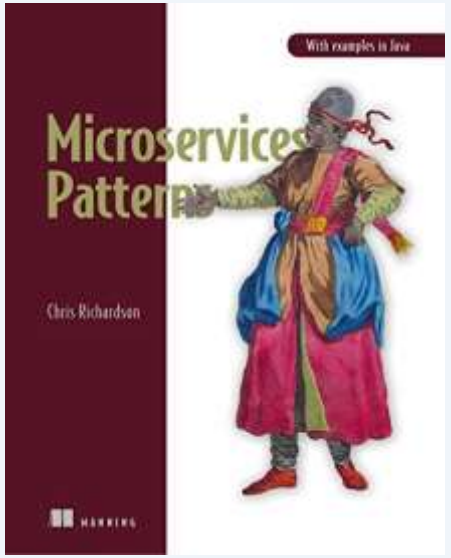
Transaction Management

The SAGA Pattern

Objectives

- What do we need from transaction infrastructure?
- Saga communication
- Saga coordination
  - Choreography
  - Orchestration
  - Saga as a state machine



Transactions

- In some sense, *transactions* form the heart of our service.
  - We take data in the form of a client request then transact with the client on the basis of that request.

- *Data consistency* essentially means that there is a single, consistent ground truth at any moment in time for any given request, at any point of our system.
  - We don't want one service dispatching an order that another service has cancelled.

Transactions in a Monolithic Architecture

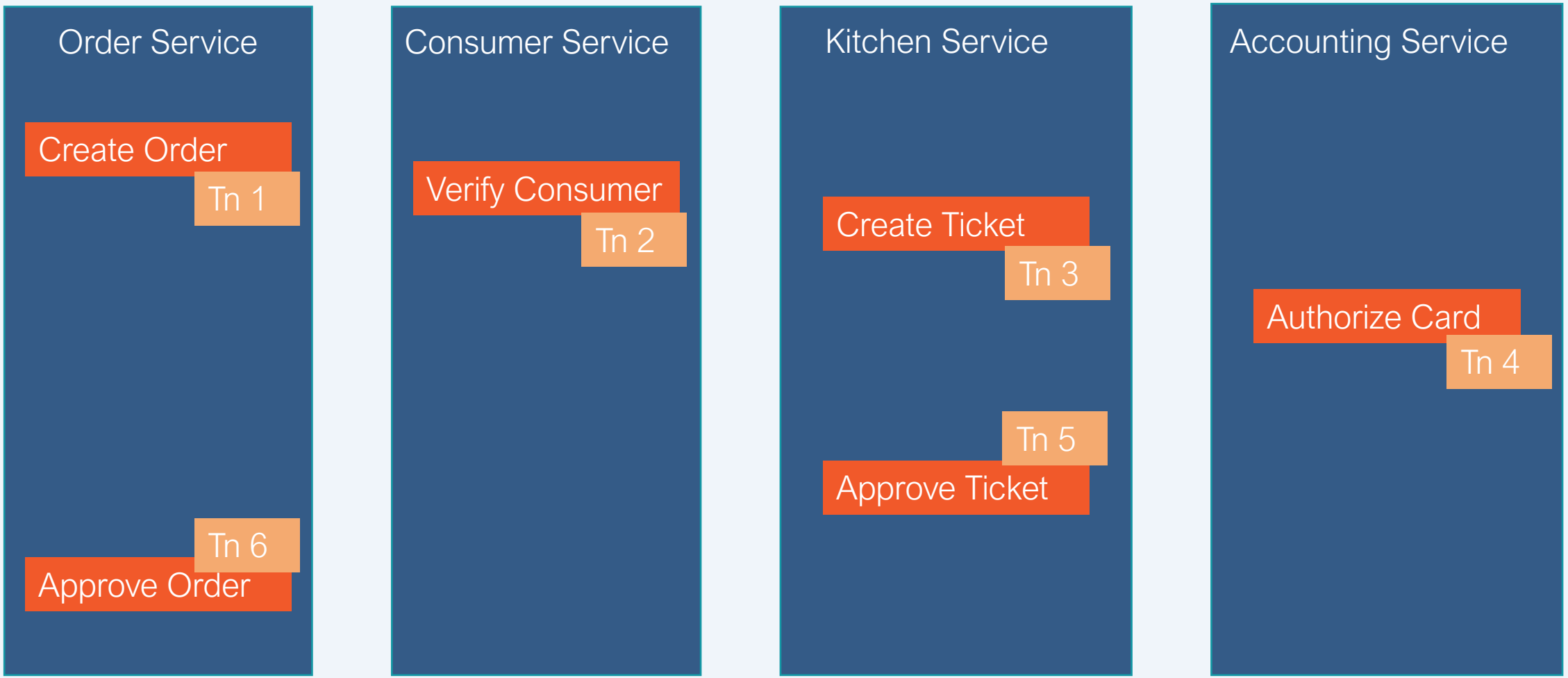
- Once again, this is greatly simplified in the monolithic architecture pattern!

