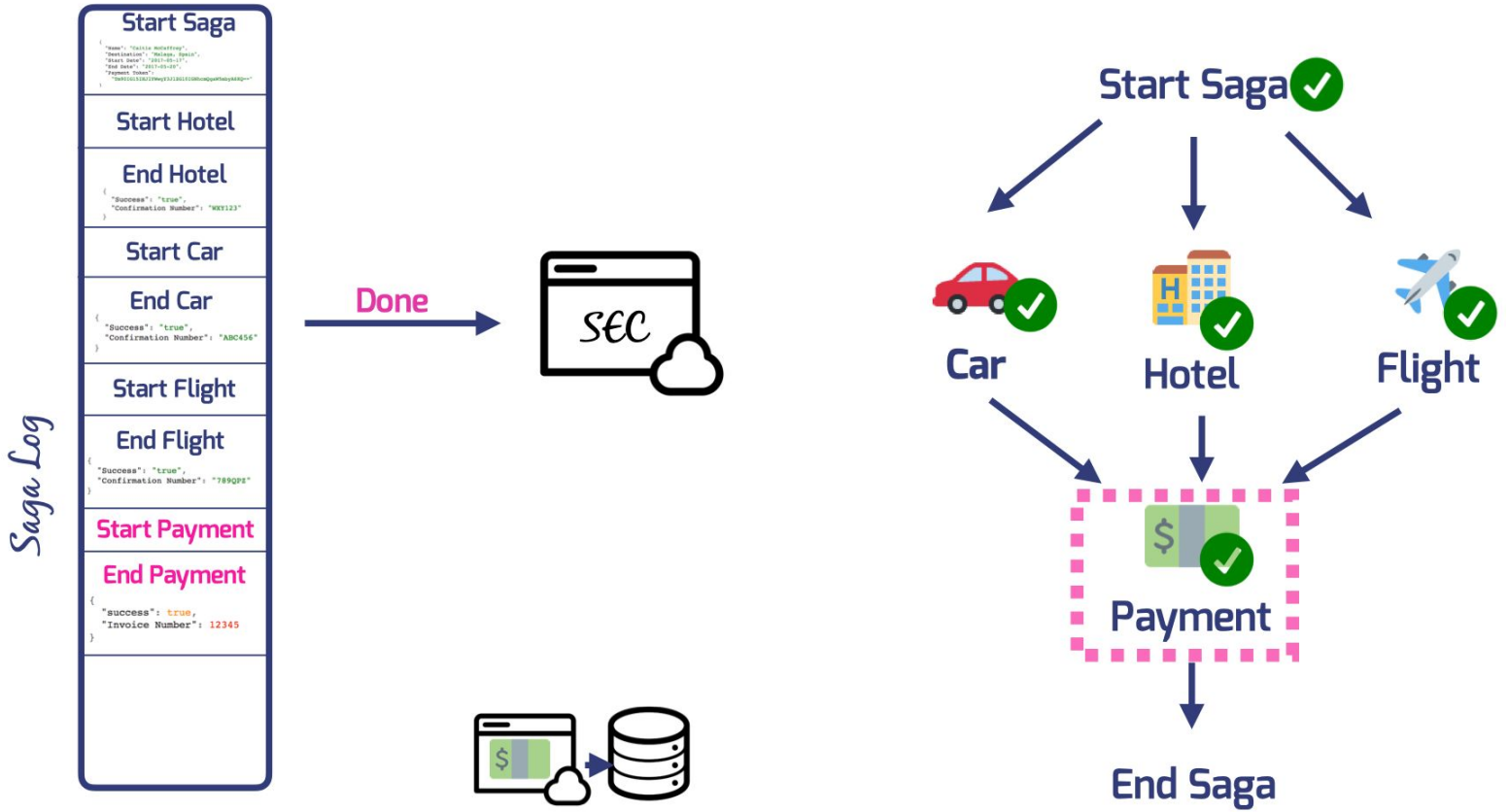
Executing a Distributed Sagas - Case Study



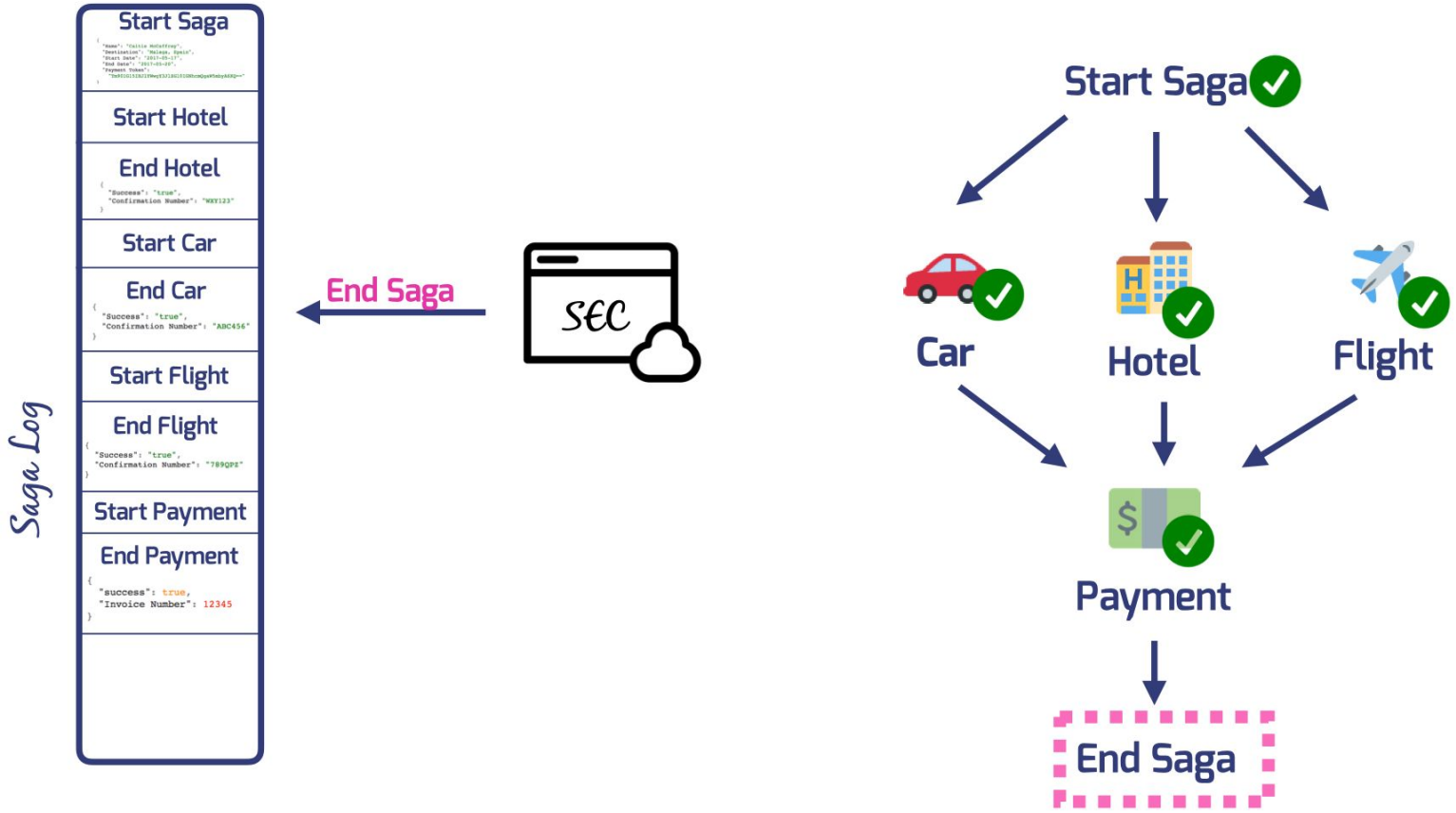




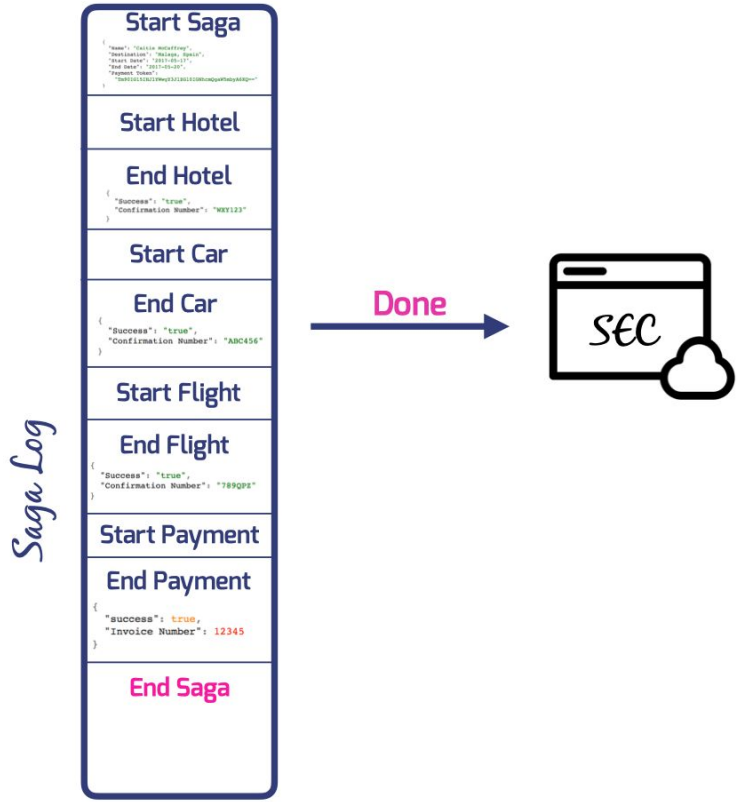
Executing a Distributed Sagas - Case Study



