nexmo® | The Vonage® API Platform

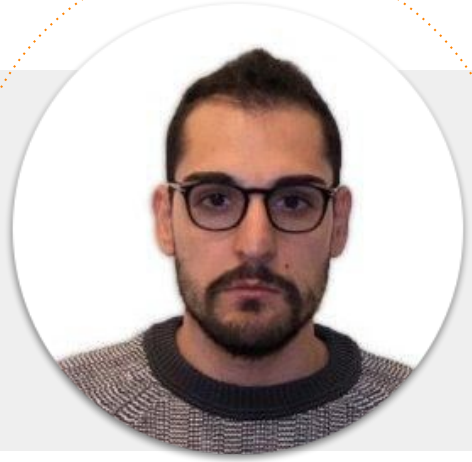
Extensible Autonomous Transactions in the world of Microservices using Apache Kafka

APACHECON

Nice to meet you

NICOLA GIACCHETTA

nicolagiacchetta



DIVYA NAGAR

divya2661

We work in a team of 3 people along with NARAIN RAMASWAMY at nexmo® The Vonage® API Platform

What is Nexmo, The Vonage API Platform?

All the world of Communications in a Single Platform

Full Product Suite

- Voice
- Messages
- Video
- Numbers Insight
- Verify
- Dispatch
- SMS
- Chat
- SIP Trunking

Enterprise DNA

Loved by developers.
Built for business.

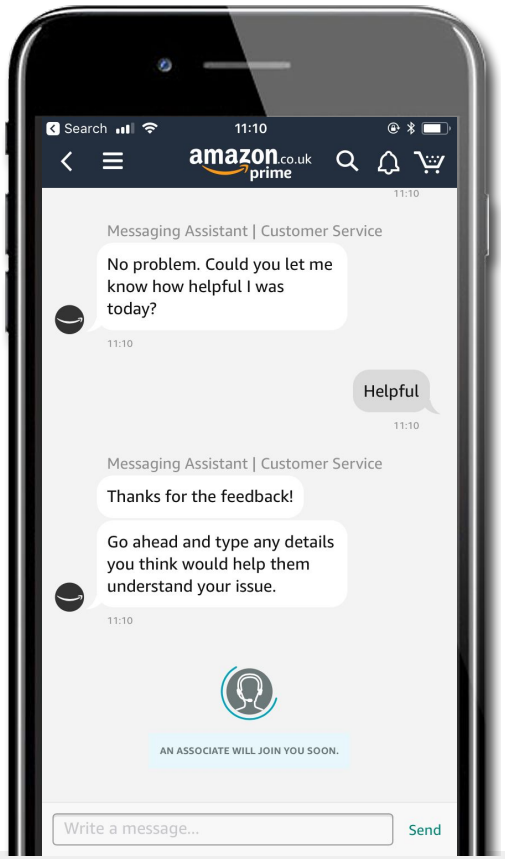
Global Coverage

225 countries and over 1,600 telecommunications networks.

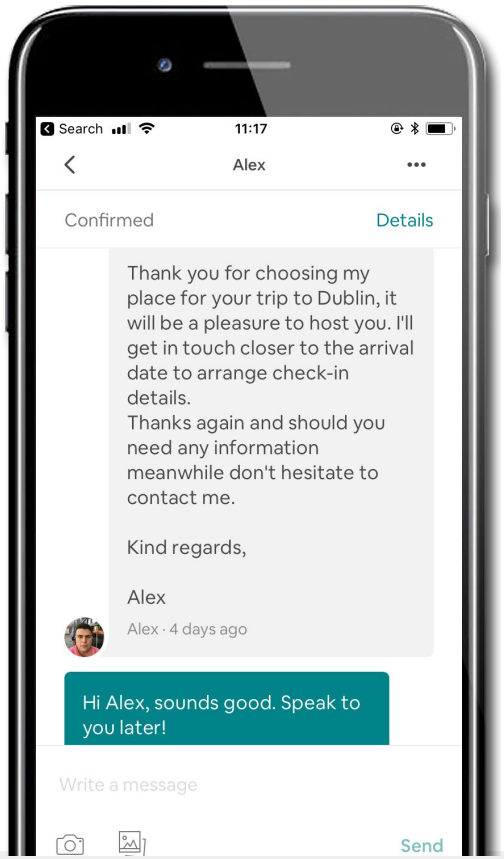
CPaaS are everywhere

Digital native companies have embraced programmable communications

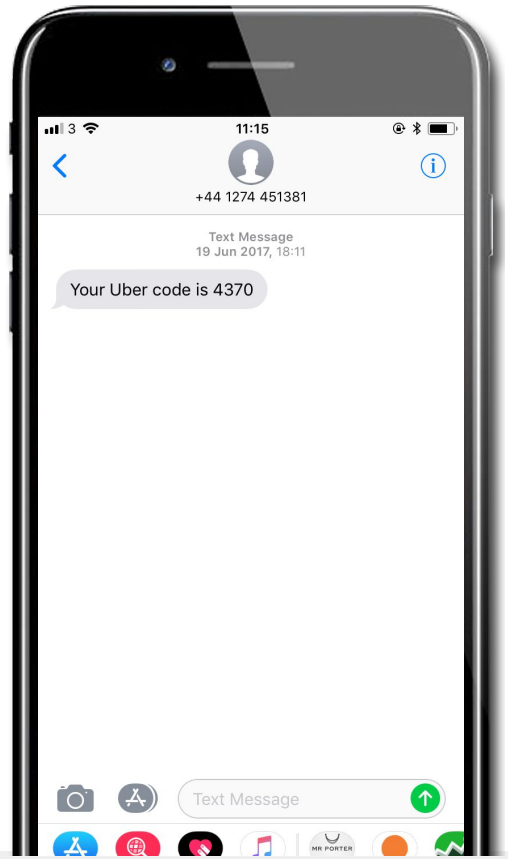
amazon



airbnb



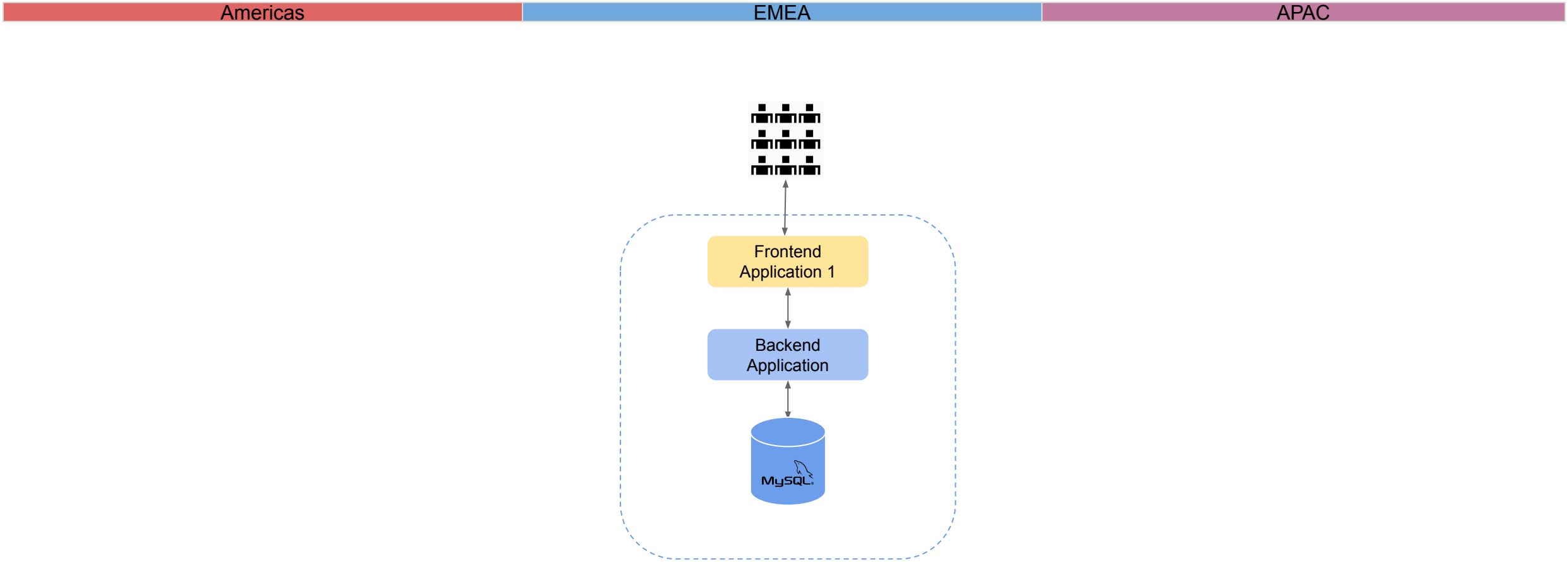
UBER



Agenda

- Small Backend Application → Monolith
- Scaling Read Workload with Volga
- Monolith → Microservices
- Transactions in the World of Microservices

Once upon a time...

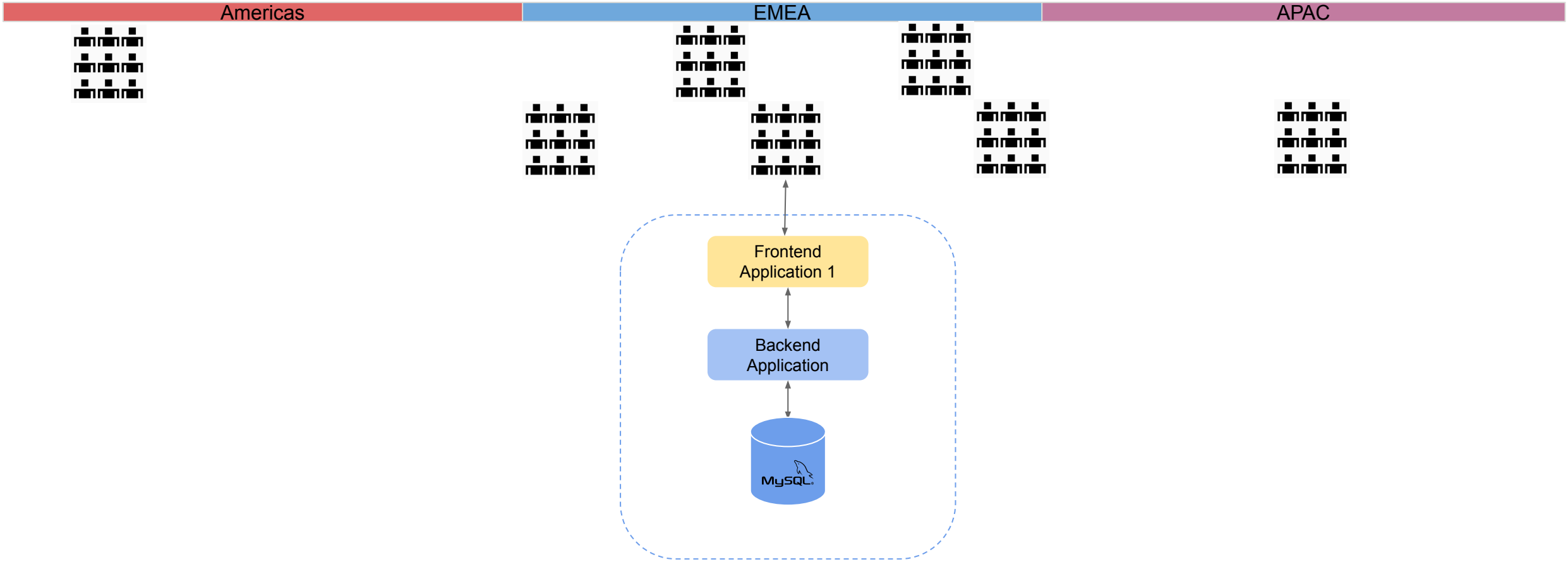


1 application in **1 DC** backed by **1 DB** handling **all reads** and **all writes**

Something more about our Backend Application

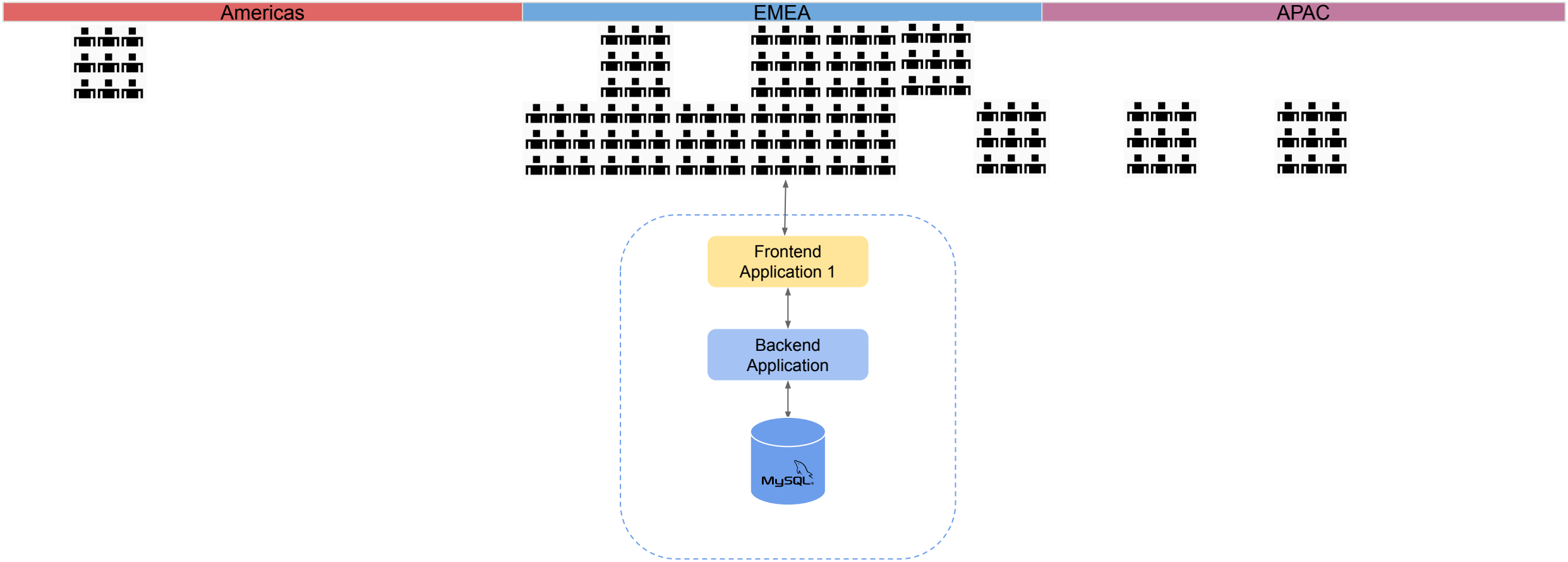
- Backed by a **MySQL** Database
- Reads **>>>** Writes

We were starting to grow



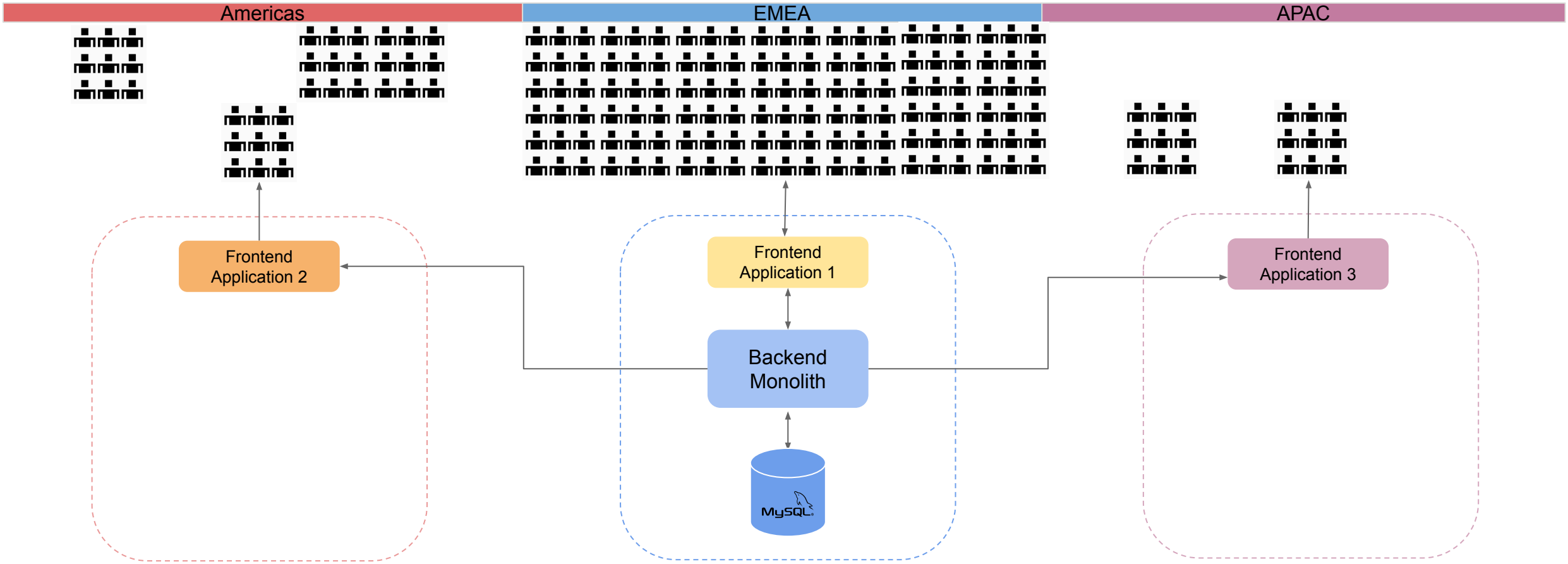
1 application in **1 DC** backed by **1 DB** handling **all reads** and **all writes**

We were starting to grow and grow



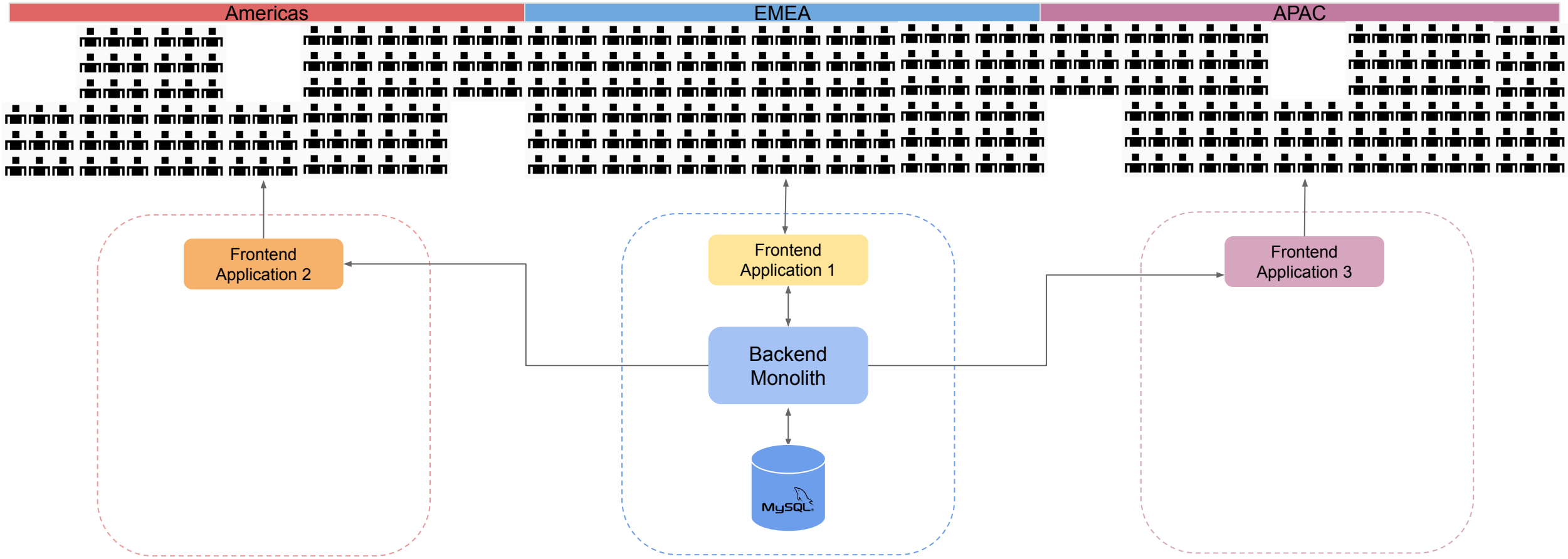
1 application in **1 DC** backed by **1 DB** handling **all reads** and **all writes**

We were starting to grow and grow and grow



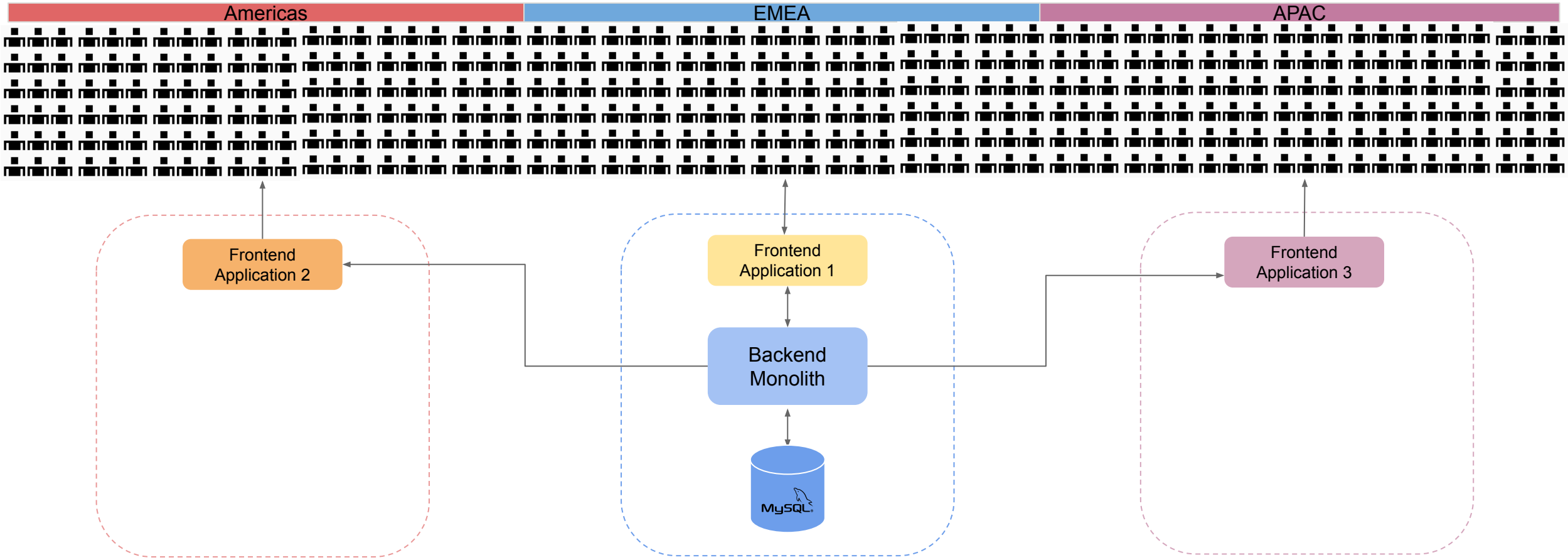
1 application in **1 DC** backed by **1 DB** handling **all reads** and **all writes**

We were starting to grow and grow and grow and grow



1 application in **1 DC** backed by **1 DB** handling **all reads** and **all writes**

We were starting to grow and grow and grow and grow and grow



1 application in **1 DC** backed by **1 DB** handling **all reads** and **all writes**

Problems

1. Read workload
2. Monolithic Architecture

Problem 1: Scaling Read workload

1. **Read workload**
2. Monolithic Architecture

Time to Scale our Read Workload

Problems:

- Too much Load
- Too high Latency

Scaling Reads Strategies

Problems:

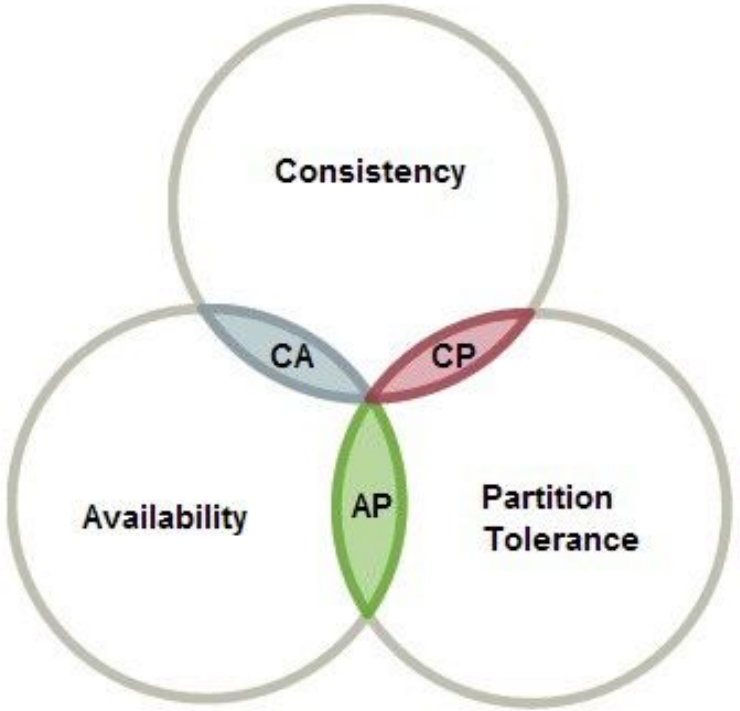
- Too much Load
- Too high Latency

Solutions:

- Adding **Caches** closer to the Applications
- Add **Read Replicas** in other regions

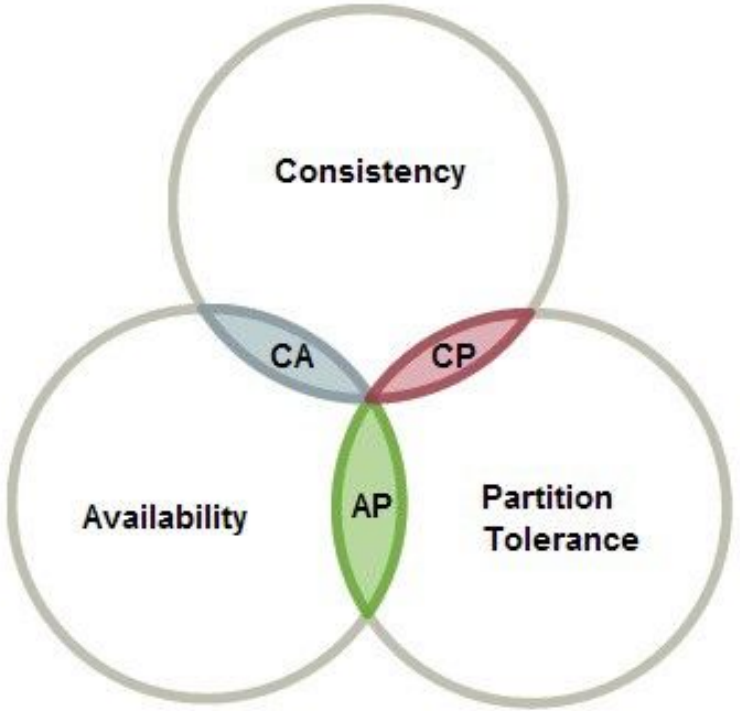
CAP Theorem

High Consistency, **High-Availability**, **Partition Tolerance**: choose **2**.

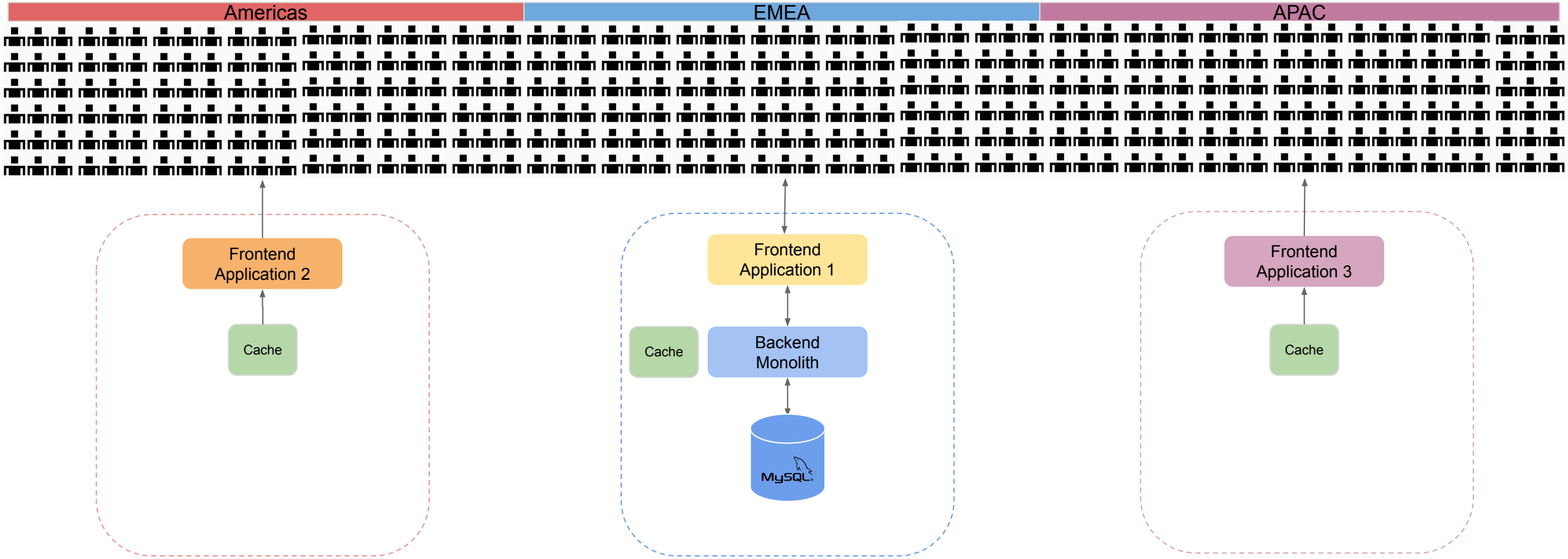


CAP Theorem

High Consistency, **High-Availability**, ~~**Partition Tolerance**~~: choose **1**.

