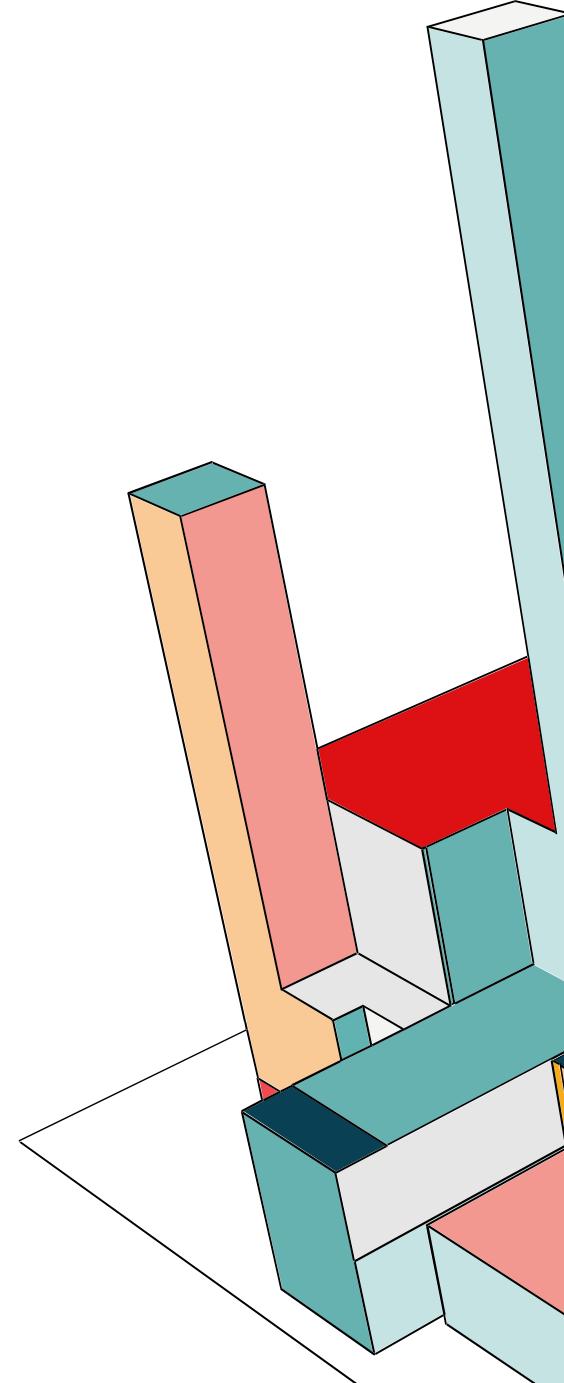
The background features a collection of 3D-style geometric shapes, primarily cubes and rectangular prisms, rendered in a variety of colors including red, orange, yellow, teal, light blue, and white. These shapes are arranged in a somewhat scattered, overlapping manner, creating a sense of depth and architectural complexity.

# КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

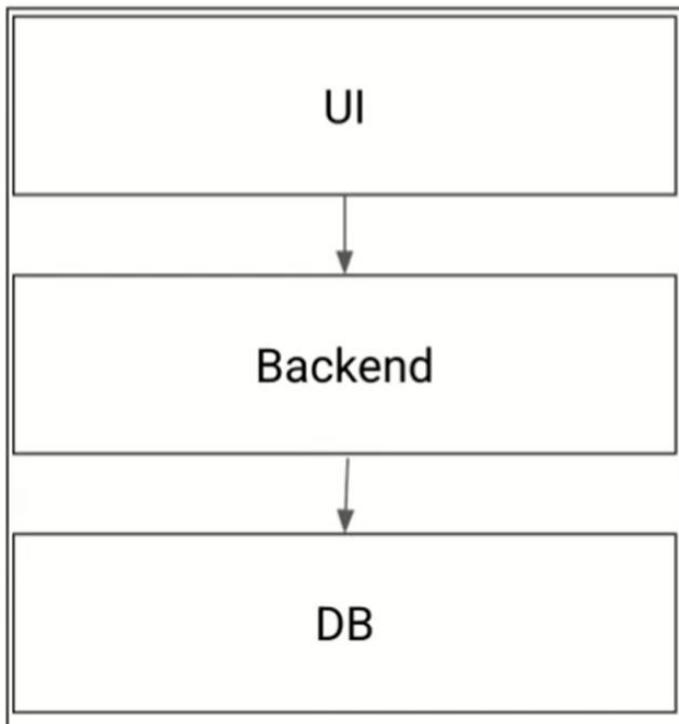
# ПЛАН ЛЕКЦИИ № 12

## Паттерны отказоустойчивости при межсервисном взаимодействии

1. О согласованности данных
2. Transaction Outbox
3. Распределенные транзакции
  1. 2-phase-commit
  2. SAGA
    1. Оркестрация
    2. Хореография



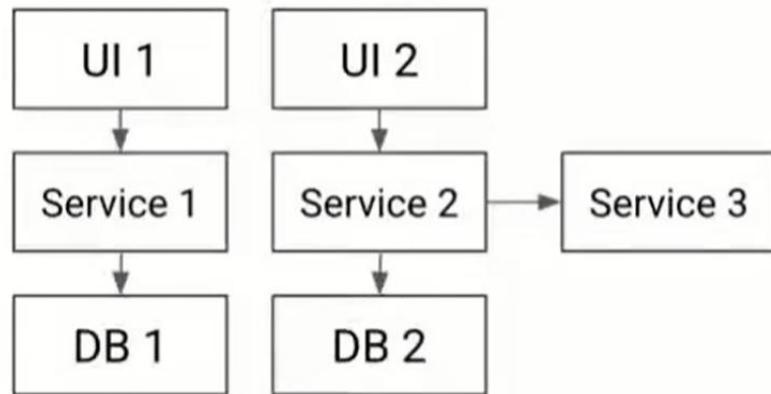
# МОНОЛИТНАЯ АРХИТЕКТУРА



- Вся логика системы в одном приложении (единый деплой, единая база)
- Проста на старте, сложна при росте
- Хорошо: стартапы, MVP, prototyping
- Плохо: масштабируемость, изоляция ошибок