- ACID transactions are enforced throughout, which is easy to do when every module accesses the same database!
  - **Atomicity** - each statement in a transaction (to read, write, update or delete data) is treated as a single unit. Either the entire statement is executed, or none of it is executed.
  - **Consistency** - ensures that transactions only make changes to tables in predefined, predictable ways. Transactional consistency ensures
  - **Isolation** - when multiple users are reading and writing from the same table all at once, isolation of their transactions ensures that the concurrent transactions don't interfere with or affect one another. Each request can occur as though they were occurring one by one, even though they're actually occurring simultaneously.
  - **Durability** - ensures that changes to your data made by successfully executed transactions will be saved, even in the event of system failure.

Transactions in a Microservices Architecture

- A transaction that occurs wholly within one service can continue to follow the ACID transaction requirement.
- However transactions that span services cannot be isolated by nature.
  - We lose this facility by enforcing the Database per service pattern.
- One pattern to address this is the Saga pattern.
  - A saga is a sequence of local transactions, each of which updates data within a single service.

Create Order Example Saga



Saga communication

- Sagas use asynchronous messaging
- A service publishes a message when a local transaction completes
- The message then triggers the next step in the saga.

- This ensures
  - The saga participants are loosely coupled
  - The saga completes
    - Even if a recipient of a message is temporarily unavailable, the message broker buffers the message until it can be delivered.

What could possibly go wrong?

- Firstly, ACID transactions have a simple rollback mechanism in the case a transaction fail.

