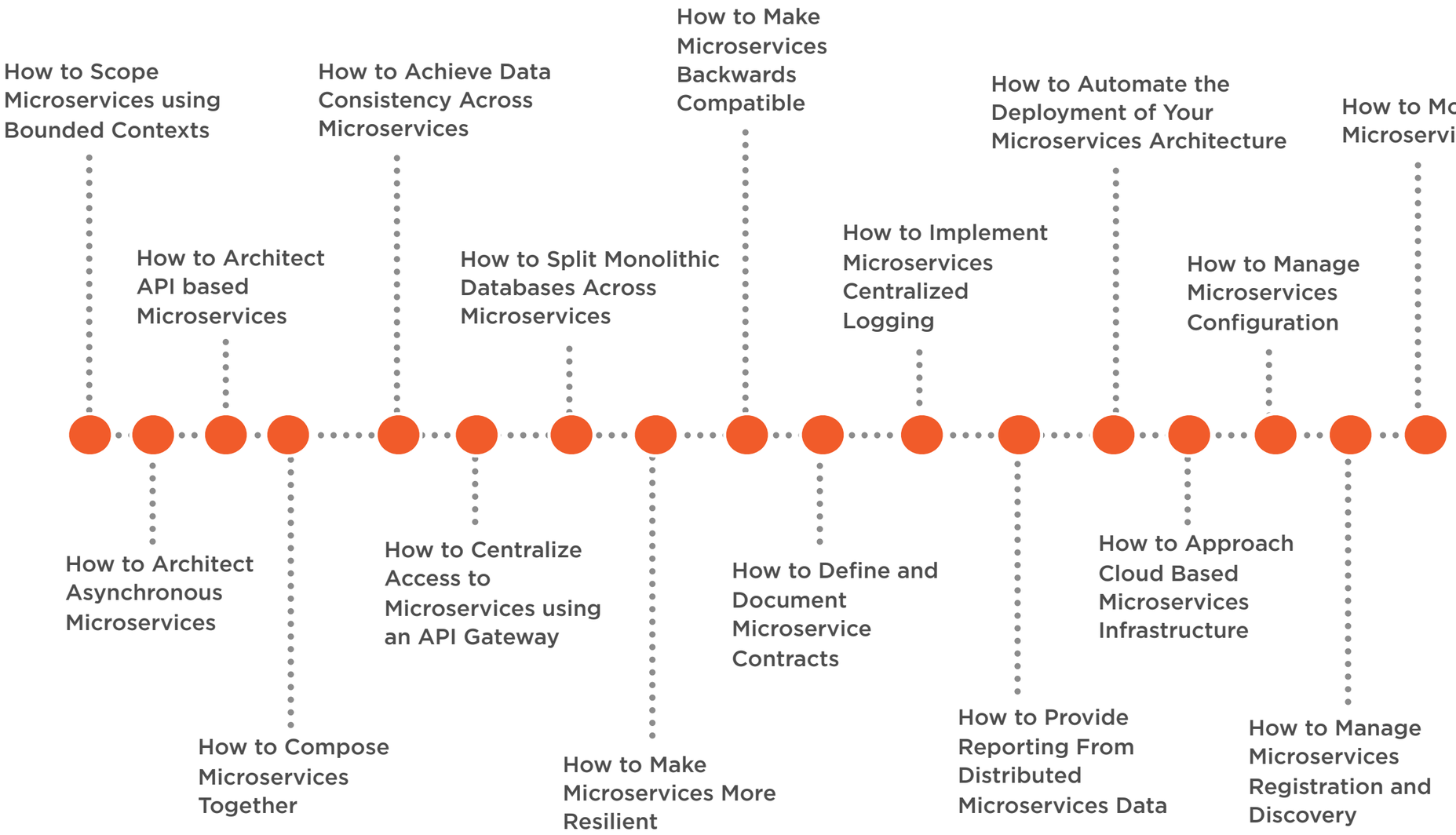
How to Achieve Data Consistency Across Microservices