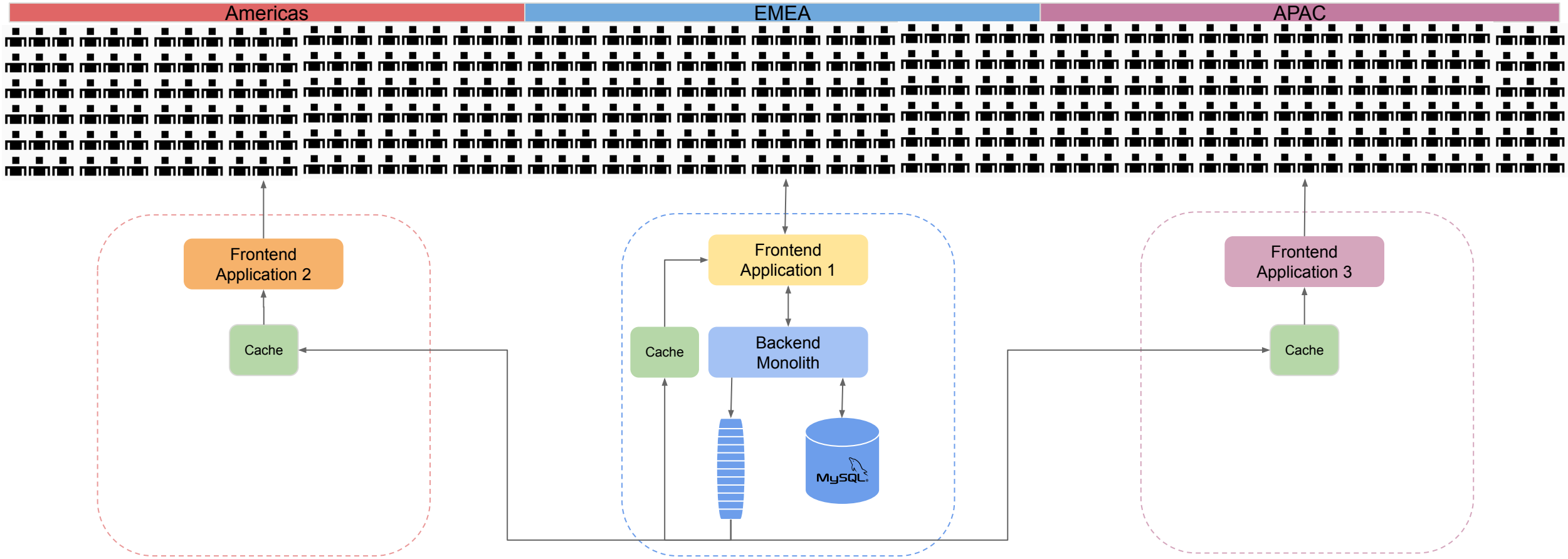


How do we notify the caches?



«There are only two hard things in Computer Science: cache invalidation and naming things. » -- **Phil Karlton**

Dual Writes



The application writes to the database and to another messaging system **in parallel**

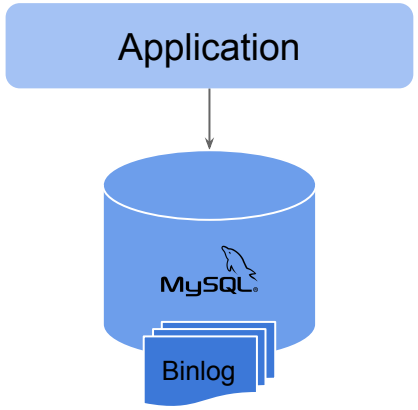
The problem of Dual Writes

The application writes to the database and **in parallel** to another messaging system creating two main issues:

1. Without a complex **coordination protocol** (e.g. 2-Phase Commit or Consensus) it is hard to achieve sequential consistency between the database and the messaging system

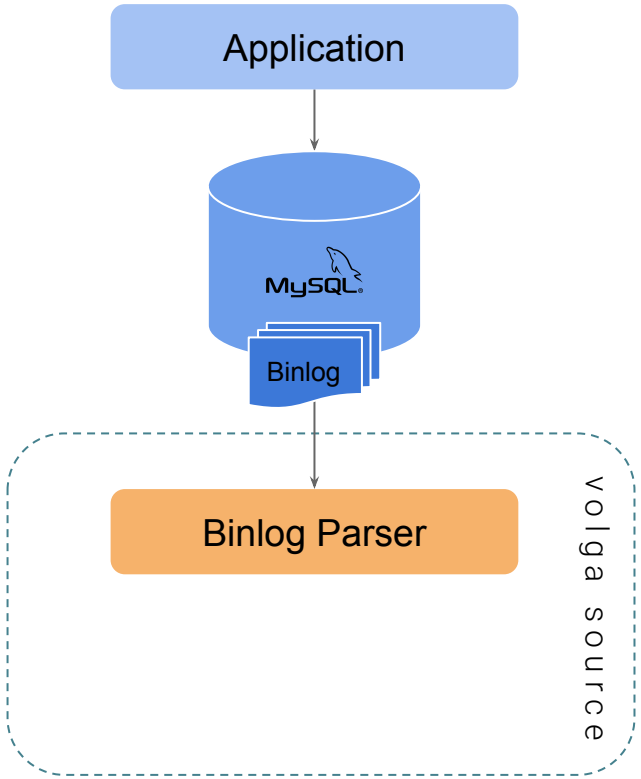
2. The logic to write to the messaging system is **strictly-coupled** with the application

Log Mining with Volga



The database is the **single source-of-truth** and changes are extracted from its transaction or commit log.

Log Mining with Volga

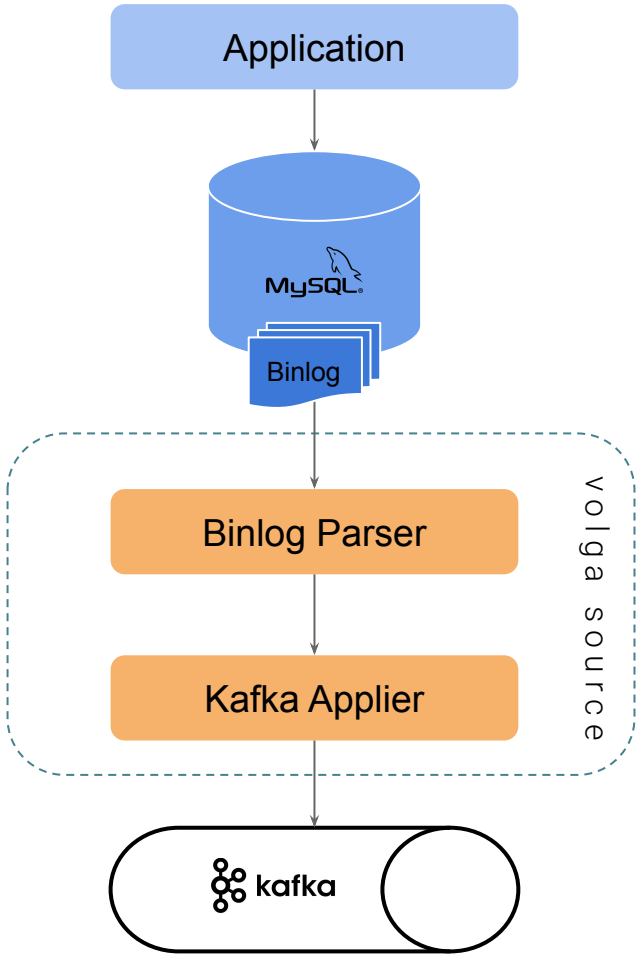


The database is the **single source-of-truth** and changes are extracted from its transaction or commit log.

Volga:

- **Connects** to the Commit Log of the Source DB

Log Mining with Volga

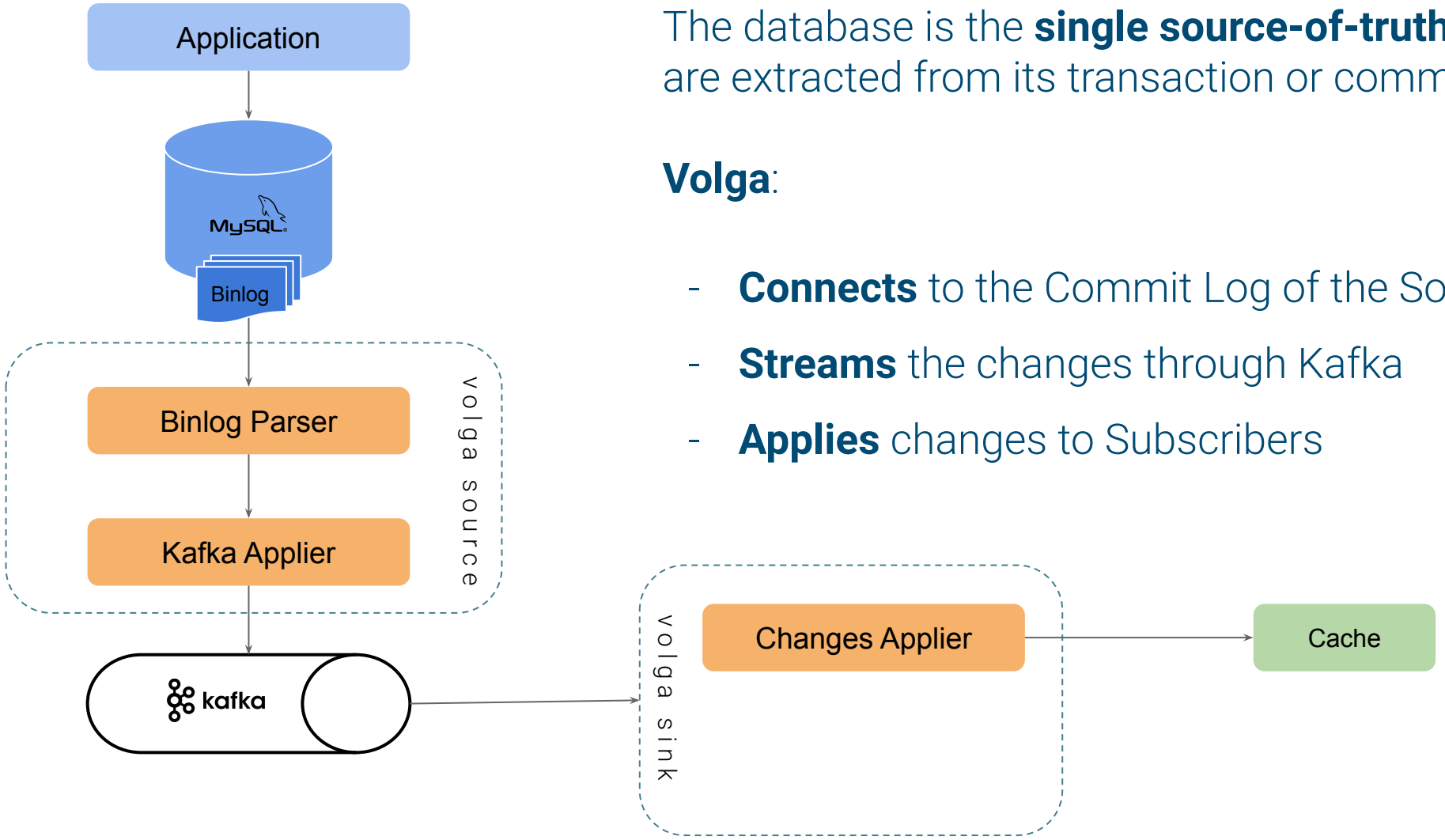


The database is the **single source-of-truth** and changes are extracted from its transaction or commit log.

Volga:

- **Connects** to the Commit Log of the Source DB
- **Streams** the changes through Kafka

Log Mining with Volga

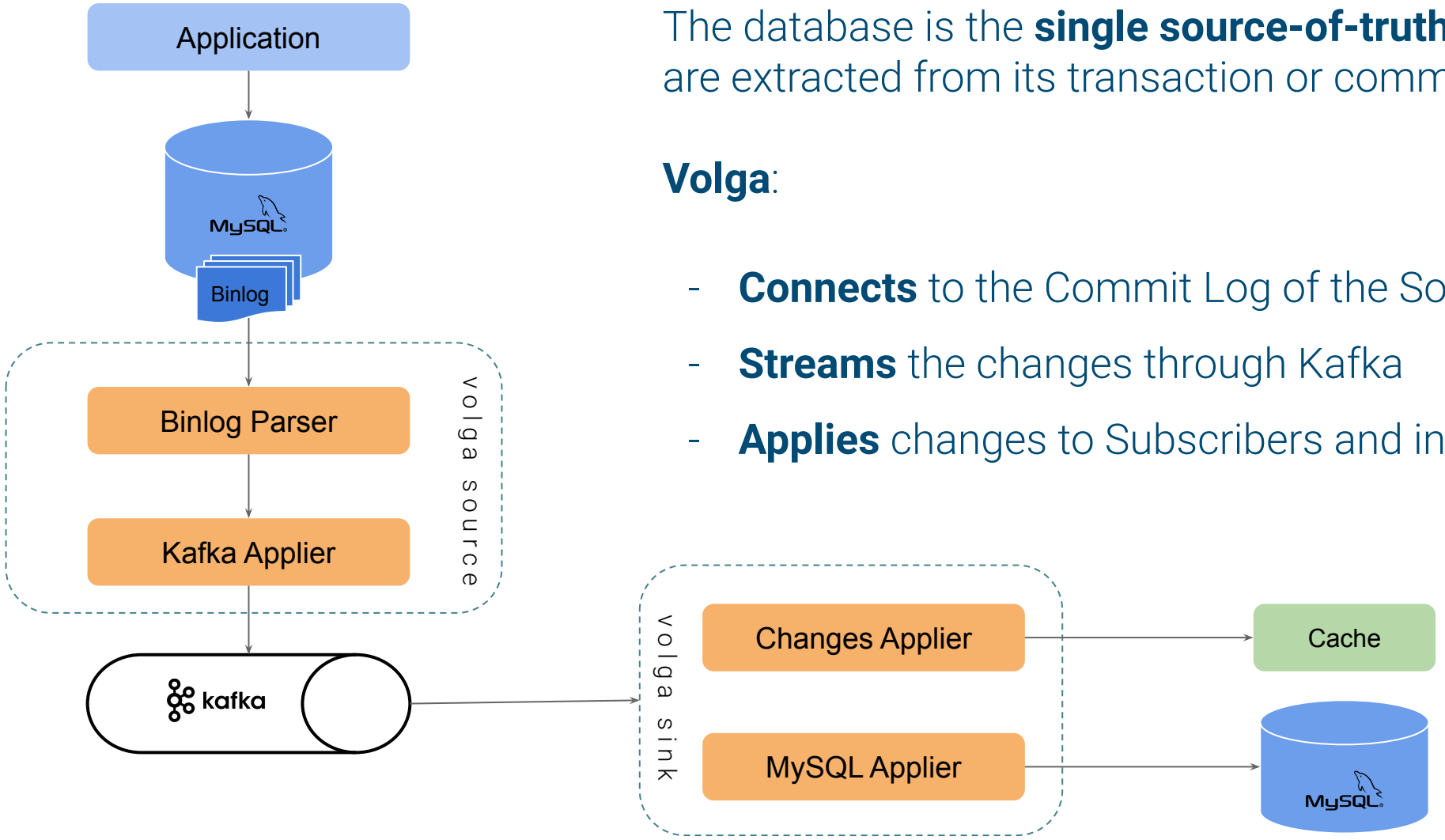


The database is the **single source-of-truth** and changes are extracted from its transaction or commit log.

Volga:

- **Connects** to the Commit Log of the Source DB
- **Streams** the changes through Kafka
- **Applies** changes to Subscribers

Log Mining with Volga

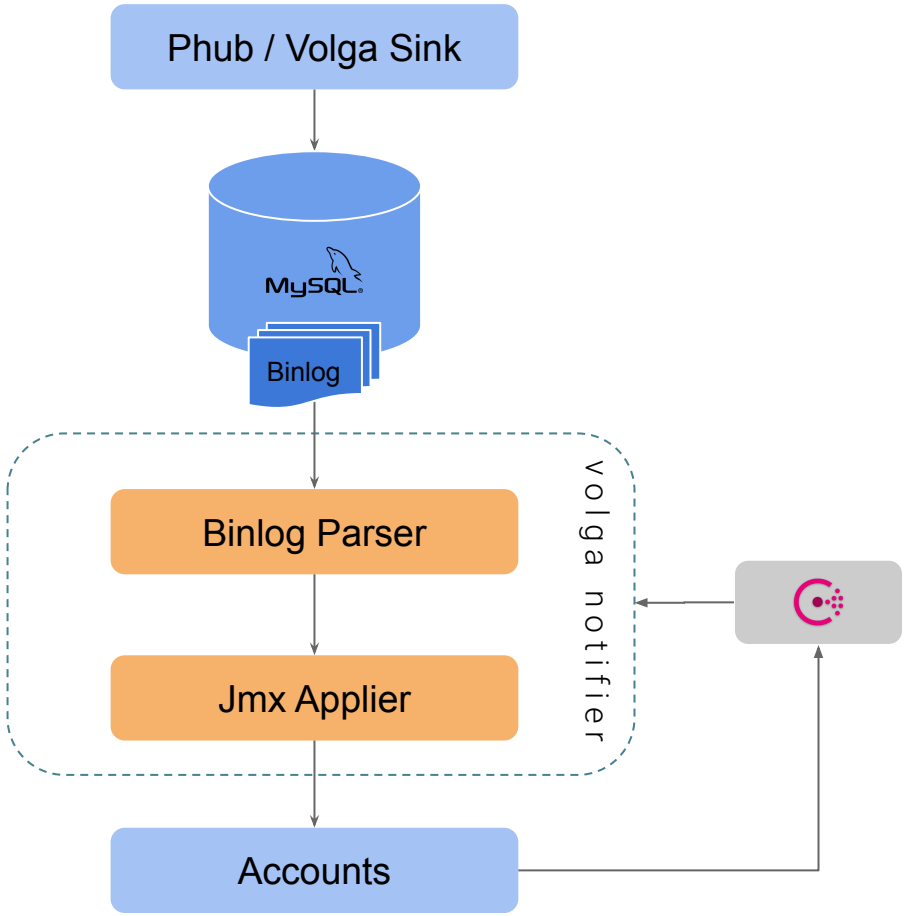


The database is the **single source-of-truth** and changes are extracted from its transaction or commit log.

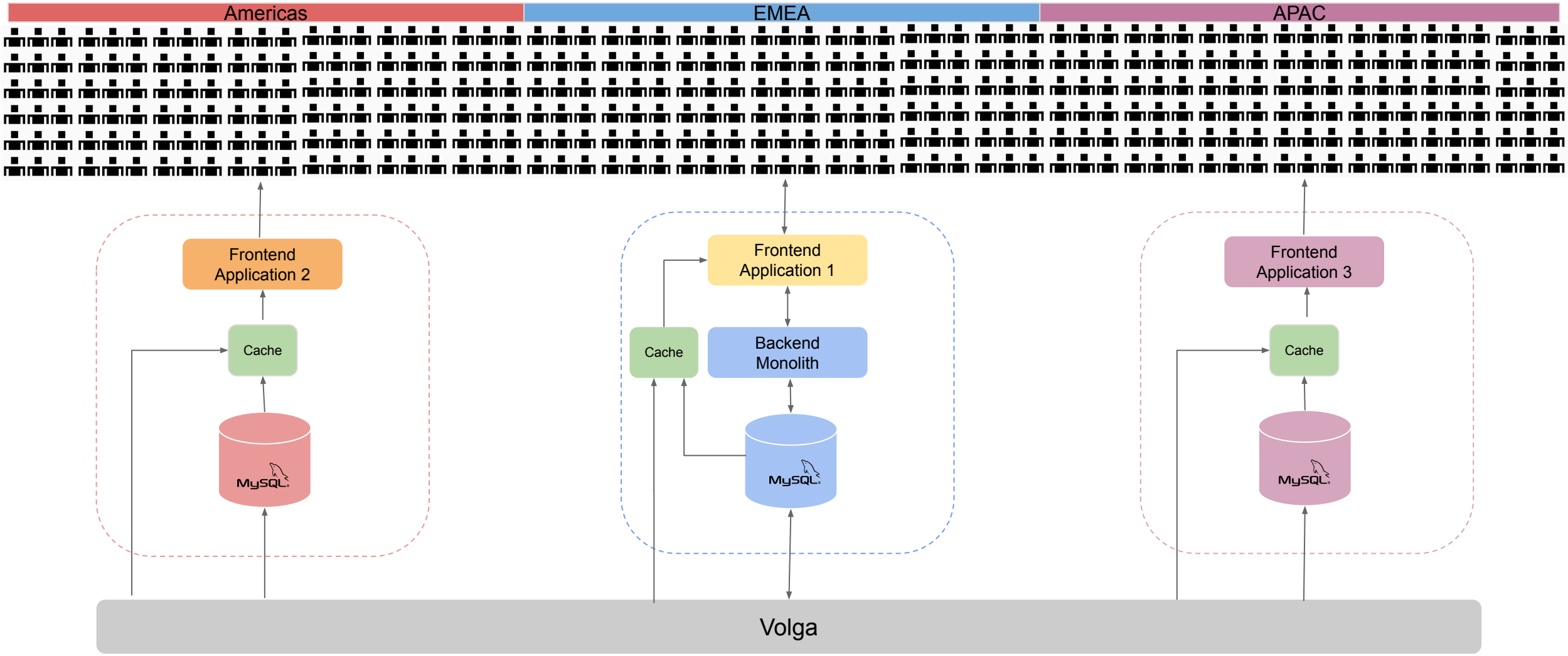
Volga:

- **Connects** to the Commit Log of the Source DB
- **Streams** the changes through Kafka
- **Applies** changes to Subscribers and in-sync Replicas

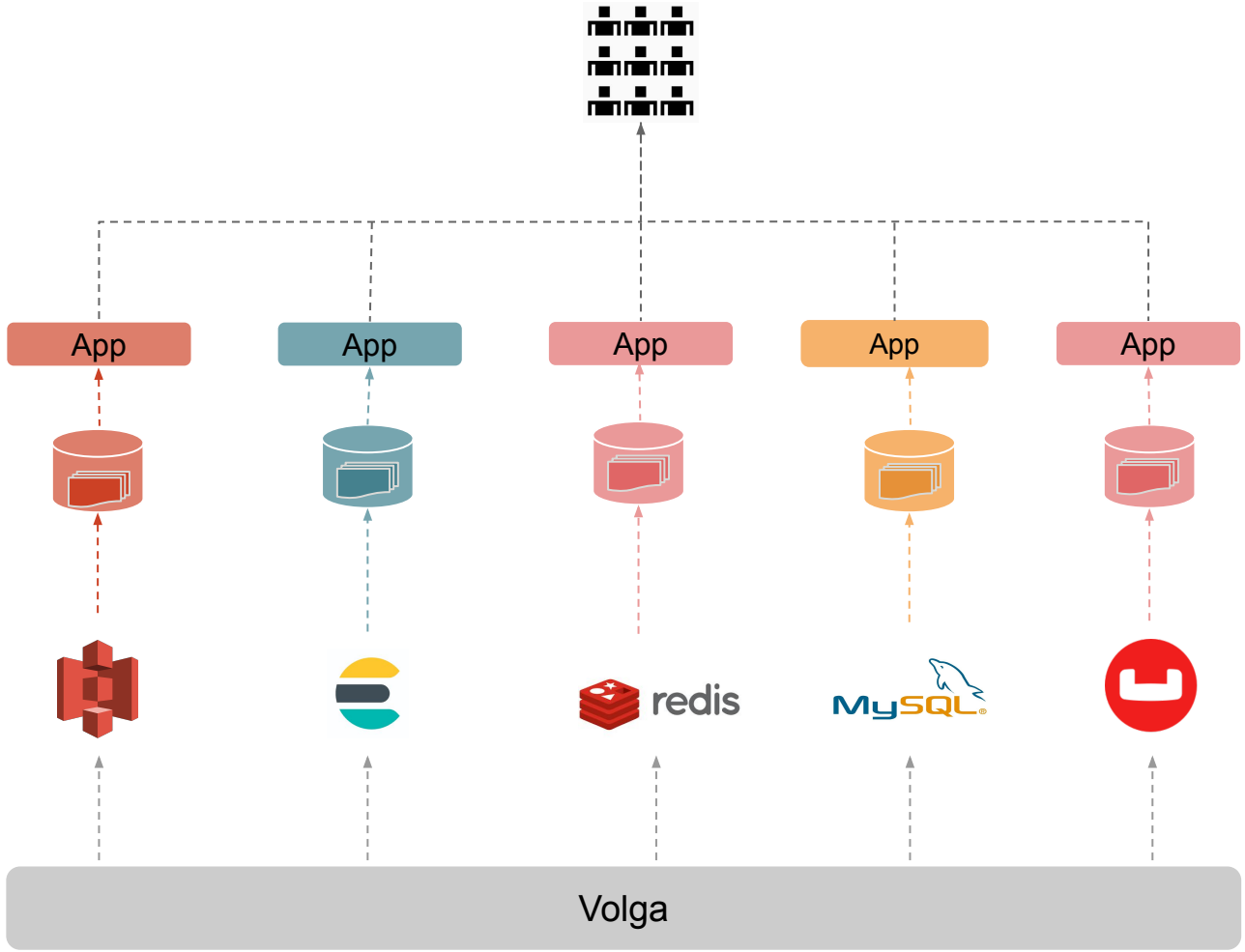
Log Mining with Volga



No Dual Writes with Volga



Make you data available for everyone, as they want



Via Volga, we can send data to:

- secondary indexes
- analytic platforms
- caches
- homogeneous in-sync replicas
- heterogeneous in-sync replicas

The Volga Contract

- **At-least-once** semantic
- **Sequential Consistency** (Binlog ordering + Kafka in-order delivery)

Additional Features:

- Snapshotting for Bootstrap, Correctness Check and DR
- Database Write Access Patterns Metrics
- Different level of partitioning (key-based, table, transactions)
- Homomorphic hashing

Problem 2: The monolith

1. Scaling Read workload across regions ✓
2. **Monolithic Architecture**

Drawbacks of the monolith

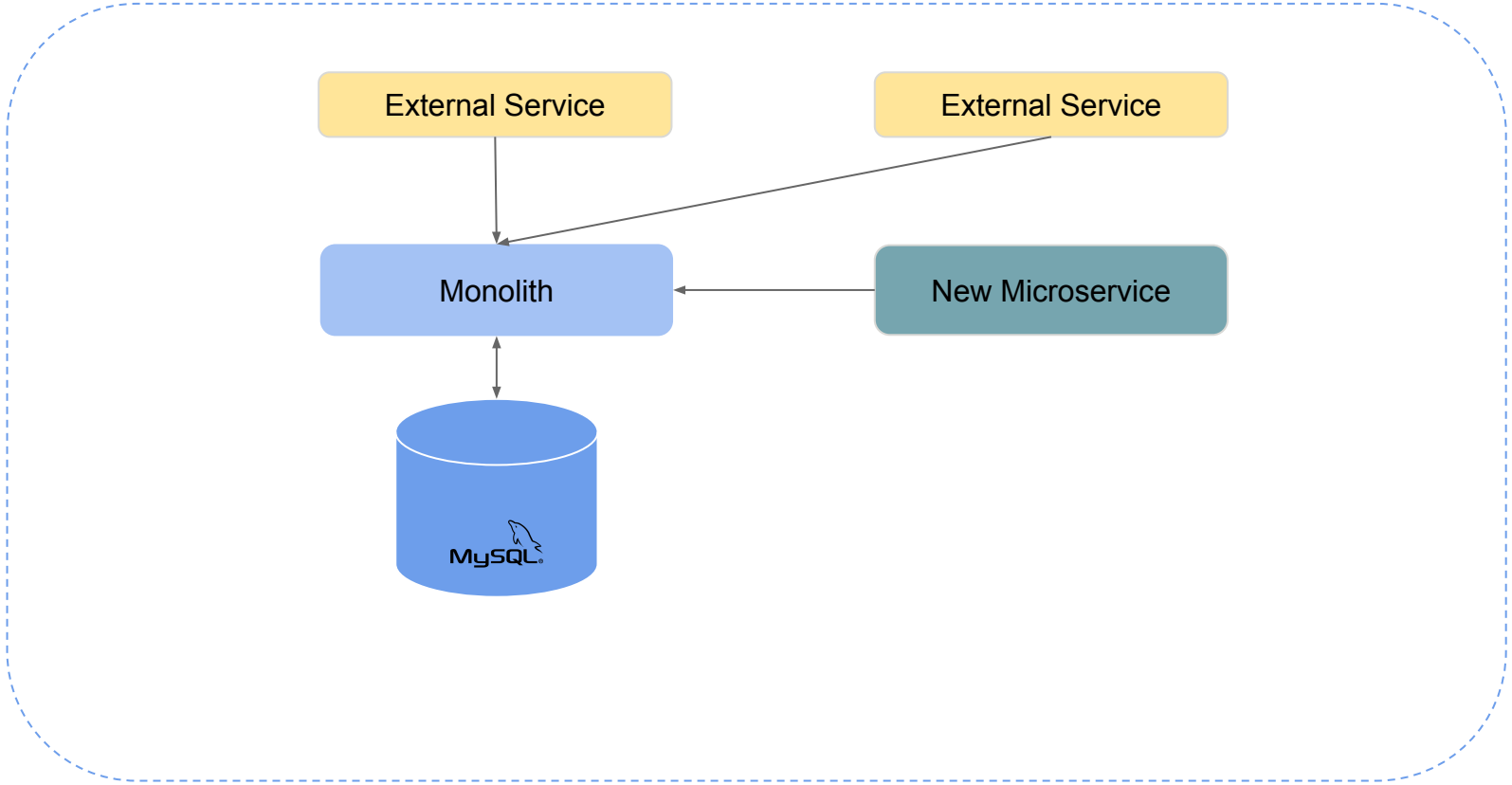
1. Incapacity to independently scale business domains
2. Barrier to innovation
3. Difficult coordination
4. Developers bad mood