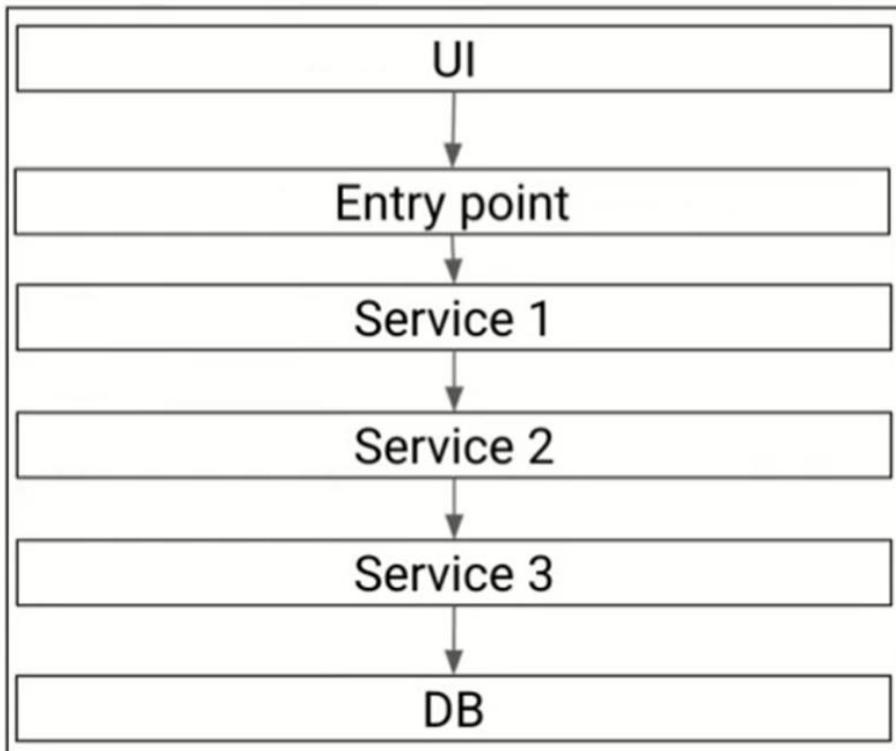
# МИКРОСЕРВИСНАЯ АРХИТЕКТУРА



- Каждый сервис – изолированное приложение с собственными зависимостями
- Коммуникация через API/сообщения
- Хорошо: масштабирование, независимые команды
- Плохо: сложность, DevOps, сетевые сбои
- Часто дополняется: API Gateway, Service Mesh



# МОДУЛЬНЫЙ МОНОЛИТ



## Что это:

Один сервис, один деплой, но **строгое разделение логики по модулям**.

## Плюсы:

- Простота разработки и отладки как у монолита
- Легкий переход к микросервисам: каждый модуль может стать сервисом
- Лучше управляемость зависимостей и границ

## Когда использовать:

- Когда проект растёт, но микросервисы пока избыточны



## НО И ЭТО ЕЩЕ НЕ ВСЁ...

- Сервисные модули (Modules as Services)
- Self-contained Systems (SCS)
- Micro Frontends
- Бэкенд как платформа (Backend-for-Frontend / BFF)
- Lambda-архитектура / Serverless



# СОГЛАСОВАННОСТЬ В РАЗНЫХ КОНТЕКСТАХ

- ACID — набор требований к транзакционной системе, обеспечивающий наиболее надёжную и предсказуемую её работу — атомарность, согласованность, изоляцию, устойчивость; сформулированы в конце 1970-х годов Джимом Греем
- Теорема CAP — эвристическое утверждение о том, что в любой реализации распределённых вычислений возможно обеспечить не более двух из трёх :
  - согласованность данных;
  - доступность;
  - устойчивость к фрагментации.



# ВИДЫ СОГЛАСОВАННОСТИ

Strong consistency	После записи все узлы сразу видят одно и то же (как в БД)
Eventual consistency	Узлы со временем придут к одному состоянию (например, в Cassandra, DynamoDB)
Causal consistency	События, связанные причинно-следственно, читаются в правильном порядке
Read your writes	Клиент, записавший данные, гарантированно видит их при чтении
Monotonic reads	Клиент не видит более старые значения, чем раньше
Session consistency	Гарантии согласованности сохраняются в рамках сессии



## ПРИМЕР

- У тебя два узла: А и В
- Ты обновил запись на А → хочешь, чтобы на В она тоже появилась

Разные модели:

- **Strong**: В сразу видит обновление
- **Eventual**: В увидит через секунду-две
- **Causal**: если В увидит следующее действие, зависящее от этого, то и само действие должно быть видно
- **Read-your-writes**: только ты как автор точно видишь своё изменение
- **Monotonic reads**: ты не увидишь "более старую" версию, чем видел до этого



# **В КАКОЙ АРХИТЕКТУРЕ КАКАЯ СОГЛАСОВАННОСТЬ?**

**Монолит** – чаще всего strong consistency через ACID

**Микросервисы** – чаще всего eventual или causal consistency

**Event-driven архитектура** – почти всегда eventual



# РИСК РАЗЛИЧНЫХ АРХИТЕКТУР

Монолит	✗ Нет (легко поддерживать)
Модульный монолит	⚠ Иногда, если модули → отдельные хранилища
Микросервисы	🔥 Да, всегда
Self-contained Systems	🔥 Да
Serverless	⚠ Да, особенно при комплексной логике
Layered / Clean / Hexagonal	✗ Нет, если в пределах одного сервиса



# ПОЧЕМУ СОГЛАСОВАННОСТЬ ЭТО СЛОЖНО

- Сетевые проблемы – норма
- Невозможно синхронизировать часы
- Отказ одного узла – это нормально
- Нет глобальных транзакций
- Повторяемость и идемпотентность – отдельная боль
- Выбор: **доступность или согласованность** (CAP-теорема)



# СЦЕНАРИЙ РАБОТЫ ИНТЕРНЕТ МАГАЗИНА

Ты оформляешь заказ в интернет-магазине.

Сервисы:

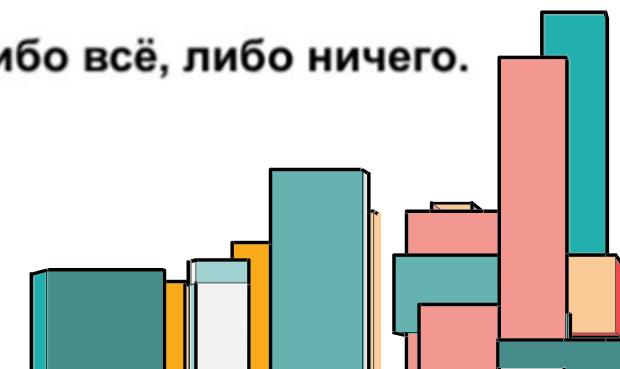
1. OrderService — создает заказ
2. PaymentService — списывает деньги

**В монолите проблем нет**

BEGIN TRANSACTION  
→ Создать заказ  
→ Списать деньги  
COMMIT

Если что-то упало → ROLLBACK

Гарантия: либо всё, либо ничего.



# ВАРИАНТЫ СЦЕНАРИЕВ В МИКРОСЕРВИСНОЙ АРХИТЕКТУРЕ

## Вариант 1

OrderService:

- заказ создан ✓
- отправка события оплаты ✗ (сеть, ошибка, брокер упал)

PaymentService:

- ничего не знает, деньги не списаны

# ВАРИАНТЫ СЦЕНАРИЕВ В МИКРОСЕРВИСНОЙ АРХИТЕКТУРЕ

## Вариант 2

OrderService:

→ заказ создаётся, но сервис падает до коммита

PaymentService:

← получил событие об оплате, списал деньги 



# ВАРИАНТЫ СЦЕНАРИЕВ В МИКРОСЕРВИСНОЙ АРХИТЕКТУРЕ

## Вариант 3

PaymentService:

- ← получил событие дважды (повторная отправка, сбой ACK)
- списал деньги 2 раза 😱



# ГАРАНТИРОВАННАЯ ДОСТАВКА СООБЩЕНИЙ

1

At-most-once. Используется в большинстве случаев. Отсутствие каких-либо гарантий, что сообщение дойдет потребителю.

2

At-least-once. 100% гарантия отправки сообщения. Но в случае сбоя сообщение будет отправлено более одного раза.

3

Exactly-once. Реализация дополнительных механизмов, чтобы избежать дублирования сообщений.