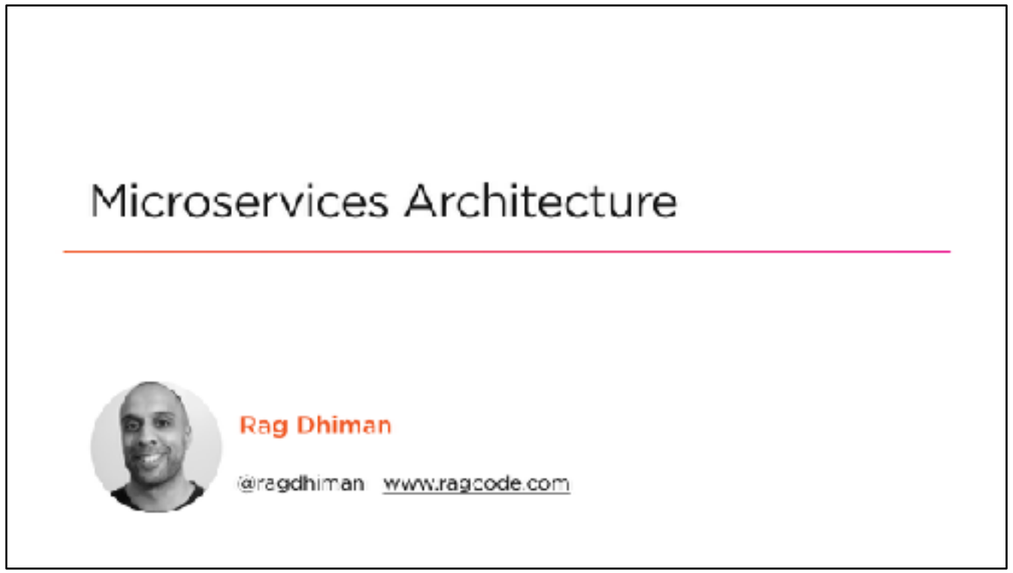
Rag Dhiman

@ragdhiman www.ragcode.com

Microservices Architectural Design Patterns Playbook



Microservices Architectural Design Patterns Playbook



Overview

Introduction

Options

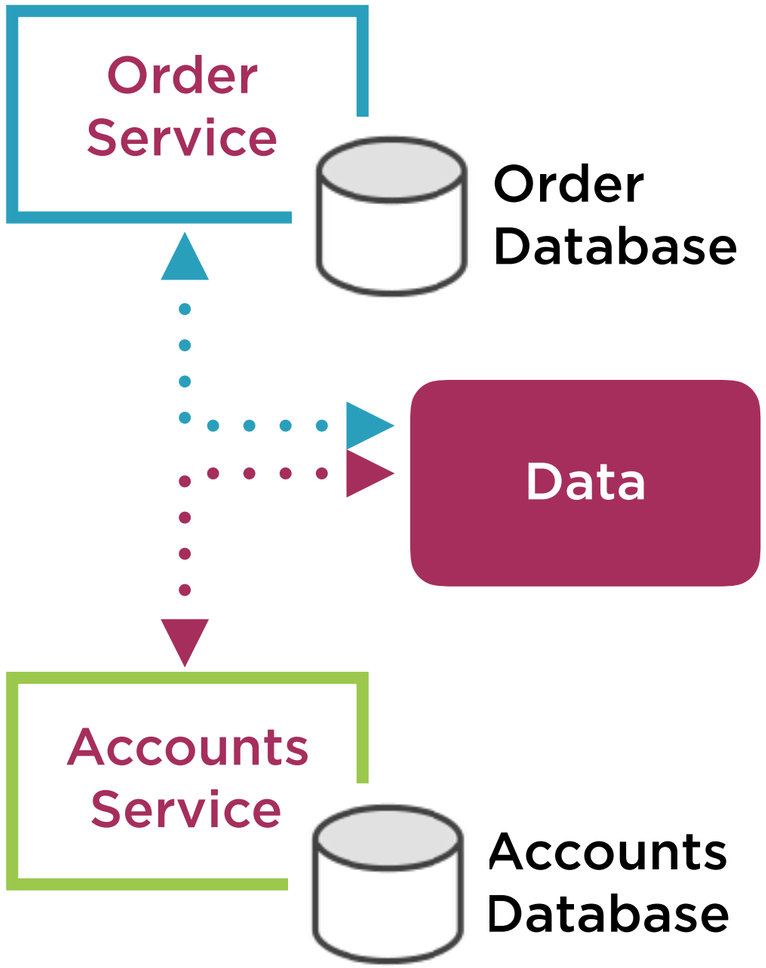
Two Phase Commit

Saga Pattern

Eventual Consistency

Introduction

Introduction



Data consistency

- Transactions are the traditional approach

Monolithic system transactions

- Single database which is the single truth

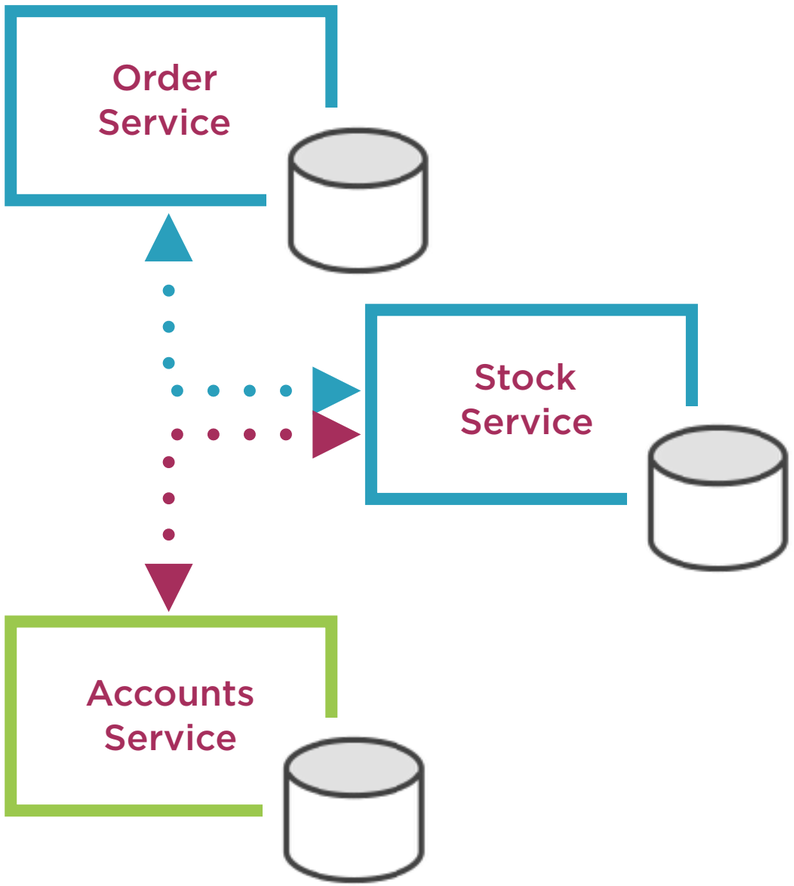
Microservice transactions

- Distributed architecture
- Distributed data
- Distributed transactions

CAP theorem

- Network failure will happen
  - Data availability or data consistency?

Options



Traditional ACID transactions

- Atomicity, consistency, isolation and durability

Two phase commit pattern

- ACID is mandatory
- CAP Theorem: Choosing consistency

Saga Pattern

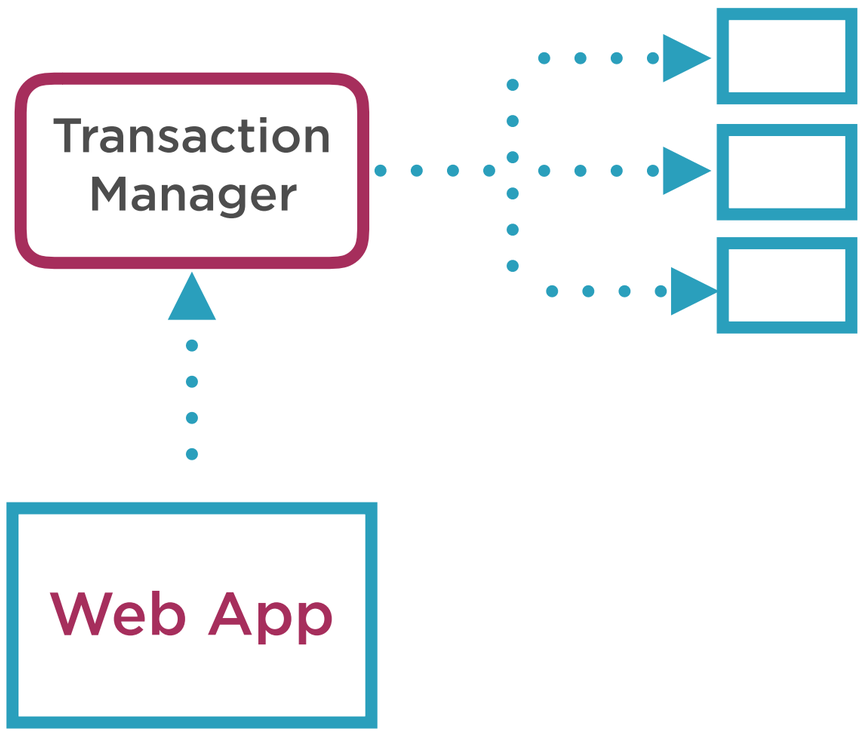
- Trading atomicity for availability and consistency