The Strangle Pattern

The migration can be decomposed in 3 steps:

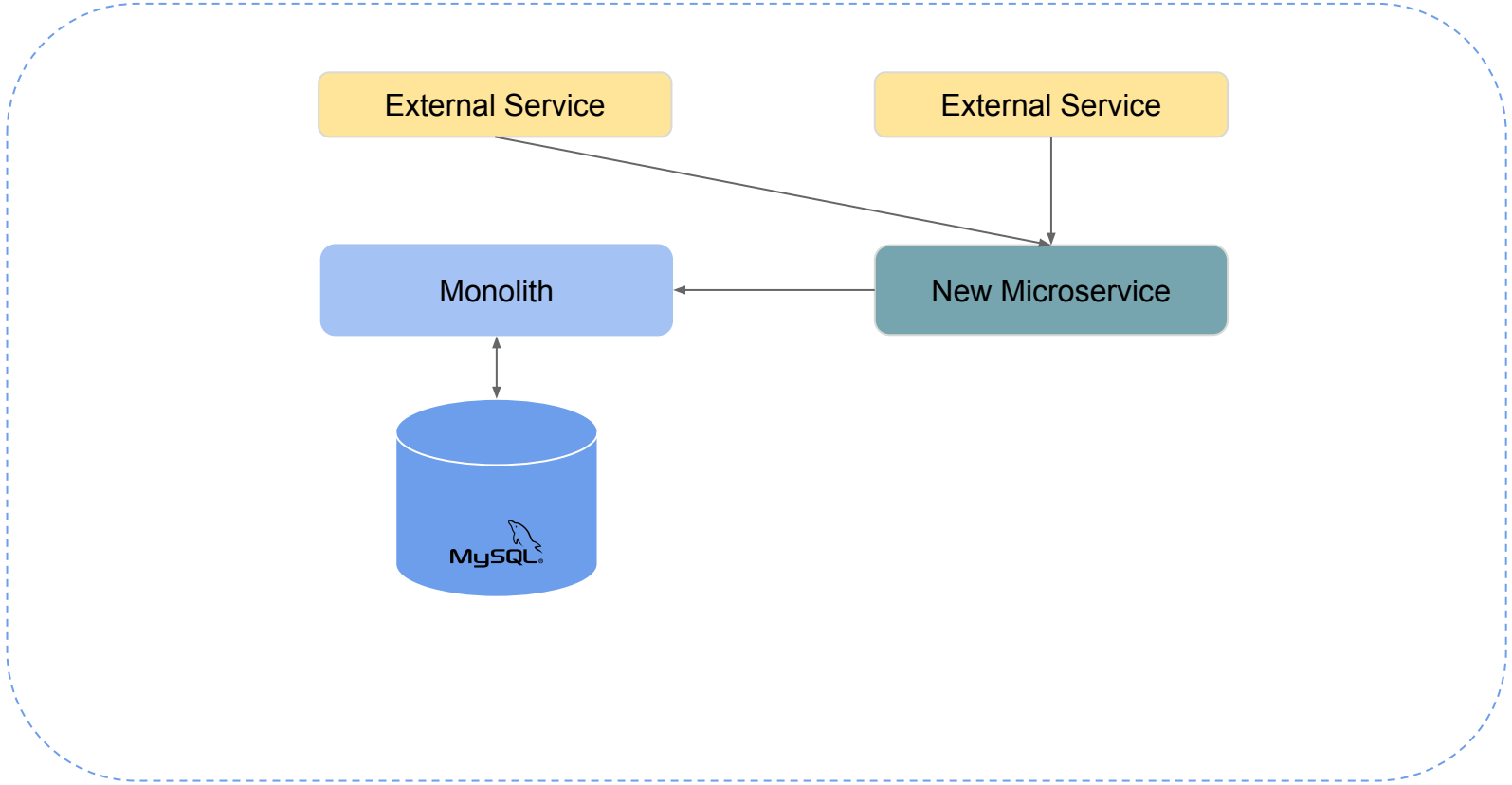
- **Transform** - Creation of the new Microservice extracting the new generation of a subset of functionalities from the Monolith
- **Coexist** - The functionalities are exposed from both the Monolith and the Microservice at the same time
- **Eliminate** - Remove the functionalities from the monolith

Strangler Pattern: Transform



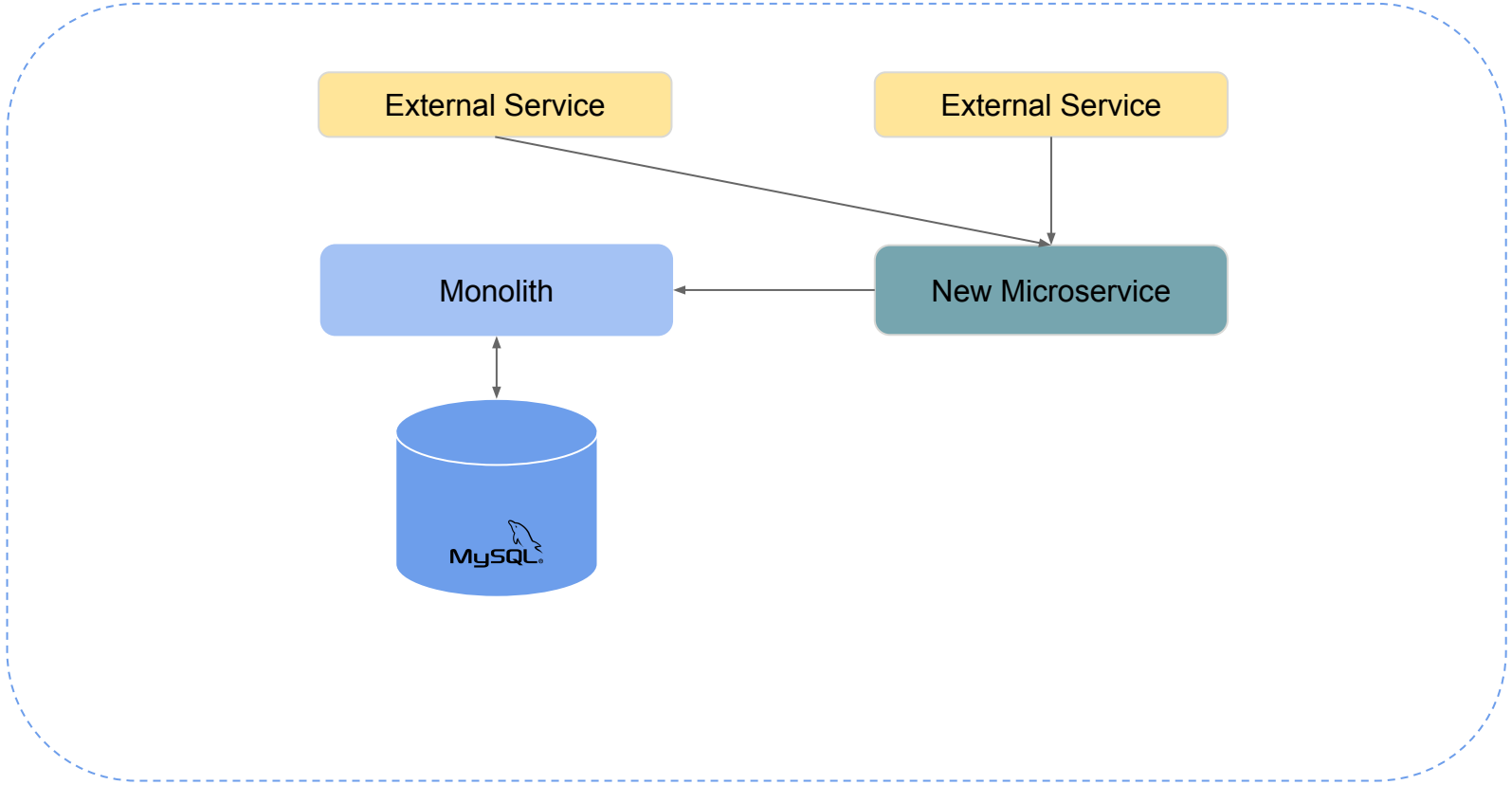
Build the new microservice still dependent on the Monolith

Strangler Pattern: Transform



Migrate the dependencies

Strangler Pattern: Transform

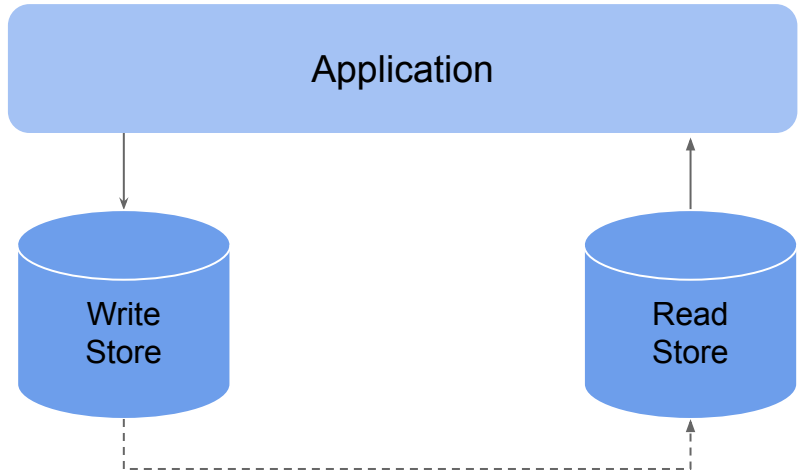


How do we migrate the dependency to the data store?

Read model migration via CQRS

CQRS stands for Command Query Responsibility Segregation

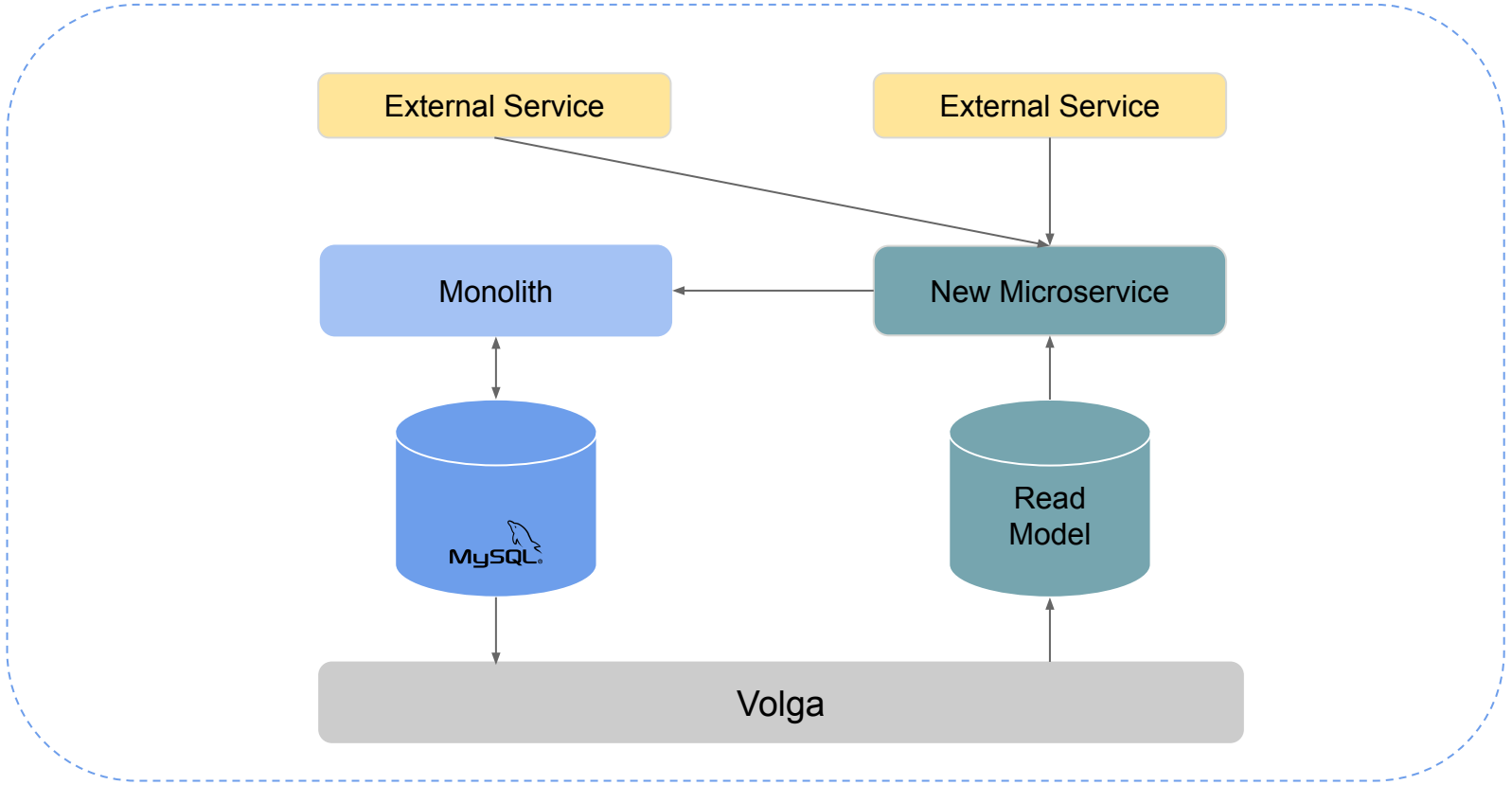
Reads and Writes are segregated into separate models



Benefits of Command Query Responsibility Segregation

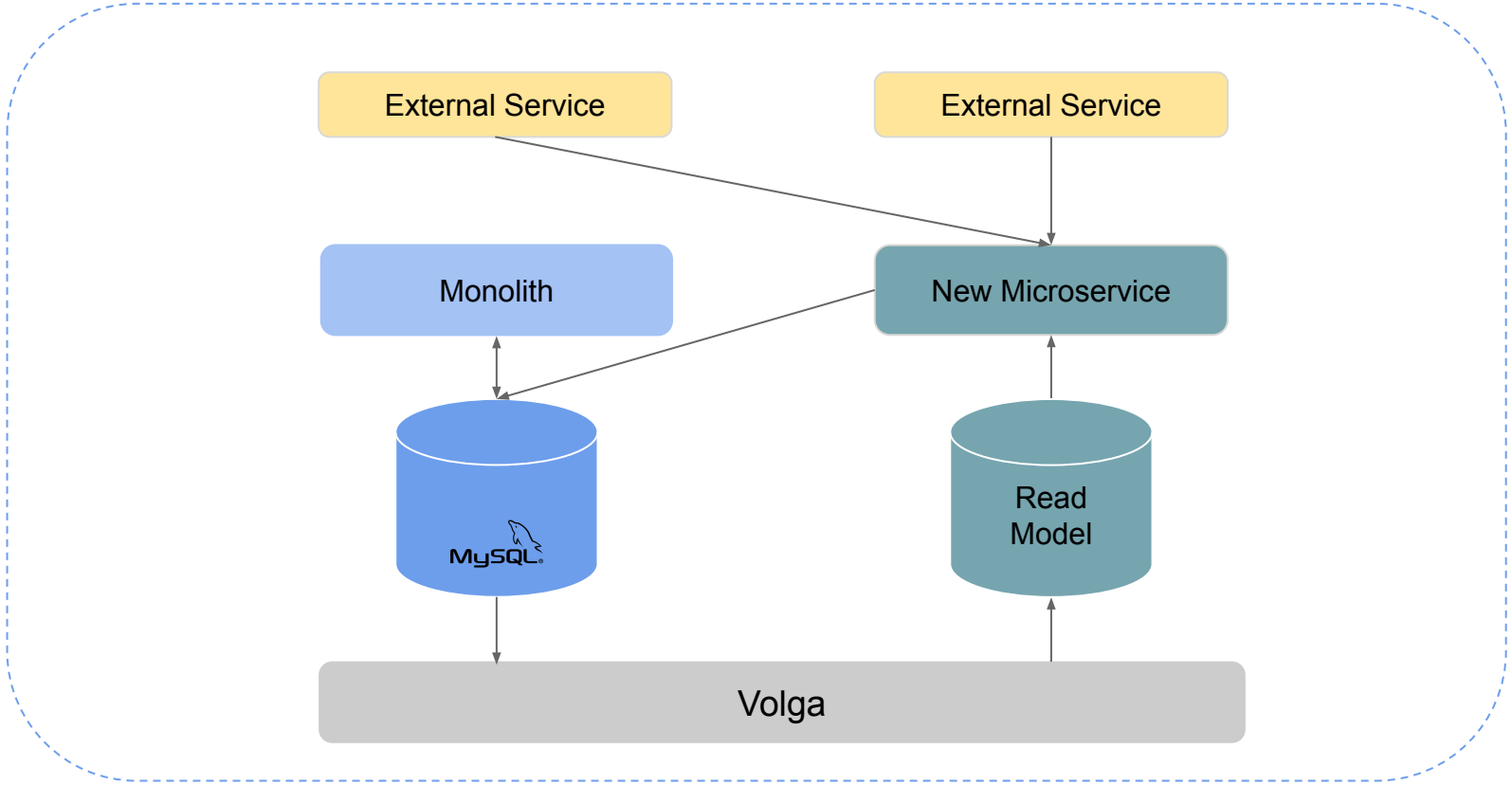
- Independent scaling of write and read workloads
- Independent Optimized data schemas
- Separation of concerns
- Improved security
- Loose coupling
- Support for incremental Migration

Strangler Pattern: Coexist



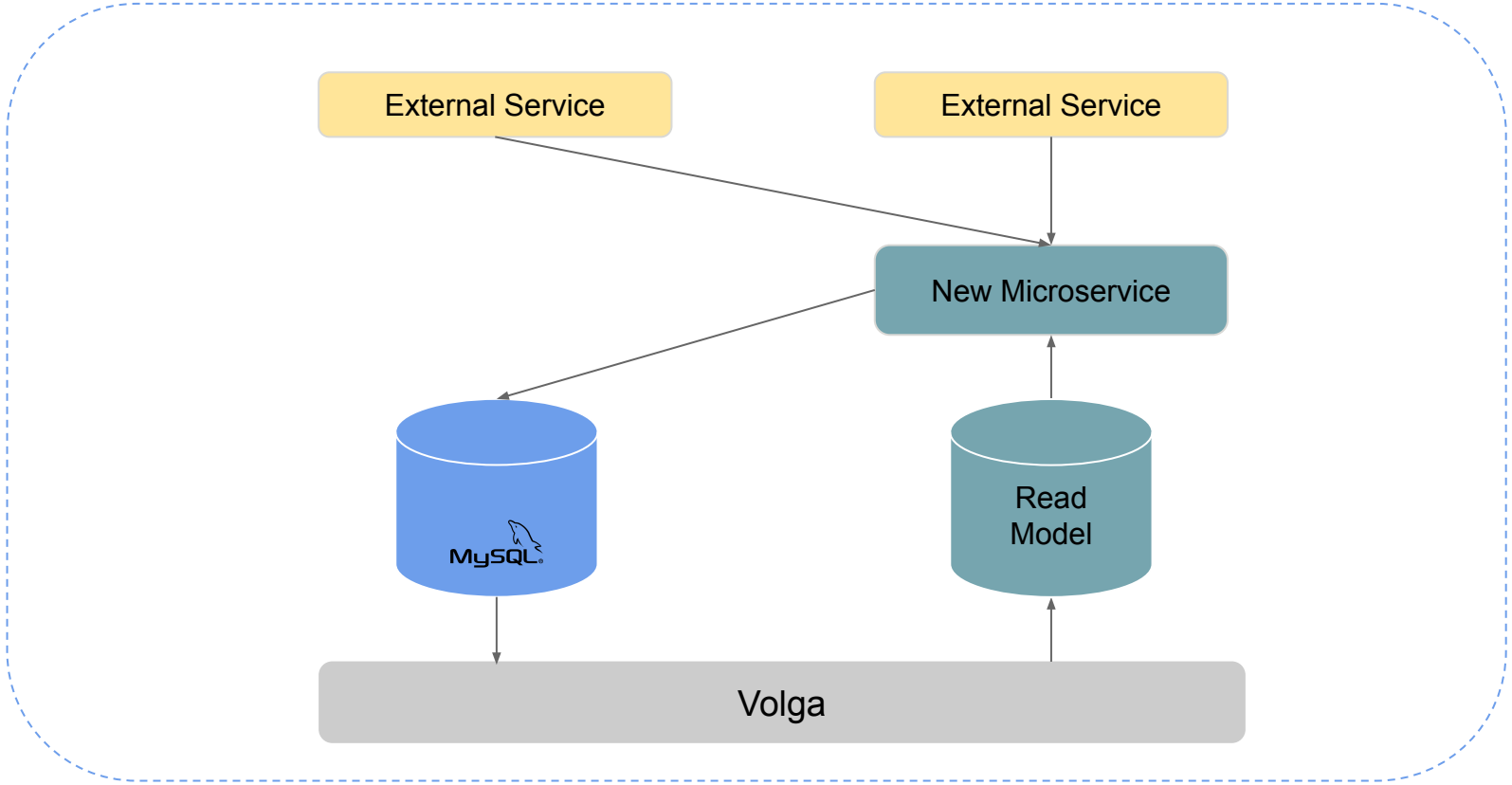
Writes are served by the old model, Reads by the new Read Model

Strangler Pattern: Eliminate



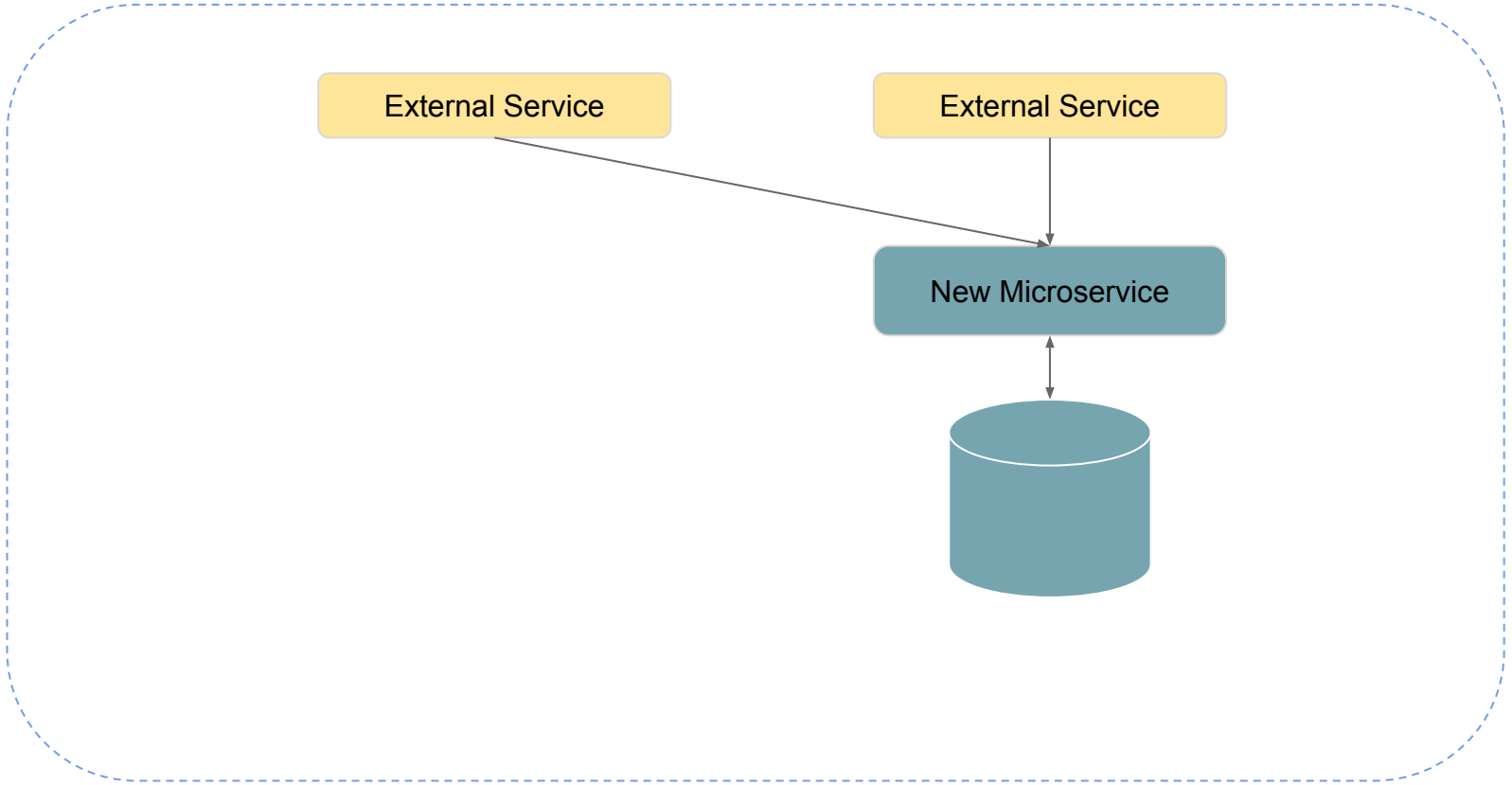
Remove the dependency to the Monolith

Strangler Pattern: Eliminate



Remove the dependency to the Monolith

Strangler Pattern: Eliminate

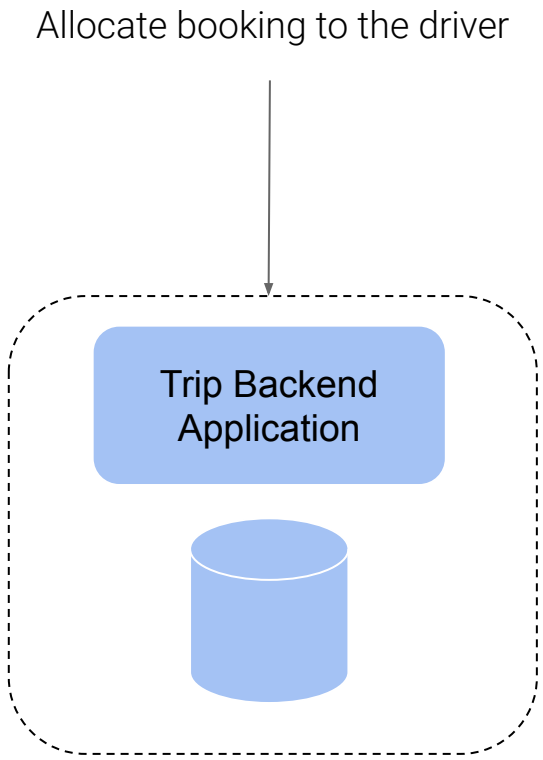


Migrate the Write Model

Problem 2: The monolith

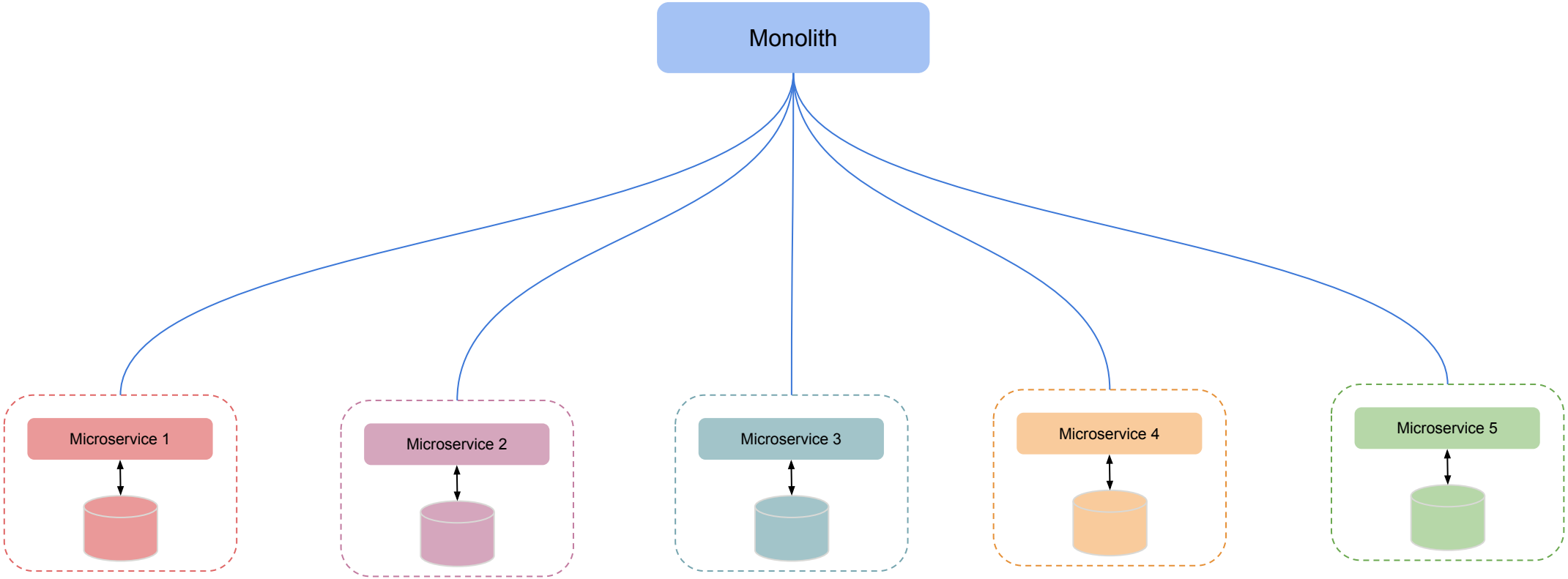
1. Read workload ✓
2. Drawbacks of the Monolithic Architecture ✓