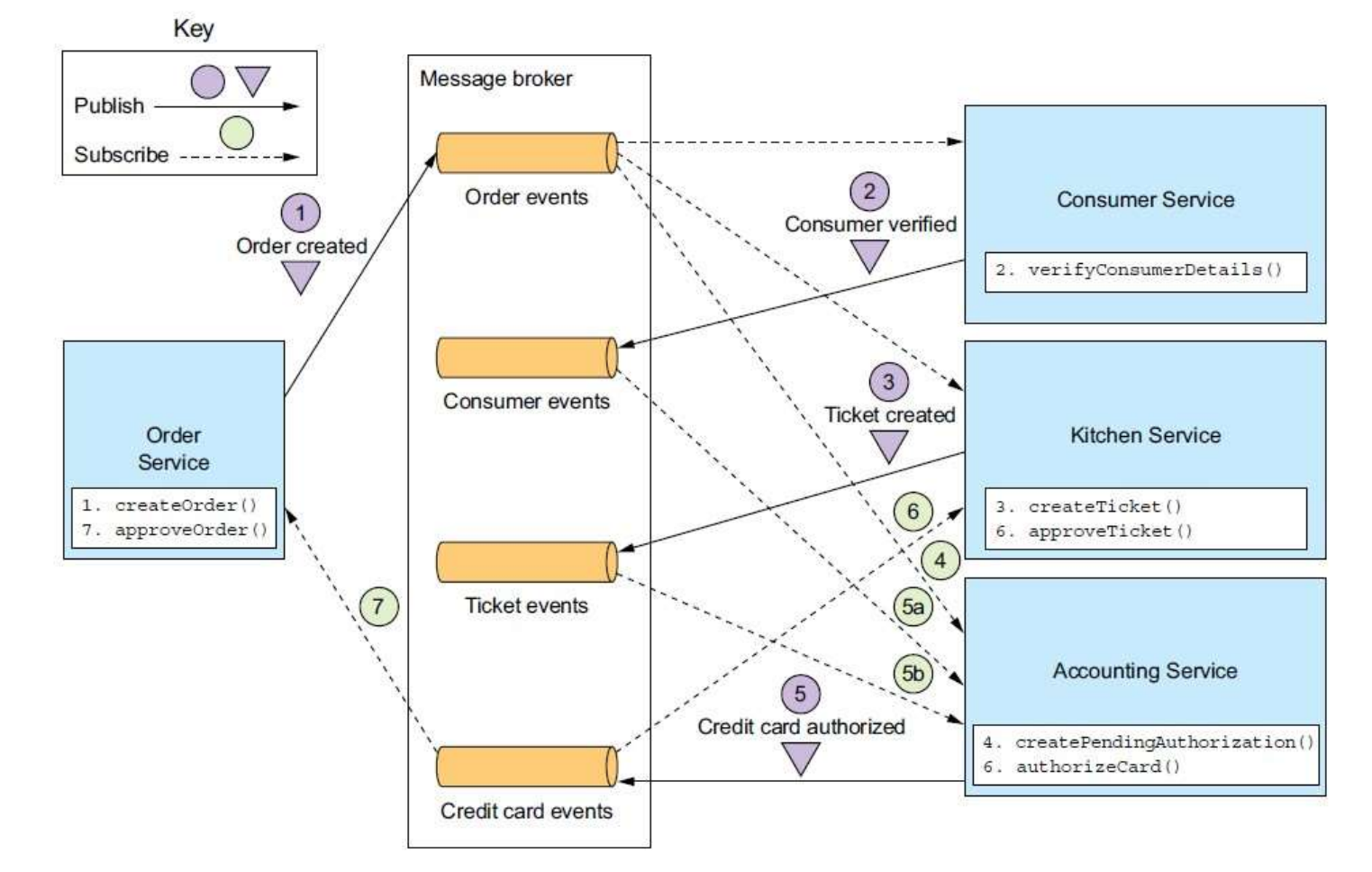
- Sagas don't have a simple rollback
  - Each step in the sequence of transactions can make changes to a local database
  - All of these changes need to be undone in the event of a transaction failure.
- These are called *compensating transactions*.

- Suppose T_1 … T_n succeed and T_{n+1} fails, then we must implement compensating transactions in the reverse order
  - C_n … C_1
  - Same messaging mechanism as the forward transactions.

Coordinating Sagas

- As we have seen, a saga initiates when the system logic selects the first saga participant to commence the system command (e.g. create order)
- Once the first transaction completes the saga's coordination invokes the next saga step and so forth.
- If at any point a step fails, the coordination logic must then initiate the required compensating transactions.
- We can structure the coordinating logic of a saga in two ways:
  (1) Choreography – saga participants coordinate by exchanging events
  (2) Orchestration - the saga coordination logic is centralized into an orchestrator class

Saga Choreography



Benefits and Drawbacks of Saga Choreography

- Benefits
  - Simplicity – services publish events on key actions (create, update, delete objects)
  - Loose Coupling – participants subscribe to events and don't have direct knowledge of each other.