Executing a Distributed Sagas - Case Study

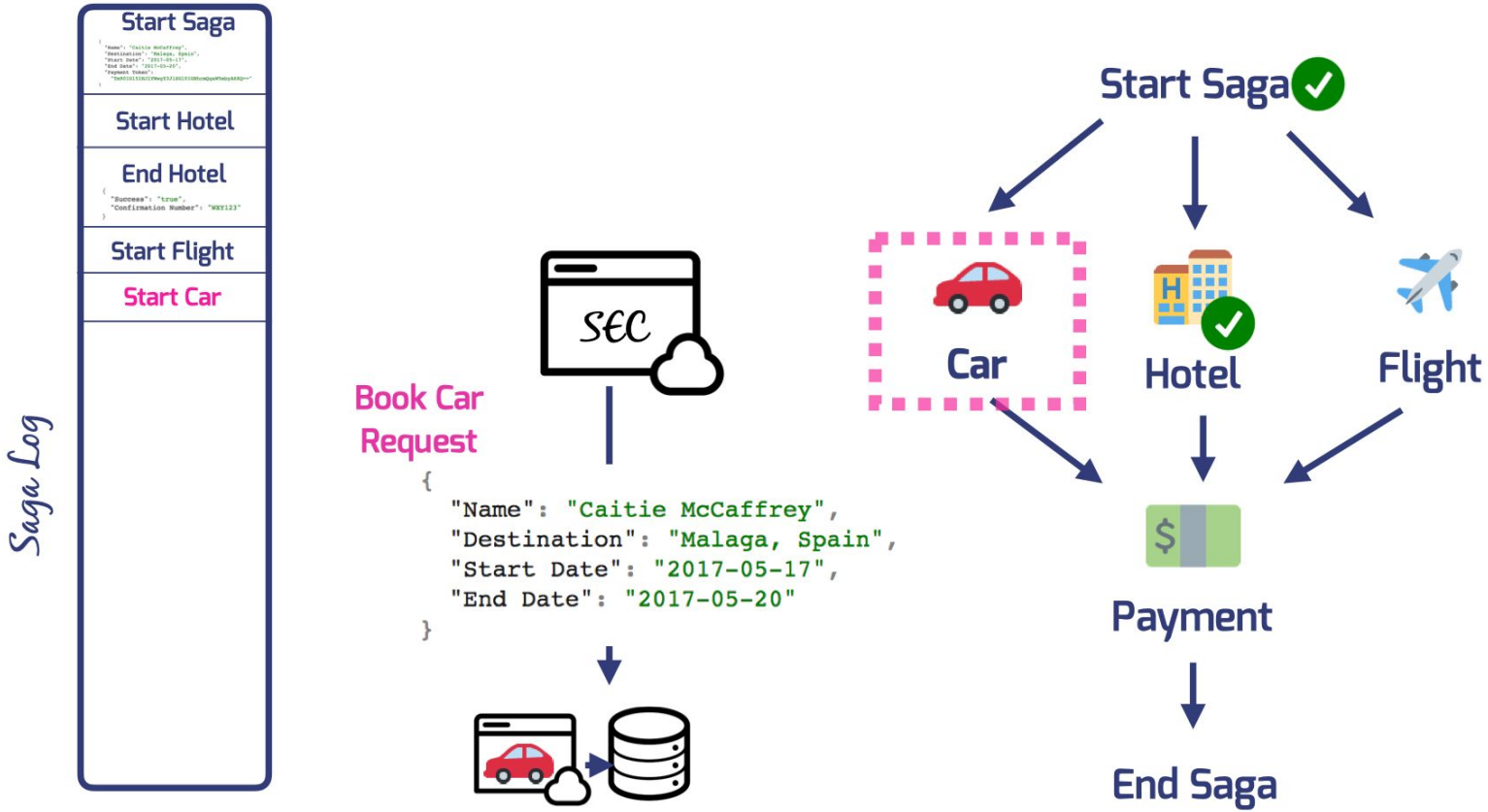


Executing a Distributed Sagas - Case Study

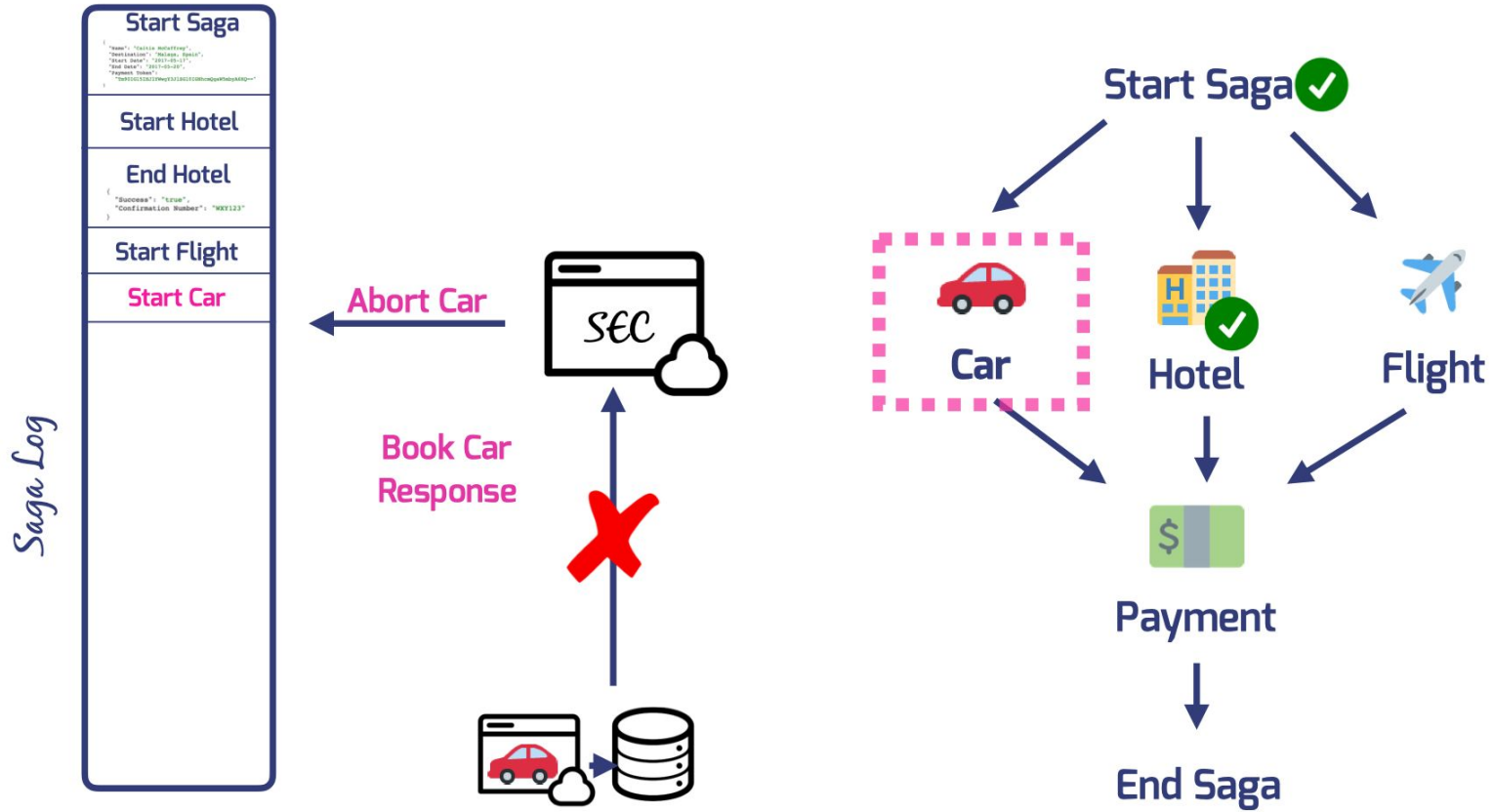


Failure of a Distributed Sagas

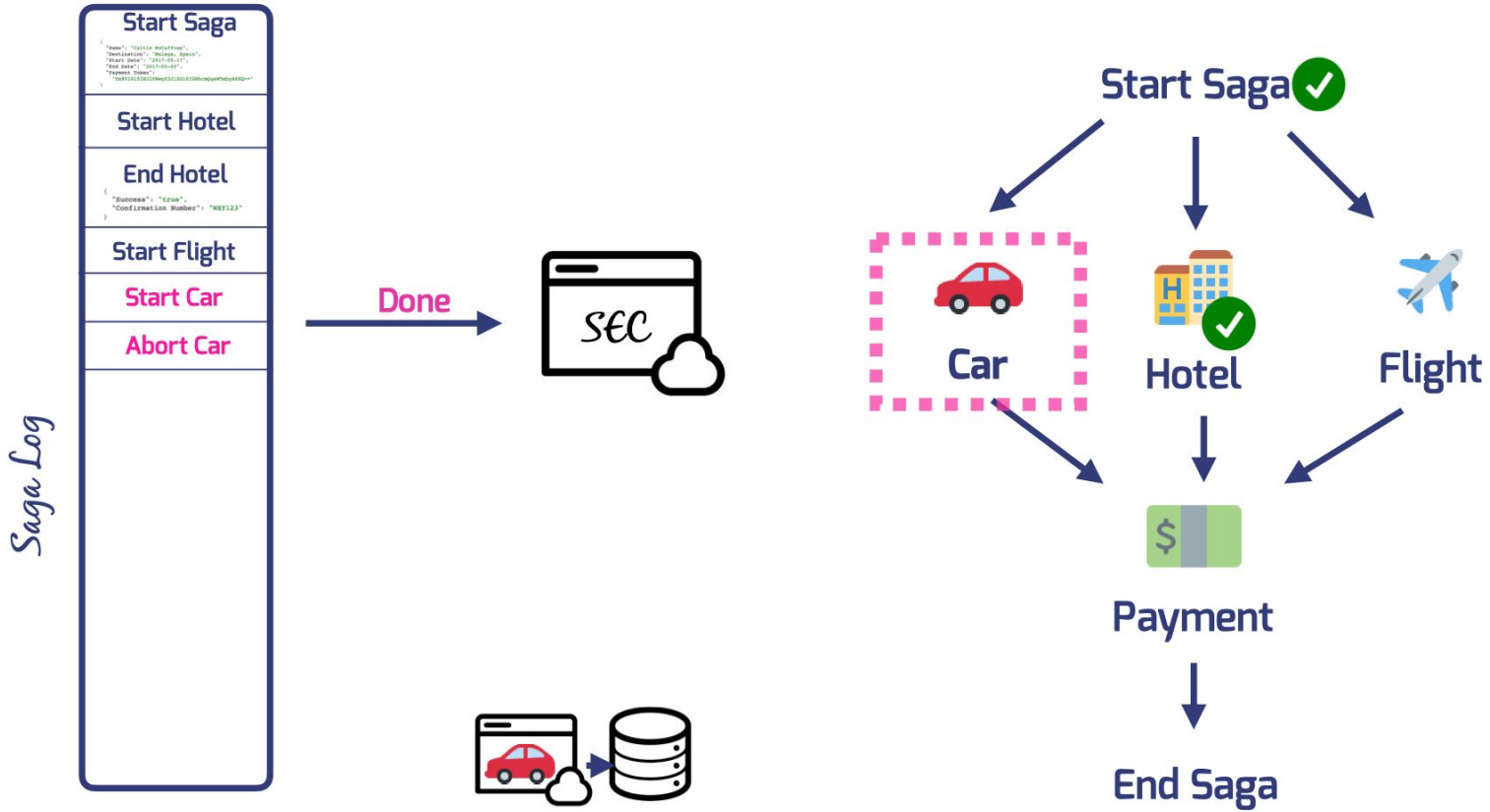
Distributed Sagas - Failure Rollback Recovery



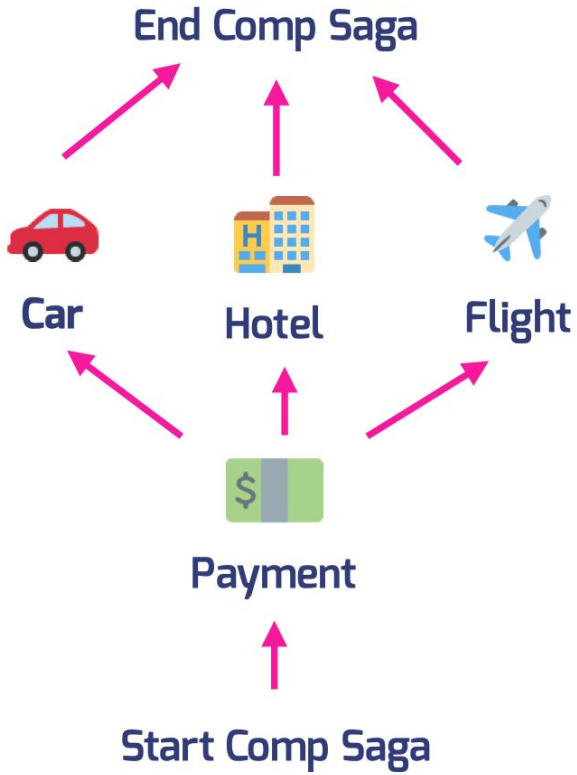
Distributed Sagas - Failure Rollback Recovery