The Delight of Transactions

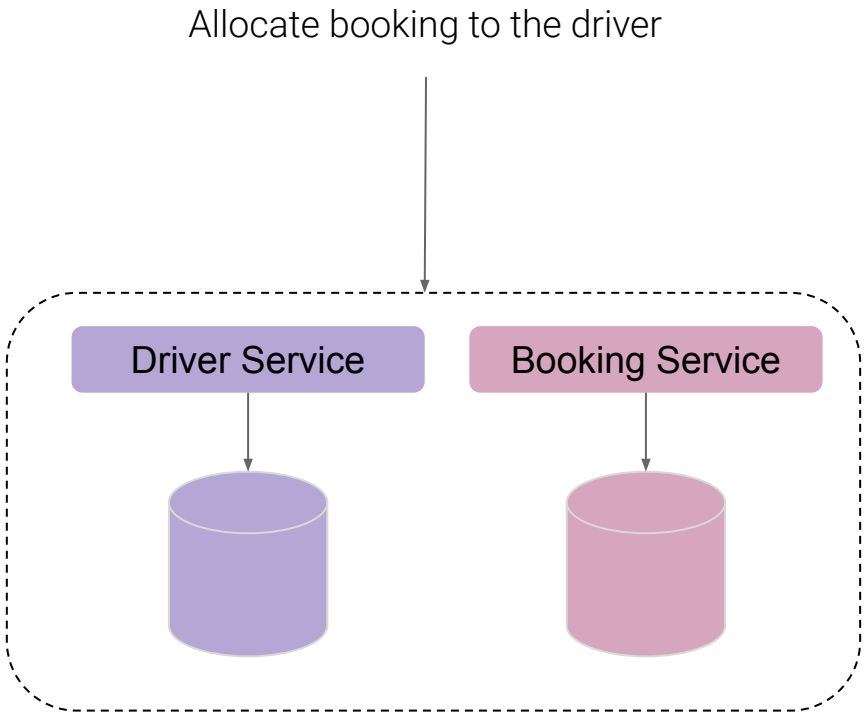



```
Start Transaction;
set driver_status=busy;
set booking_status=allotted;
Commit Transaction;
```

Microservices



The Horror of transactions



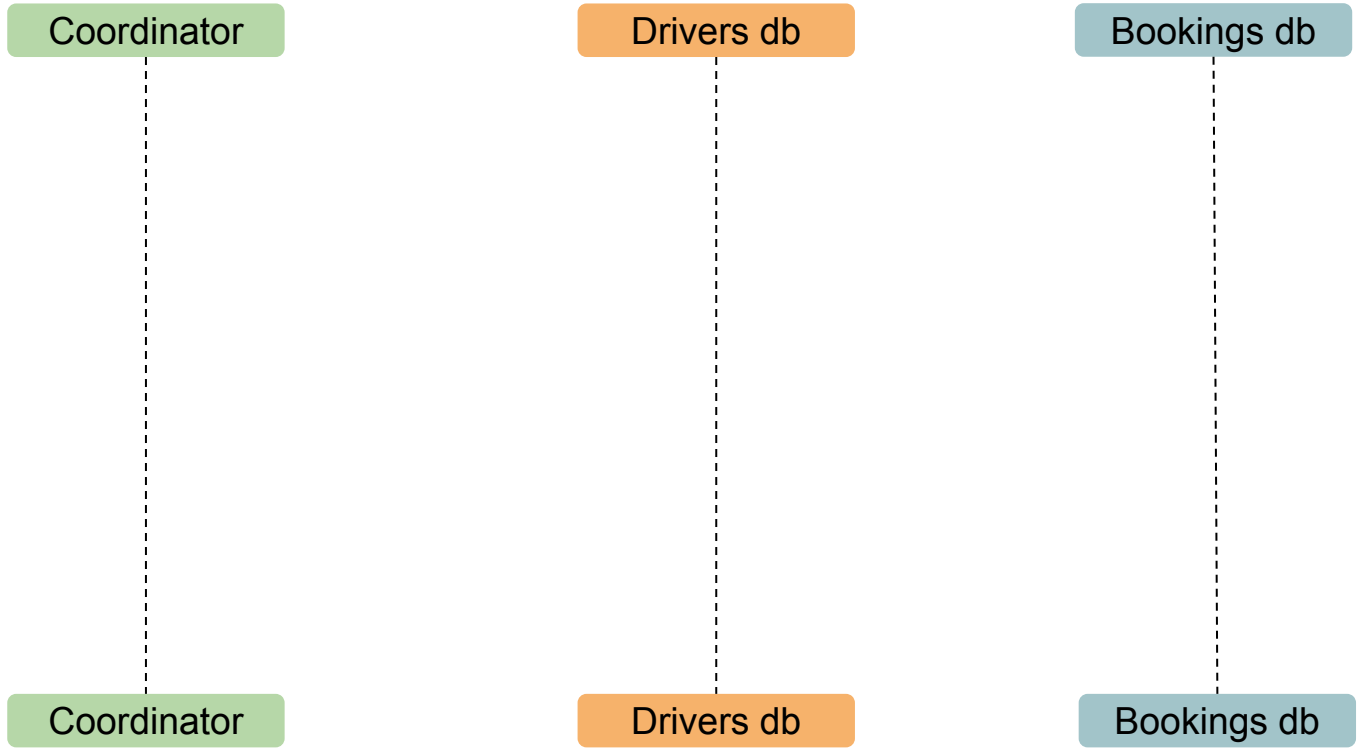

```
Start Transaction;
set driver_status=busy
set booking_status=allotted
Commit Transaction;
```

How to implement Transactions with Microservices?