Eventual consistency pattern

- Compromise ACID
- CAP Theorem: Choosing availability

Two Phase Commit

Two Phase Commit



Pattern for distributed transactions

- Transaction manager manages transactions

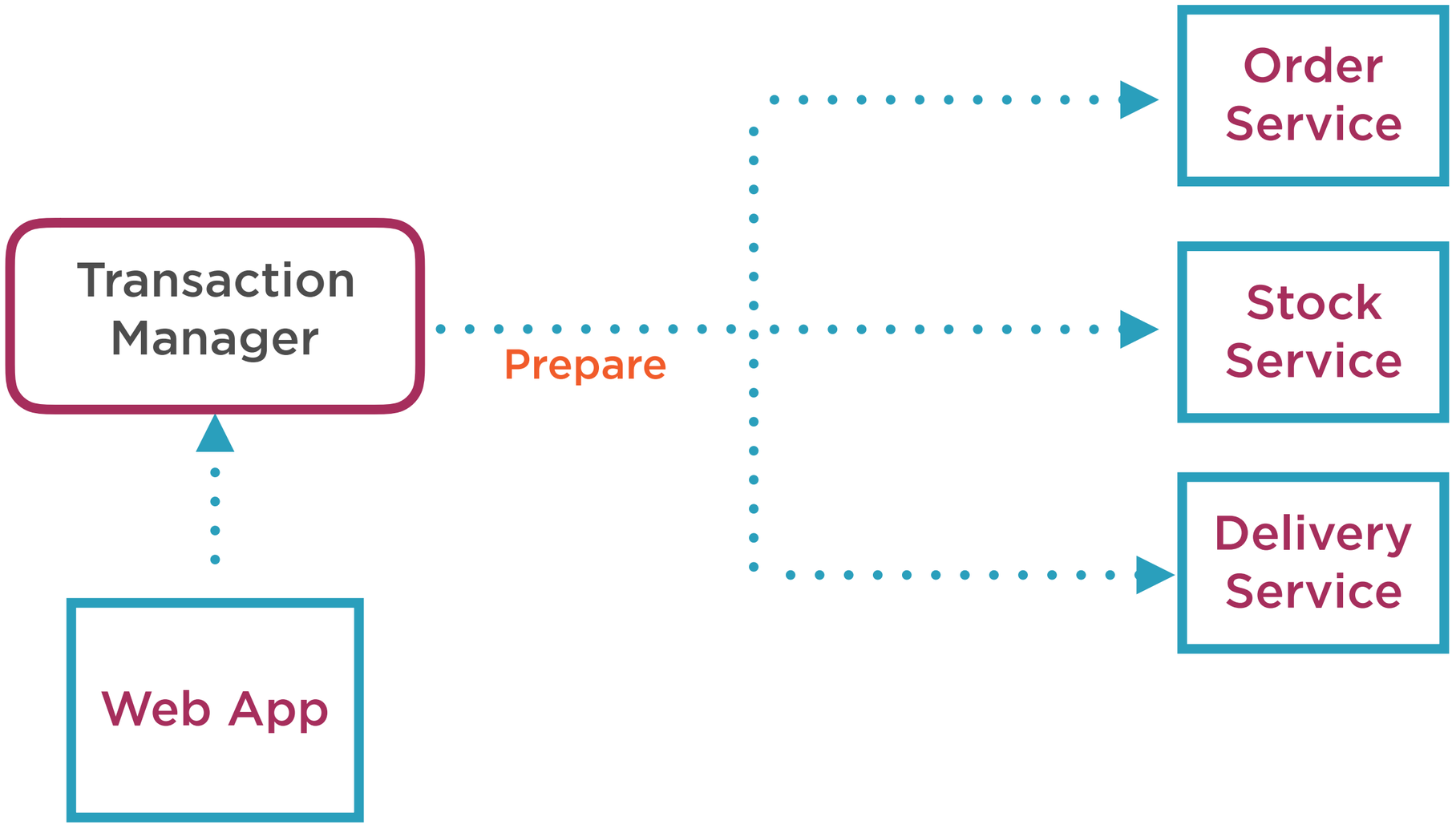
Prepare phase

- Transaction manager asks to prepare

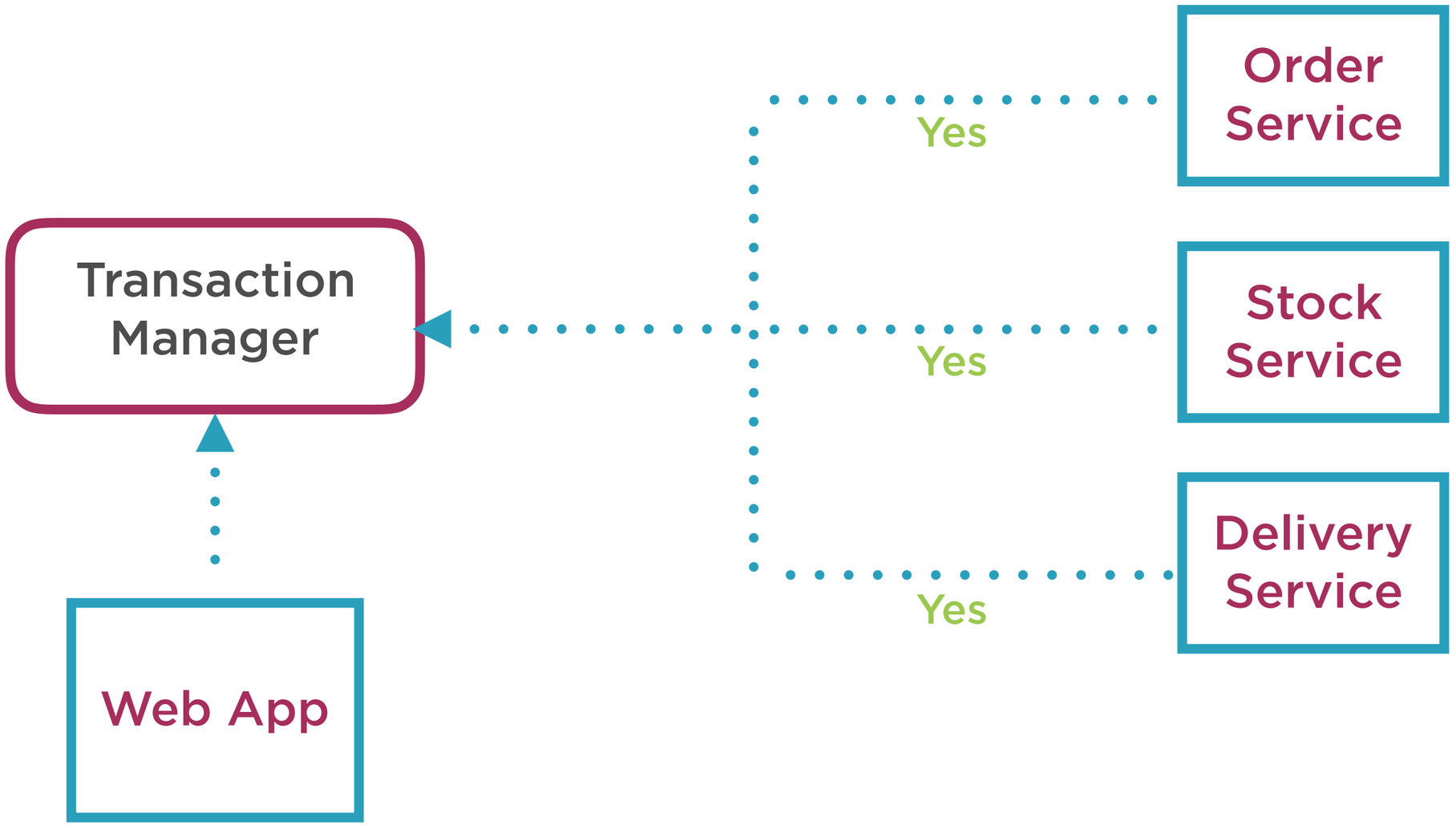
Voting phase

- Transaction manager receives votes
- Transaction manager
  - Issues a commit on all yes votes
  - Issues a rollback on any no vote