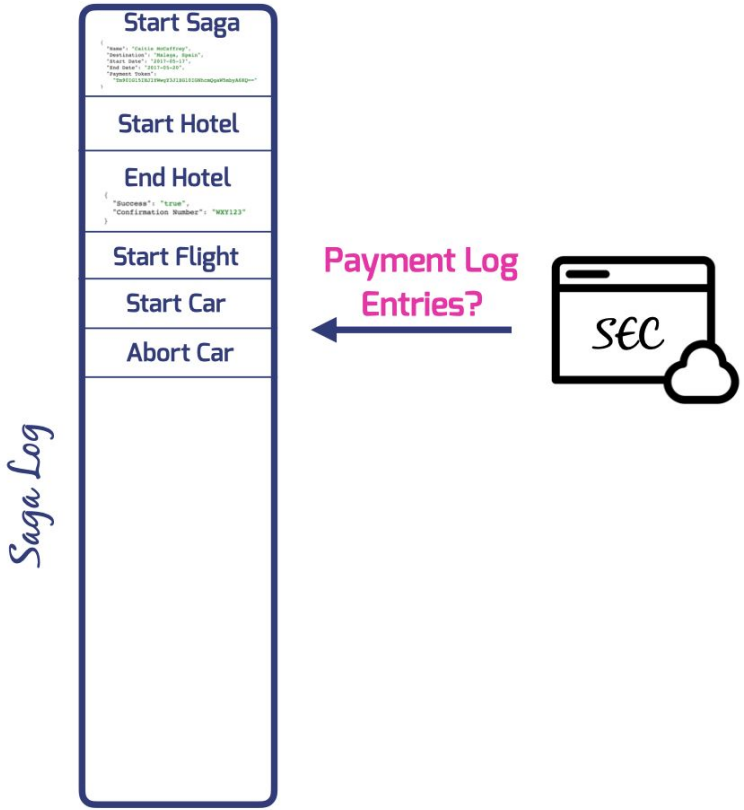
Distributed Sagas - Failure Rollback Recovery