# ОДНОГО ПАТТЕРНА НЕ ДОСТАТОЧНО

## Transaction Outbox (в OrderService)

- Сначала заказ + событие оплаты пишутся в одной транзакции в БД
- Отдельный процесс/демон читает outbox и отправляет сообщение в Kafka/RabbitMQ

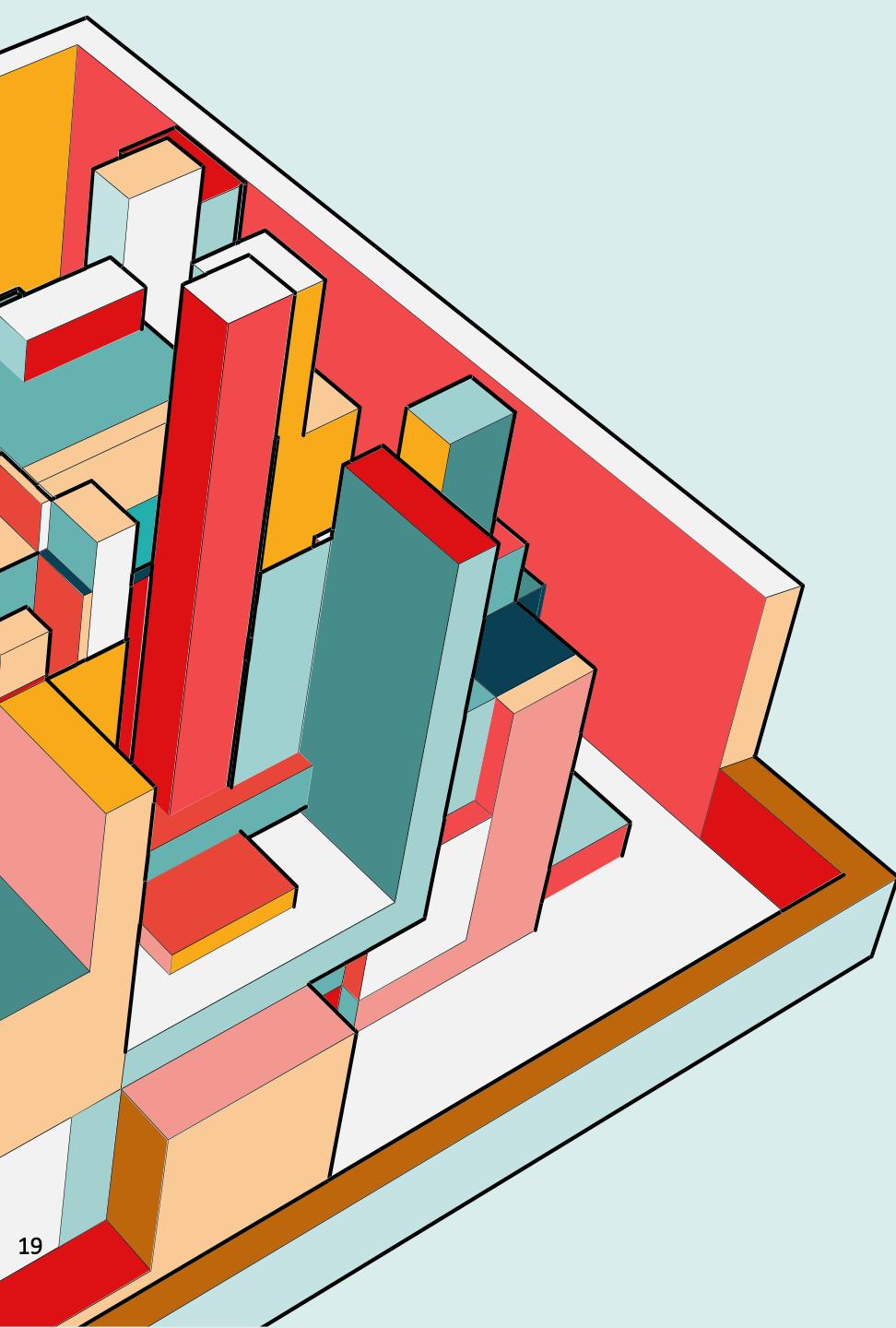
## Идемпотентность в PaymentService

Сервис проверяет payment\_id или order\_id, и игнорирует повторные события

## Использование паттерна Saga

- Каждое действие сопровождается compensating action
- Если оплата не прошла — заказ отменяется
- Если заказ отменён после оплаты — деньги возвращаются

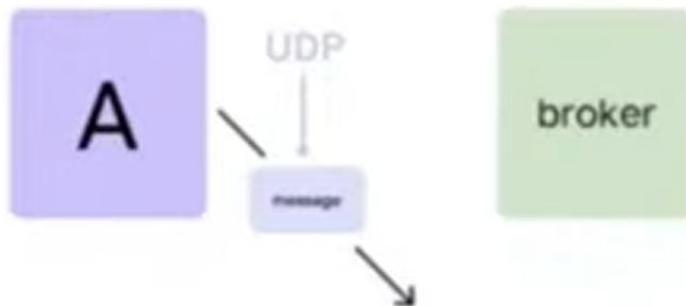




# **НАЧНЕМ С УРОВНЯ ВЗАЙМОДЕЙСТВИЯ С БРОКЕРОМ СООБЩЕНИЙ**

Transaction Outbox

# СООБЩЕНИЕ НЕ ПОПАДАЕТ В БРОКЕР



**Решение:**

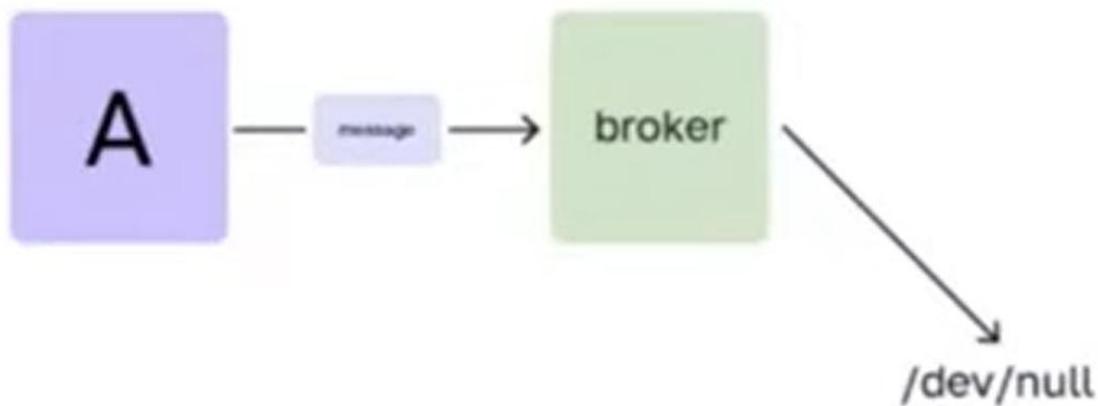
Настроить подтверждение на стороне брокера о том, что сообщение получено.

**Недостатки:**

Увеличивается время доставки сообщения.



# БРОКЕР НЕ СОХРАНЯЕТ СООБЩЕНИЕ



Решение:

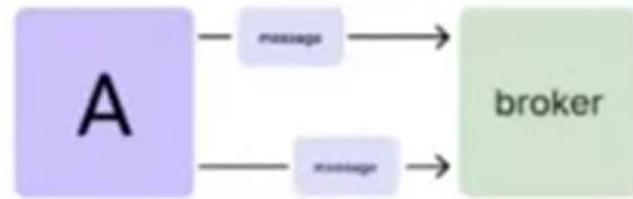
Настроить отправку реплики сообщения брокеру.