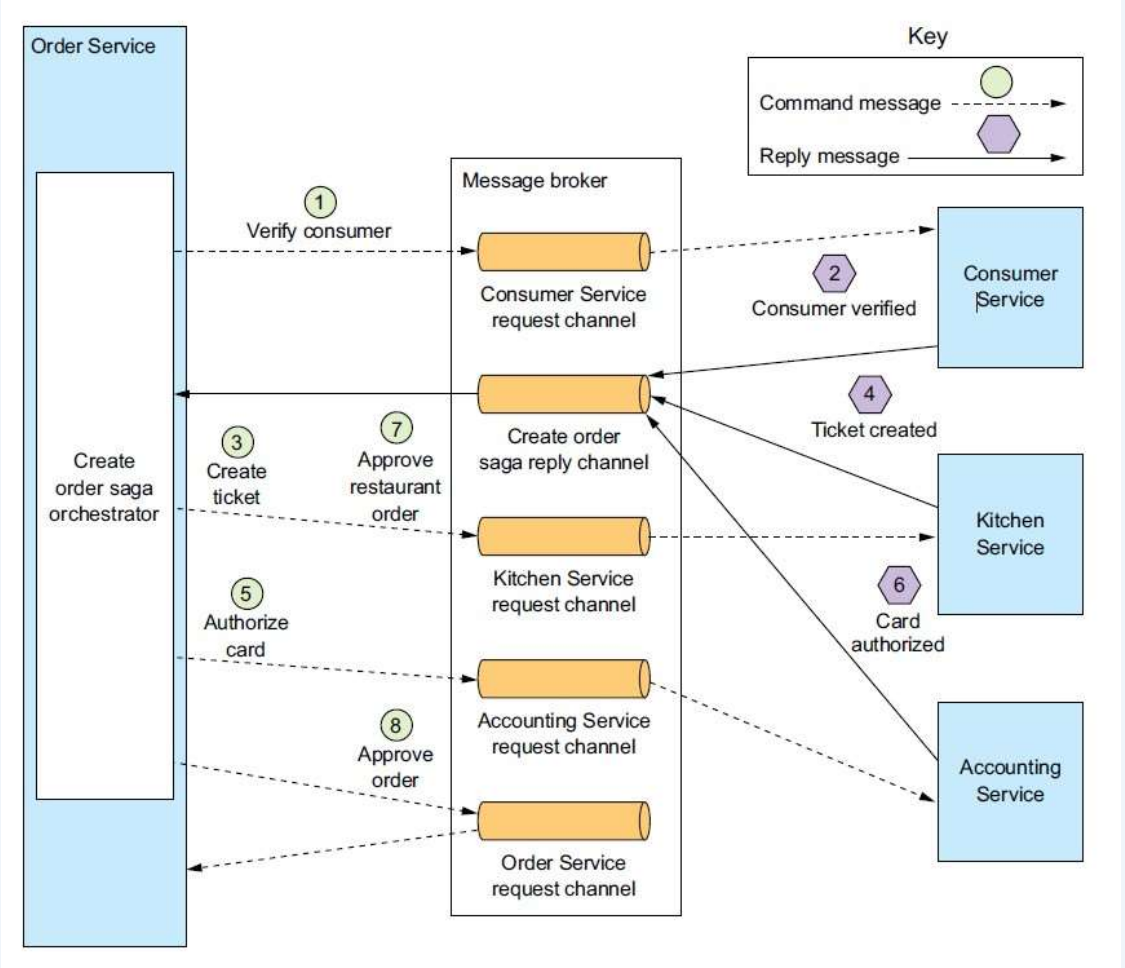
- Drawbacks
  - More difficult to understand – as we saw in the example, you have to track the saga path carefully.
  - Cyclic dependencies between services – you can get stuck with a loop of services all waiting on each others messages.
  - Risk of tight coupling – participants might get stuck waiting for updates in their lifecycle.

- Quite good for simple sagas, but orchestration often better for more complex ones.