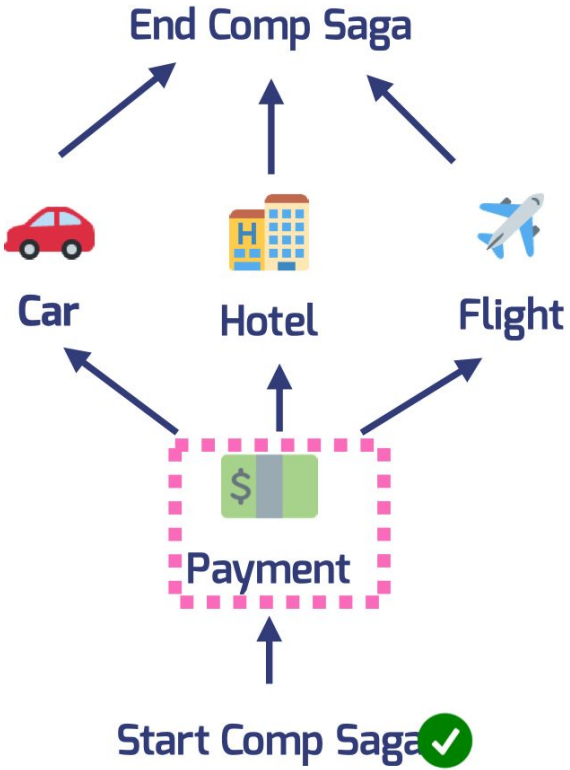


Distributed Sagas - Failure Rollback Recovery

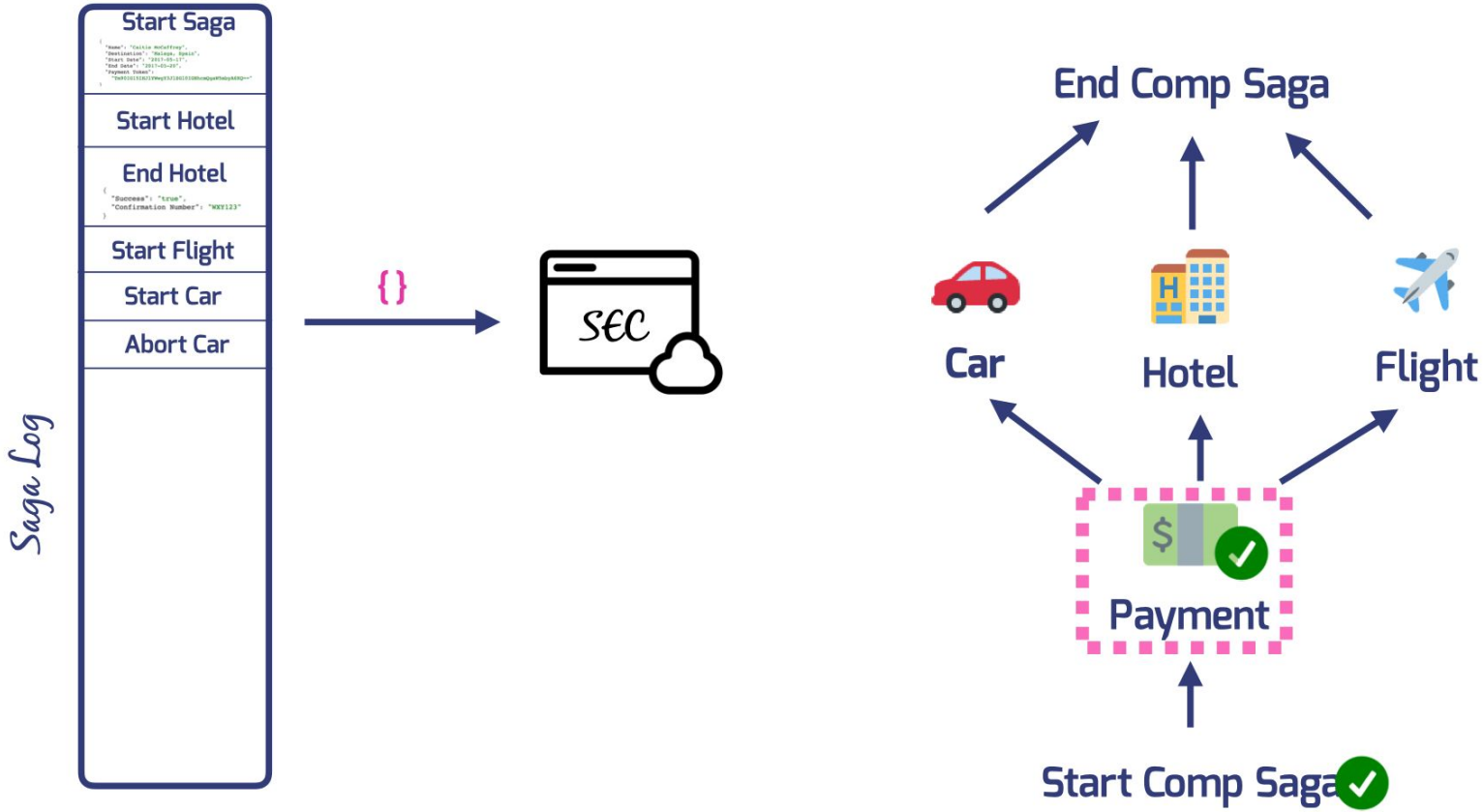


Distributed Sagas - Failure Rollback Recovery

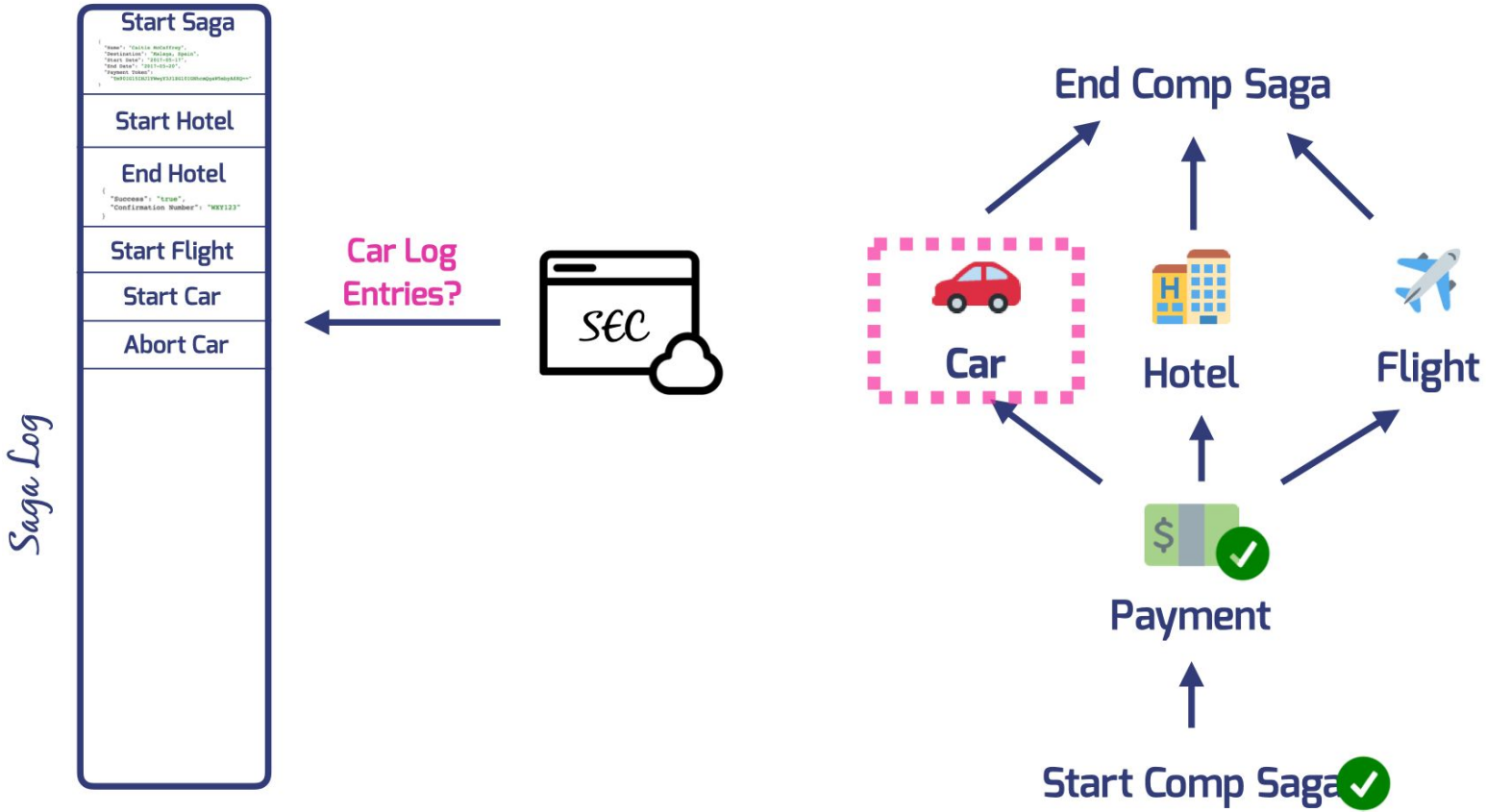




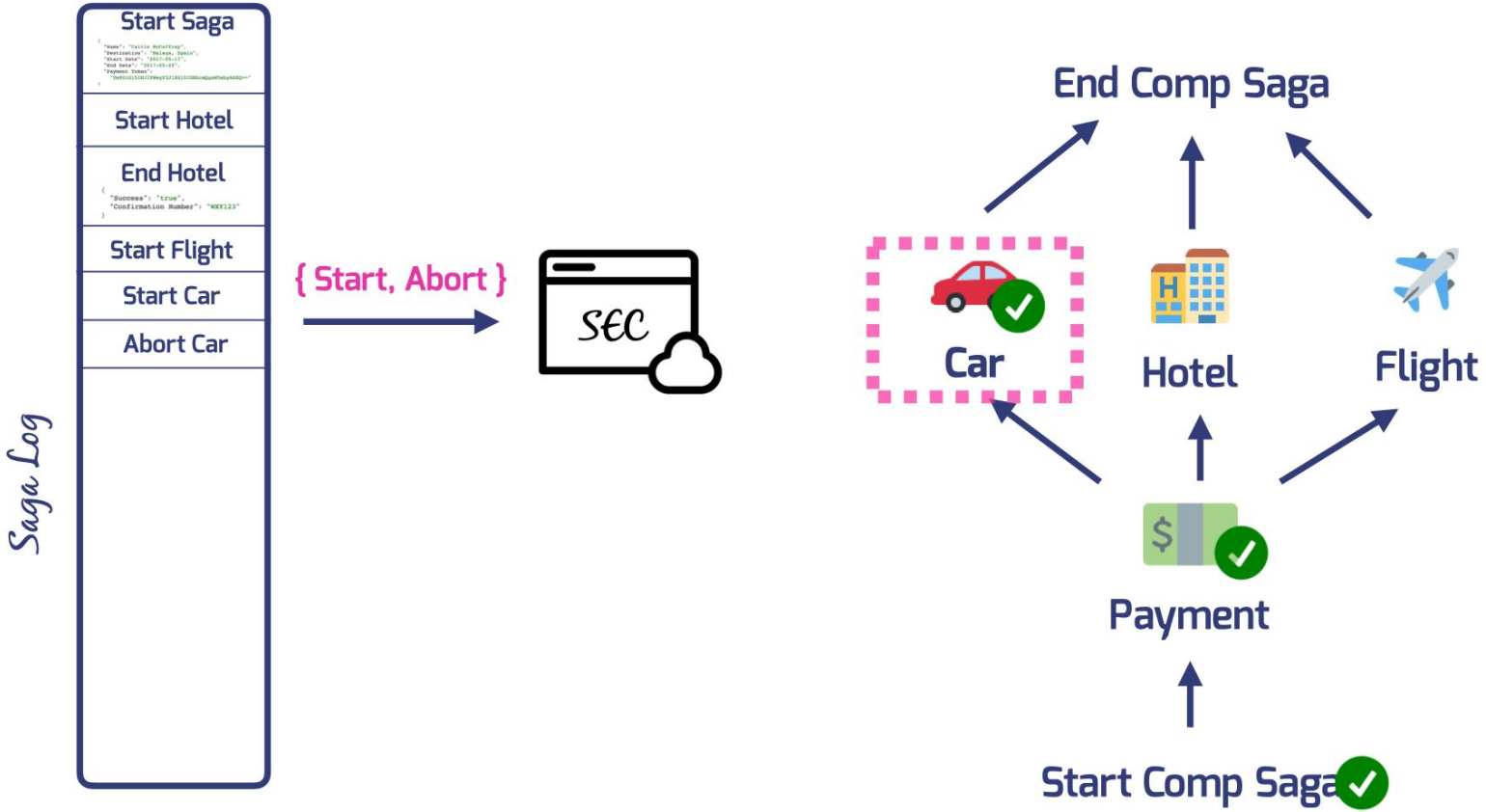
Distributed Sagas - Failure Rollback Recovery



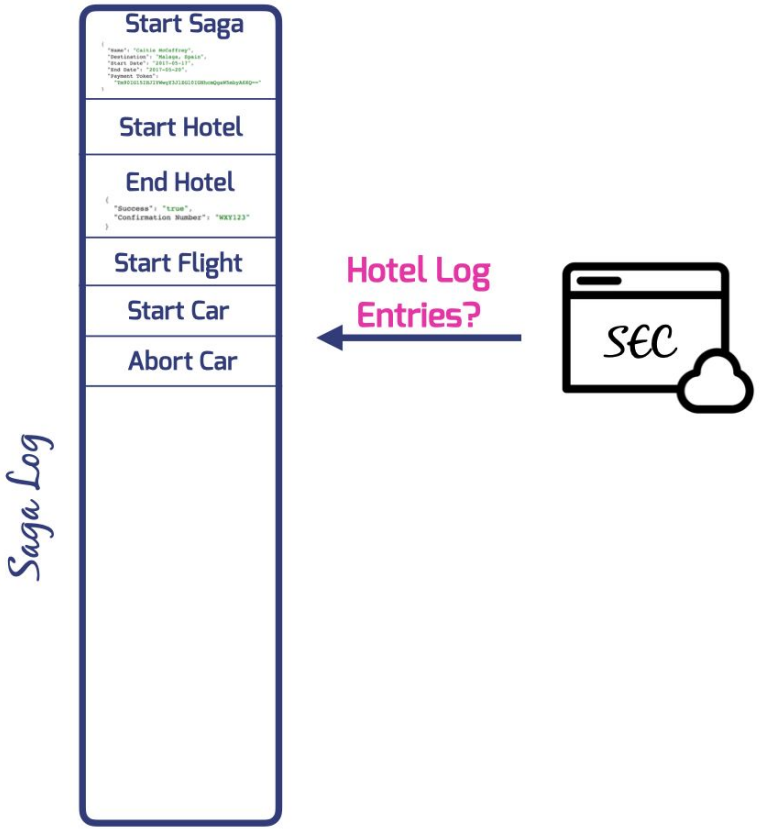
Distributed Sagas - Failure Rollback Recovery