Недостатки:

Увеличивается расход памяти, необходимый для обмена.



# ДУБЛИРОВАНИЕ СООБЩЕНИЙ



## Решение:

Настроить на брокере механизм дедупликации: на стороне поставщика устанавливать уникальный идентификатор сообщения. Тогда брокер будет сохранять только первое.

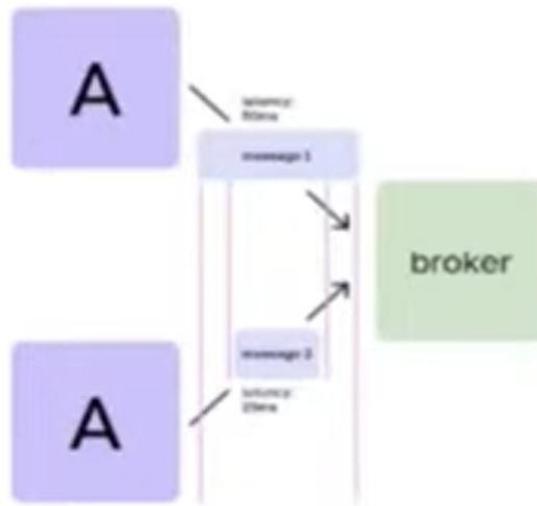
## Недостатки:

Усложнение разработки на стороне поставщика сообщений.



# ПОСТАВЩИК НЕ УВЕРЕН В ПОРЯДКЕ СОБЫТИЙ

Решение:



Научить поставщика событий присваивать событиям порядковые ключи-инкременты и научить брокера встраивать обработку сообщений по этим ключам.

Недостатки:

- Усложнение разработки
- Увеличение нагрузки на брокер
- Увеличение срока доставки сообщений



# TRANSACTION OUTBOX

Событие не отправляется сразу. Вместо этого:

1. Создаётся заказ в своей БД
2. В той же транзакции записываешь событие в **таблицу outbox**

Только потом:

3. Фоновый процесс (Outbox Processor) читает эту таблицу и **отправляет события в брокер**
4. После успешной отправки — помечает их как **sent** или удаляет

[OrderService]

→ BEGIN TRANSACTION

- Вставка в таблицу orders
- Вставка в таблицу outbox

COMMIT



[Outbox Processor]

- читает из outbox
- отправляет в Kafka / RabbitMQ
- помечает как отправленное

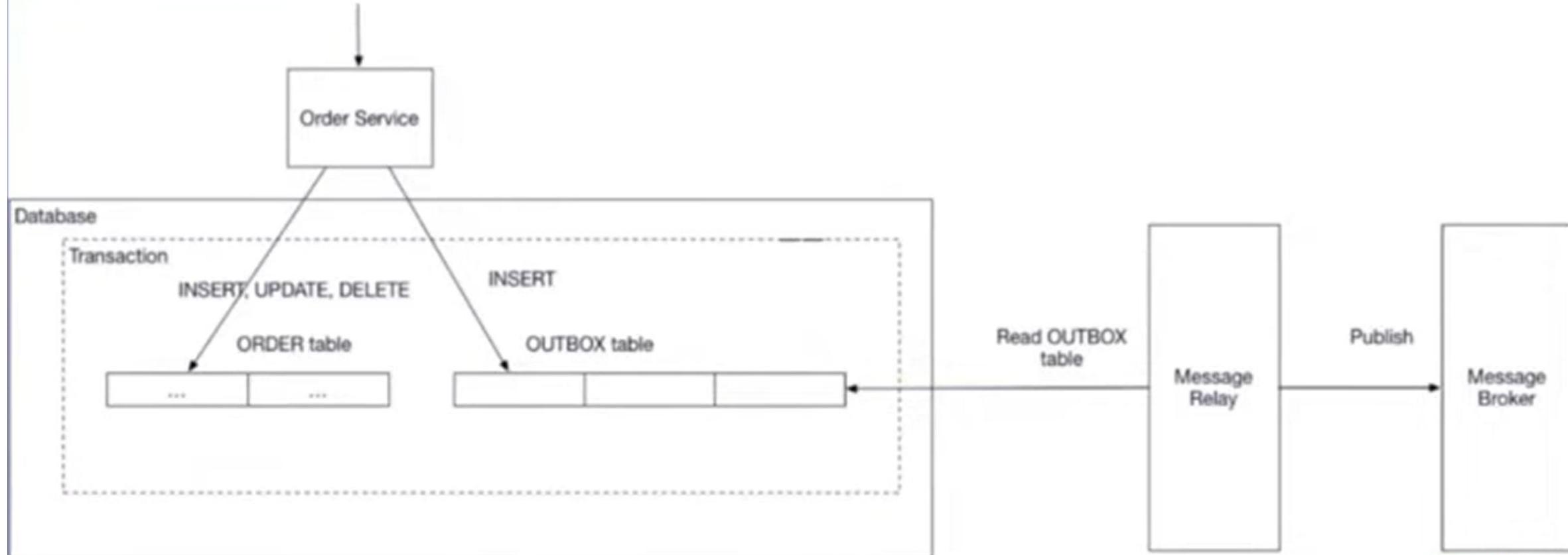
## КТО ИСПОЛЬЗУЕТ?

Netflix, Uber, Booking, Ozon, Авито, и многие другие

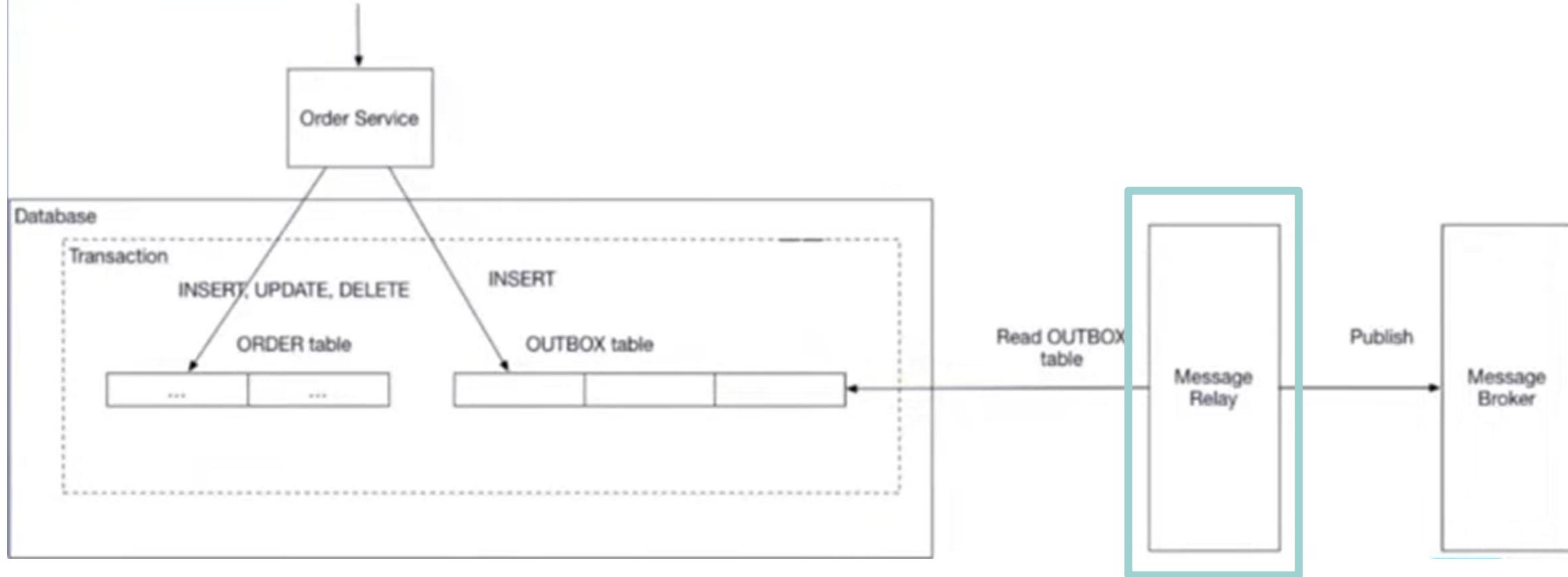
Многие используют вместе с Debezium (Change Data Capture),  
чтобы вообще не писать Outbox Processor руками