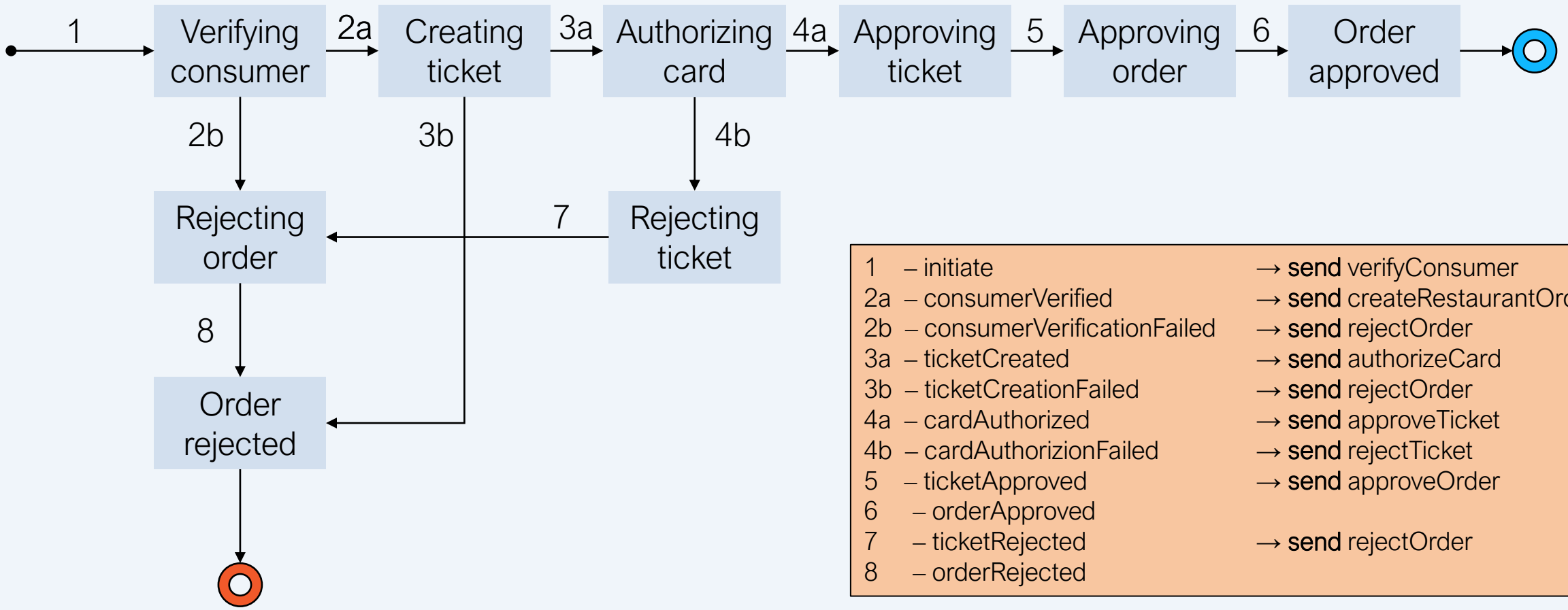
Saga Orchestration

- Orchestrator class solely responsible for telling participants what to do.
- The orchestrator alone decides what to send and when based on messages received.

- You can model a saga as a state machine, which describes all possible scenarios.



Saga State Machine



1	– initiate	→ **send** verifyConsumer
2a	– consumerVerified	→ **send** createRestaurantOrder
2b	– consumerVerificationFailed	→ **send** rejectOrder
3a	– ticketCreated	→ **send** authorizeCard
3b	– ticketCreationFailed	→ **send** rejectOrder
4a	– cardAuthorized	→ **send** approveTicket
4b	– cardAuthorizionFailed	→ **send** rejectTicket
5	– ticketApproved	→ **send** approveOrder
6	– orderApproved	
7	– ticketRejected	→ **send** rejectOrder
8	– orderRejected	