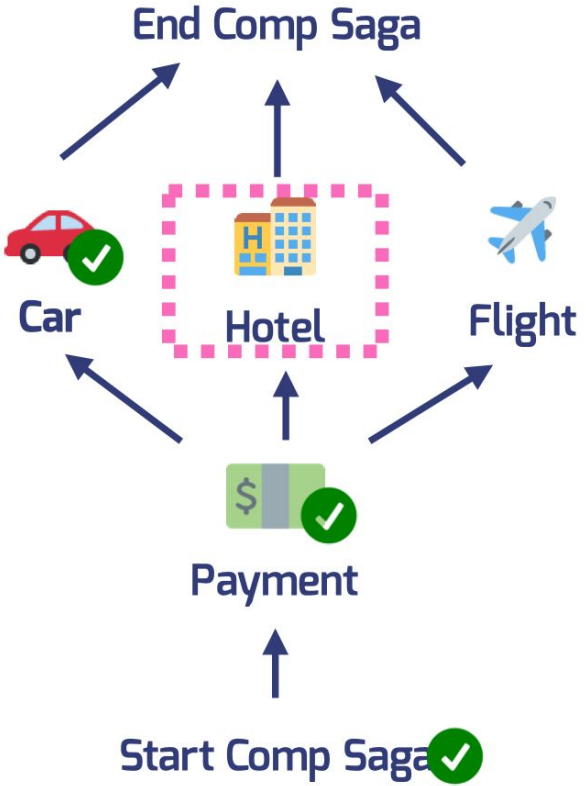


Distributed Sagas - Failure Rollback Recovery

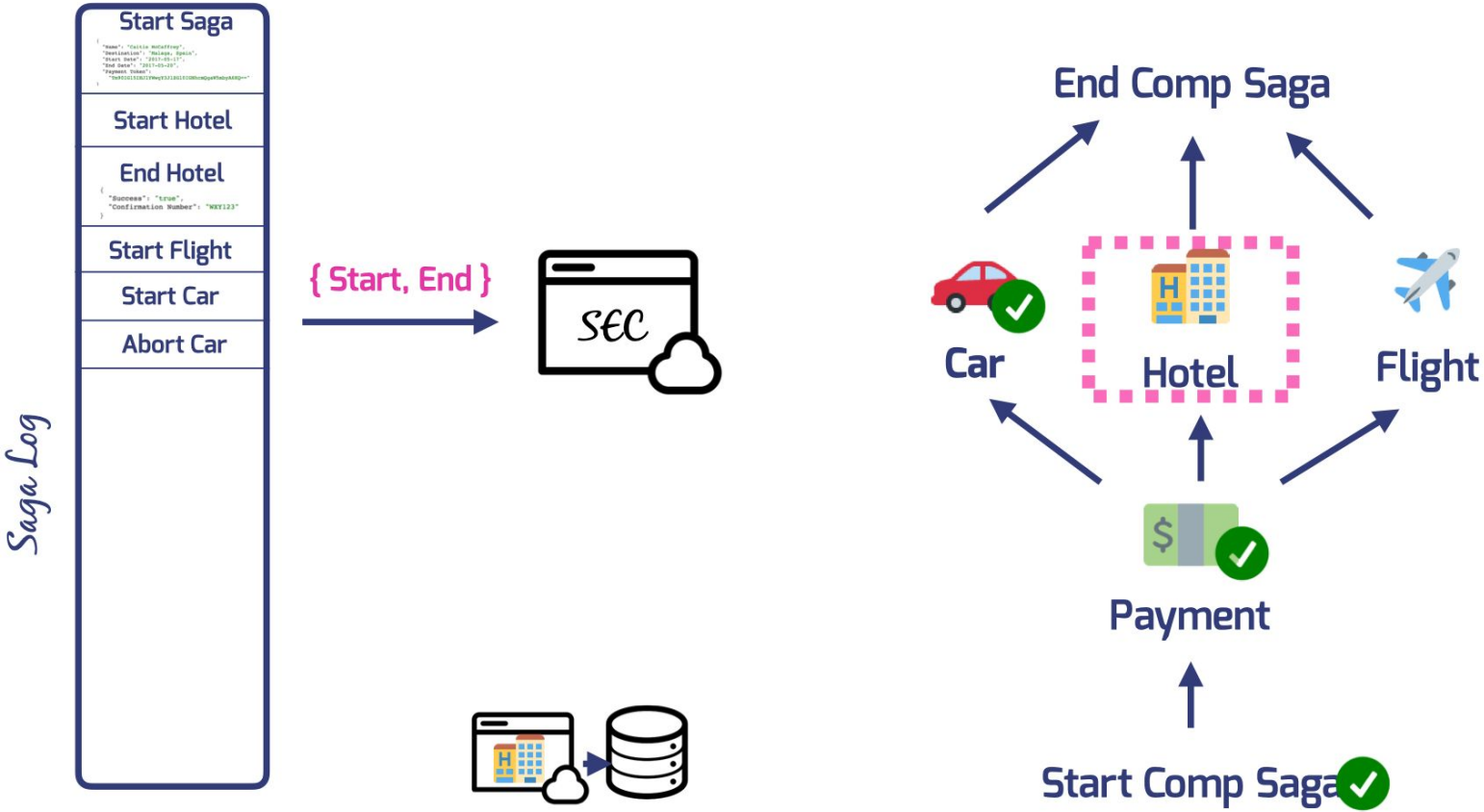


Distributed Sagas - Failure Rollback Recovery

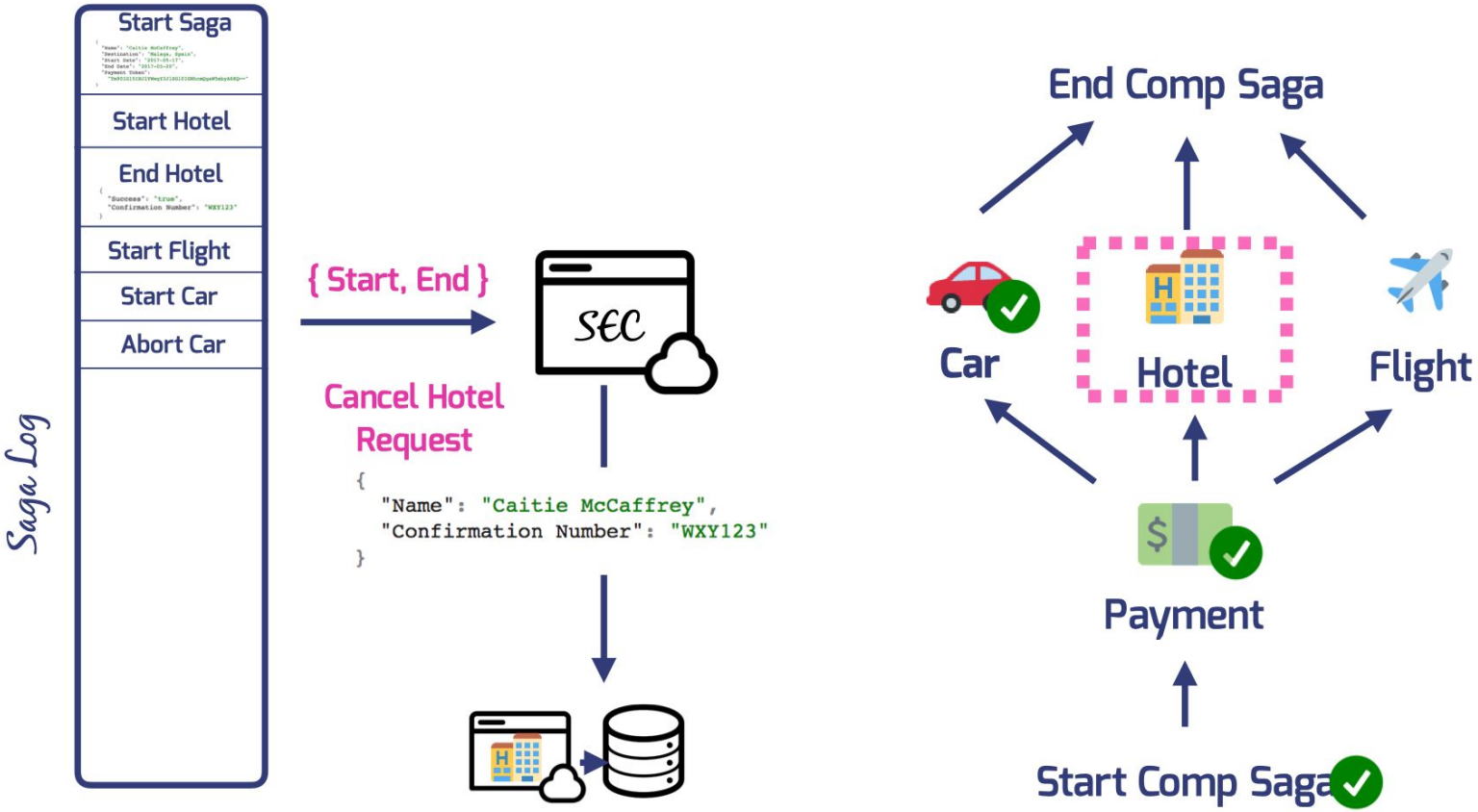




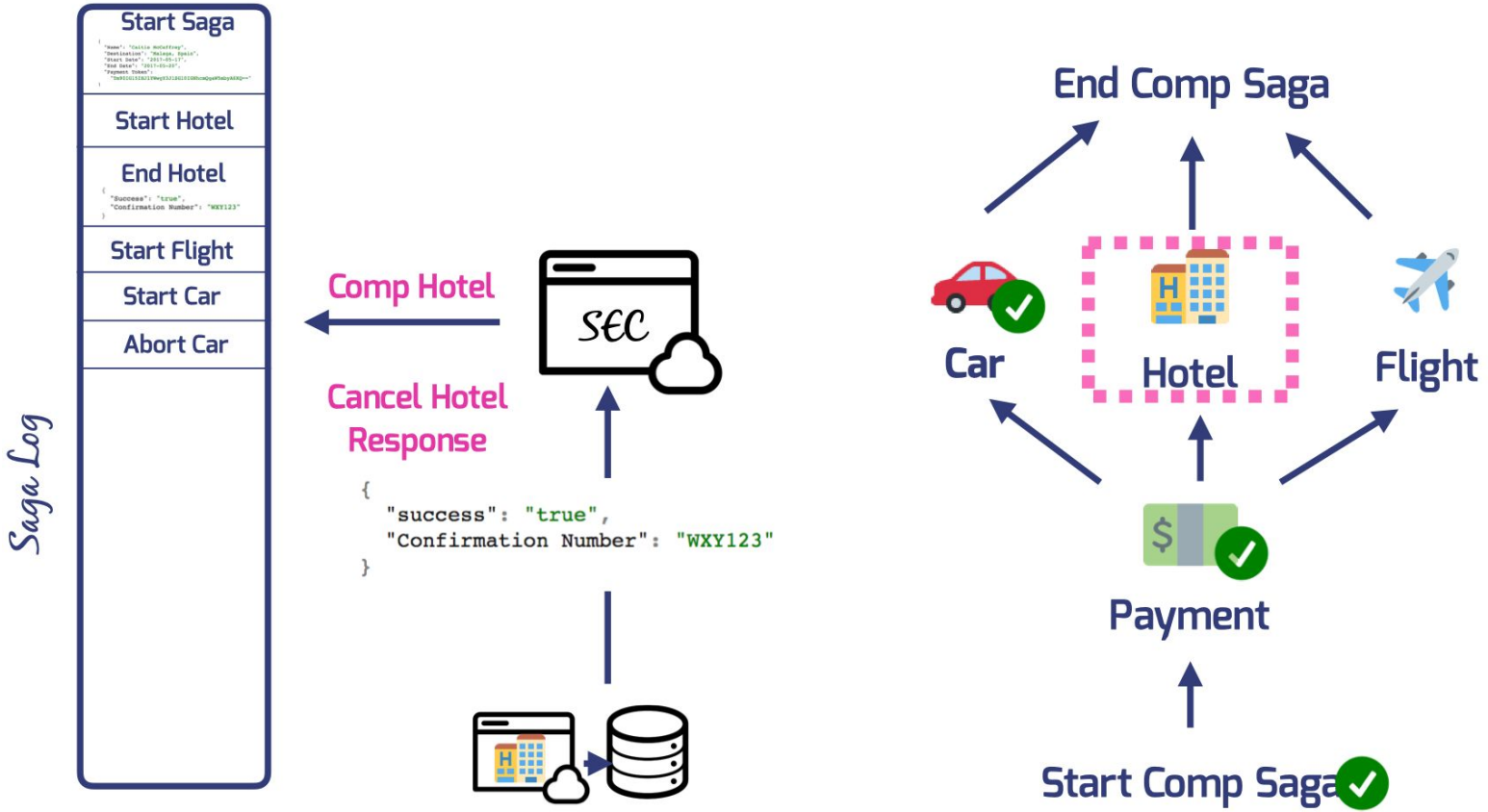
Distributed Sagas - Failure Rollback Recovery



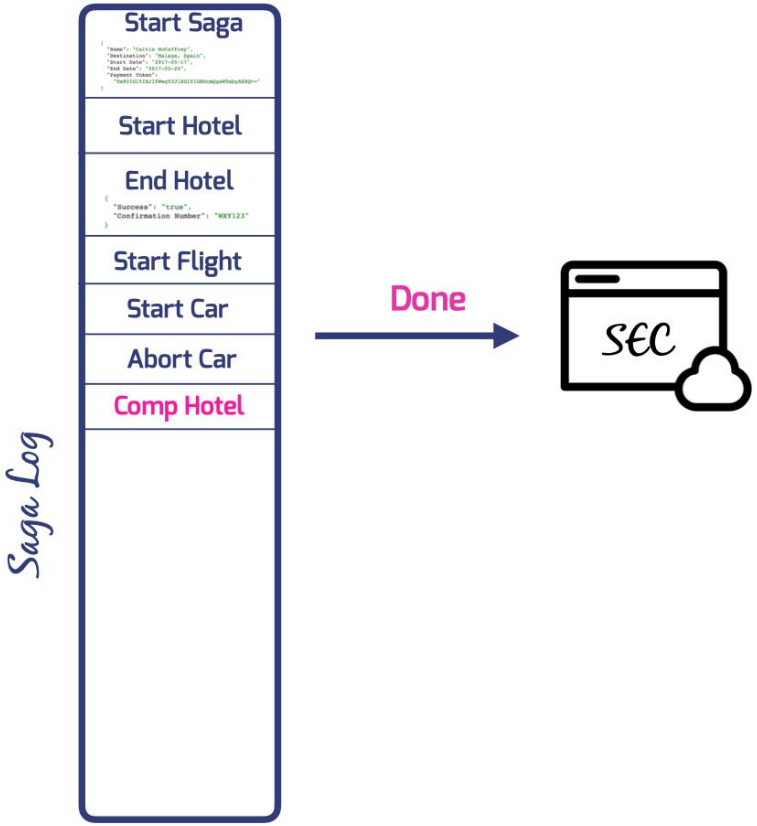
Distributed Sagas - Failure Rollback Recovery