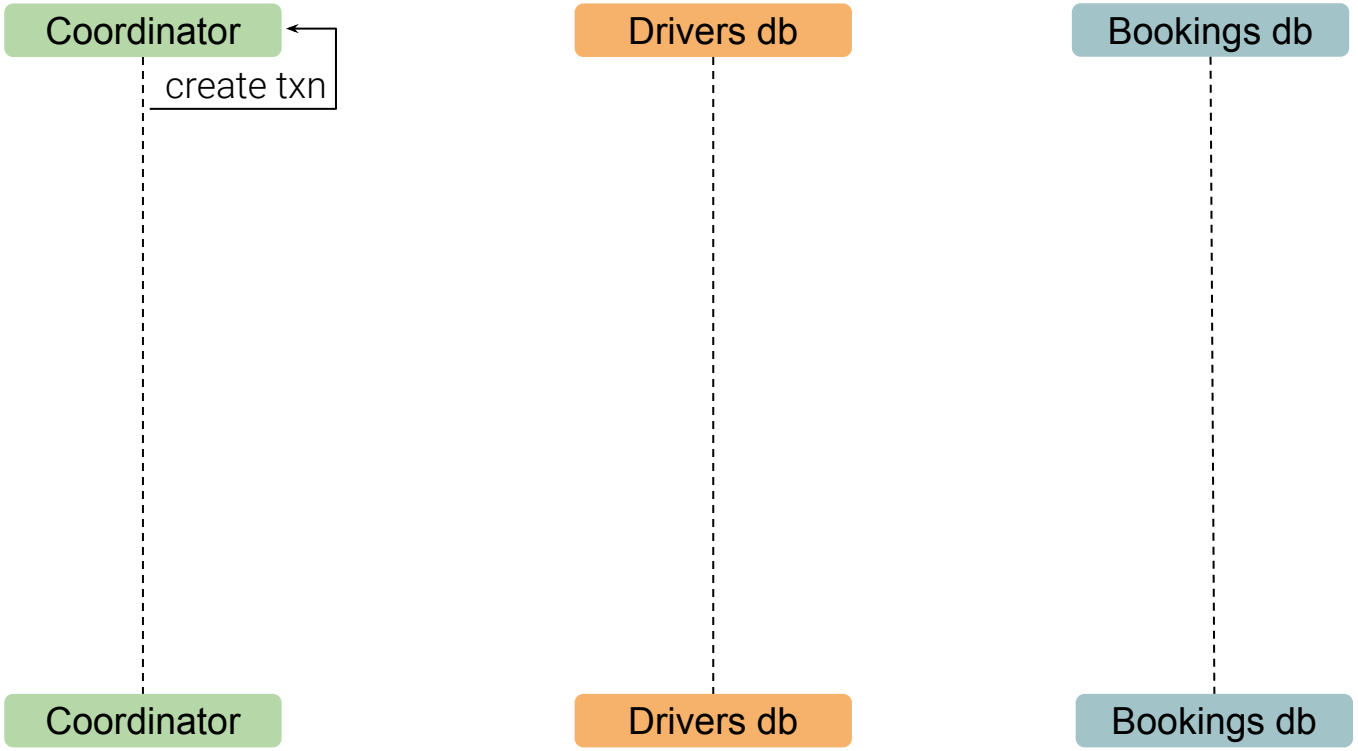
2-Phase Commit

Sagas

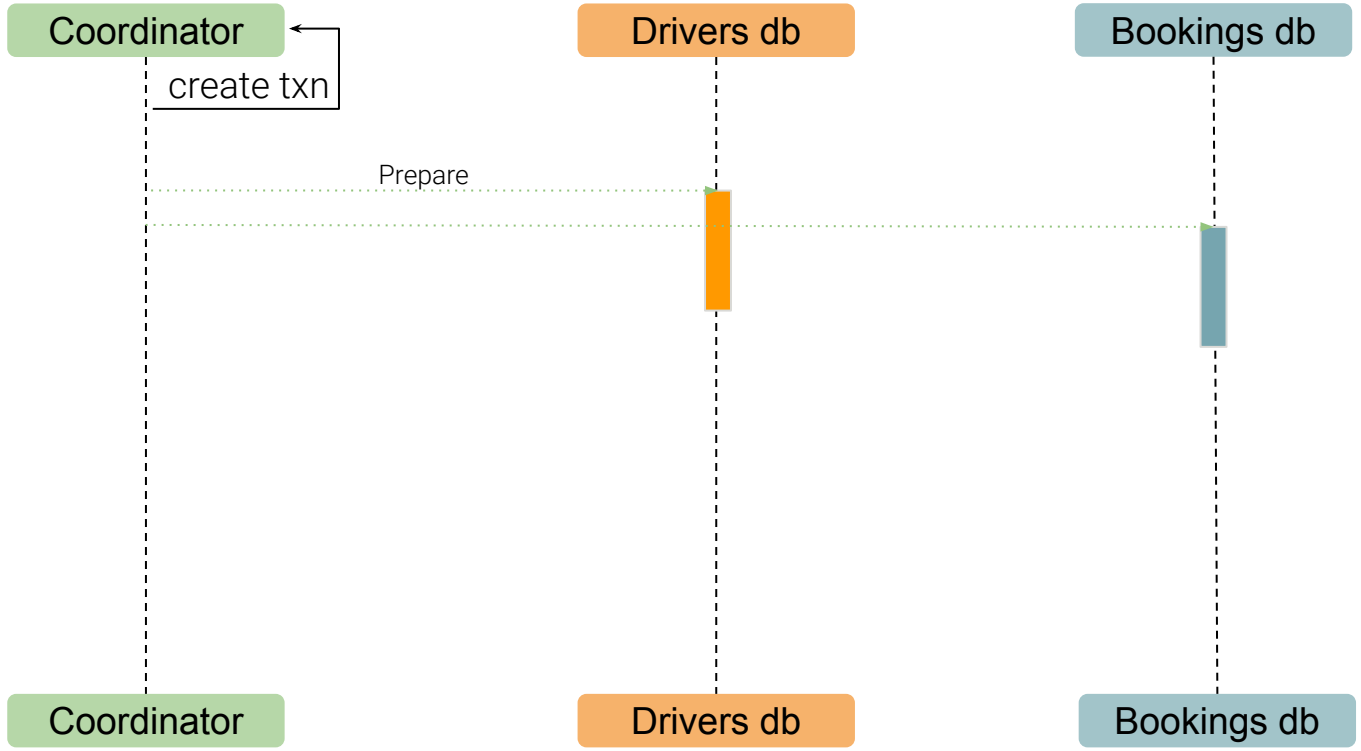
2-Phase Commit: XA Transaction



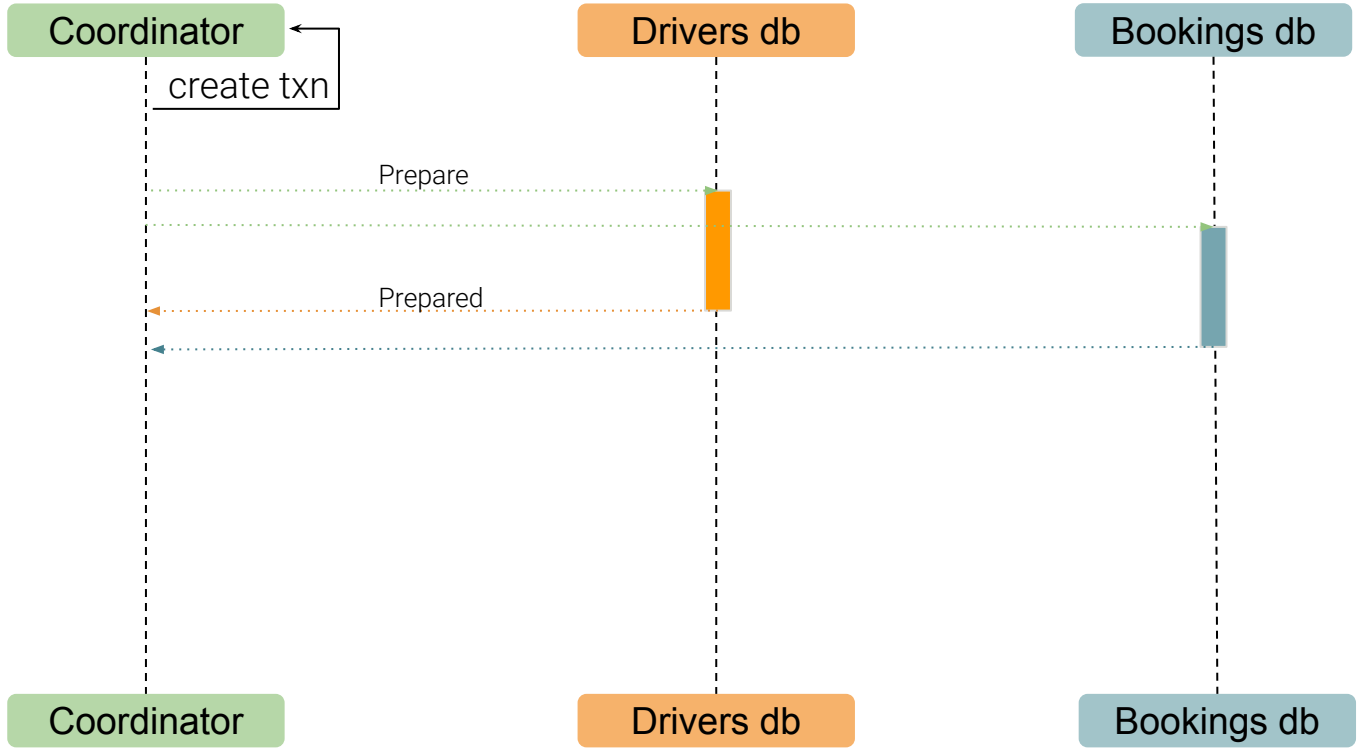
2-Phase Commit: XA Transaction



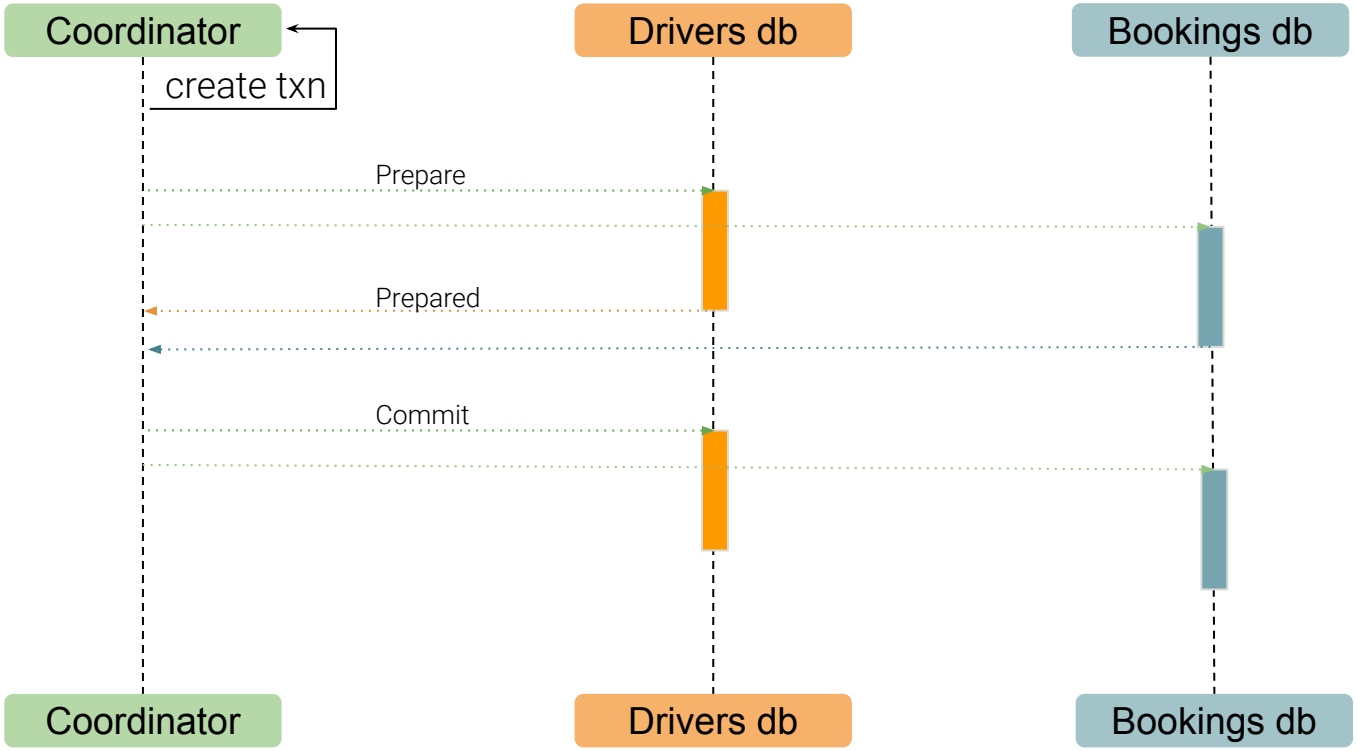
2-Phase Commit: XA Transaction



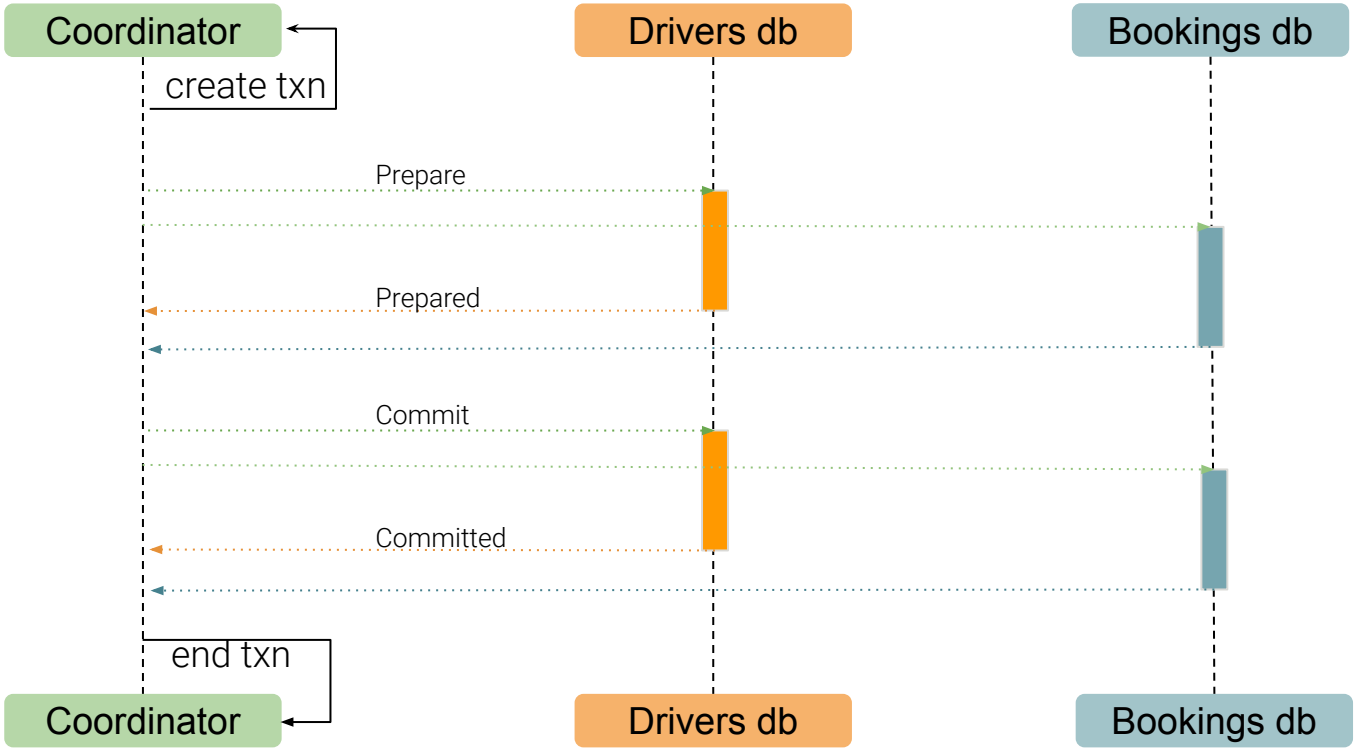
2-Phase Commit: XA Transaction



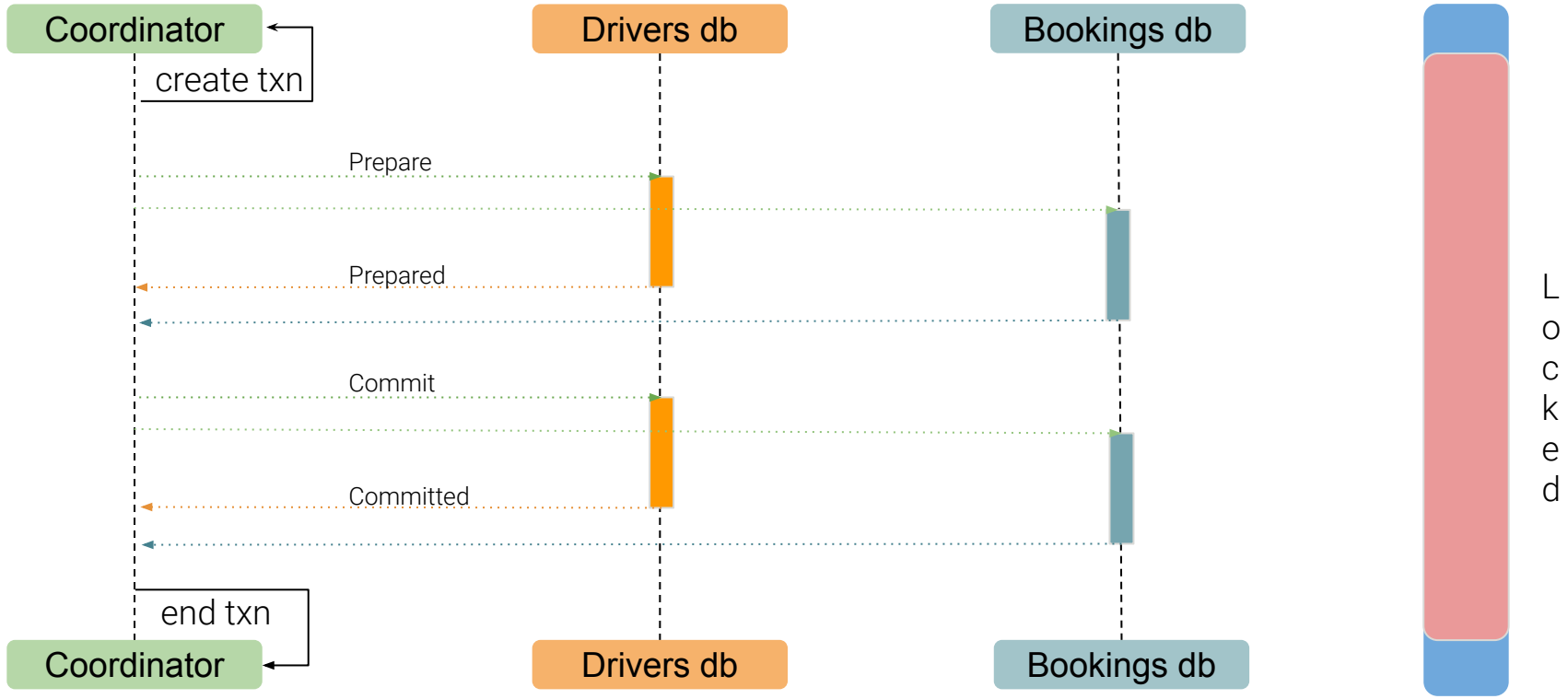
2-Phase Commit: XA Transaction



2-Phase Commit: XA Transaction



2-Phase Commit: XA Transaction

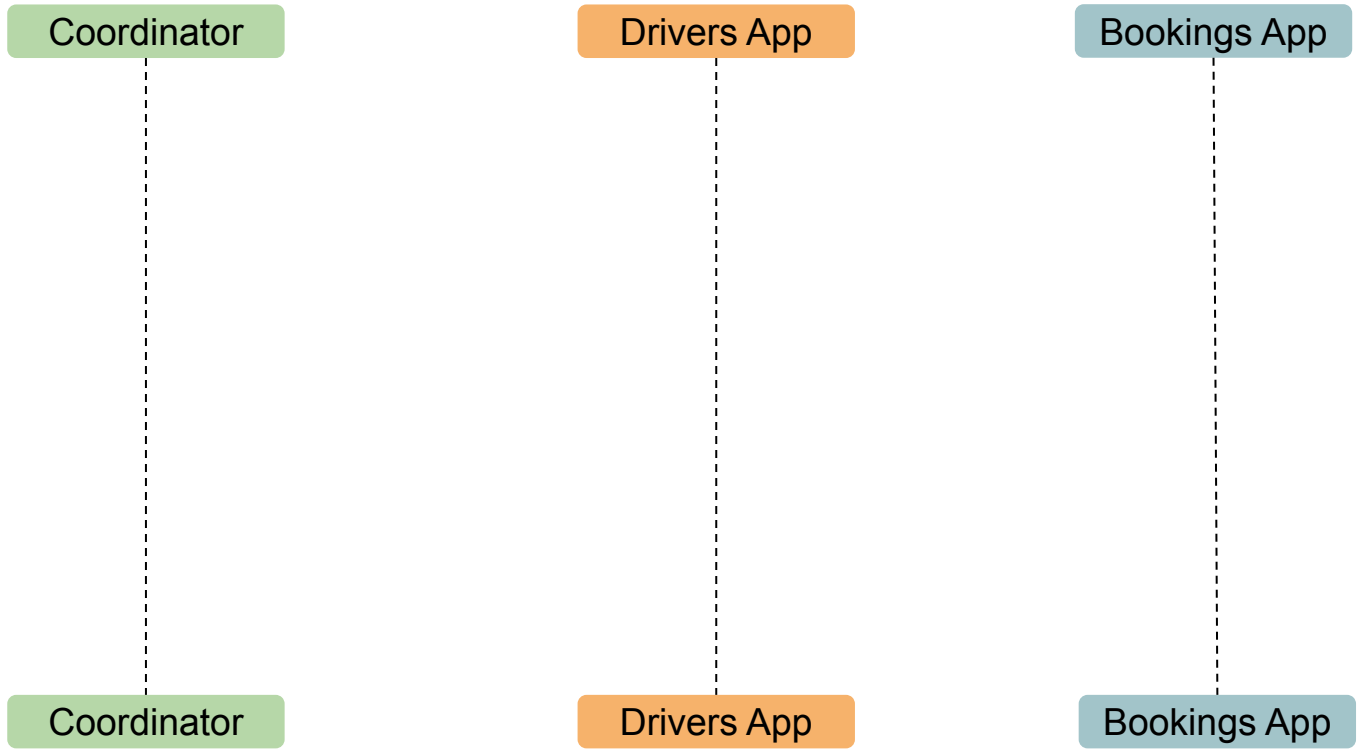


2-Phase Commit: Pros & Cons

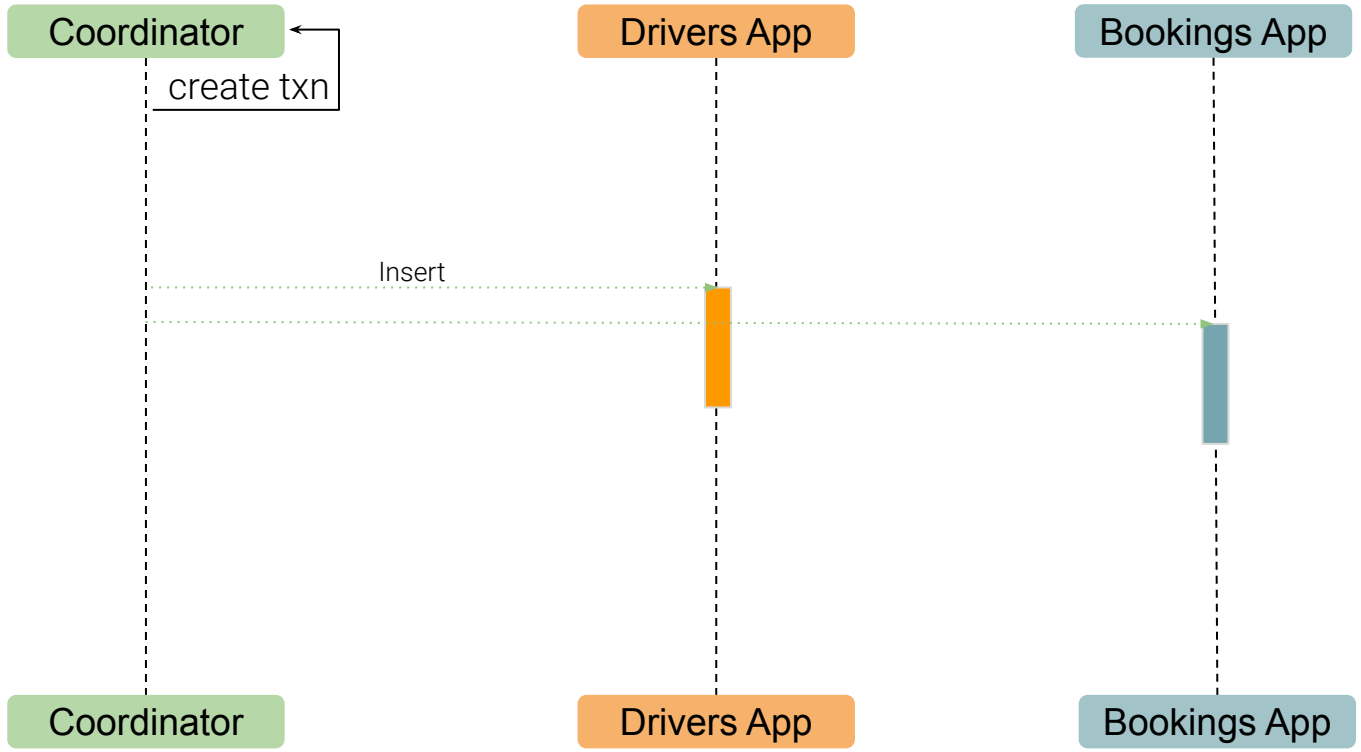
- Atomic Transaction
- Strong Consistency
- Read Write Isolation

- Slow (due to Locks)
- Deadlock

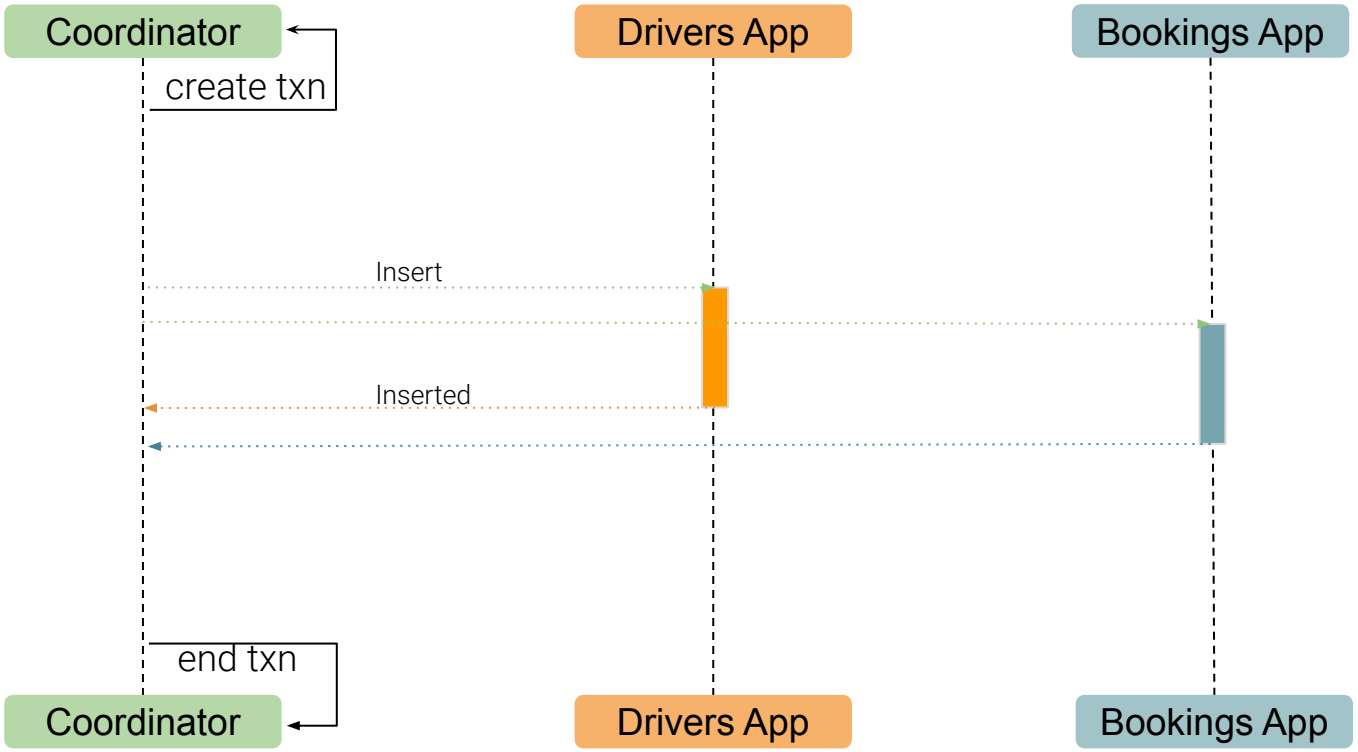
Sagas: Transaction



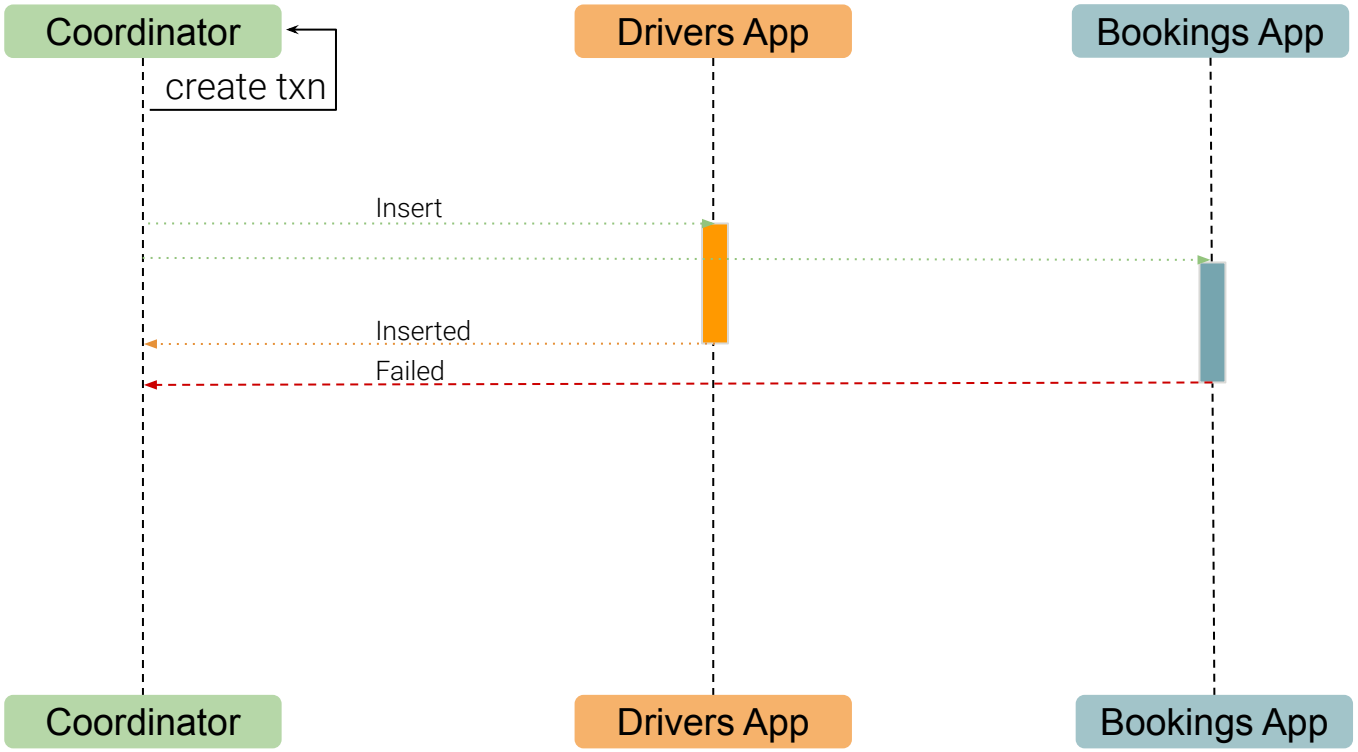
Sagas: Transaction



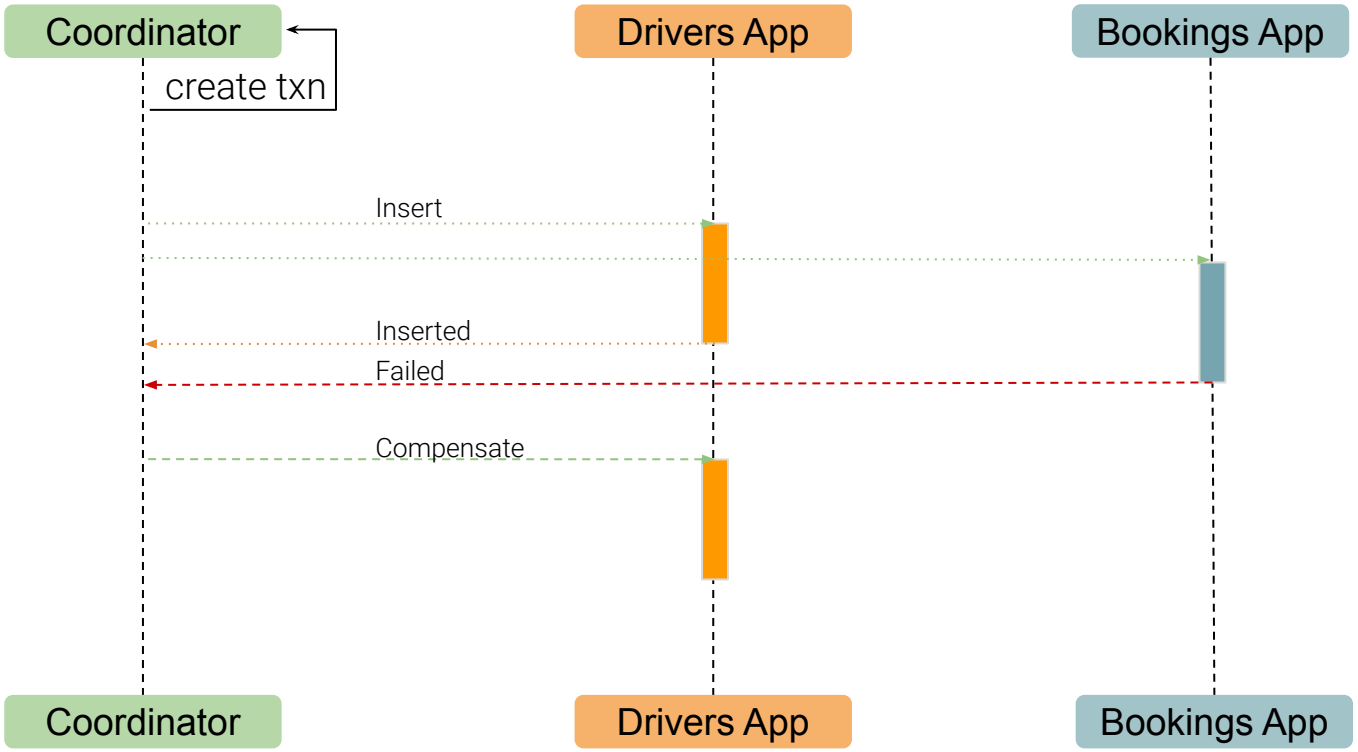
Sagas: Transaction



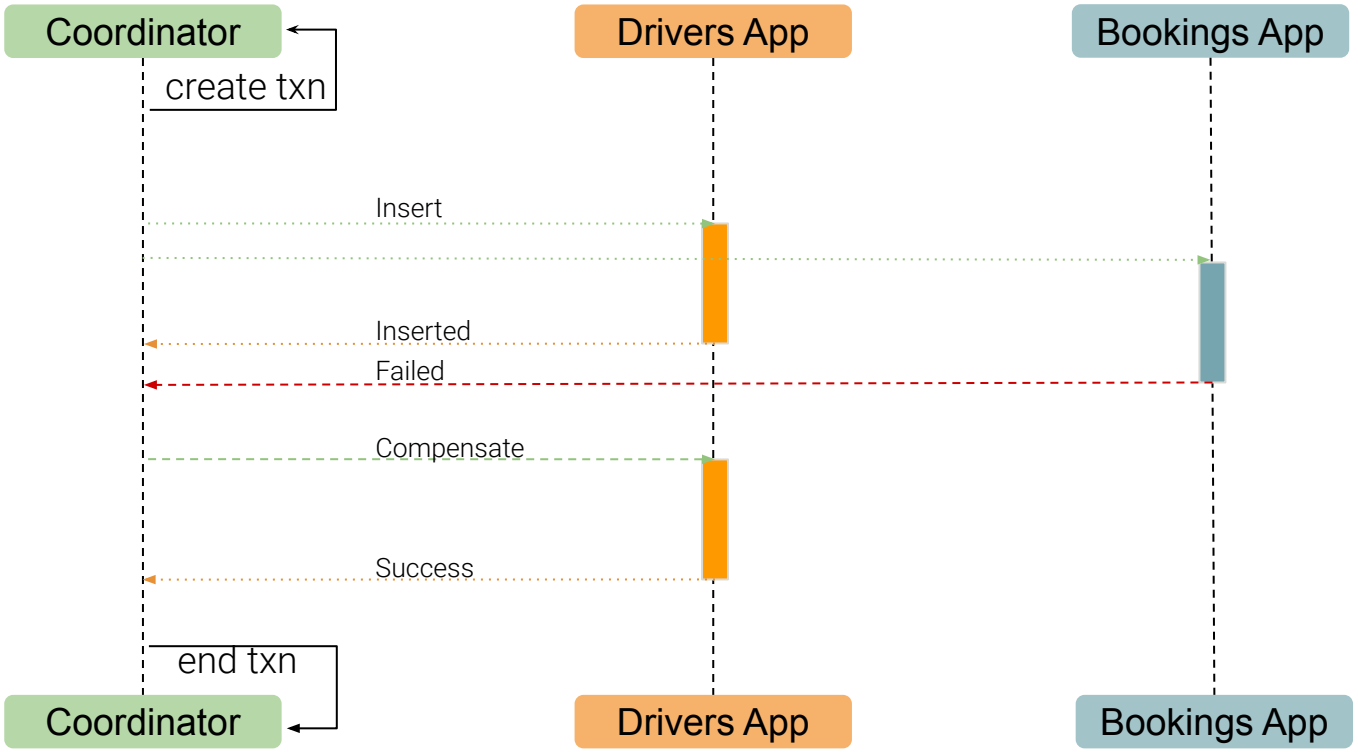
Sagas: Transaction



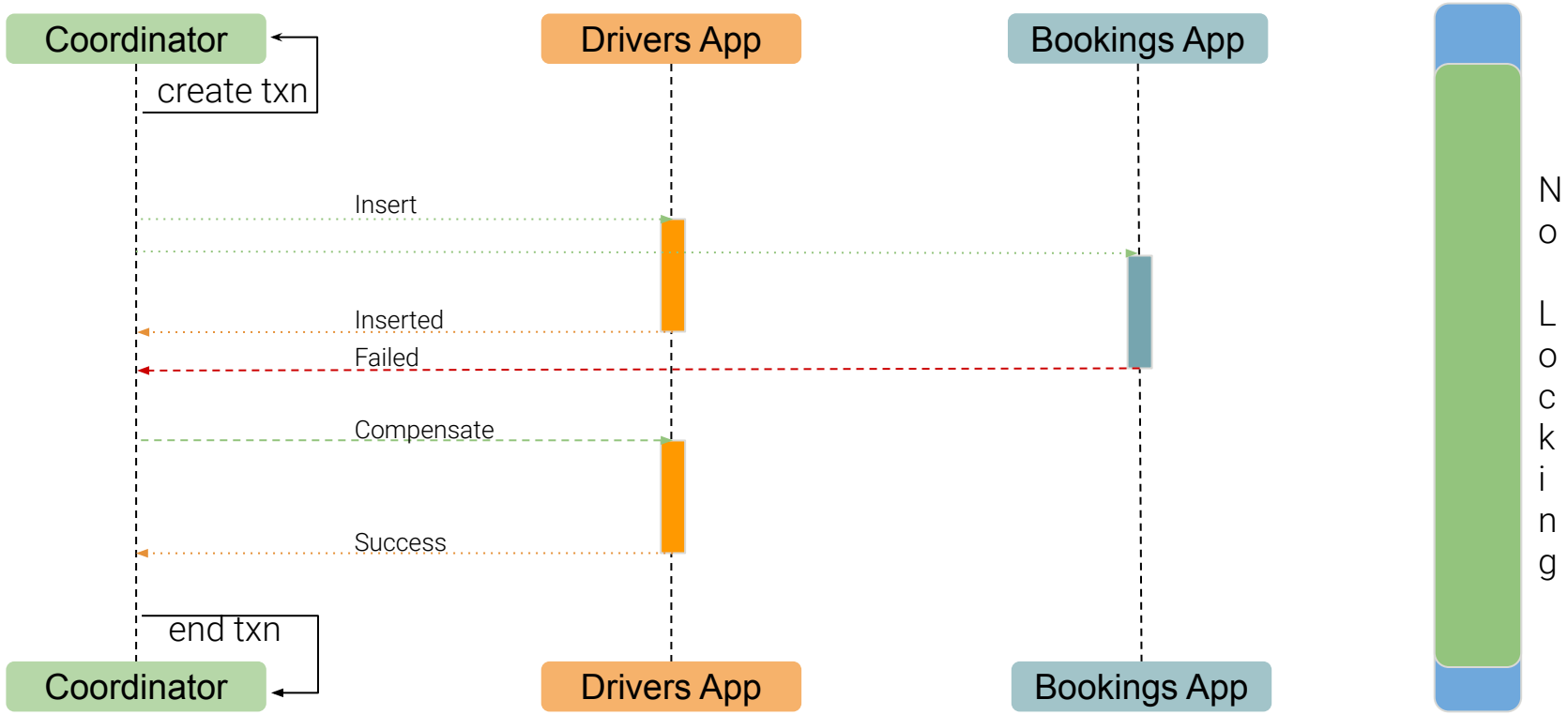
Sagas: Transaction



Sagas: Transaction



Sagas: Transaction



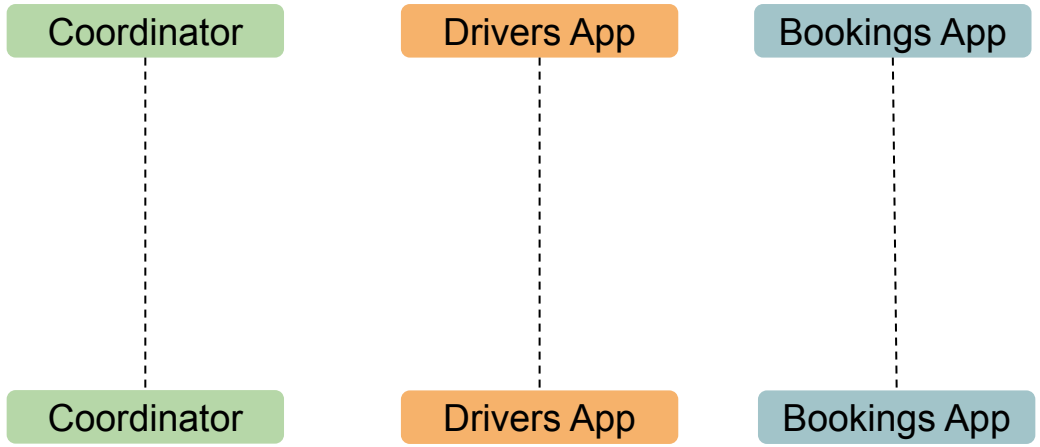
Sagas: Pros & Cons

- No Locks
- Long Lived Transaction
- No tight coupling

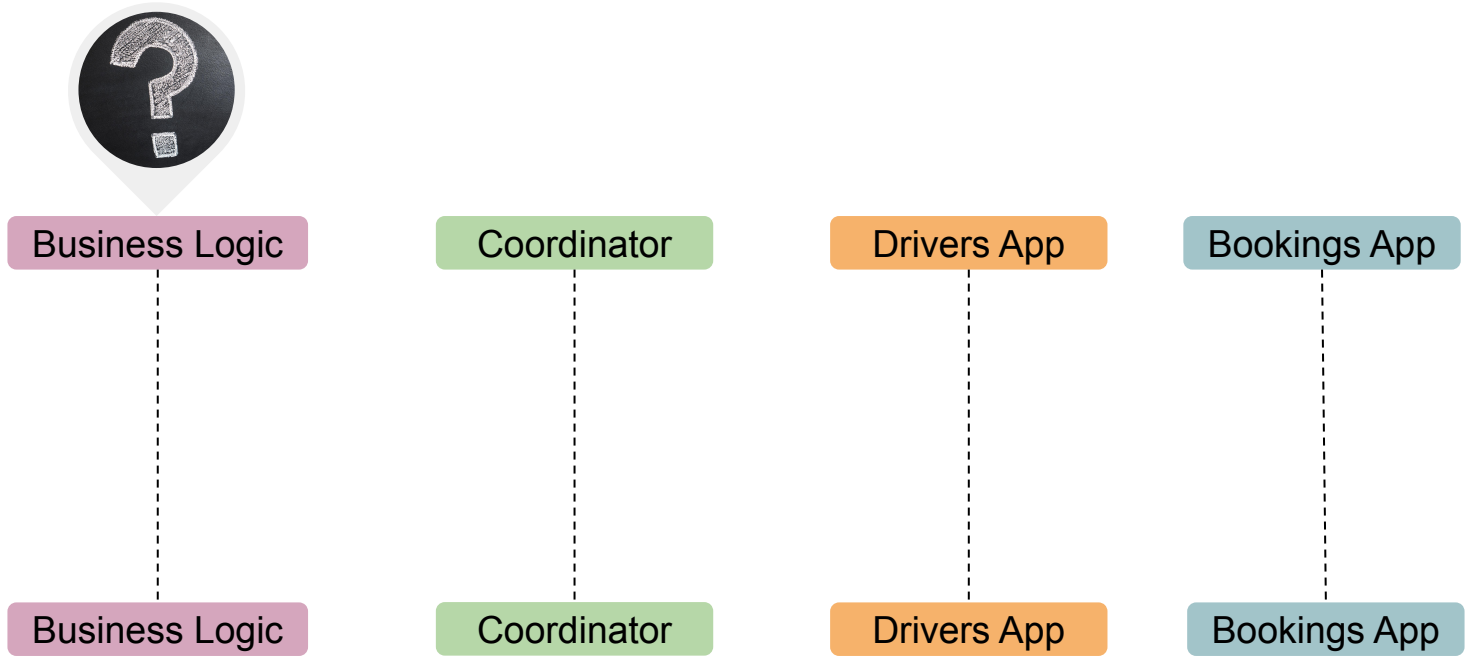
- Relaxed Isolation
- Difficult Compensations
- Bad User Experience

Is there any Better way?