# СХЕМА РАБОТЫ



# РЕТРАНСЛЯТОР СООБЩЕНИЙ

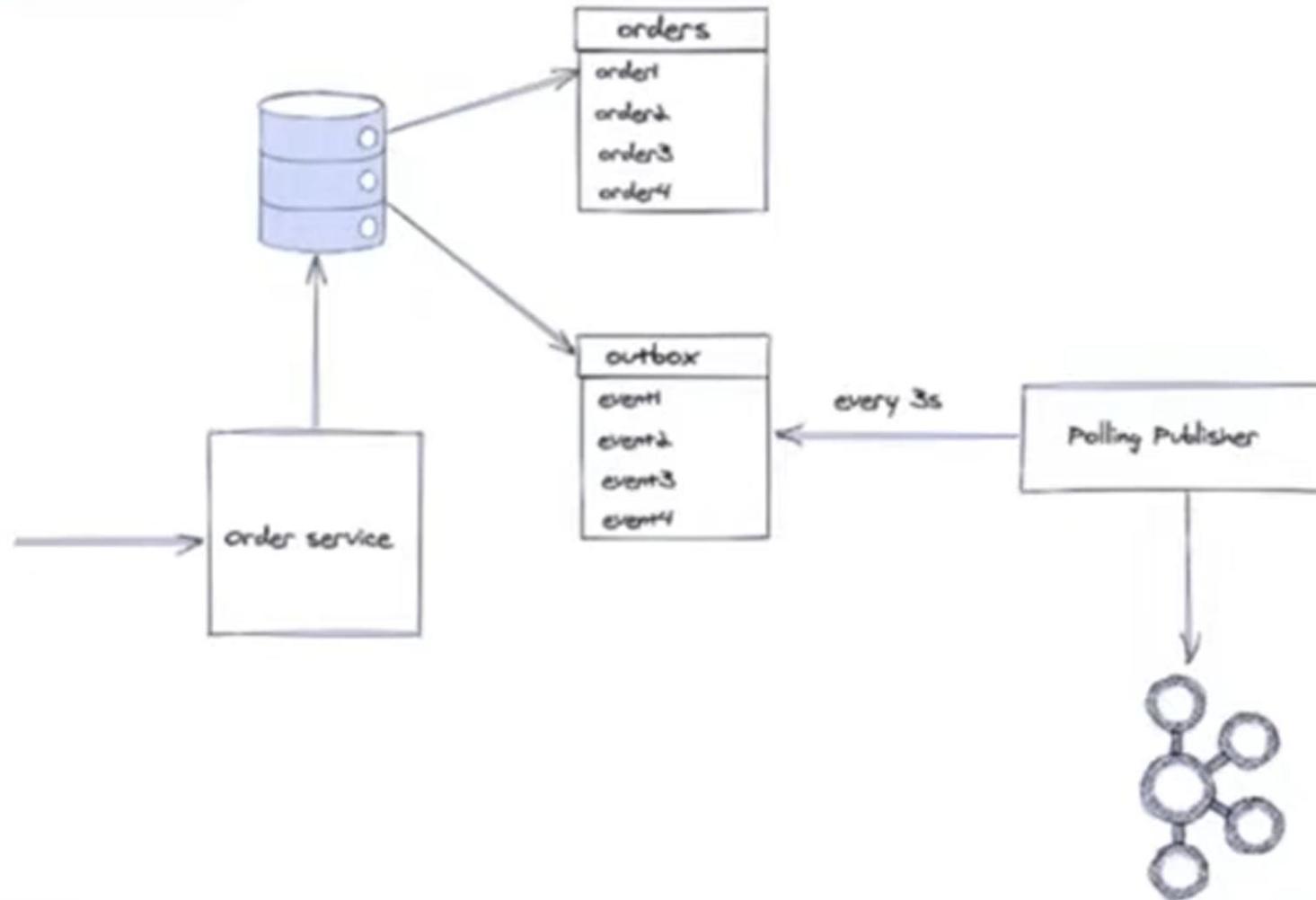


# ВИДЫ РЕТРАНСЛЯТОРОВ

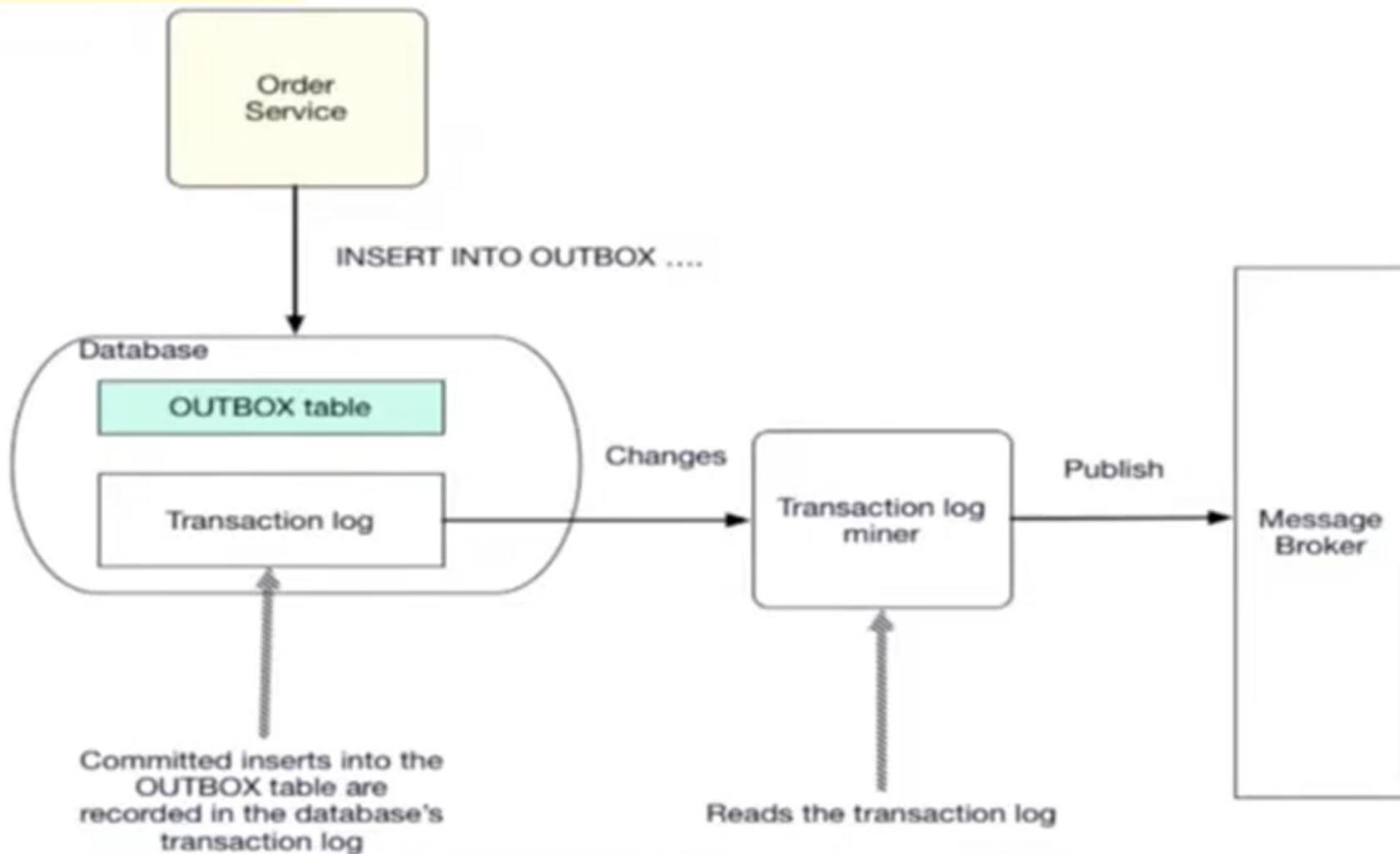
- 1 Polling publisher (Опрашивающий издатель)
- 2 Transactional log tailing (Отслеживание транзакционного журнала)