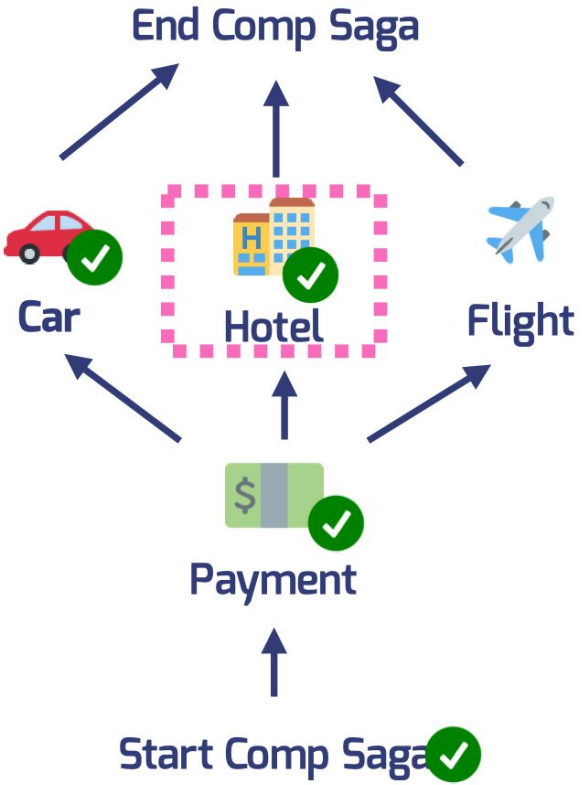


Distributed Sagas - Failure Rollback Recovery

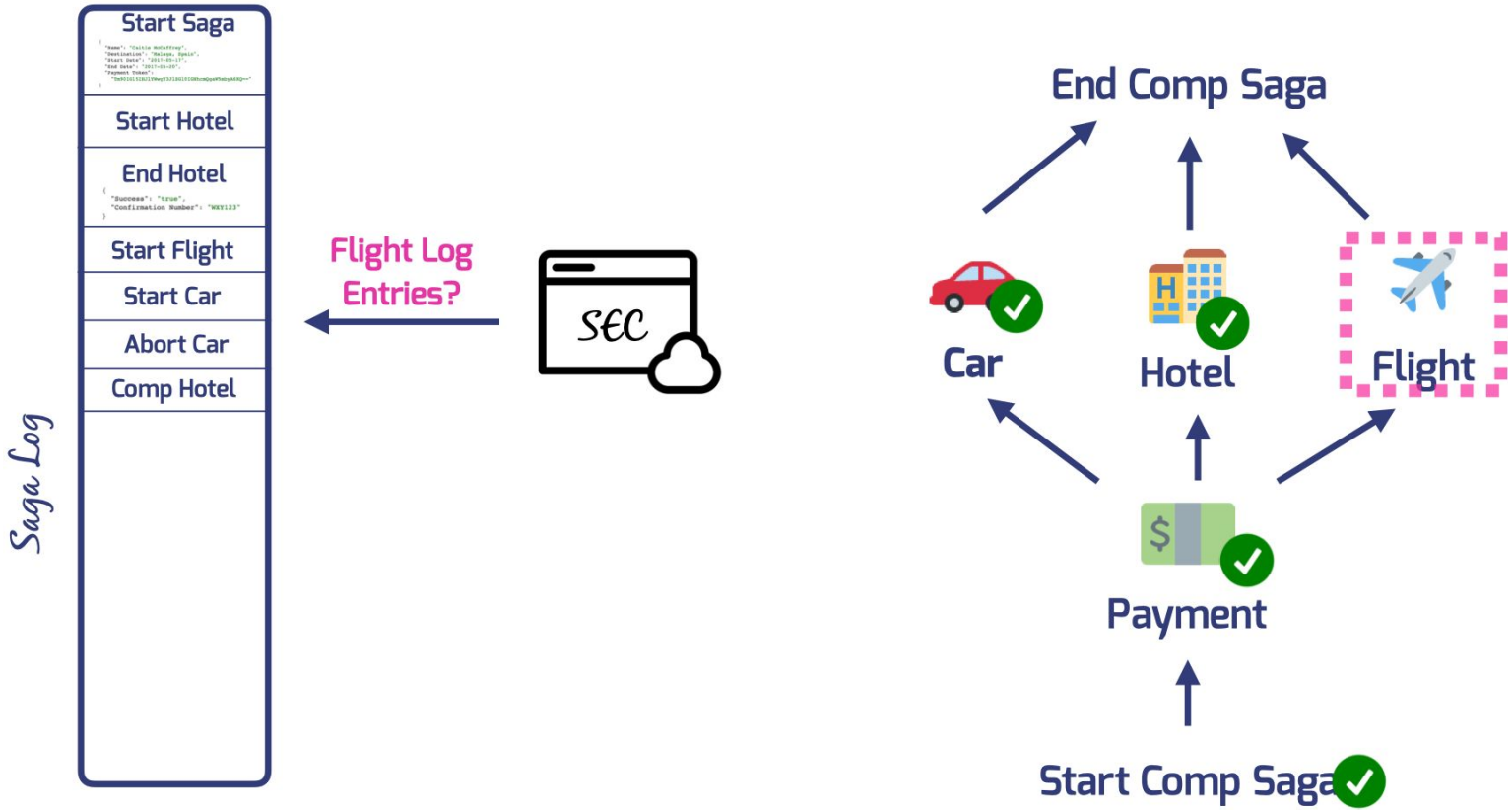


Distributed Sagas - Failure Rollback Recovery

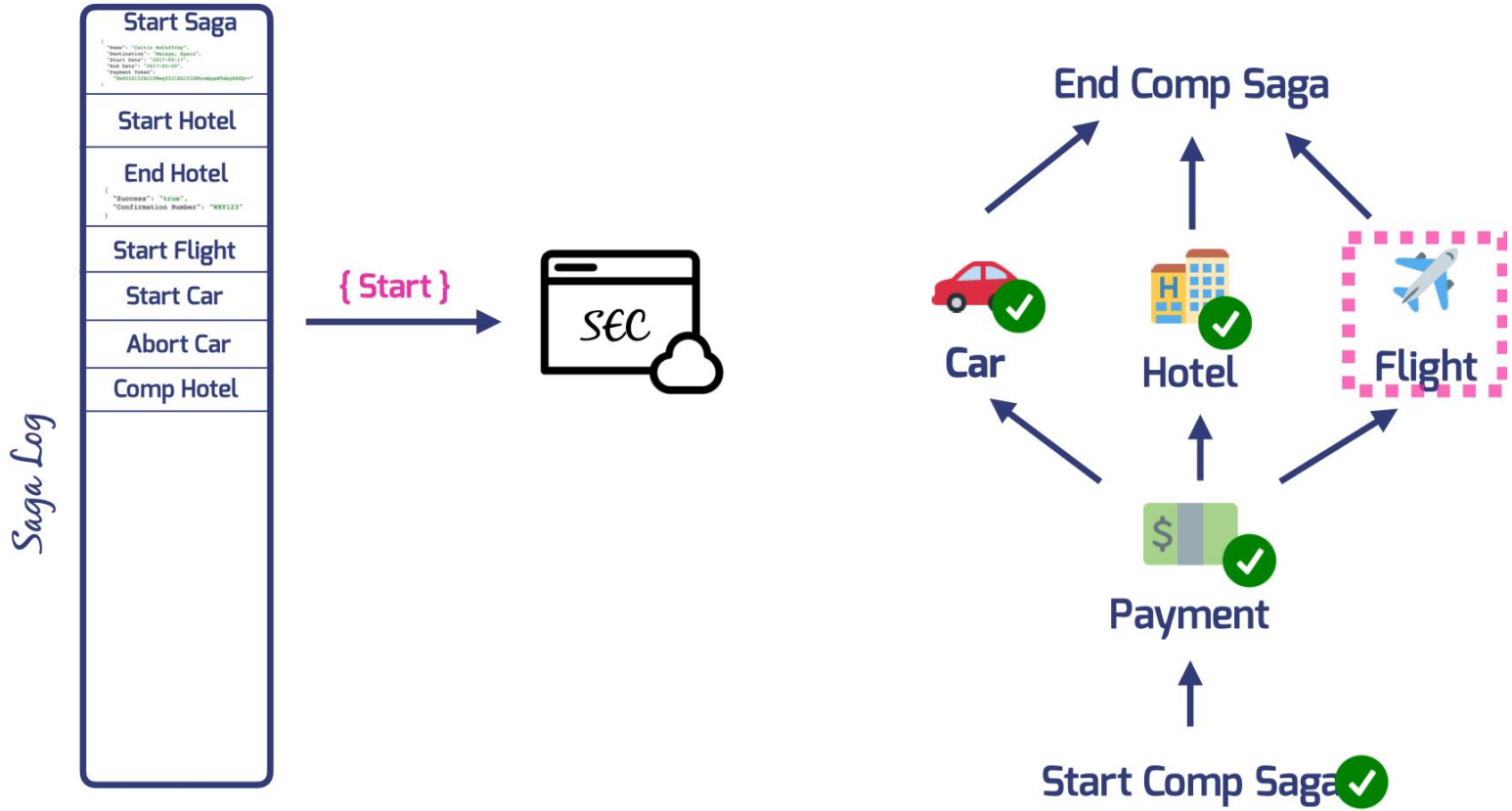




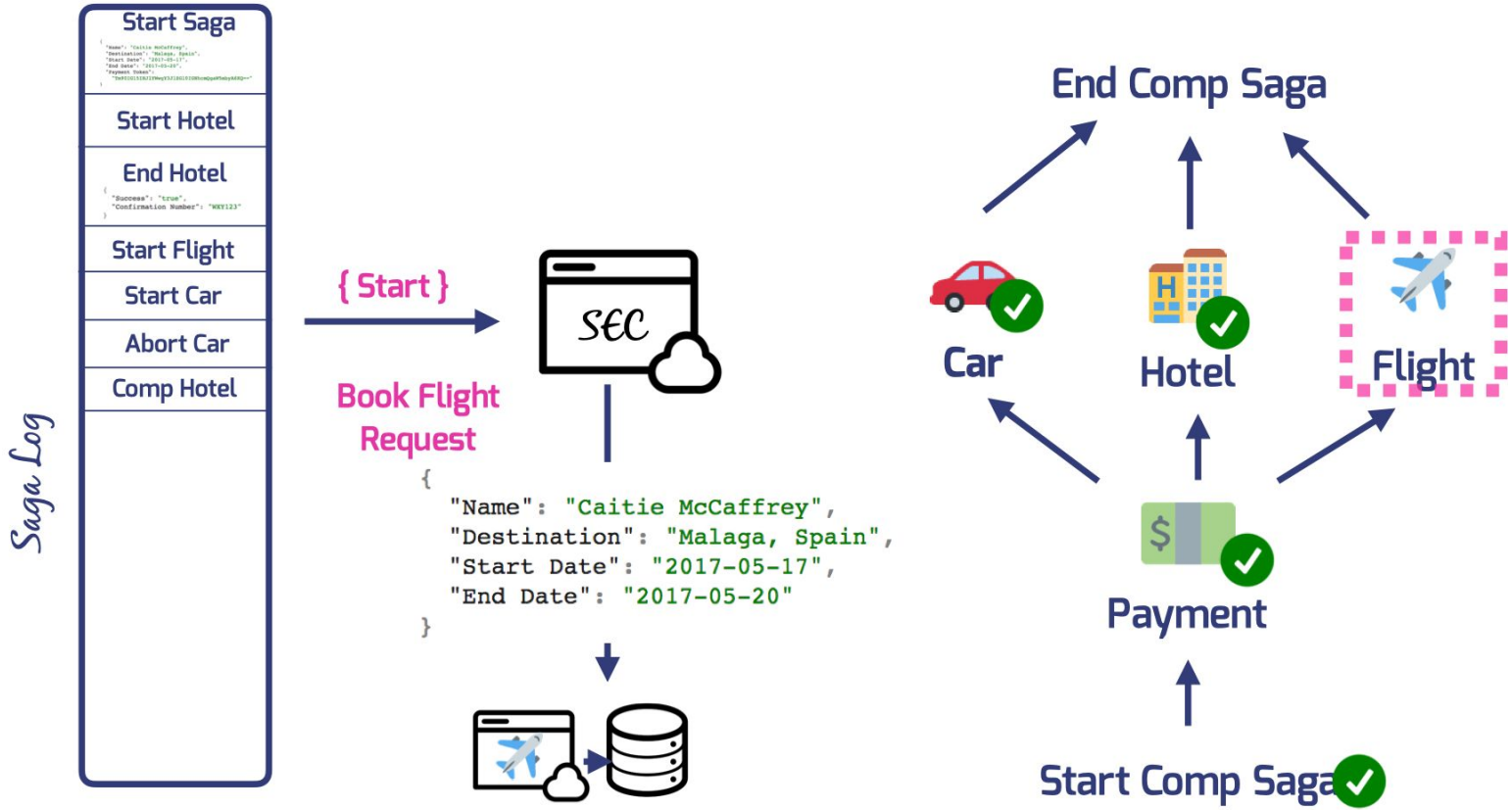
Distributed Sagas - Failure Rollback Recovery