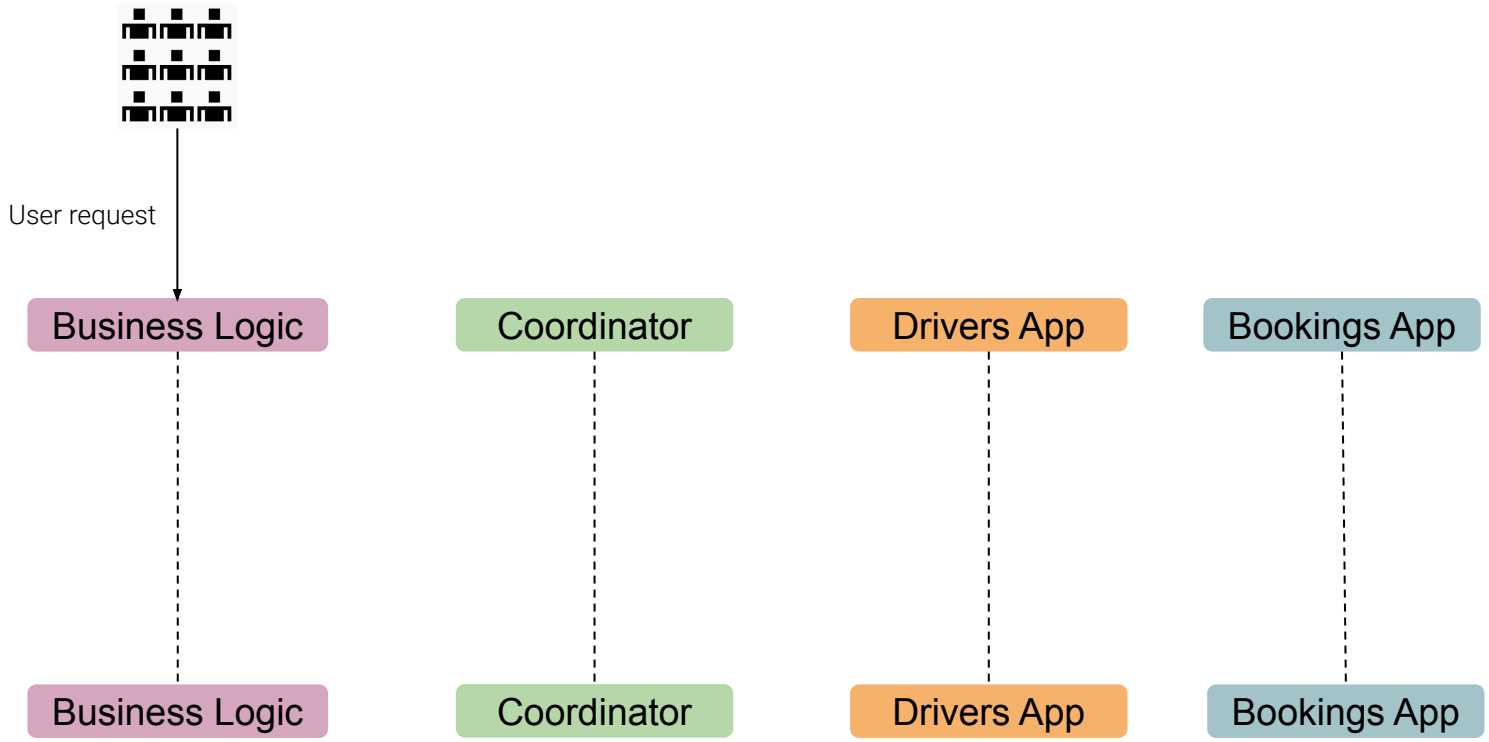
Shadows: A Distributed 2 Phase Commit



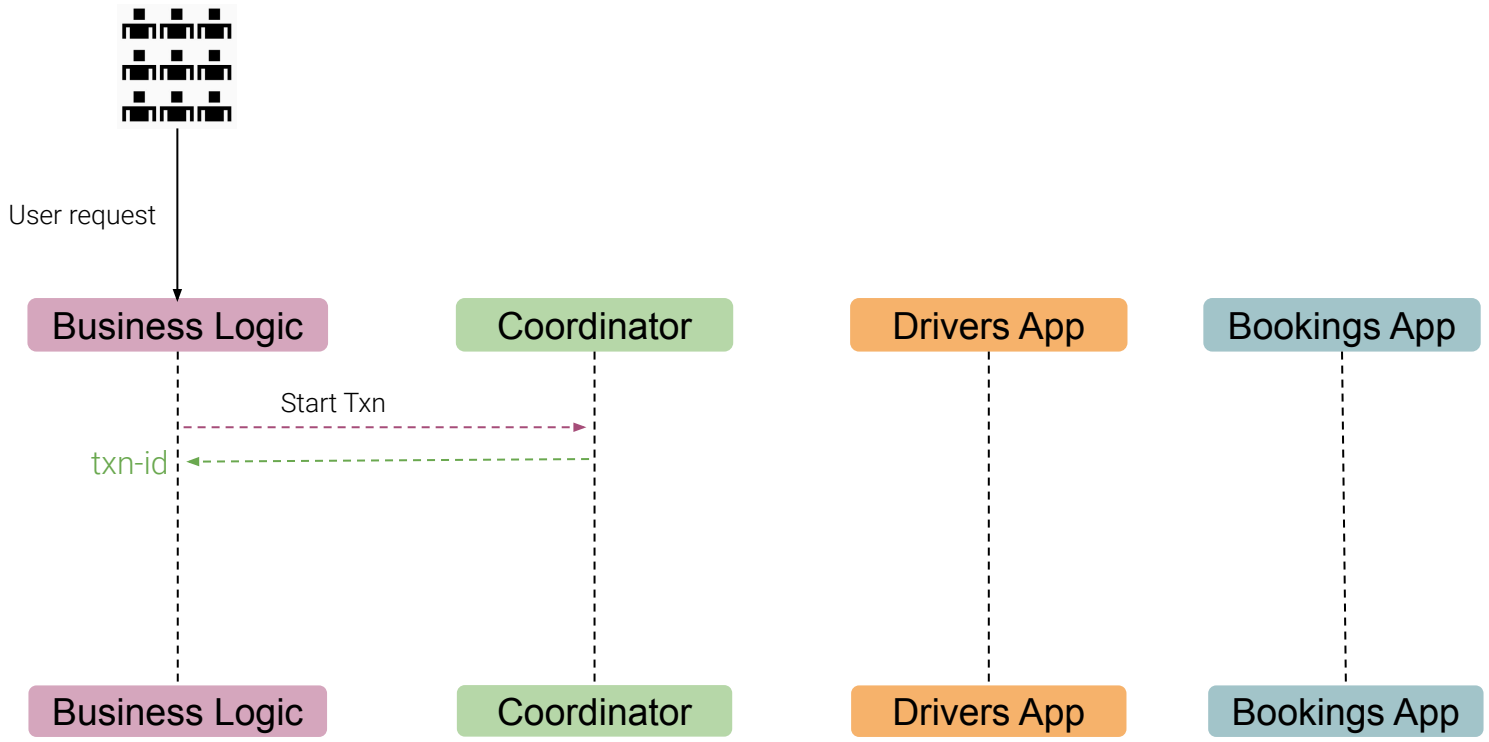
Shadows: Business Logic?



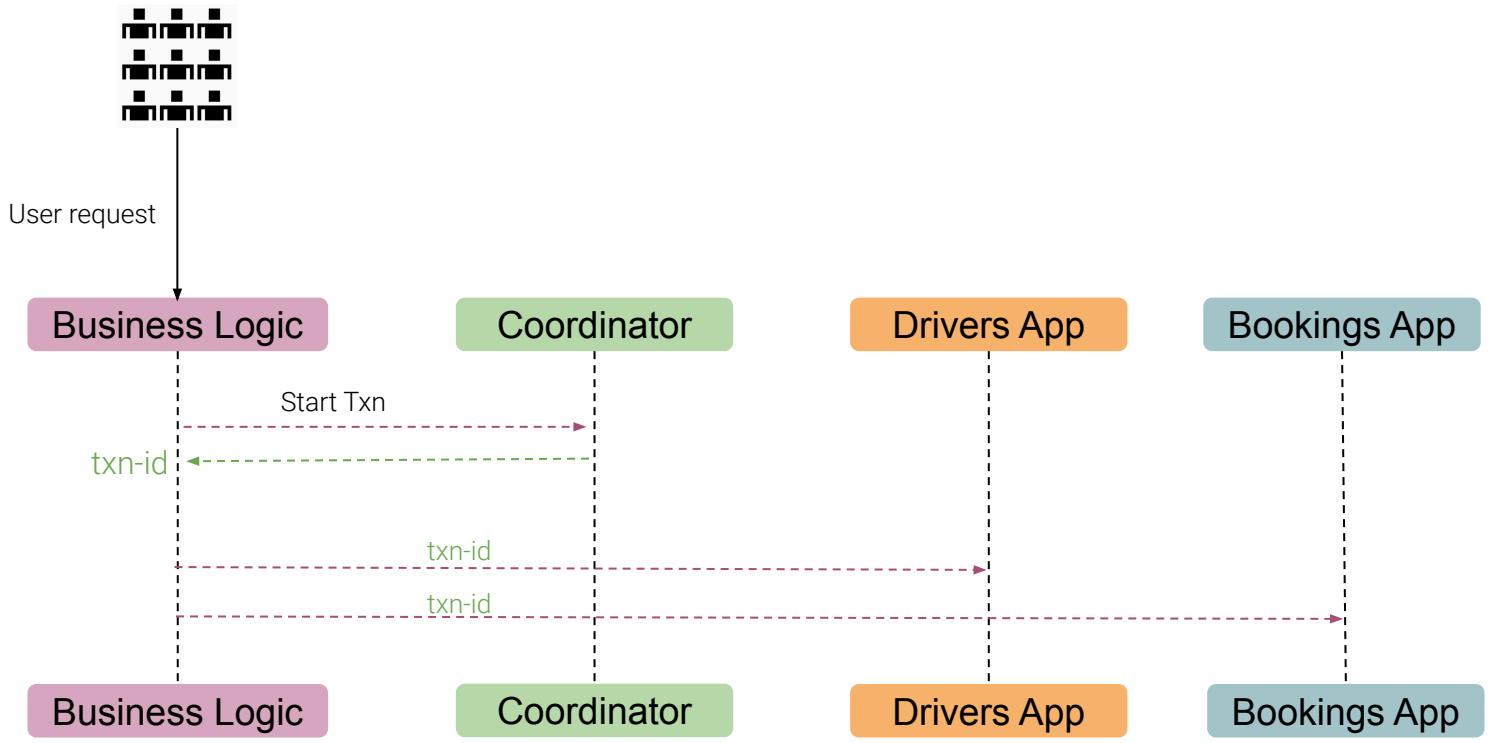
Shadows: A Distributed 2 Phase Commit



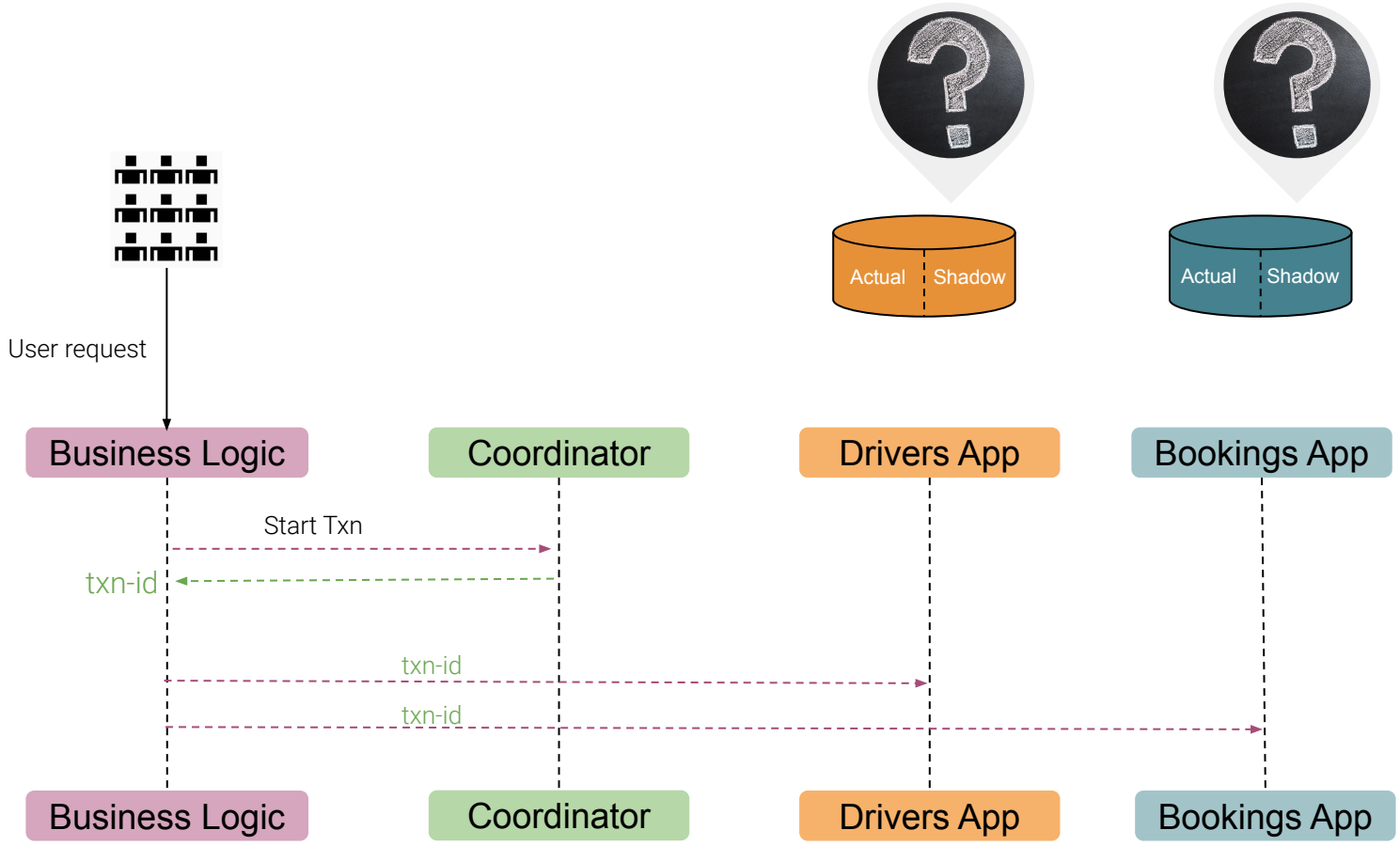
Shadows: A Distributed 2 Phase Commit



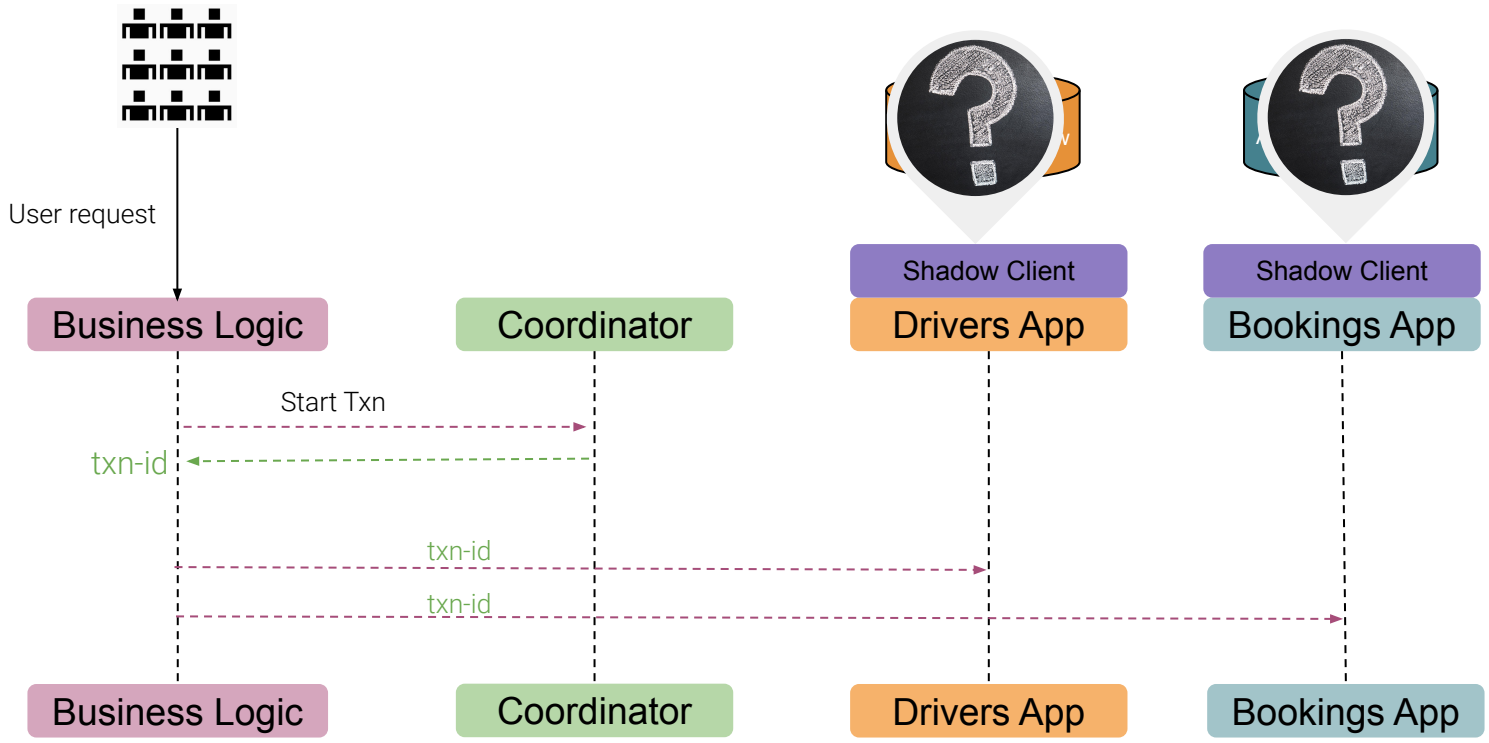
Shadows: A Distributed 2 Phase Commit



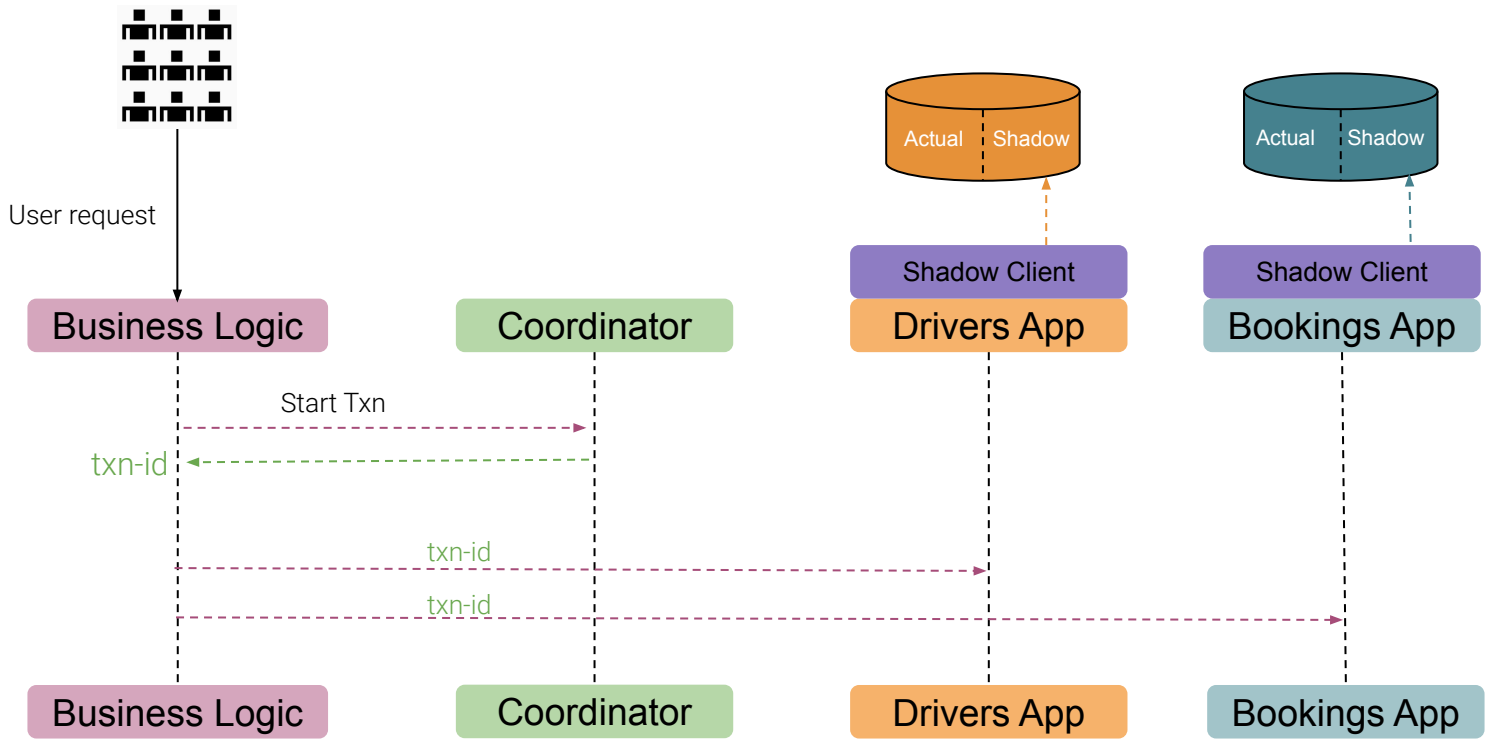
Shadows: Shadow tables?



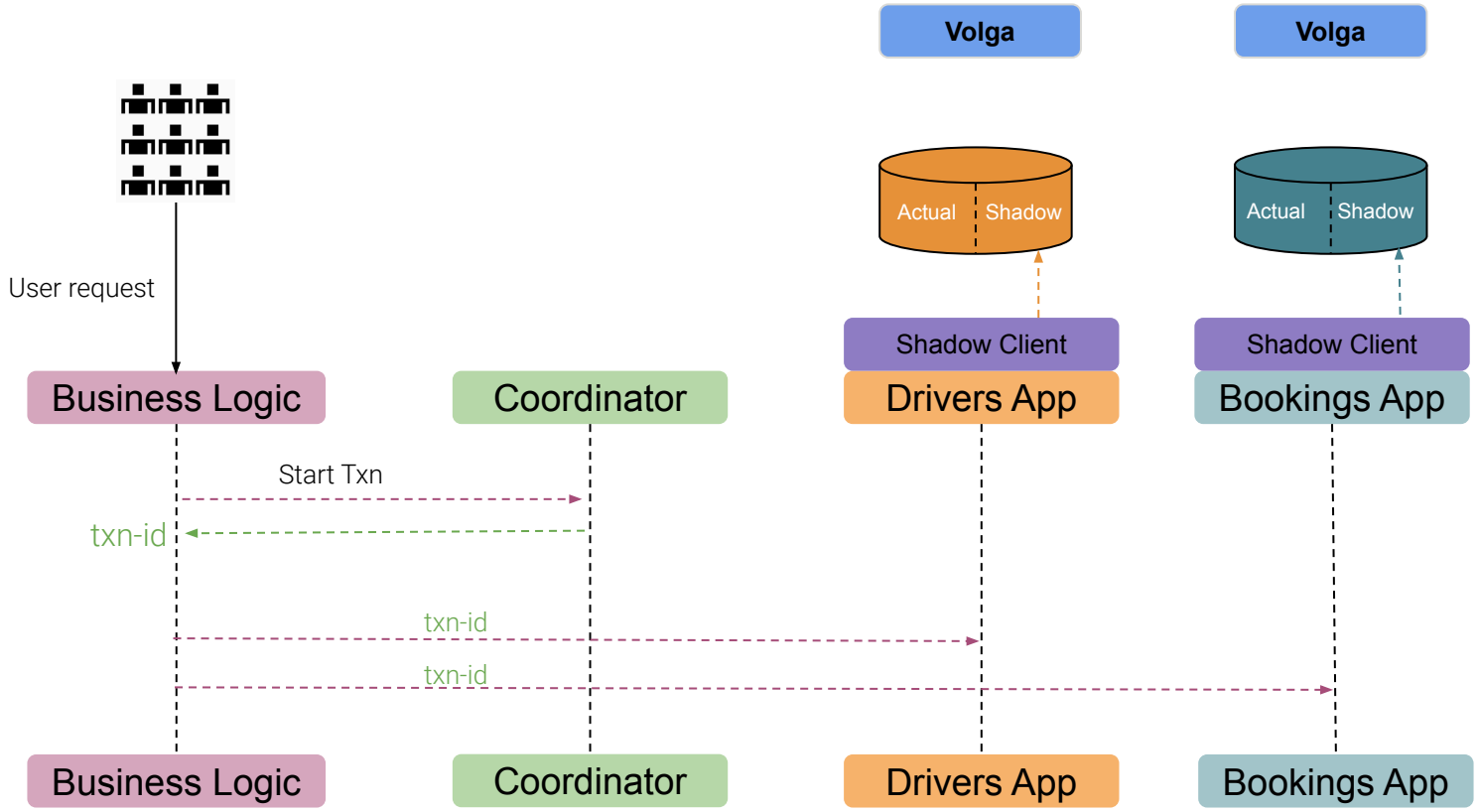
Shadows: Shadow Clients?