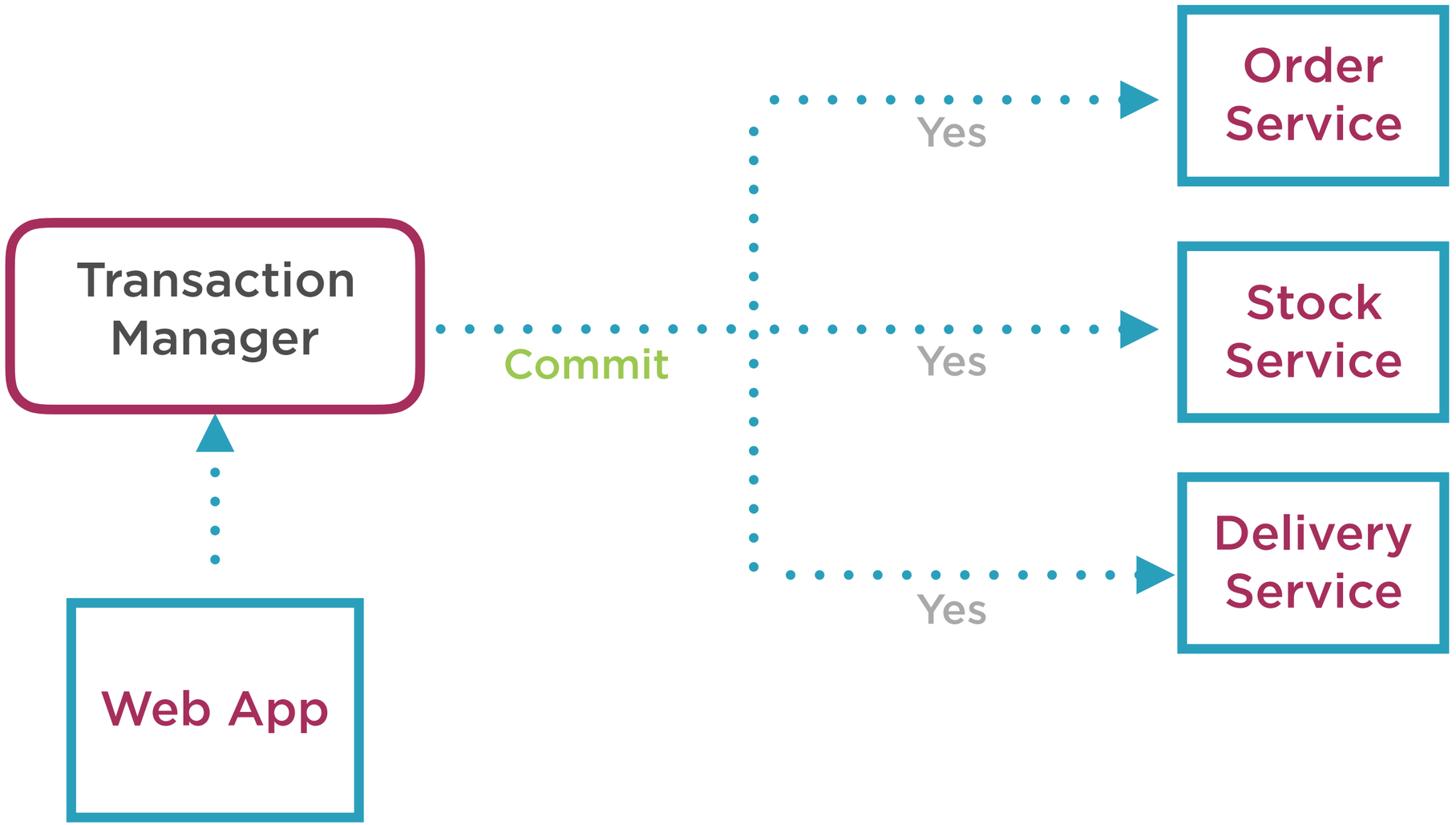
Two Phase Commit: Prepare Phase



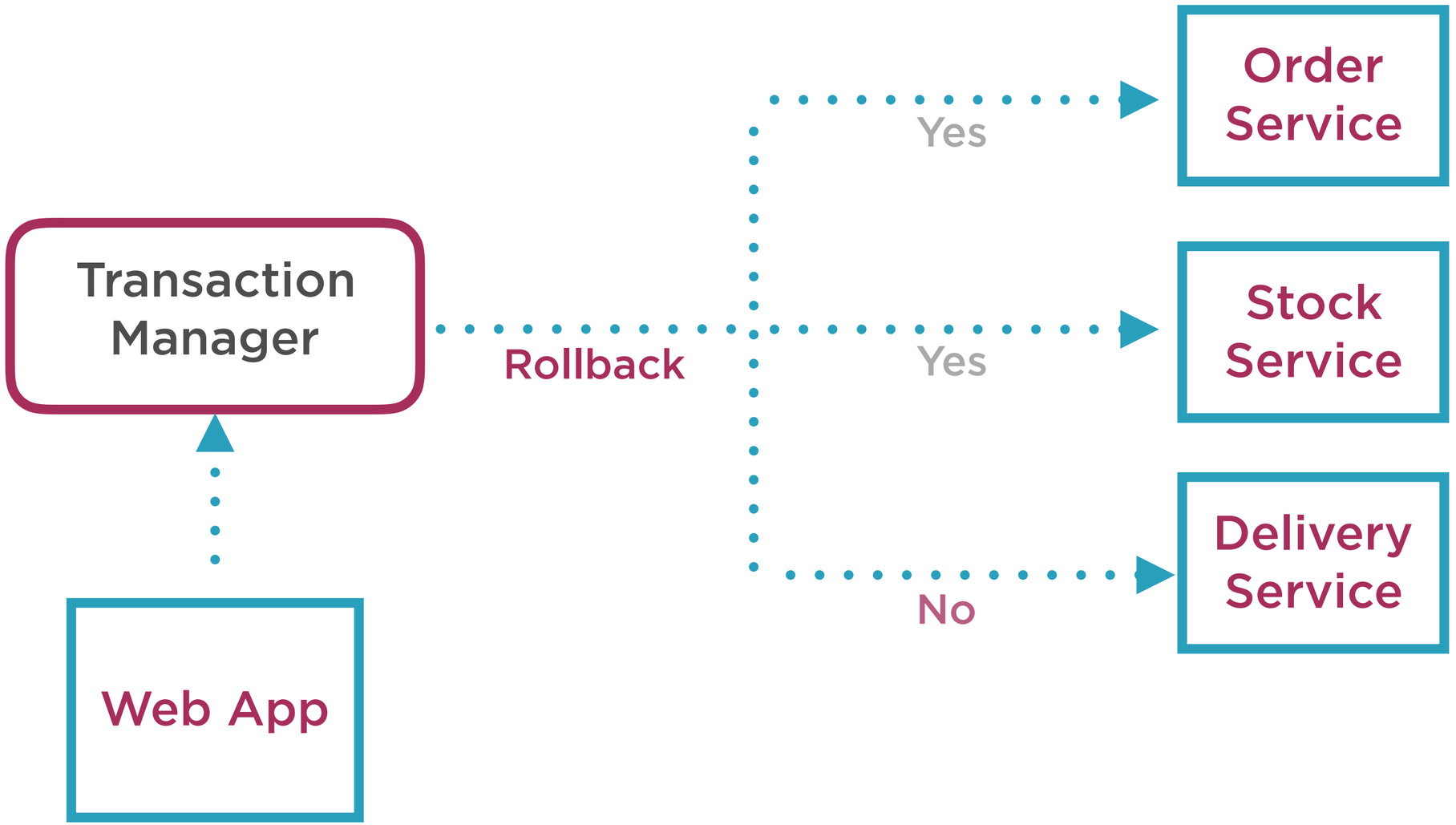
Two Phase Commit: Vote Phase



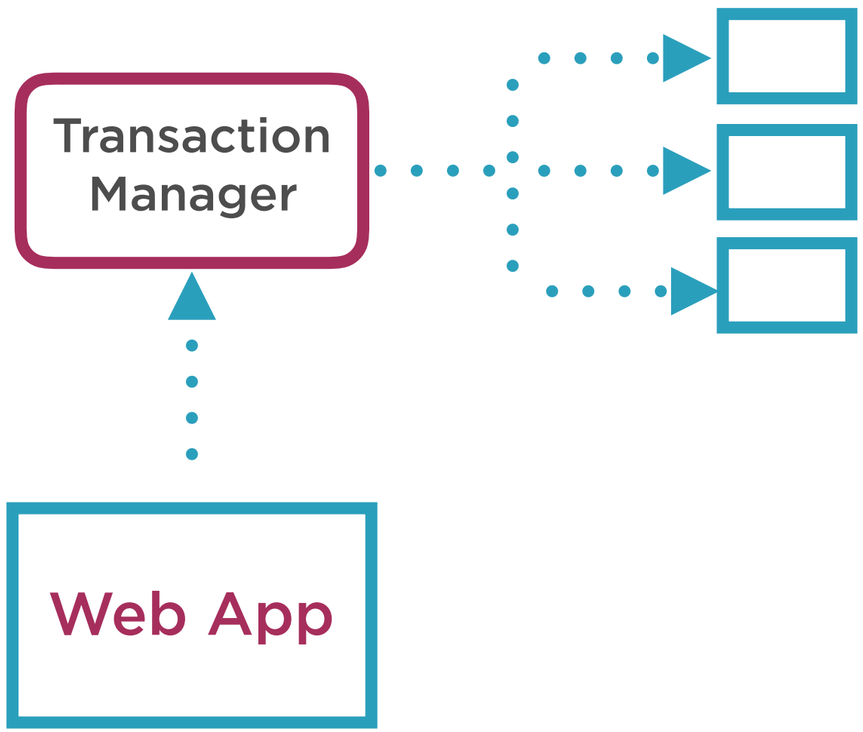
Two Phase Commit: Commit



Two Phase Commit: Rollback



Two Phase Commit



Caveats

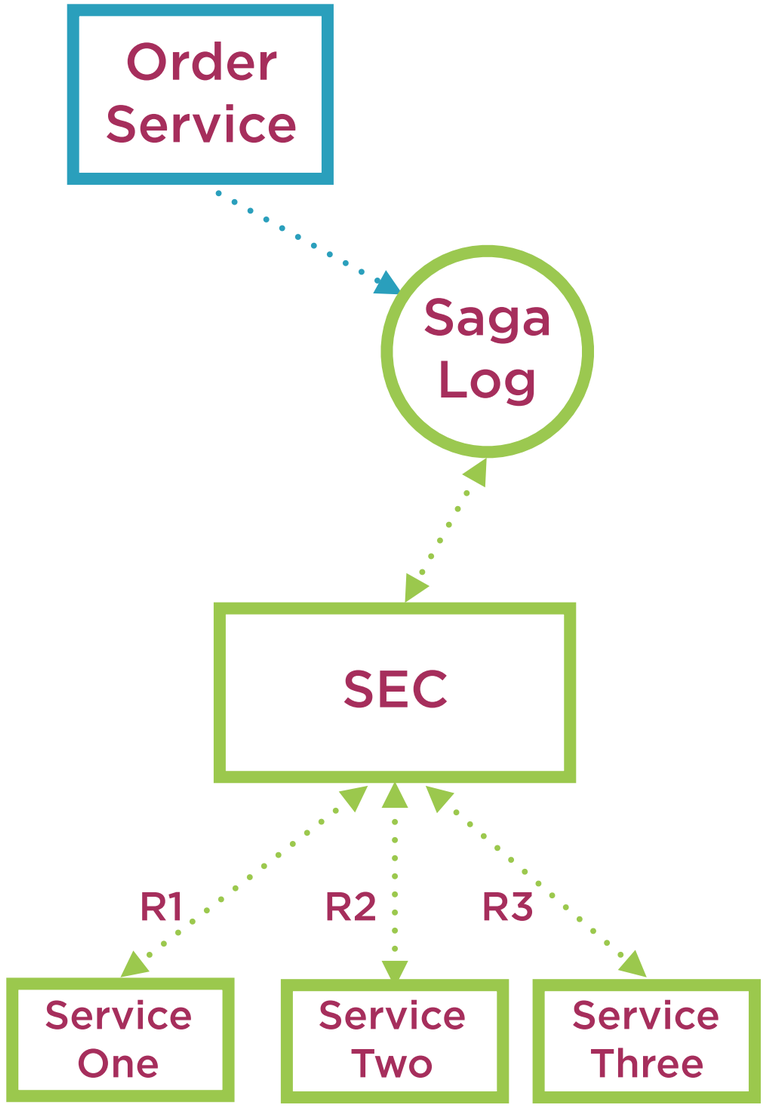
- Reliance on a transaction manager
- No voting response
- Commit failure after successful vote
- Pending transactions lock resources
- Avoid custom implementations
- Has scaling out issues
- Reduced throughput
- Anti-pattern

Consider alternatives

- Saga pattern
- Eventual consistency

Saga Pattern

Saga Pattern



Replaces a distributed transaction with a saga

- Splits transaction into many requests
- Tracks each request
- ACID: Compromise atomicity
- First described in 1987