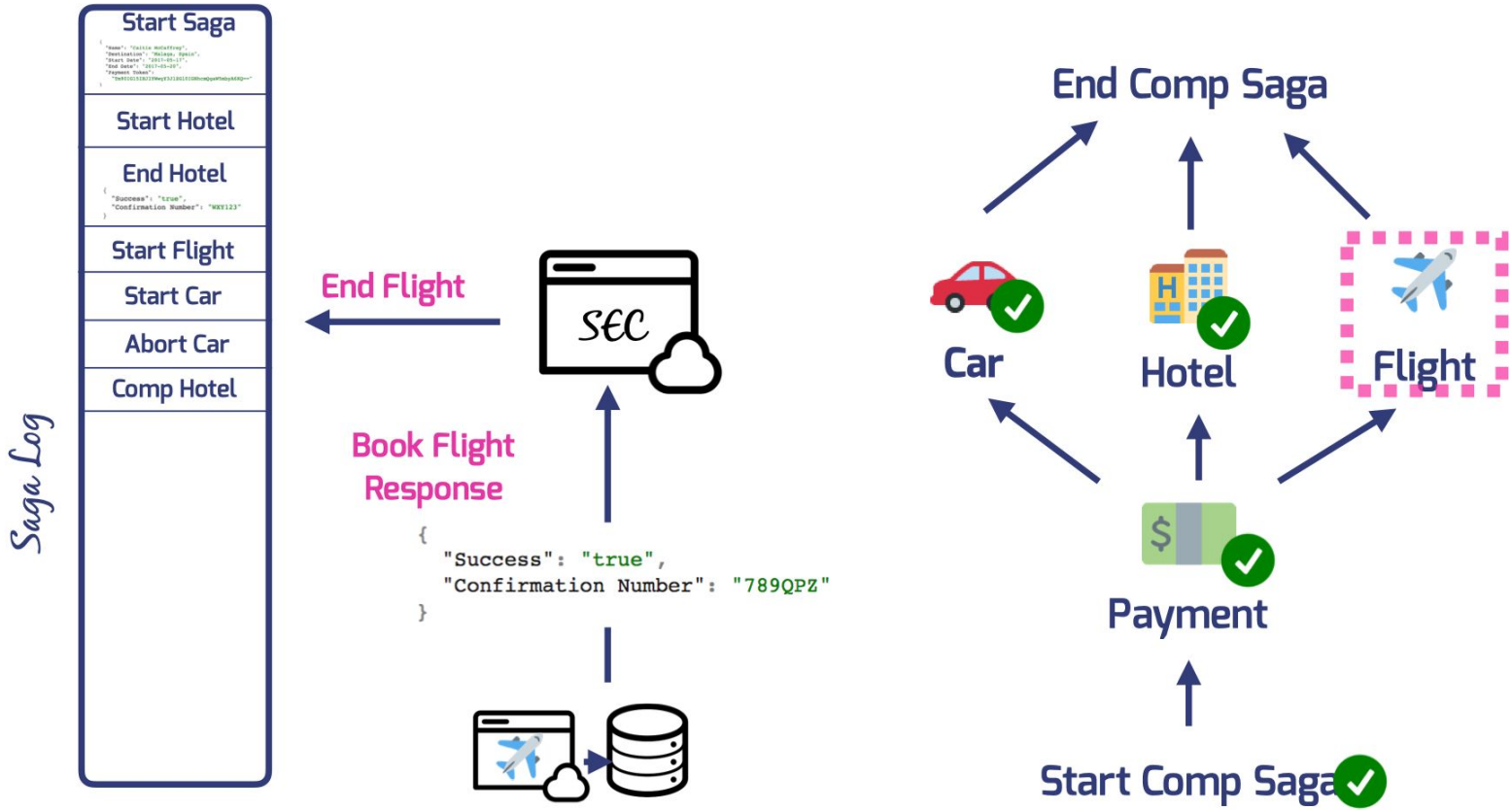
Distributed Sagas - Failure Rollback Recovery



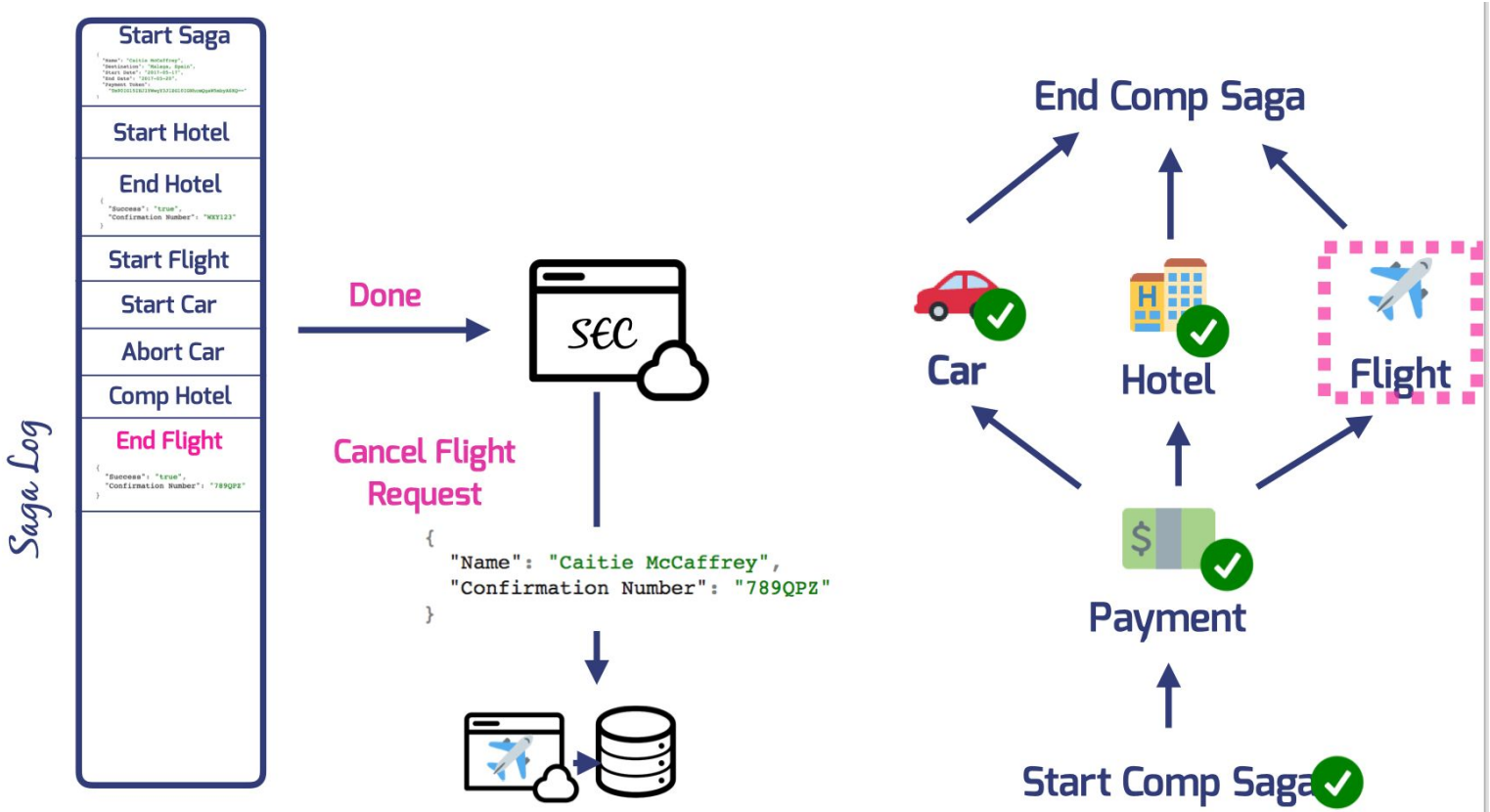
Distributed Sagas - Failure Rollback Recovery



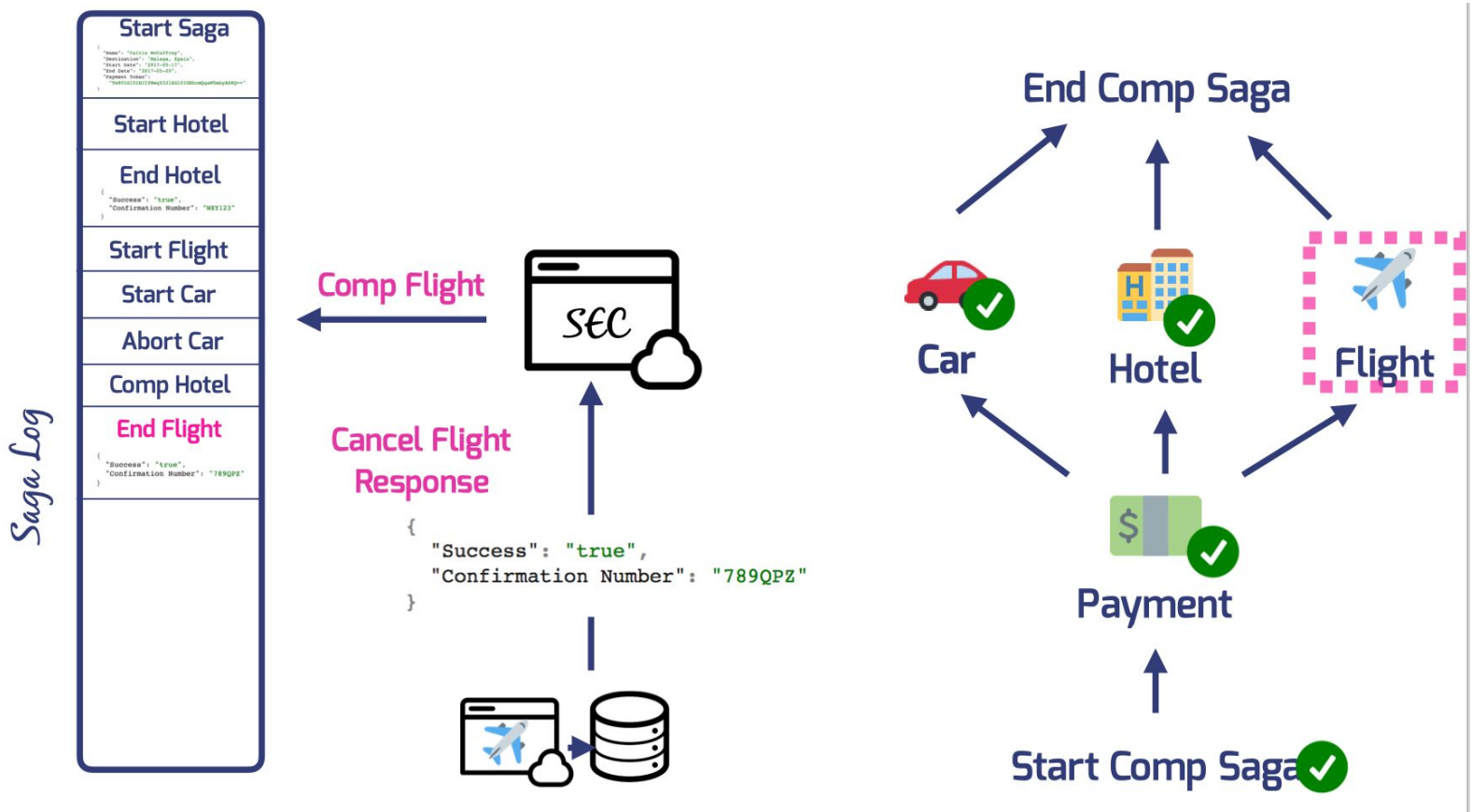
Distributed Sagas - Failure Rollback Recovery