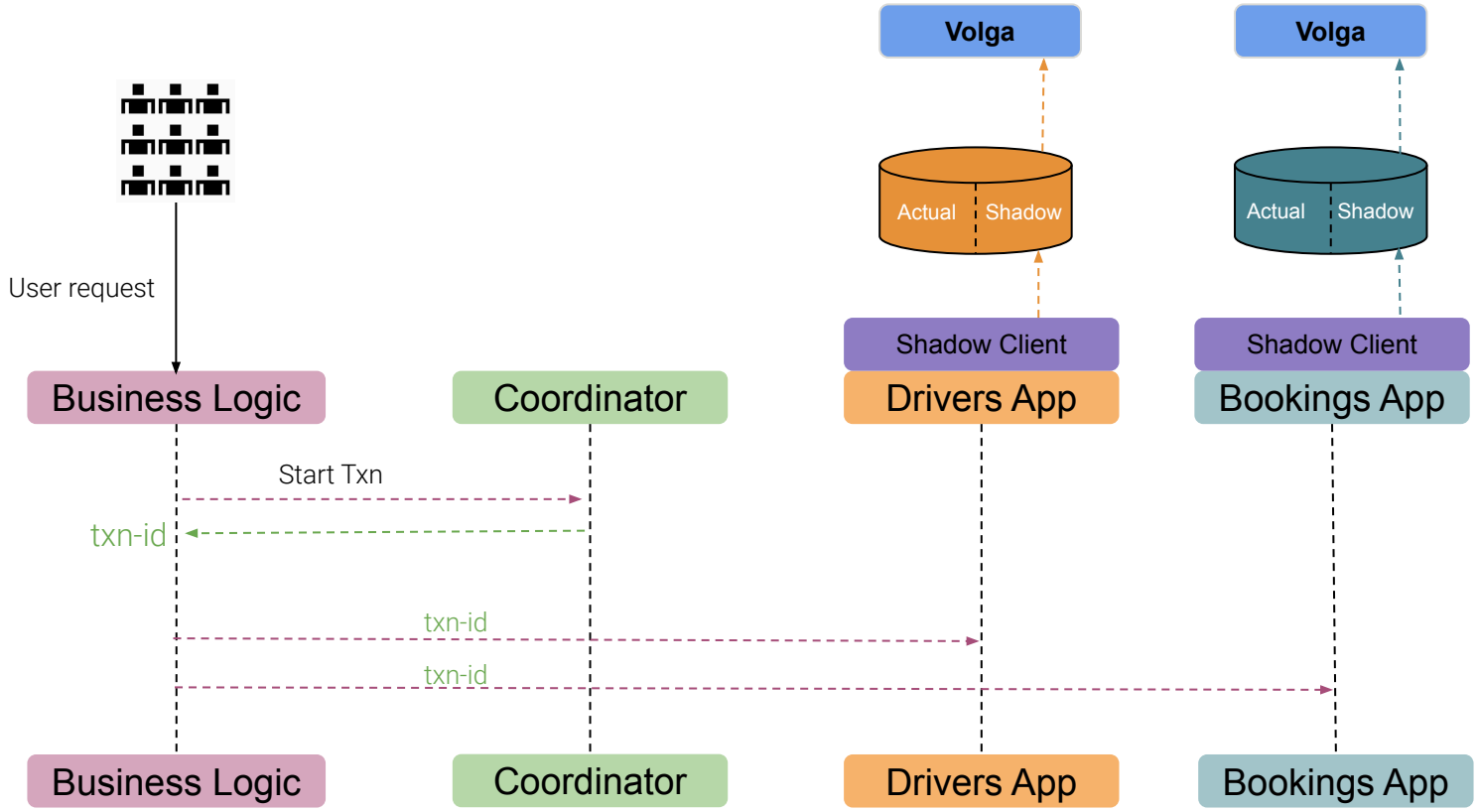
Shadows: A Distributed 2 Phase Commit



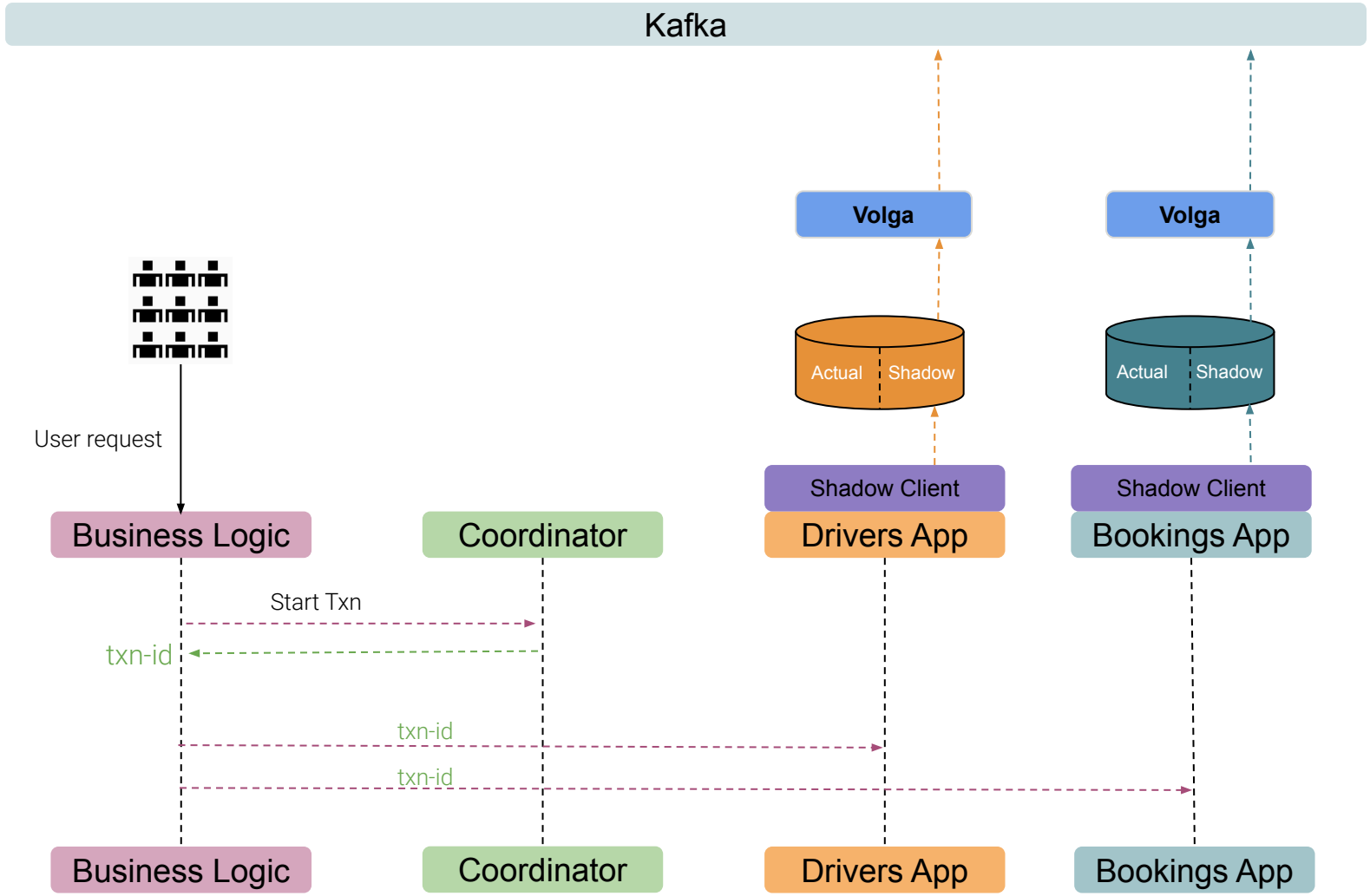
Shadows: A Distributed 2-Phase Commit



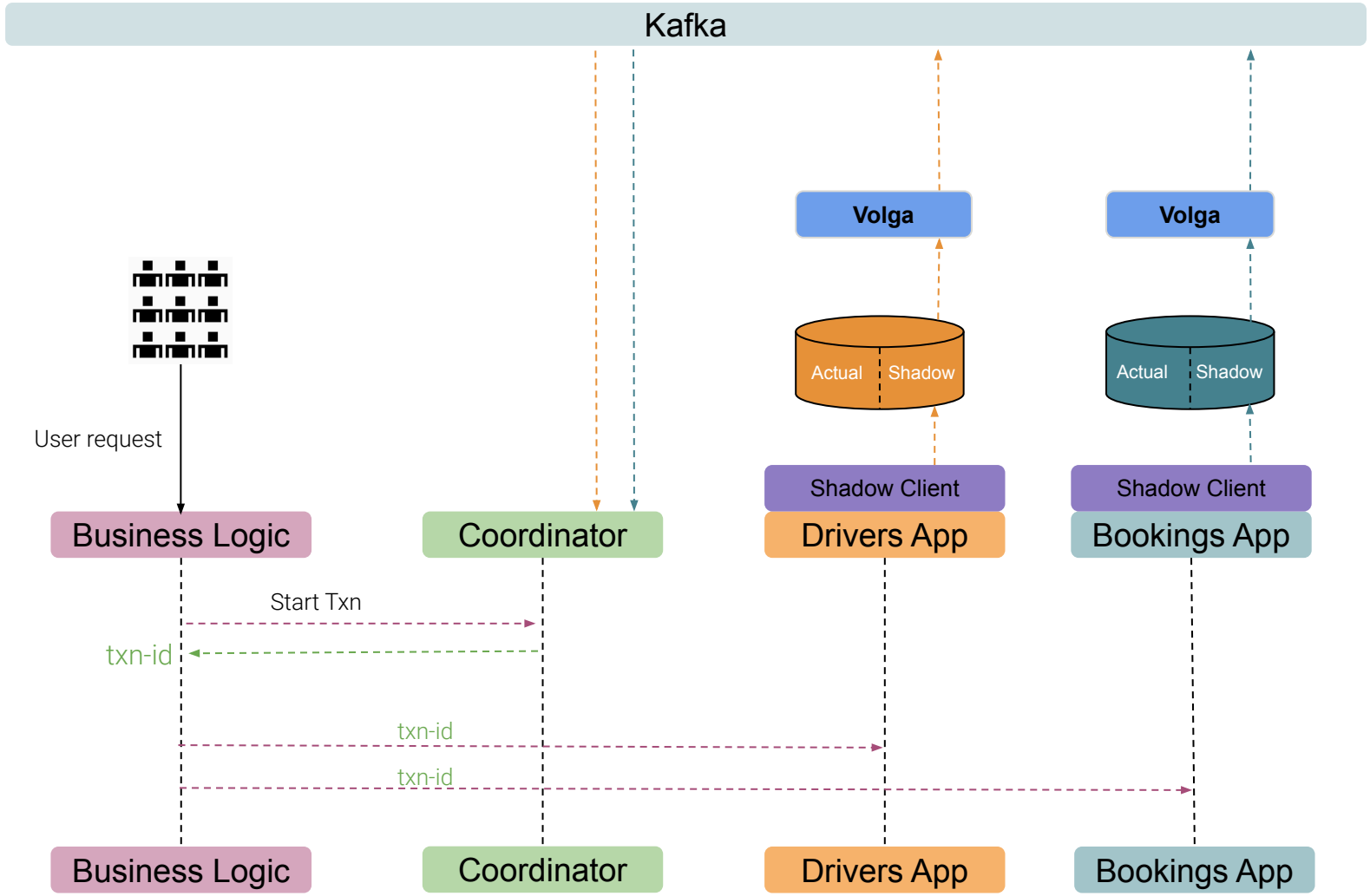
Shadows: A Distributed 2-Phase Commit



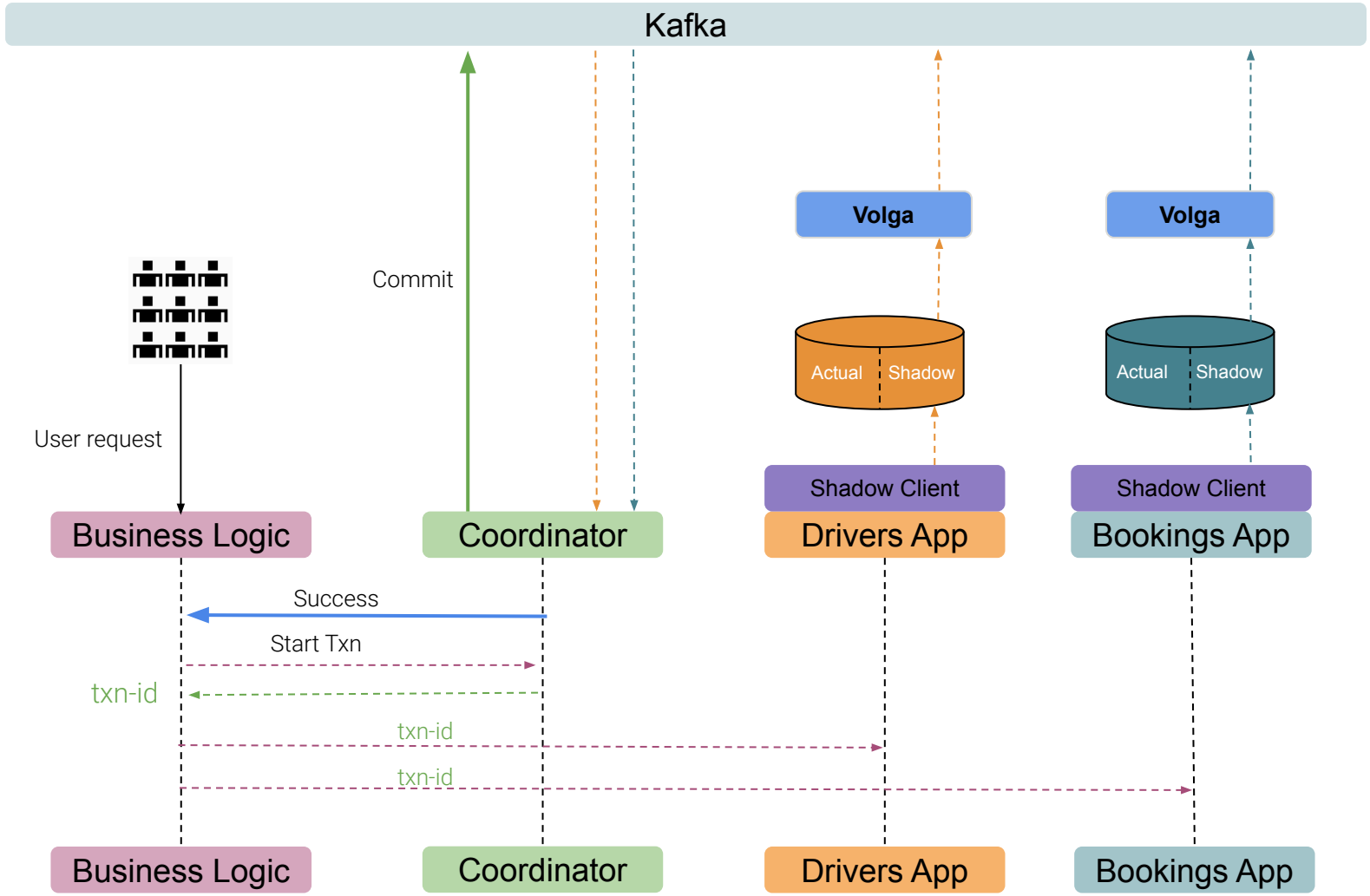
Shadows: A Distributed 2-Phase Commit



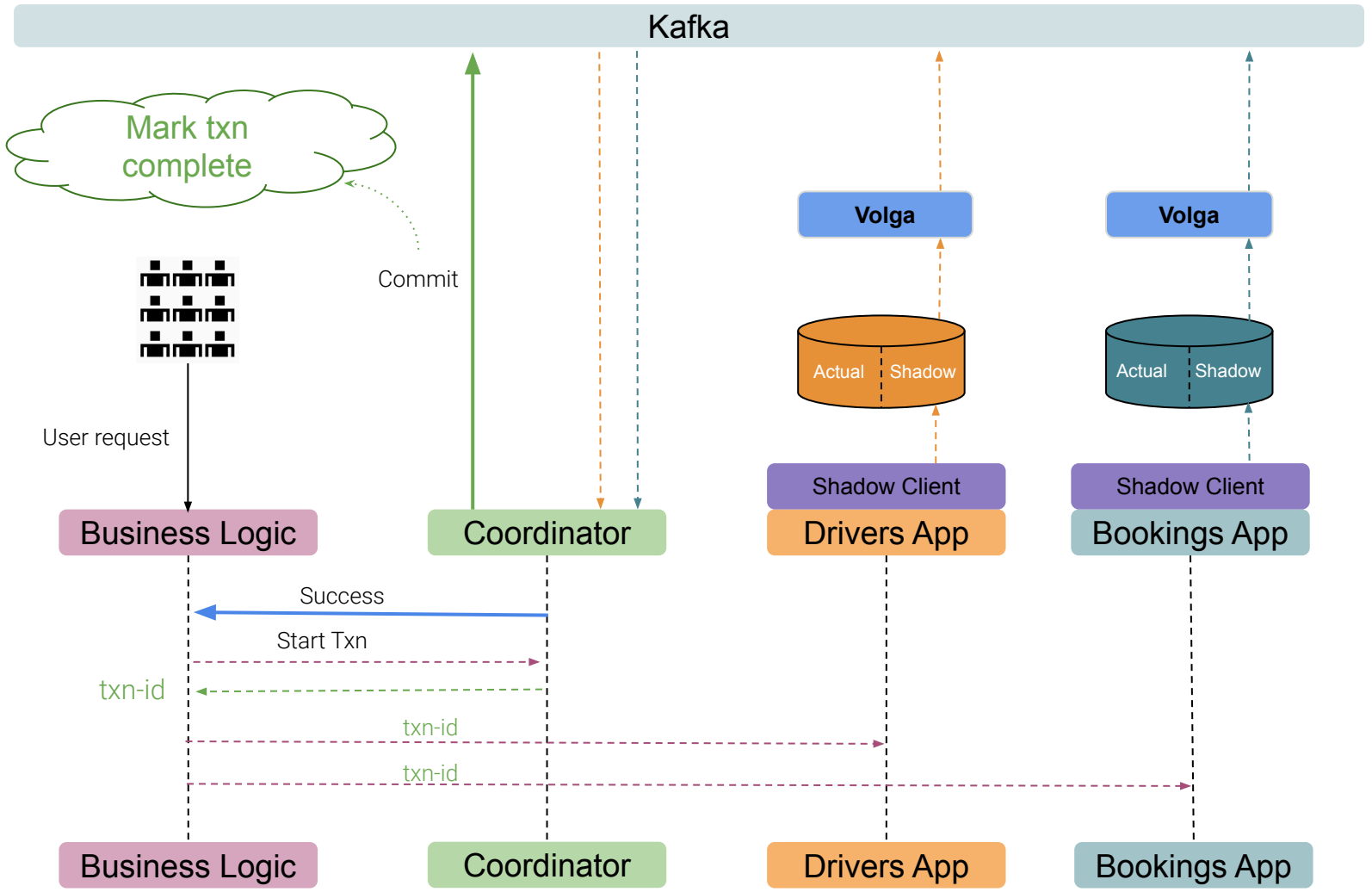
Shadows: A Distributed 2-Phase Commit



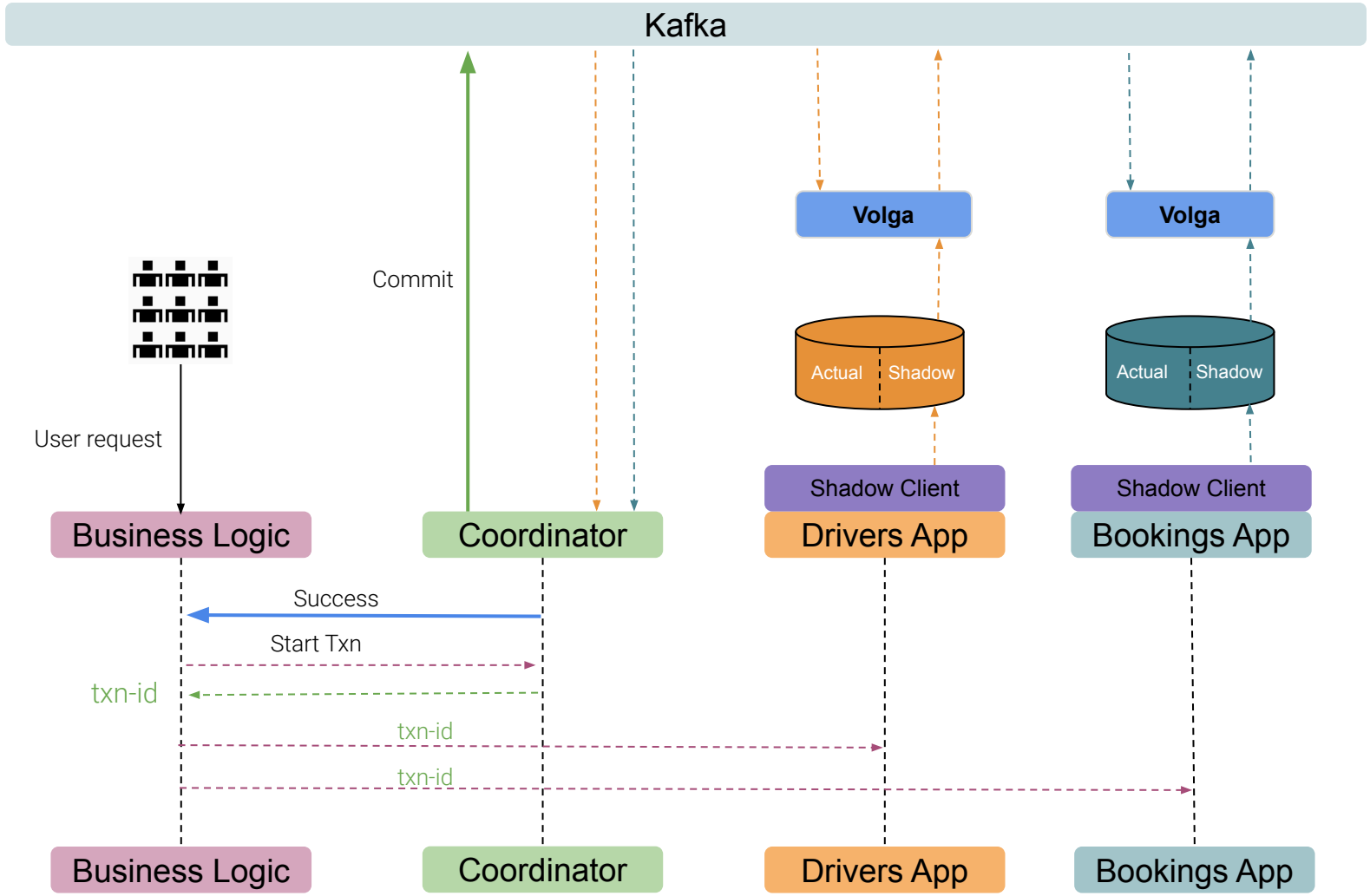
Shadows: A Distributed 2-Phase Commit



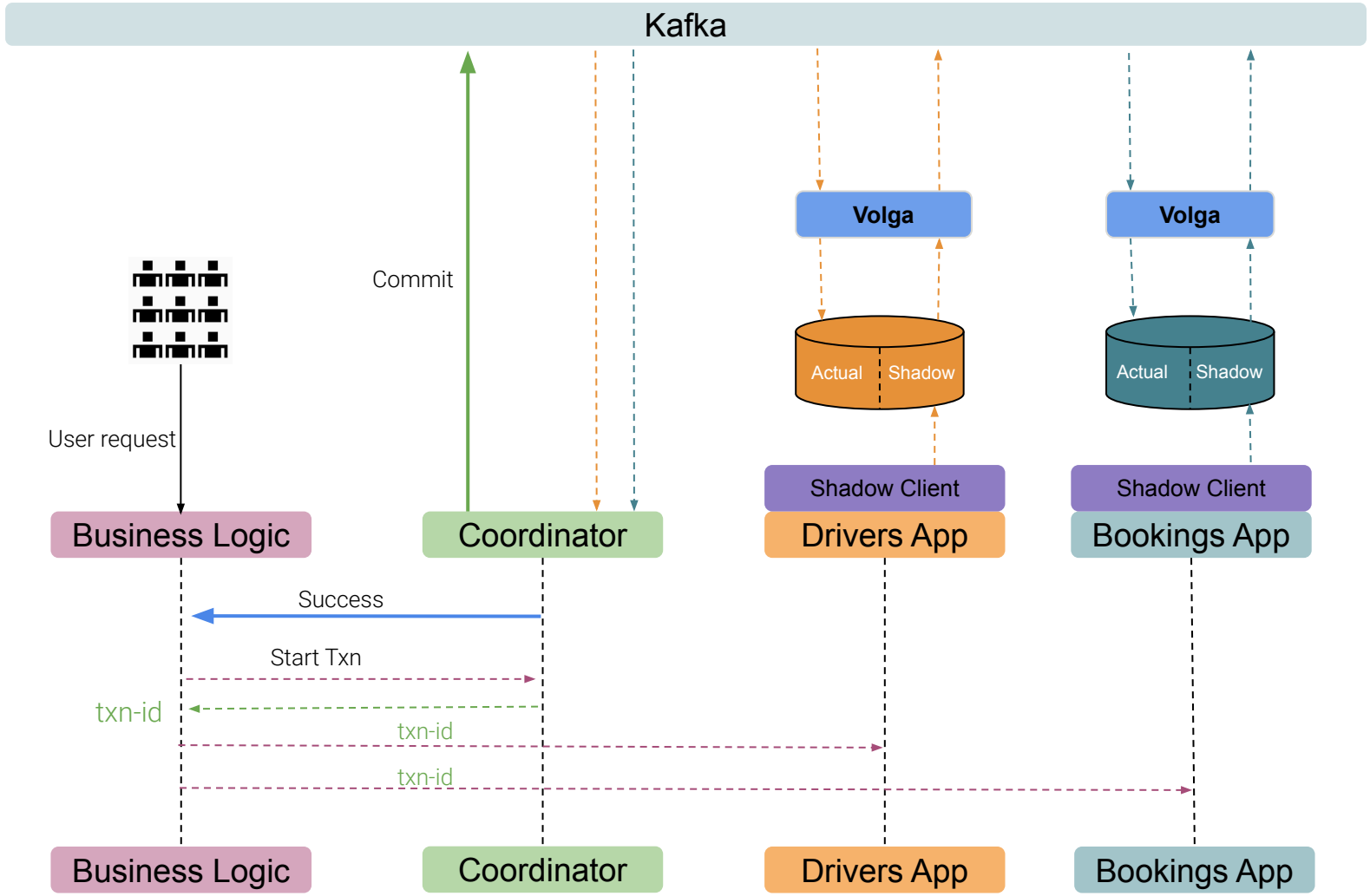
Shadows: A Distributed 2-Phase Commit



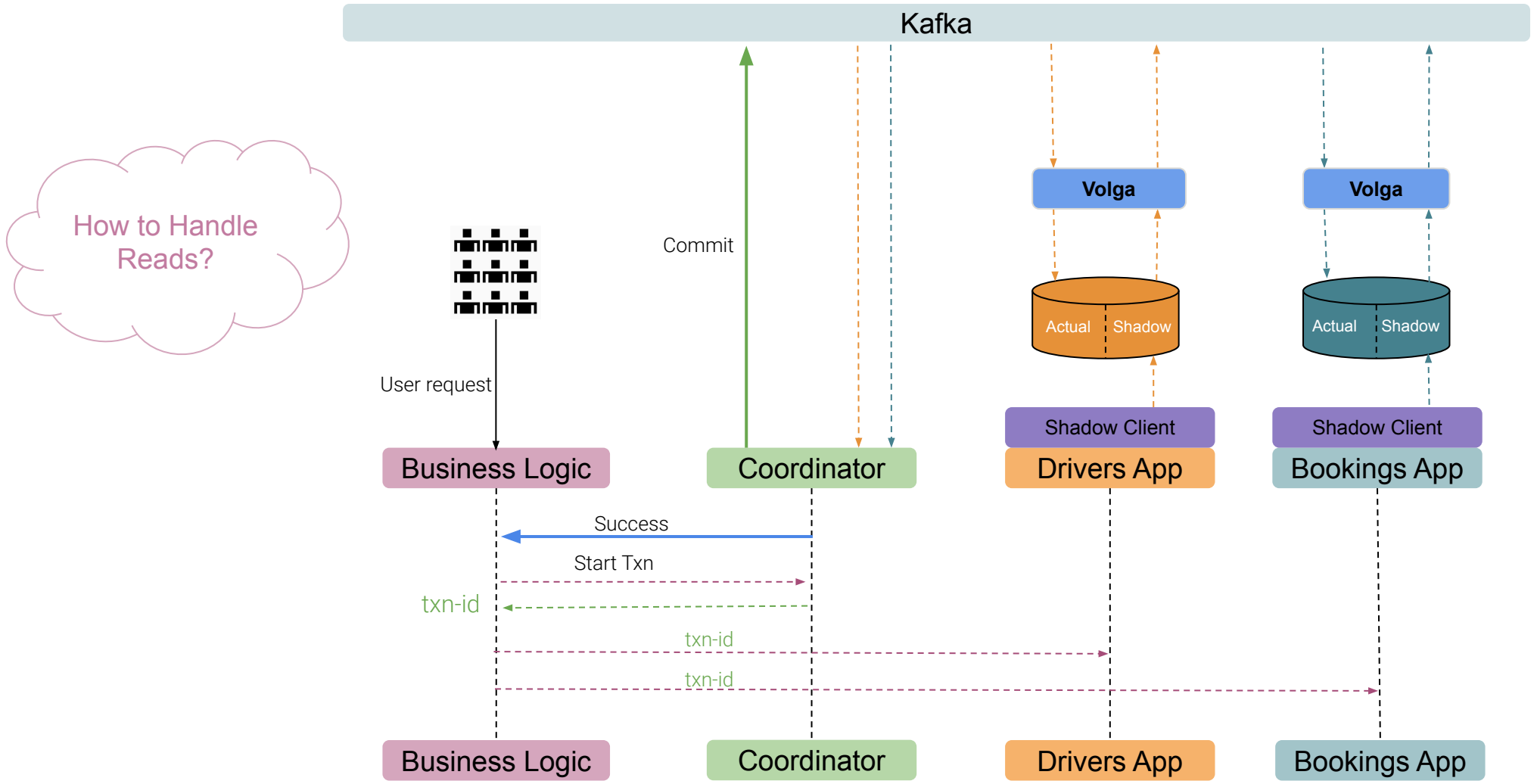
Shadows: A Distributed 2-Phase Commit



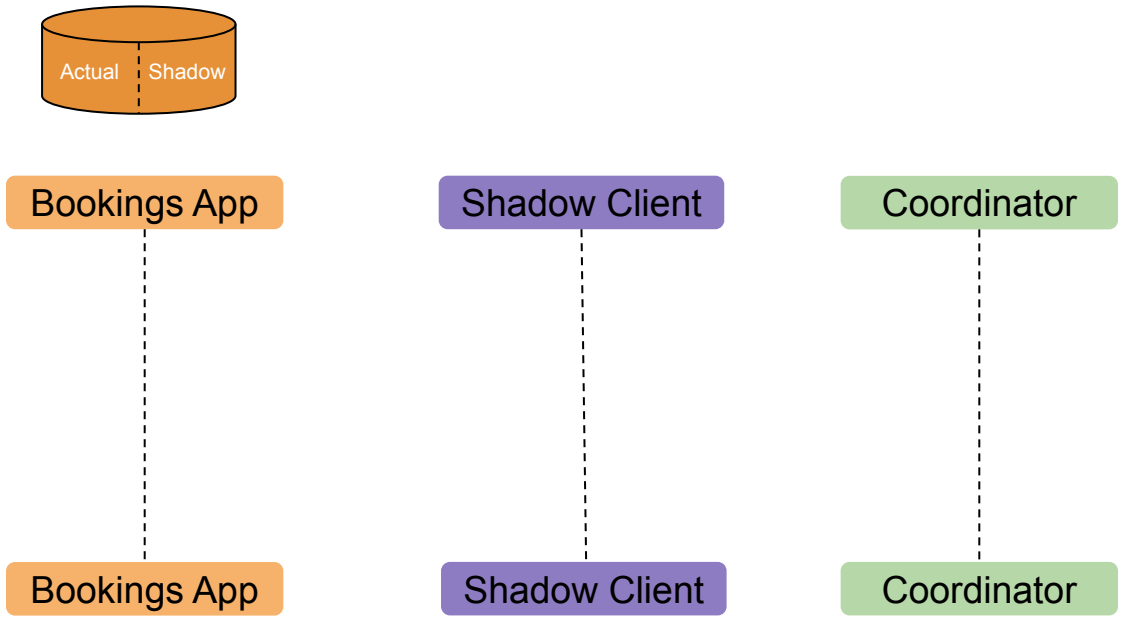
Shadows: A Distributed 2-Phase Commit



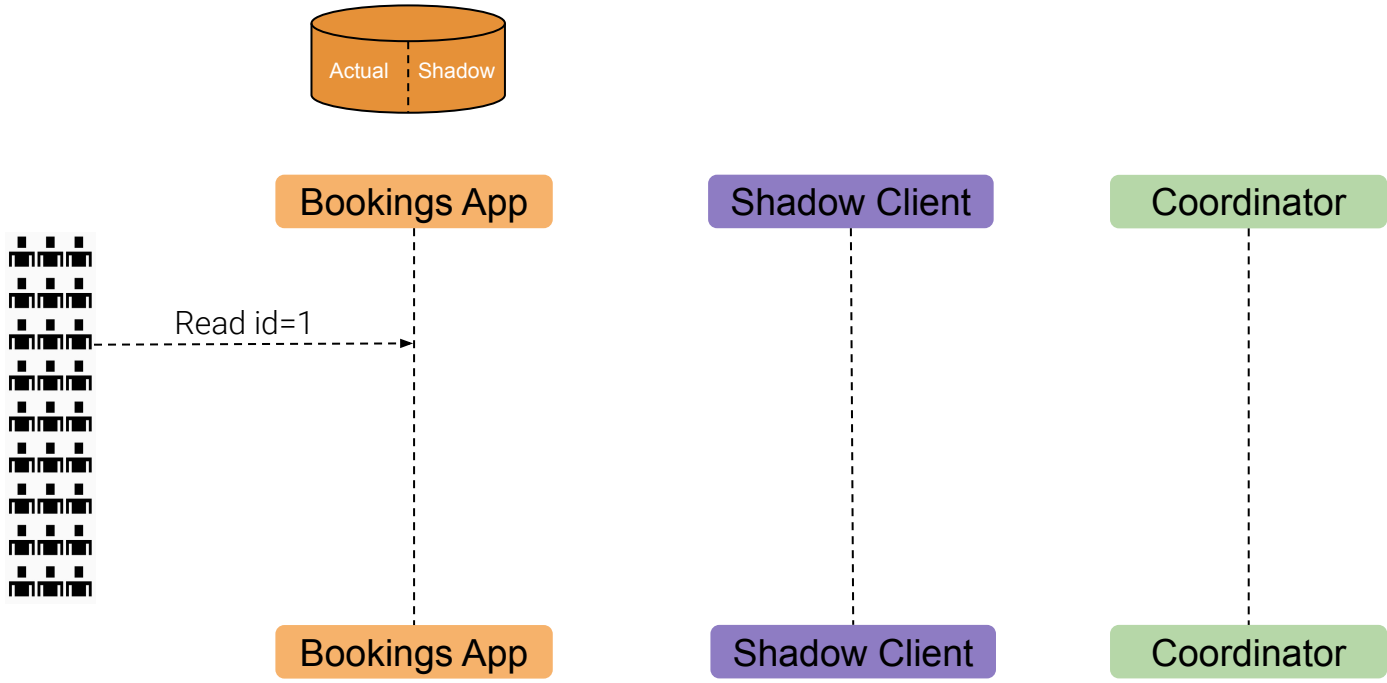
Shadows: A Distributed 2-Phase Commit



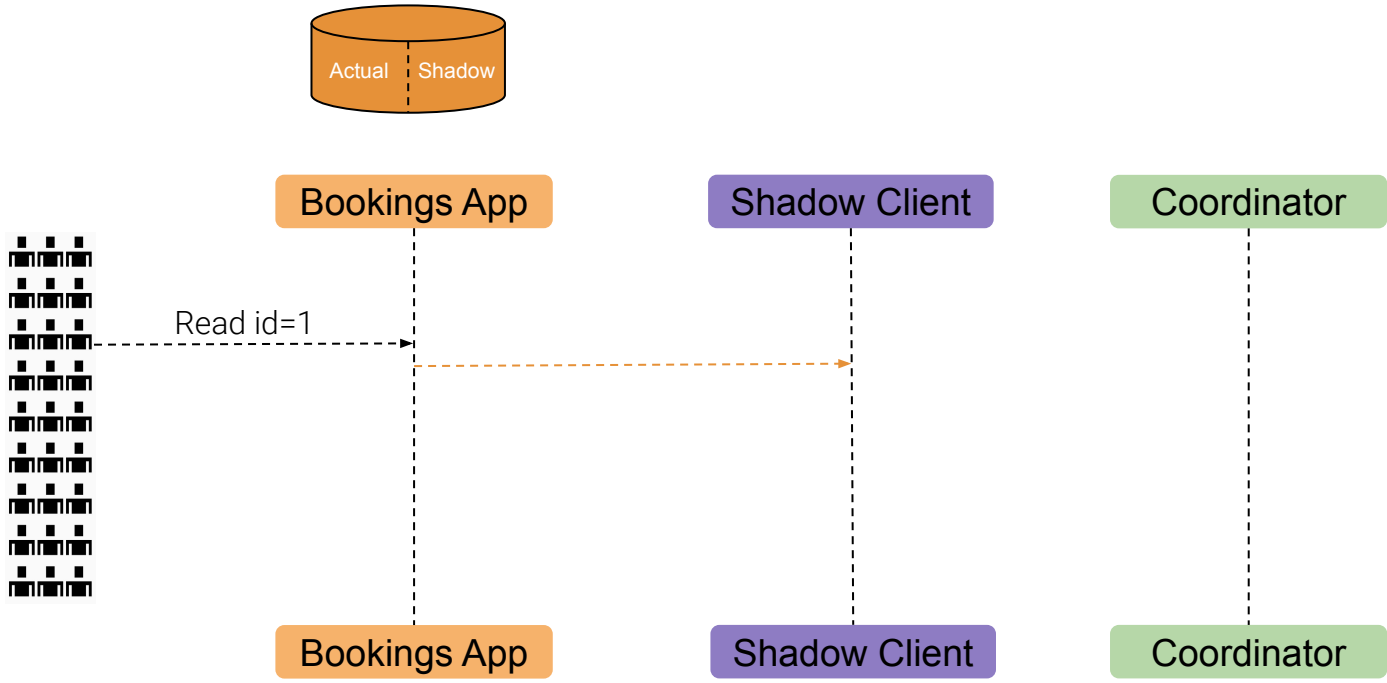
Consistent Reads



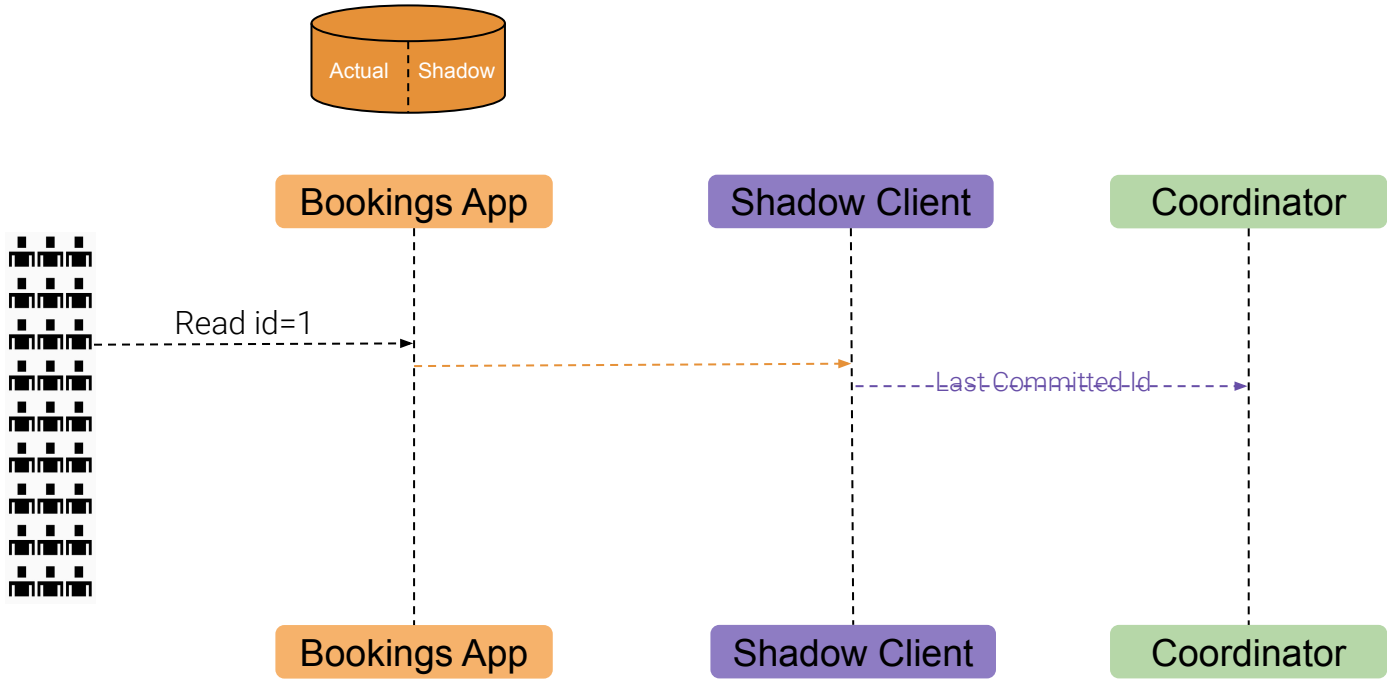
Consistent Reads



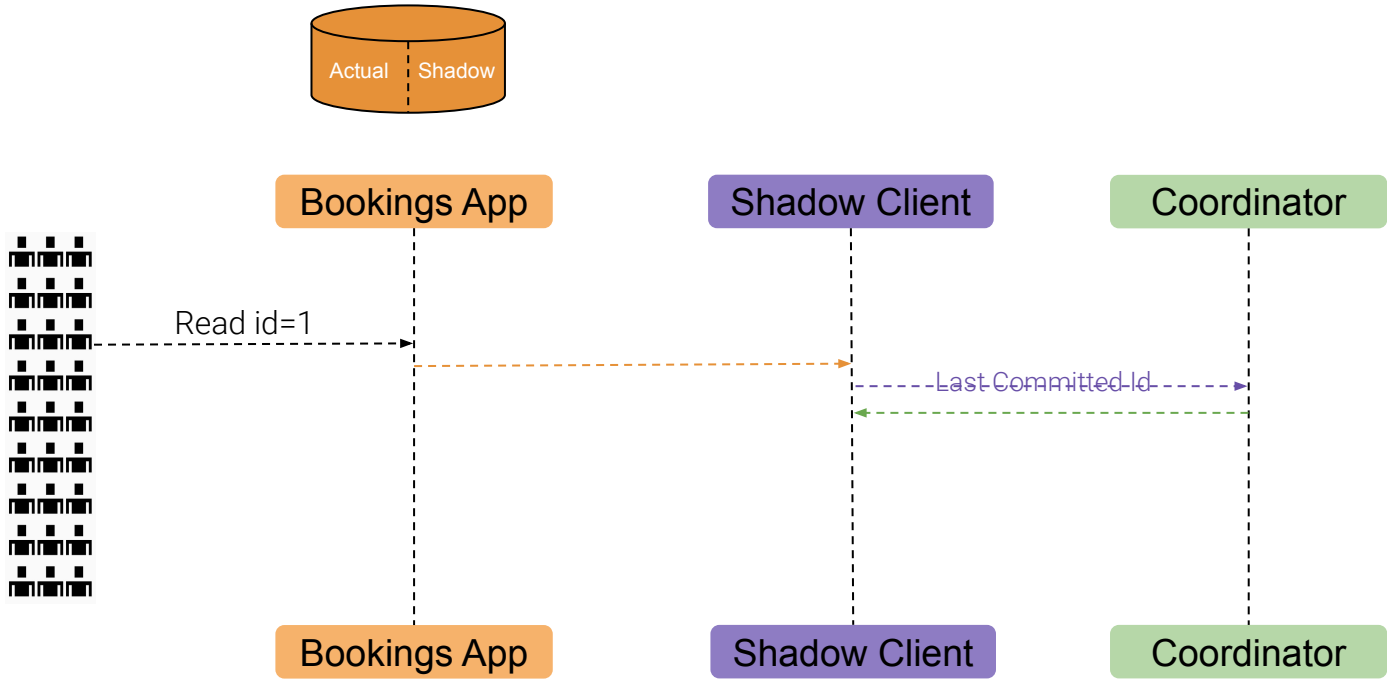
Consistent Reads



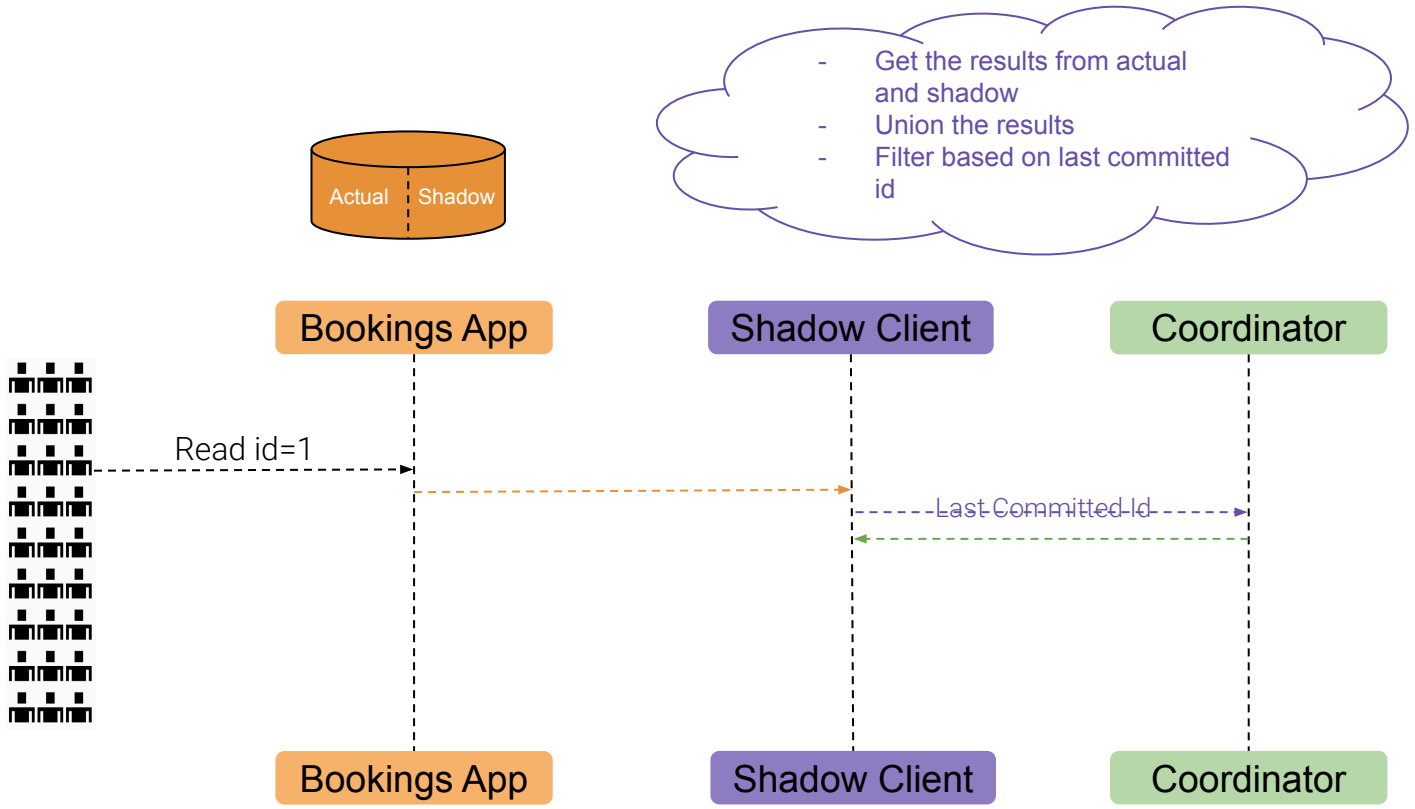
Consistent Reads



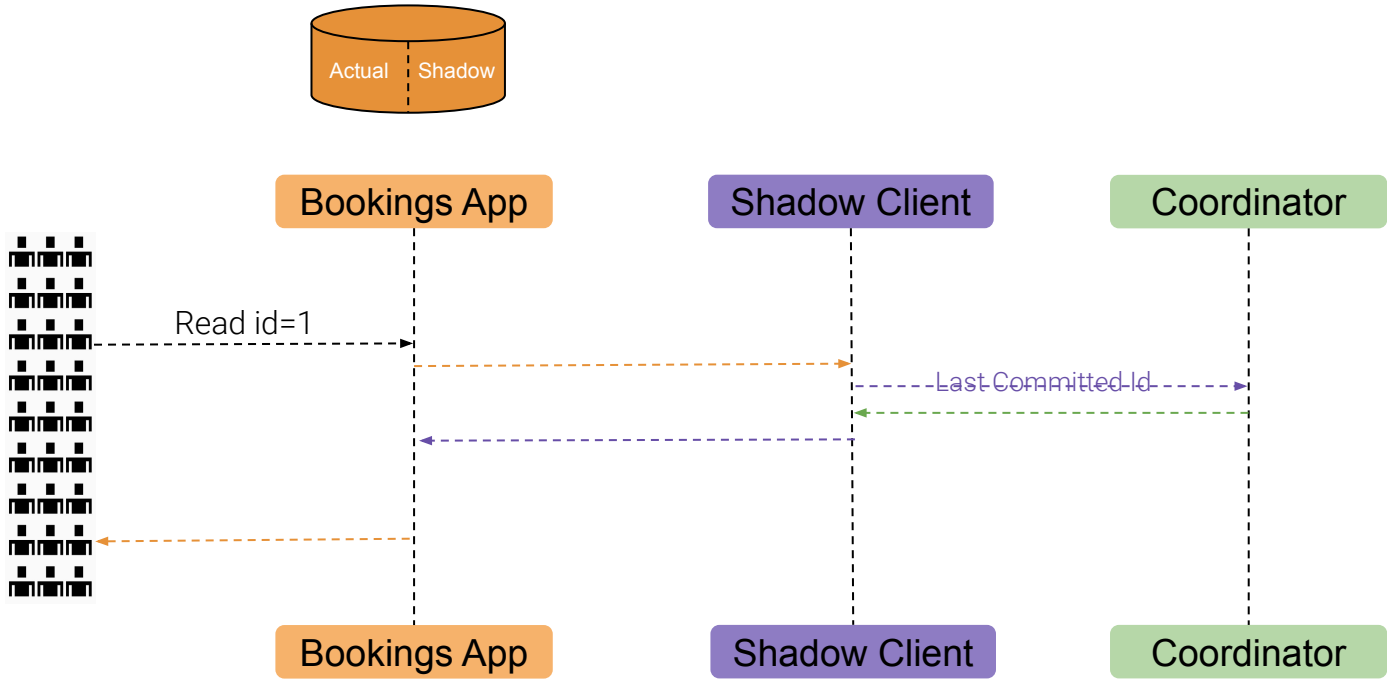
Consistent Reads



Consistent Reads



Consistent Reads



Shadows: Pros & Cons

- No locking
- Read committed isolation
- Read committed consistency

- Slower Reads
- Complex to implement

Comparison

Parameter	2 Phase Commit	Sagas	Shadows
Consistency	Linearizable	Eventual	Read Committed
Isolation	Serializable	Relaxed	Read Committed

So what to use?

So what to use?

Eventual

Shadows

High Availability

SAGAS

2PC

Isolation

Weak

Replication

Consistency

Partition Tolerance

Strong

XA Transaction