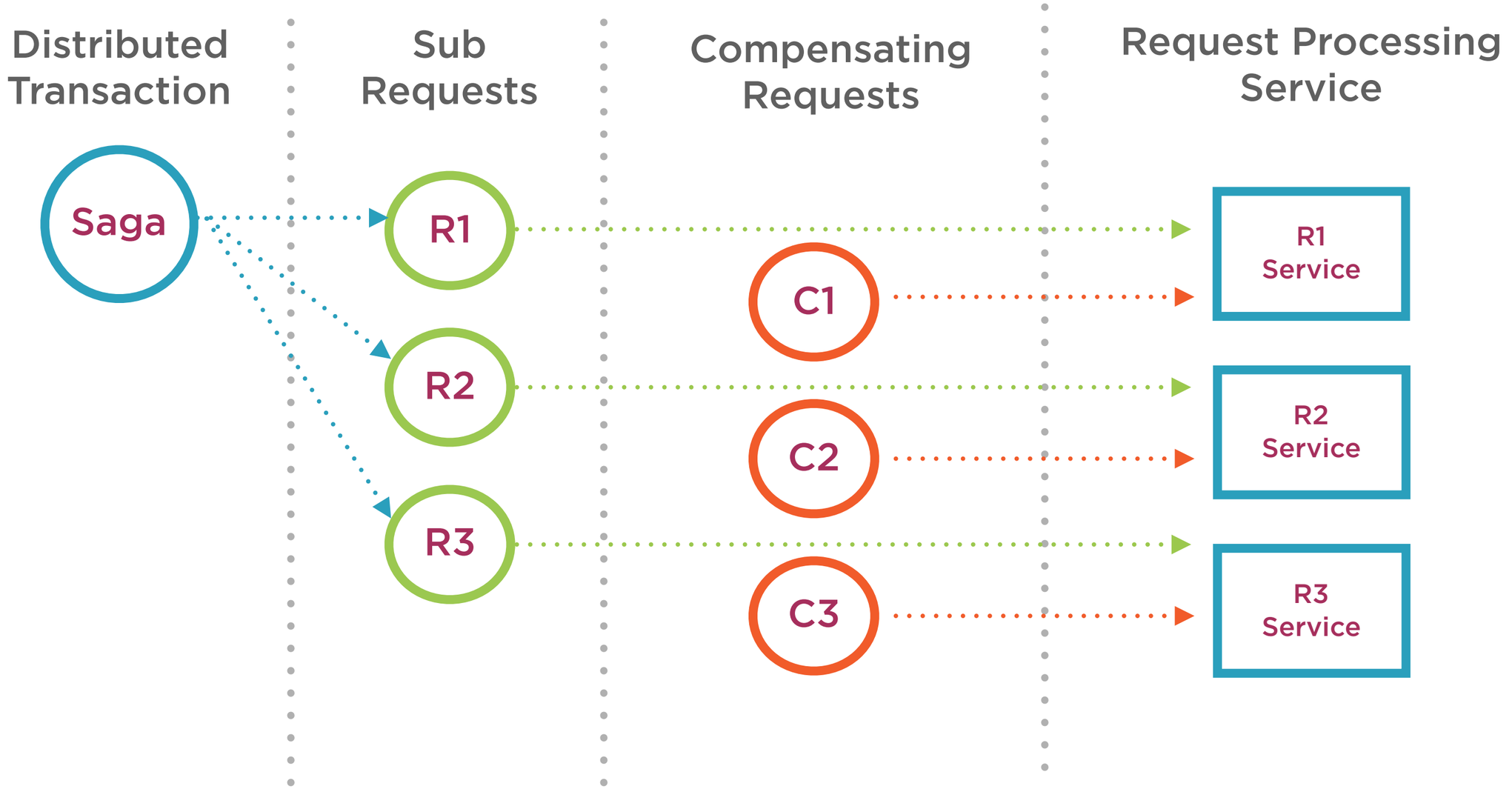
Also a failure management pattern

- What to do when one service fails
- Compensate requests

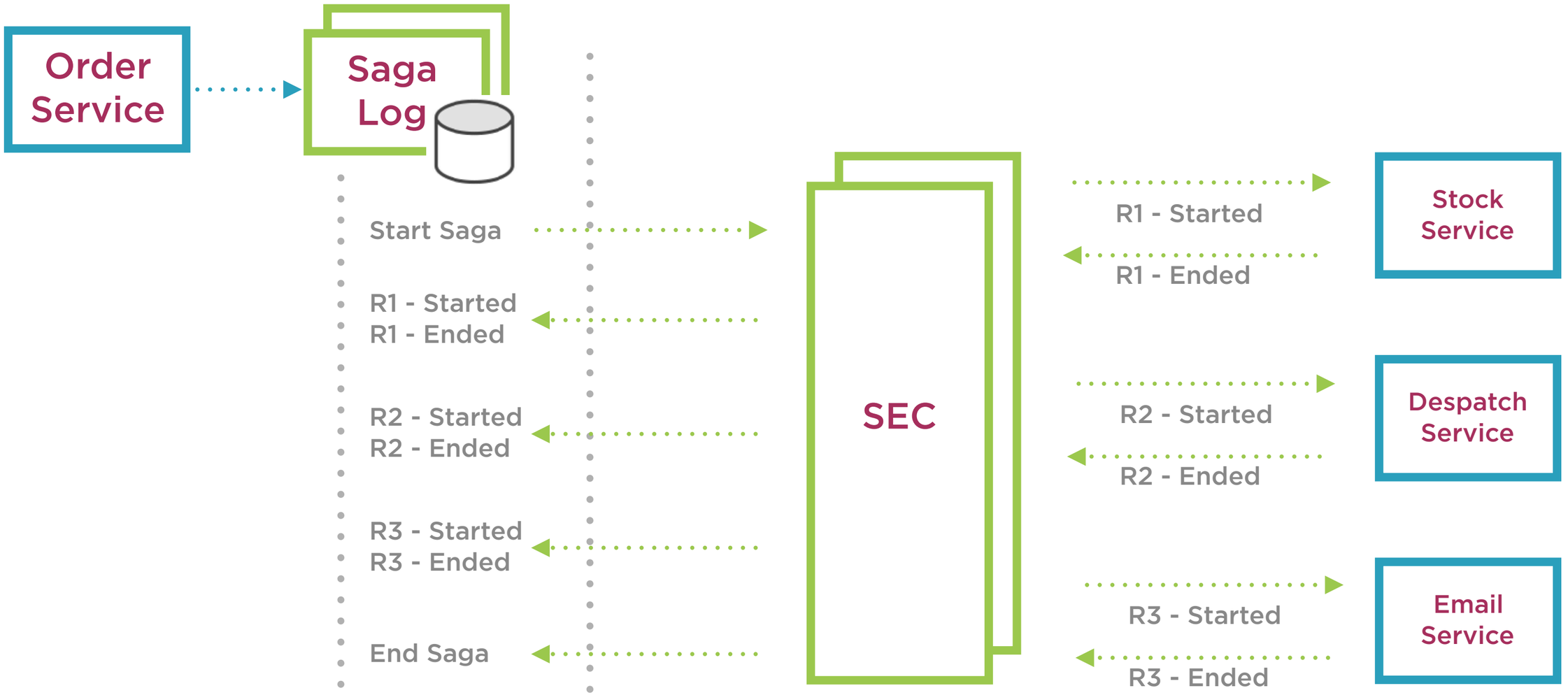
Implementation

- Saga log
- Saga execution coordinator (SEC)
- Requests and compensation requests