# ОПРАШИВАЕМЫЙ ИЗДАТЕЛЬ



# ОТСЛЕЖИВАНИЕ ТРАНЗАКЦИОННОГО ЖУРНАЛА



# ПЛЮСЫ И МИНУСЫ TRANSACTION OUTBOX

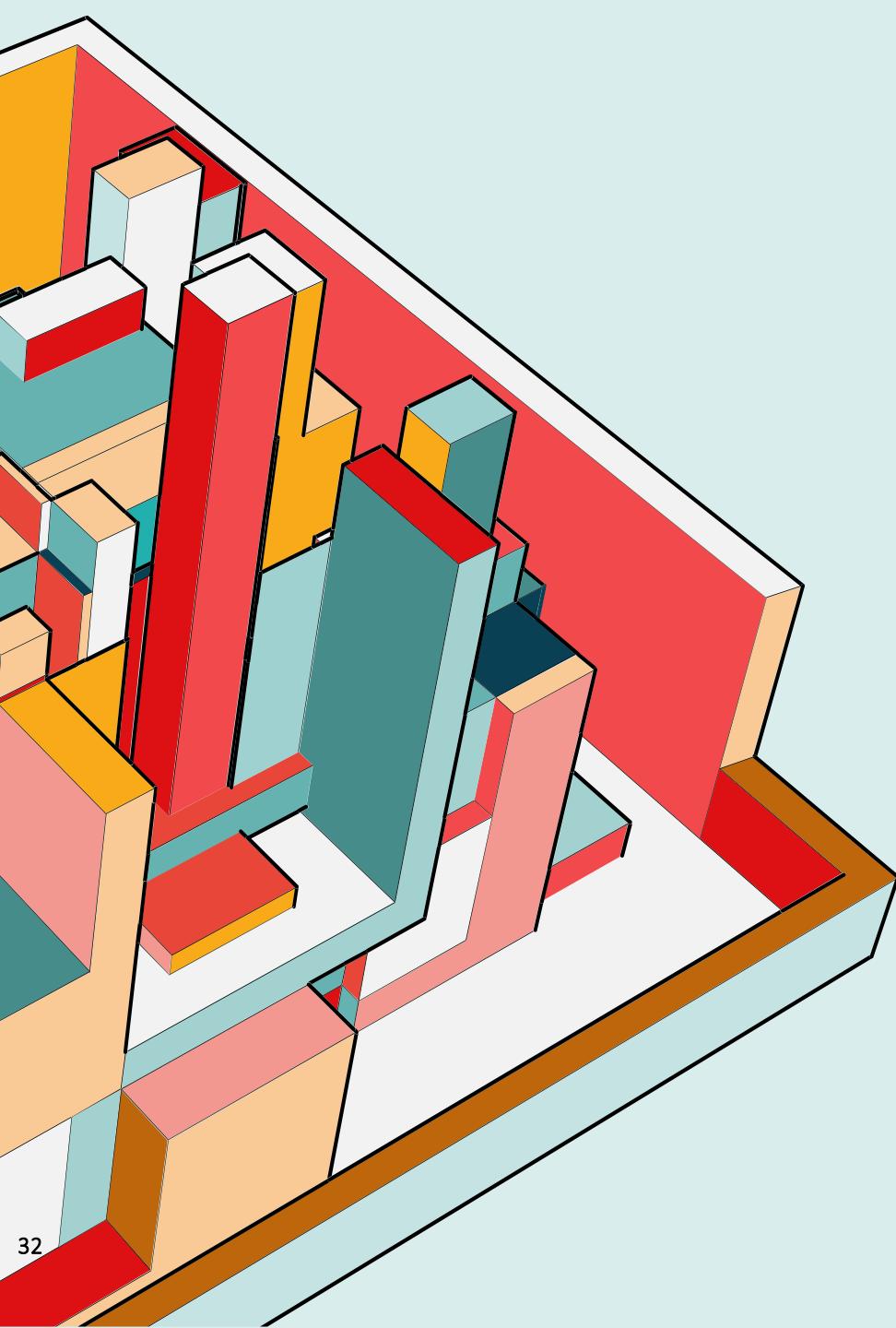
Плюсы:

- 1 Повышение гарантий доставки сообщения: оно не будет потеряно при отправке
- 2 Повышение гарантий сохранения порядка сообщений: можно запрограммировать поставщика так, чтобы сообщения доходили потребителю в необходимом порядке

Минусы:

- 1 Усложняется разработка поставщика и брокера сообщений
- 2 Замедляется отправка сообщений потребителю