Read Committed

Relaxed

Latency

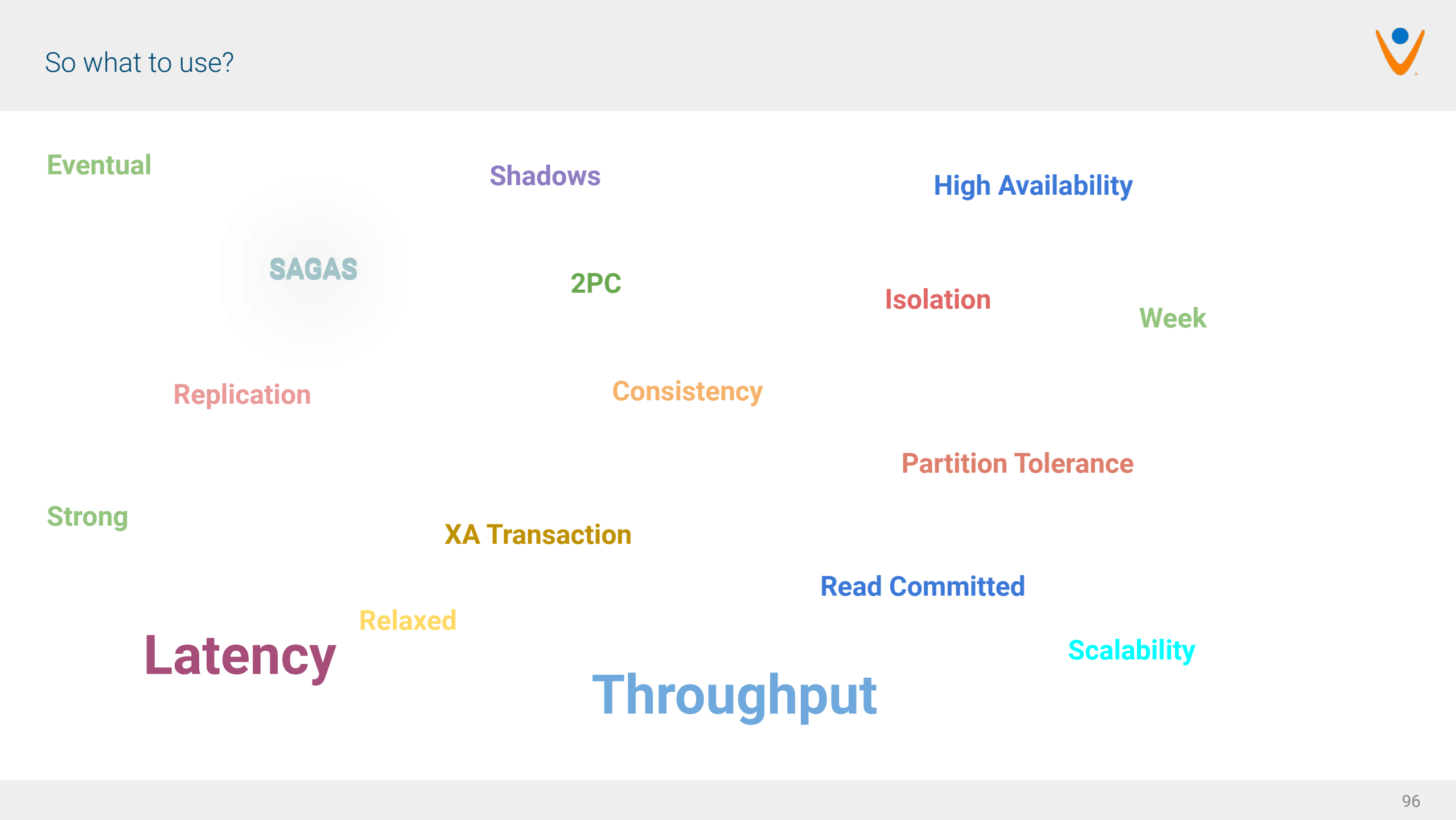
Throughput

Scalability

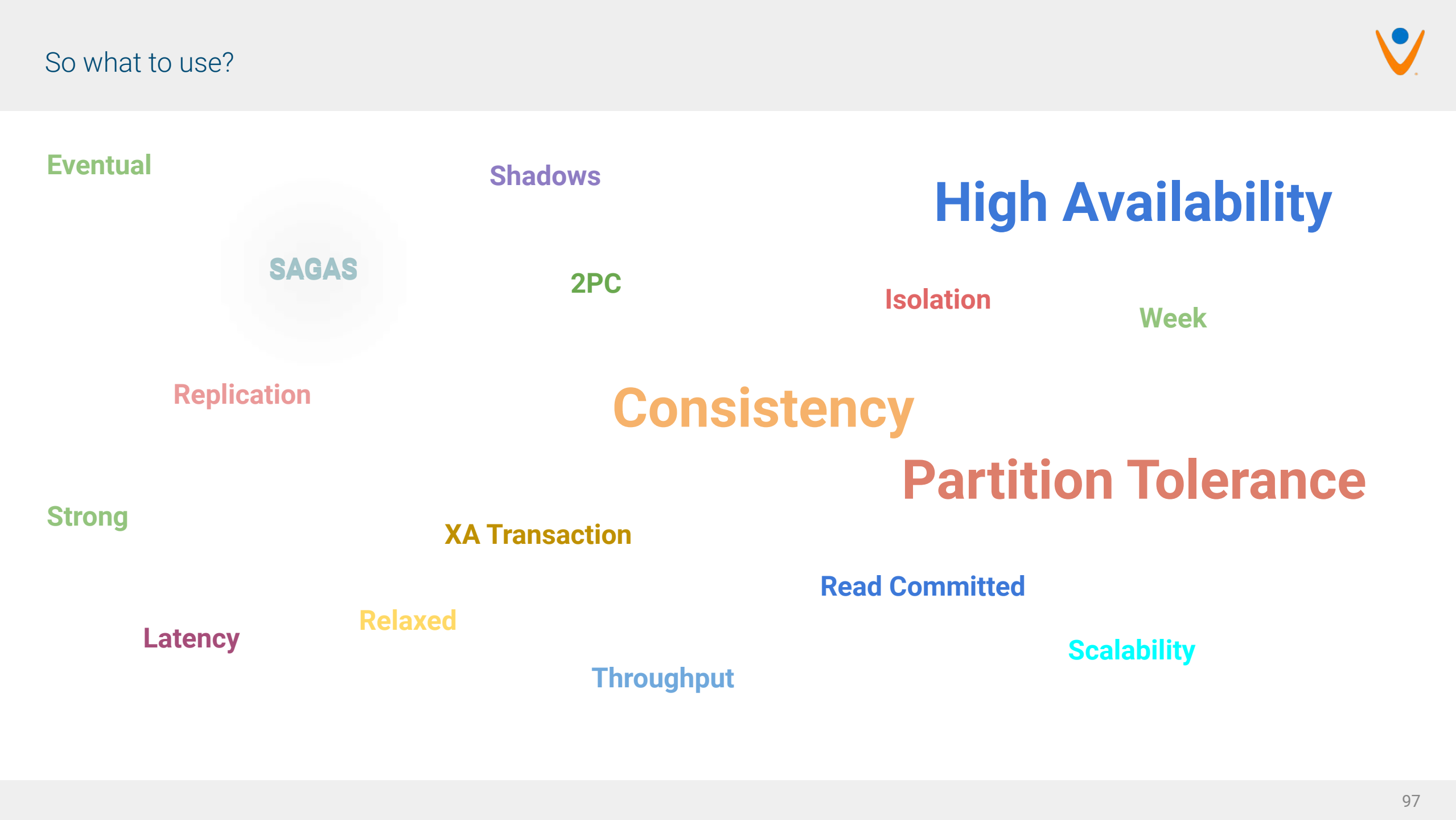
So what to use?

So what to use?

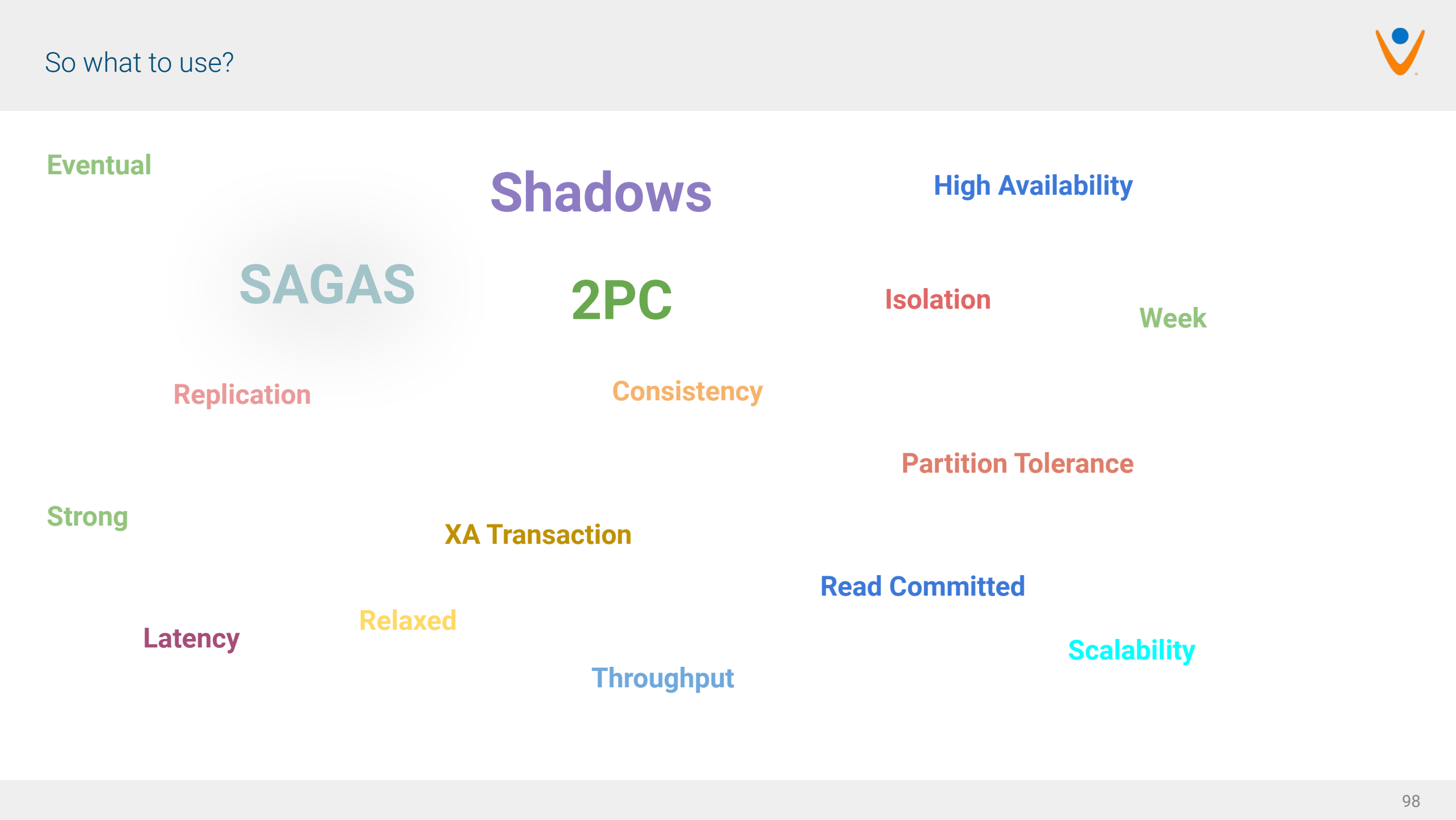
So what to use?



So what to use?



So what to use?



THANKYOU

QUESTIONS?

THANKYOU

Let’s understand the terminology

Database Consistency: if two clients can see different states at the same point in time, we say that their view of the database is inconsistent

Database Isolation: the ability of a database to allow a transaction to execute as if there are no other concurrently running transactions

* Daniel Abadi in DBMS Musings (http://dbmsmusings.blogspot.com)

How read committed

write -->1 ,, read -->2

<2

read -->1 write -->2

0-->

read -->1

resutsl --0

write-->2

read-->3 (write --2)

read->4 write -->5

read == write --2

write--6 ,, read --7

read ...write --2,write.5 (edited)

read--8 ... write2,write5,write6

- According to the coordinator timestamp the transaction which started first or which has the lower timestamp have to finish first.

This means that if a concurrent transaction happens and let's say first is read and second is write then it is okay as read will read all the data before this write but if write happens before read then the read will wait for the write to finish and then only return or wait until timeout.

if write write concurrent transaction happens then if first write is still going on then the second one will wait until timeout and if the first one finishes before the timeout then second will start or second will fail as soon as it times out.