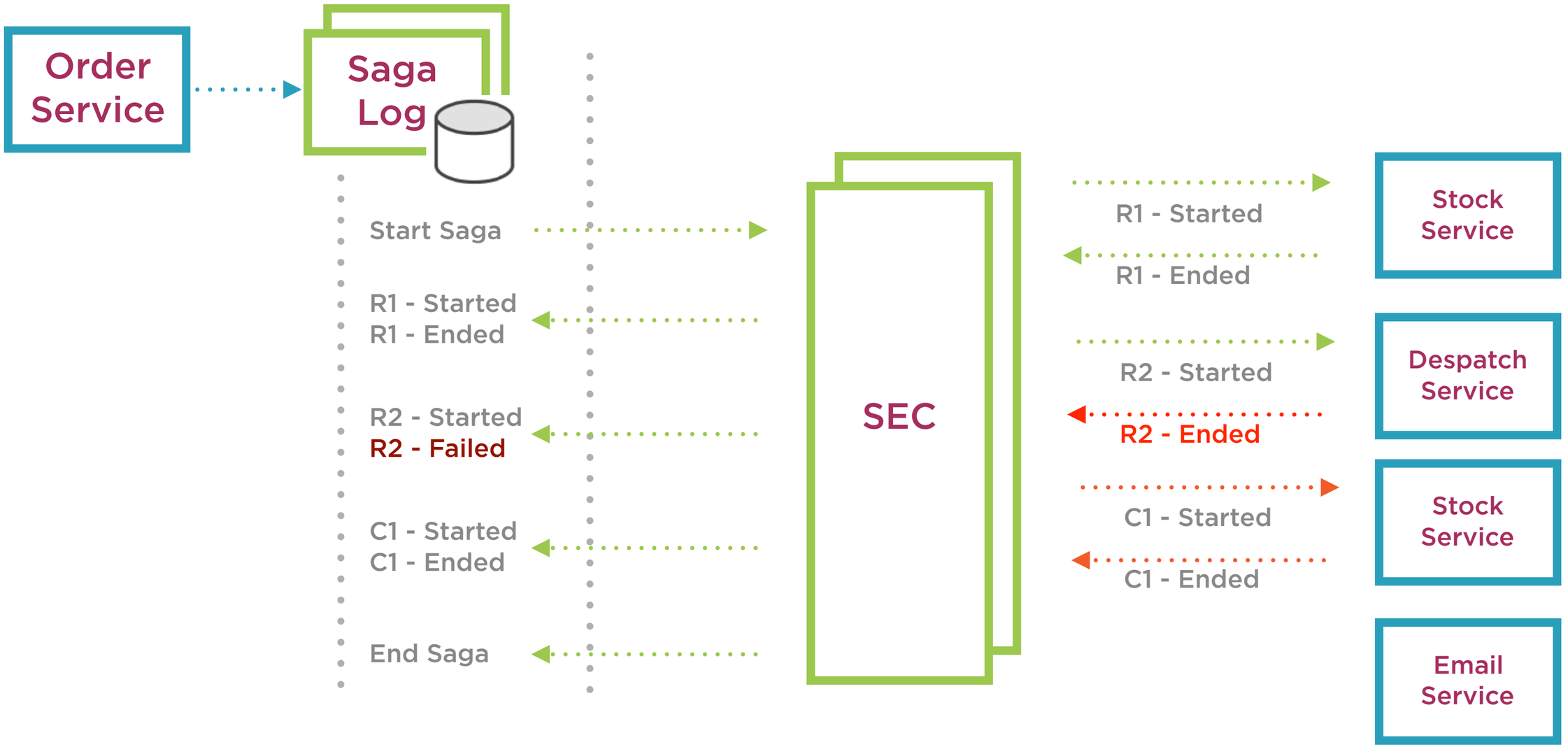
Saga Pattern



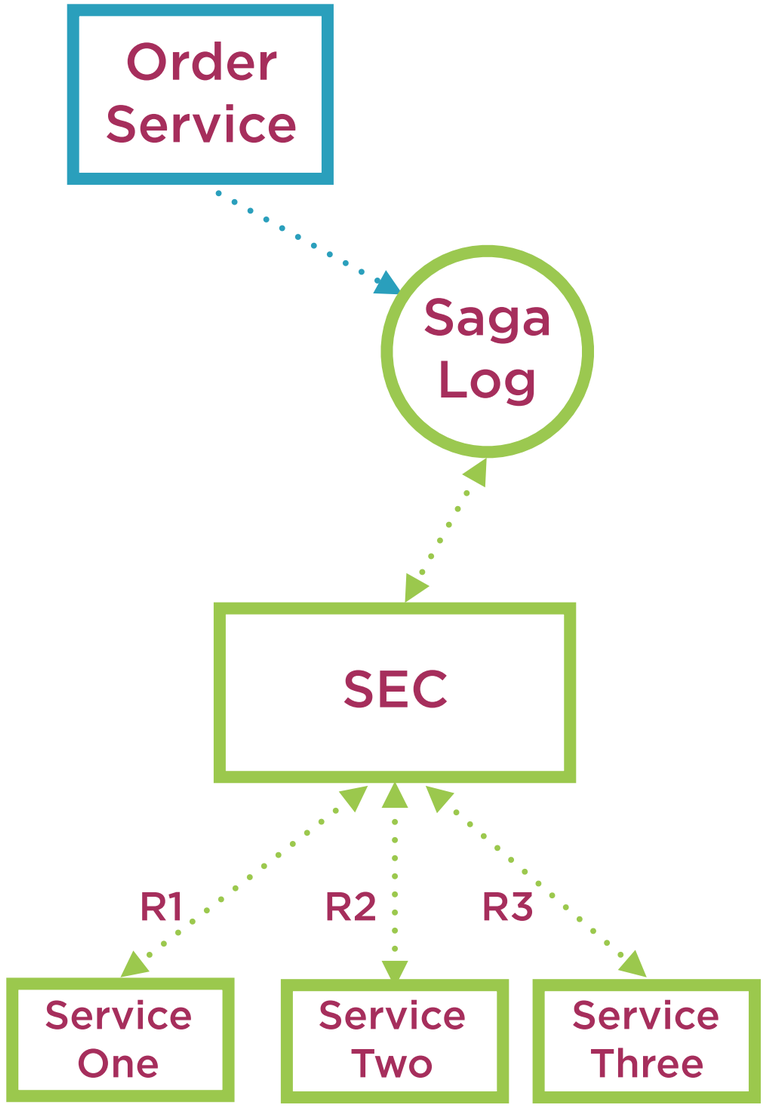
Successful Order Saga



Unsuccessful Saga



Saga Pattern Implementation



Service that initiates the saga

- Sends the saga request to the saga log

Saga Log

- Service with a database

SEC

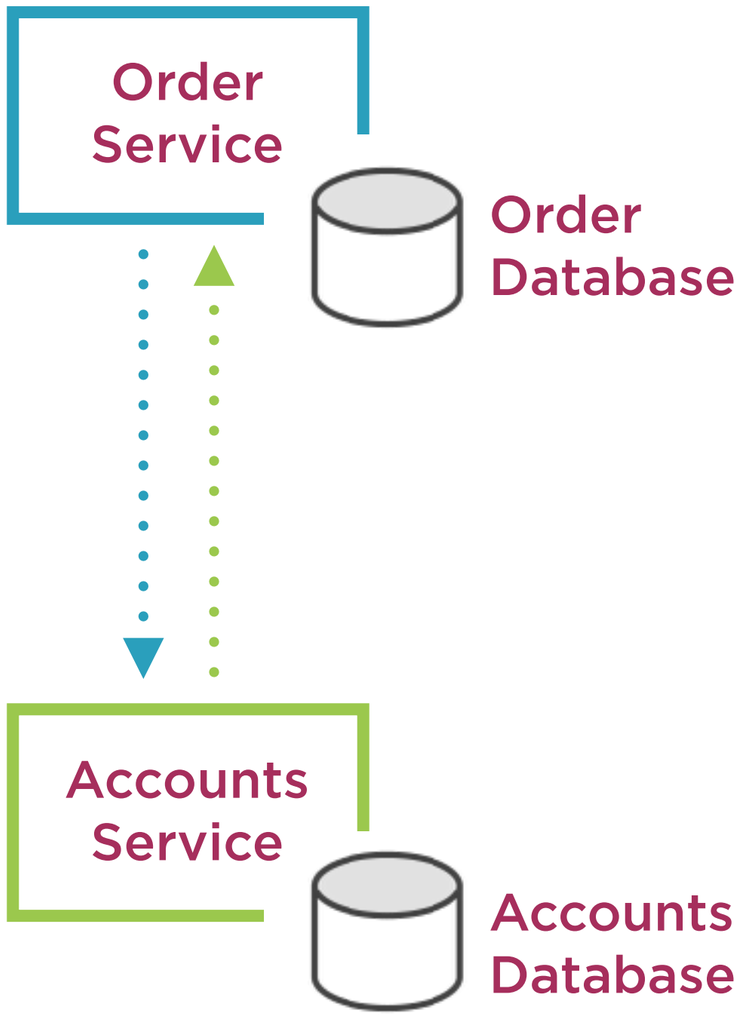
- Interprets and writes to the log
- Sends out saga requests
- Sends out saga compensation requests
- Recovery: Safe state vs unsafe state