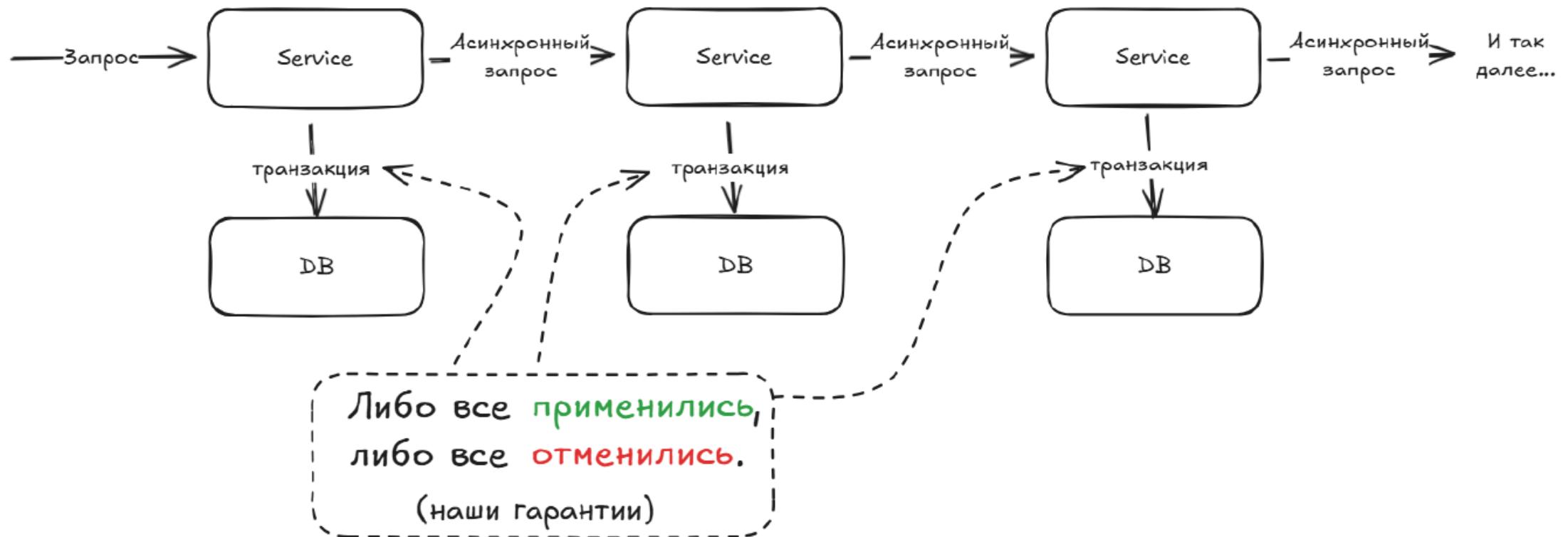


# **РАСПРЕДЕЛЕННЫЕ ТРАНЗАКЦИИ В УСЛОВИЯХ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ**

Перейдем к уровню  
межсервисного взаимодействия

# Надо вспомнить что мы хотим

Хотим гарантировать вот это



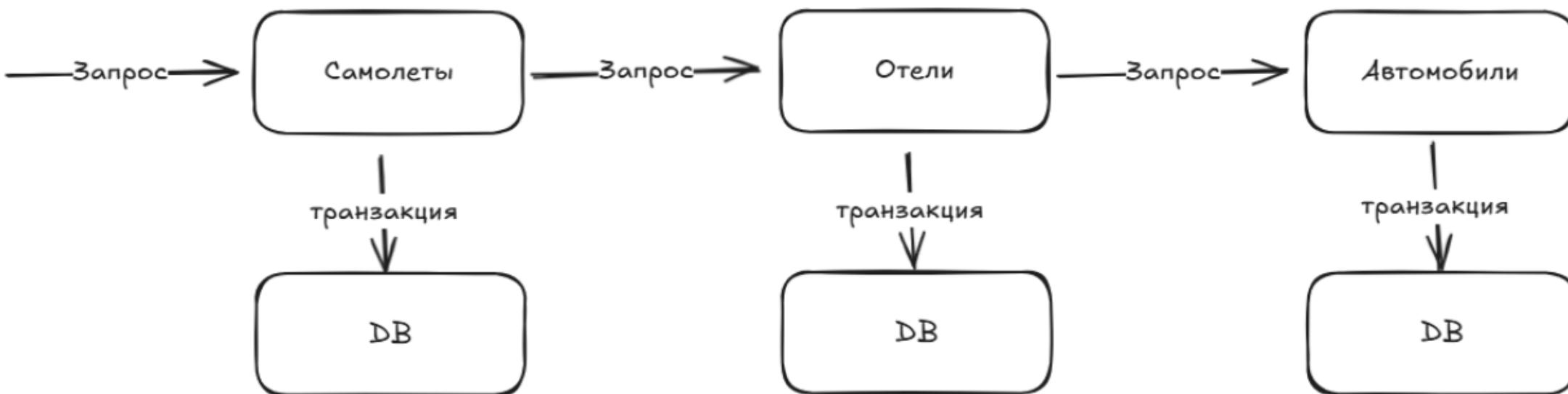
Это называется **распределенная транзакция**