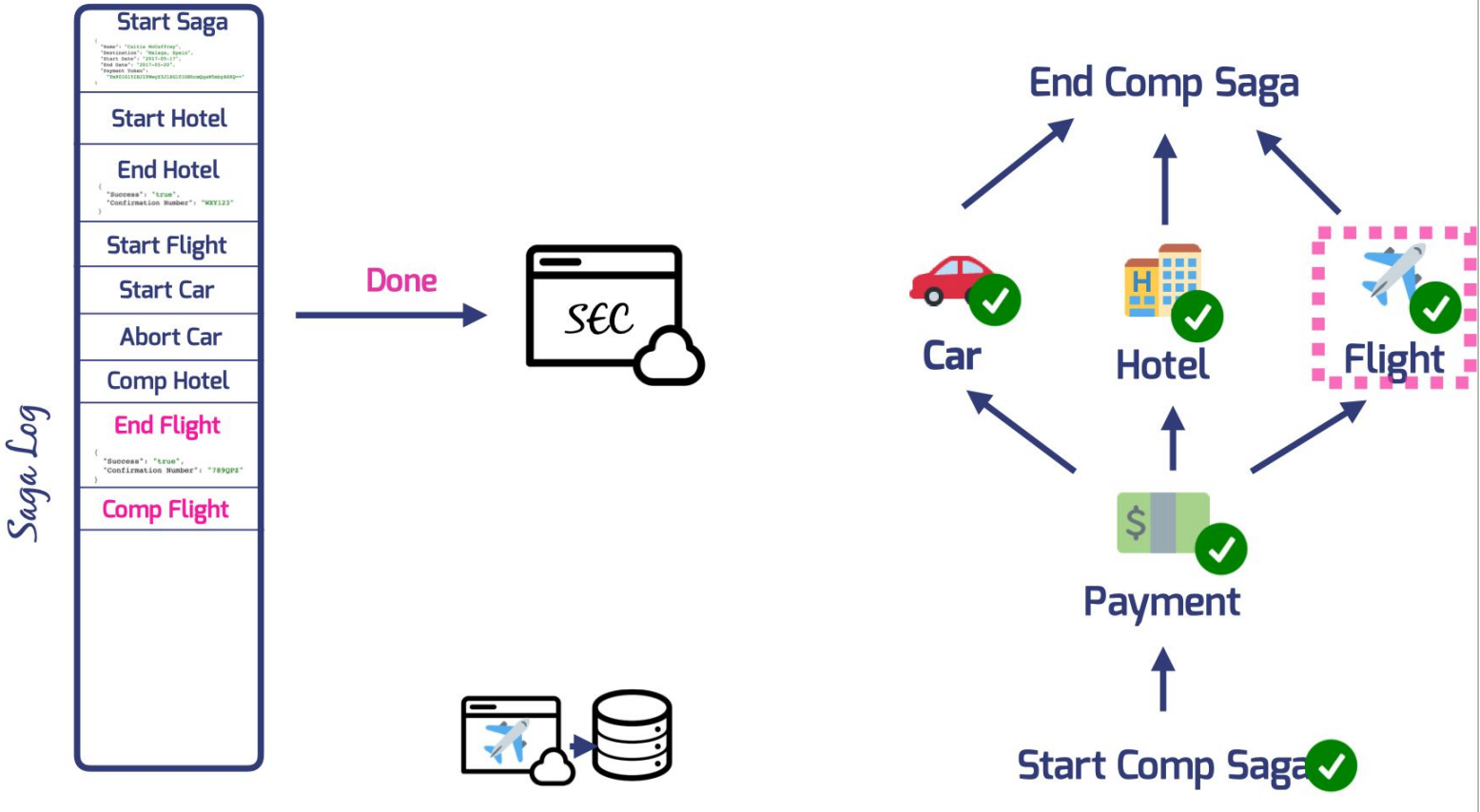
Distributed Sagas - Failure Rollback Recovery



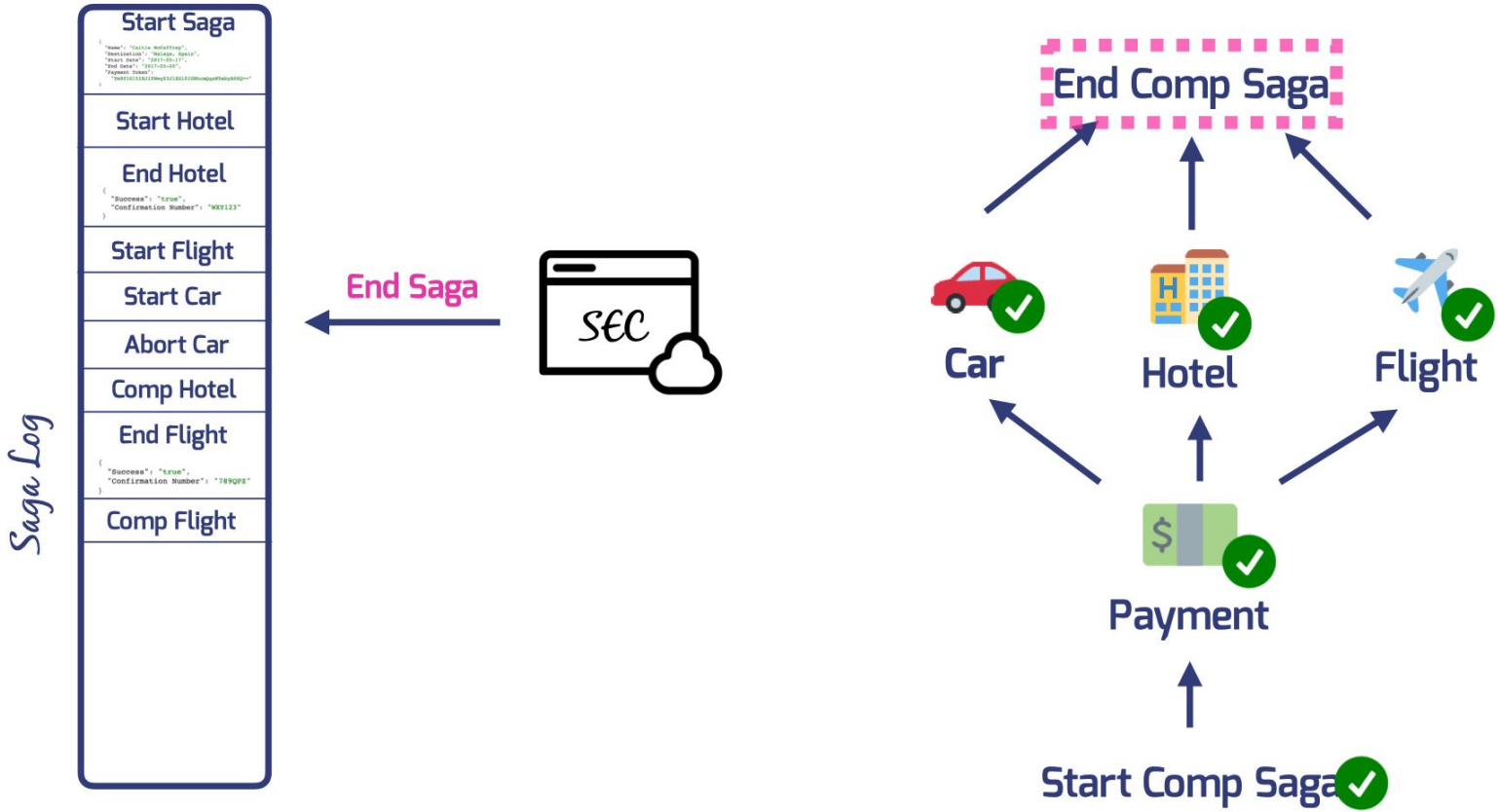
Distributed Sagas - Failure Rollback Recovery



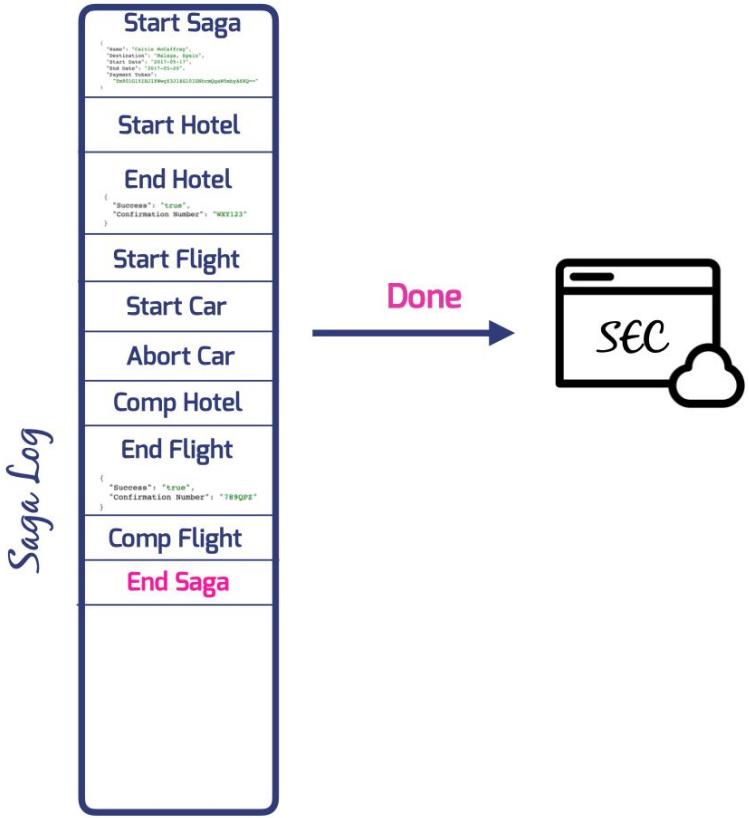
Distributed Sagas - Failure Rollback Recovery

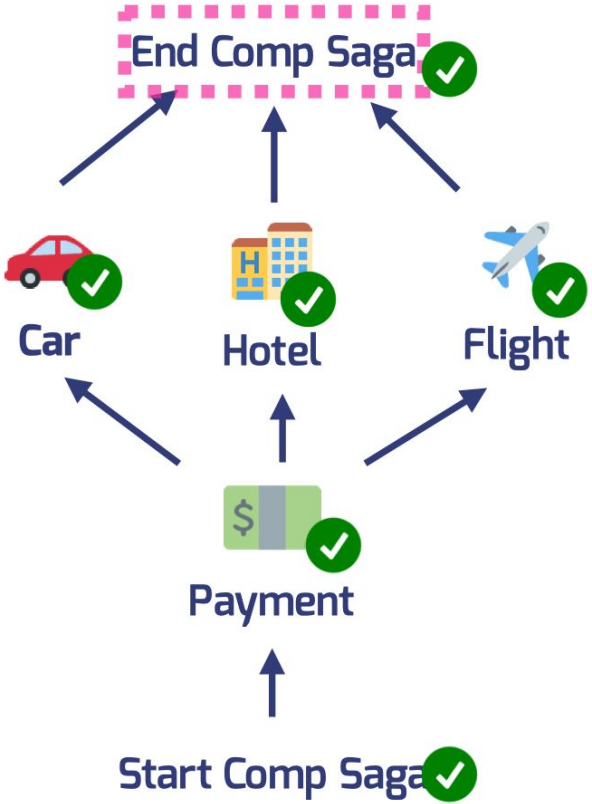


Distributed Sagas - Failure Rollback Recovery



Distributed Sagas - Failure Rollback Recovery





Future direction

- Provide isolation
- Handle the failure of compensating request
- Provide debugging tool for saga pattern

Q&A

- How is isolation achieved in Saga?
- How does distributed saga implement compensating request?
- How does distributed saga handle coordinator failing?
- What if compensating transaction failed?

Reference

https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga

https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/

https://medium.com/@ijayakantha/microservices-the-saga-pattern-for-distributed-transactions-c489d0ac0247

https://yos.io/2017/10/30/distributed-sagas/

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