Compensation requests

- Send on failure for all completed requests
- Idempotent (easy with REST)
- Each one is sent zero to many times

Eventual Consistency

Eventual Consistency



Data will eventually be consistent

- BASE
- BASE vs ACID

Availability over consistency

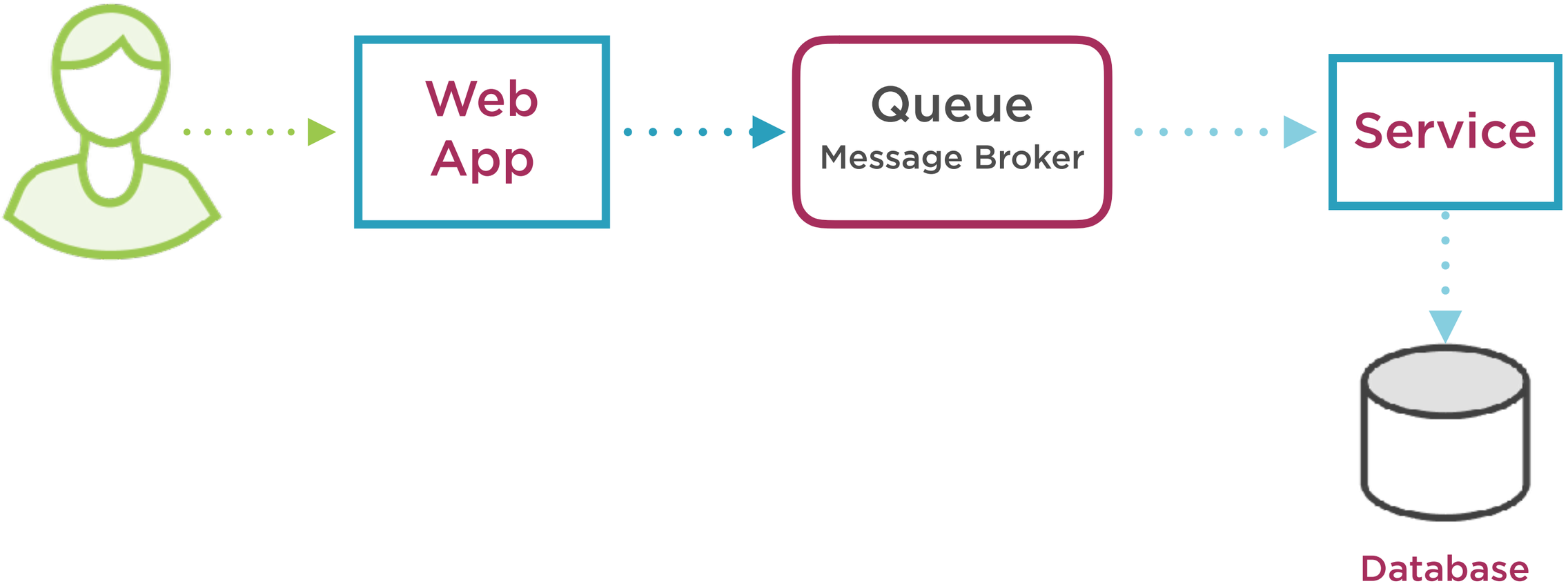
- Avoid resource locking
- Ideal for long running tasks
- Prepared for inconsistencies
- Race conditions

Data replication

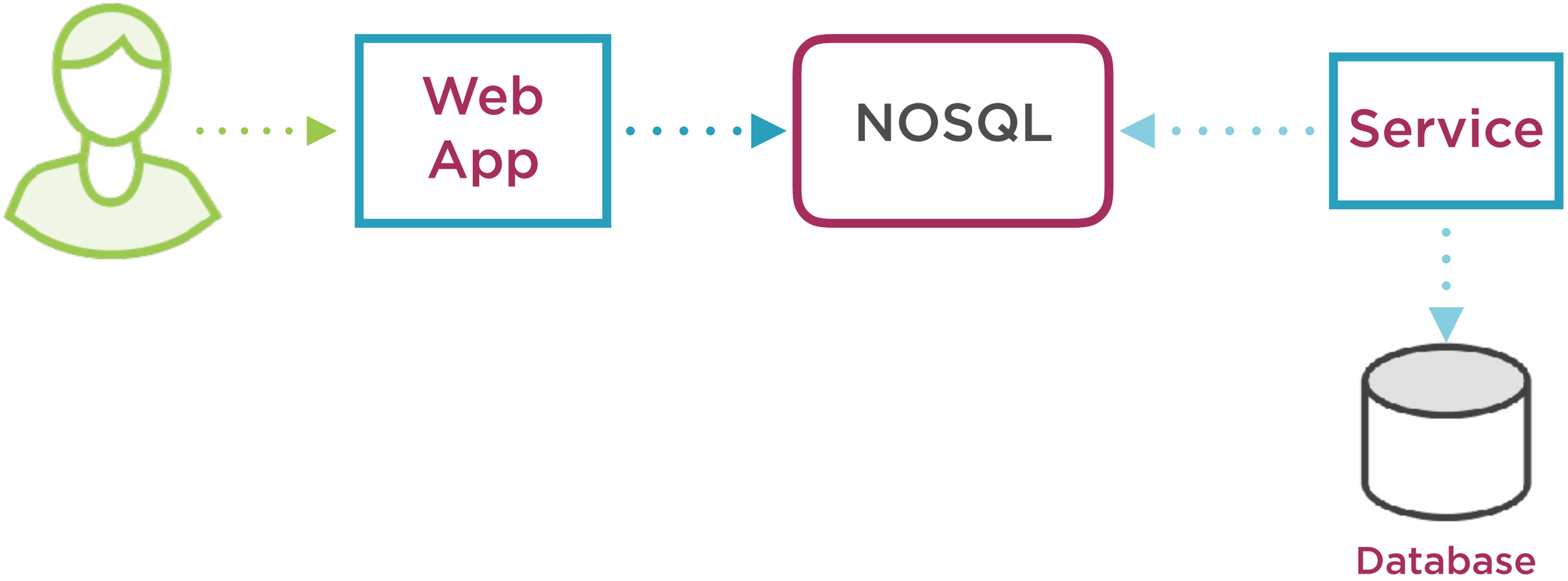
Event based

- Transaction/actions raised as events
- Messages using message brokers

Eventual Consistency



Eventual Consistency



Summary

Introduction

Options

Two Phase Commit

Saga Pattern

Eventual Consistency

Microservices Architectural Design Patterns Playbook