Owerya RESOURCING

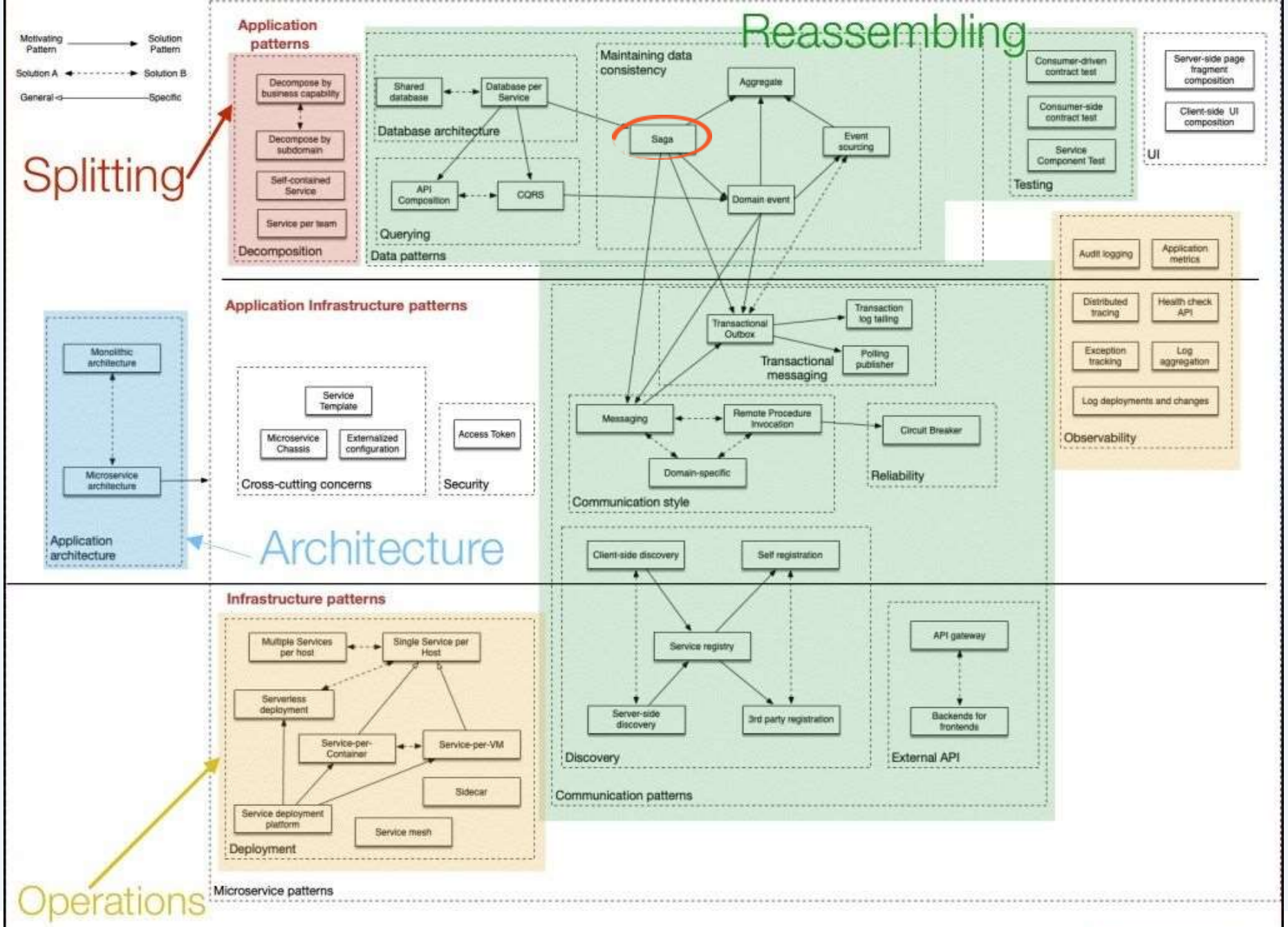
Benefits and Drawbacks of Saga Orchestration

- Benefits
  - Simpler dependencies. No cyclic dependencies.
  - Less coupling.
  - Improves separation of concerns and simplifies the business logic.

- Drawback
  - You could centralize too much business logic in the orchestrator.
  - This can be avoided by making orchestrators solely responsible for sequencing.

Resulting Context of the Saga Pattern

- Benefits
  - Application can maintain data consistency across multiple services without using distributed transactions.
- Drawbacks
  - The programming model is more complex. Every transaction must have a compensating transaction to undo changes.
- Issues
  - To be reliable a service must atomically update its database and publish a message/event.
    - There are two patterns which can solve this: event sourcing and transactional outbox. Not covered here.
  - The initiating client often uses a synchronous (e.g. REST) request to initiate an asynchronous flow.
    - Need to think about how it determines the request outcome.



Learn-Build-Assess Microservices http://adopt.microservices.io

Summary

- What do we need from transaction infrastructure?
- Saga communication
- Saga coordination
  - Choreography
  - Orchestration
  - Saga as a state machine

Questions or Comments?