# Бронируем тур перед путешествием

Хотим как то туда добраться

Хотим там где-то жить

Пока там будем хотим  
арендовать машину



## Как это сделать?

Вариант 1

## 2-phase-commit

он же двухфазный коммит

он же 2-РС

он же Гоша

он же Гога

он же Юрий

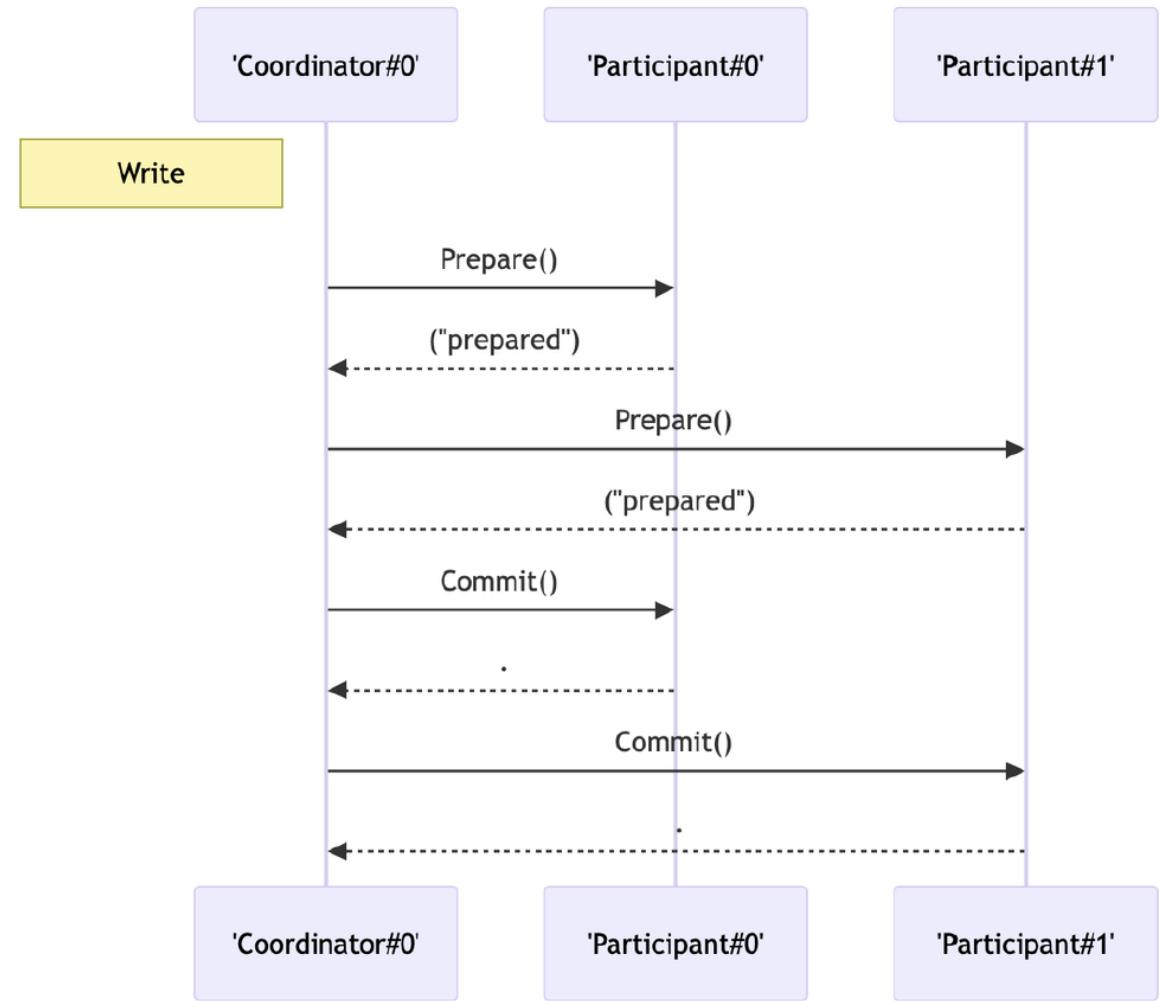
он же Гора

он же Жора

### Ключевая идея:

Транзакцию можно начать сейчас,  
а закончить позже и она не потеряется

“Двухфазный”, потому что  
первая фаза это начало транзакций,  
вторая фаза это завершение транзакций



## Вариант 1

# 2-phase-commit

### Плюсы

- Довольно просто и прозрачно
- Хорошая консистентность данных

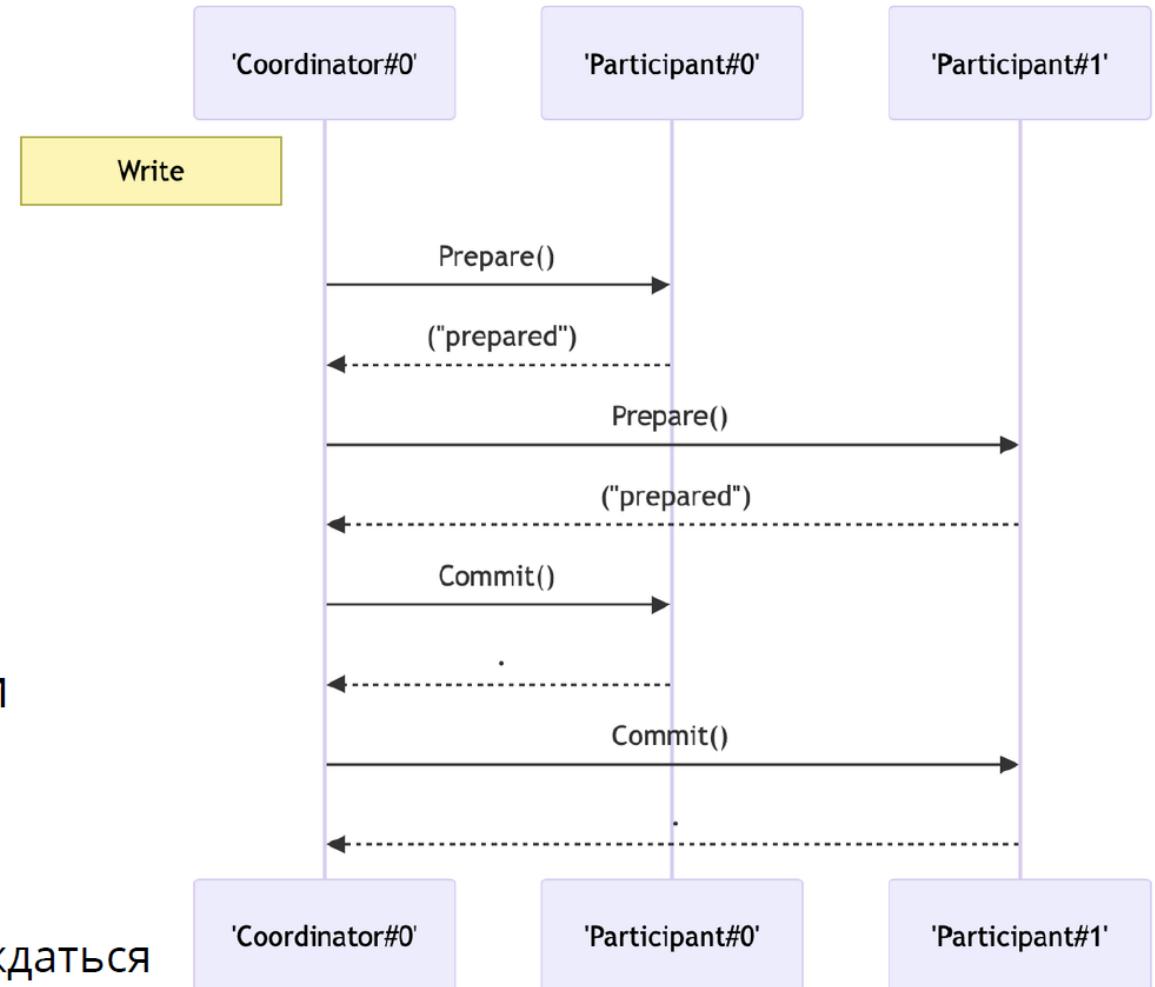
### Минусы

- Единая точка отказа\*
- Все сервисы ждут самого медленного
- Много раундов обмена сообщениями

Как итог - долго

В базах могут копиться зависшие транзакции

Если нас еще и юзер ждет - может вообще не дождаться



\*Если координатор упал - транзакции висят в базах пока он не встанет и не подтвердит их

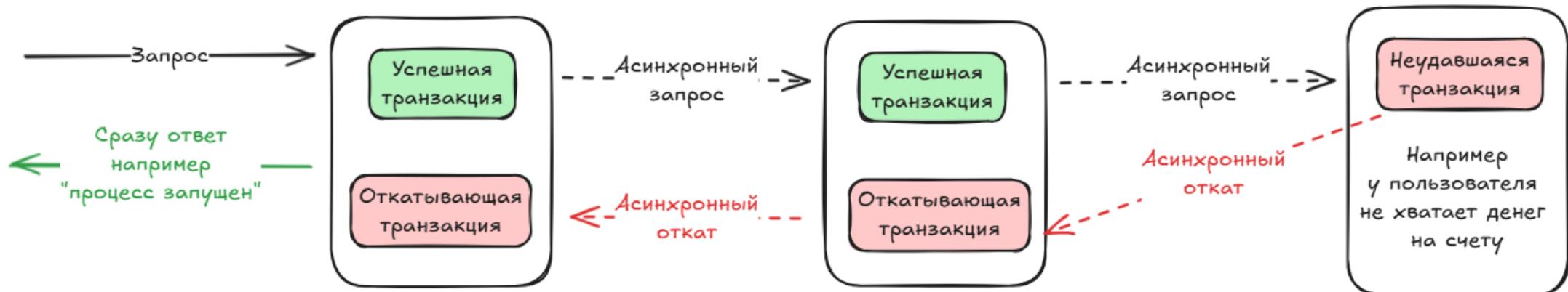
## Вариант 2

# SAGA

Почему название такое - я не нашел. Но и искал я недолго.

### Ключевая идея:

Транзакцию можно закончить сразу, сделав вид что мы уверены в ее успехе  
но на всякий случай запомнить как ее отменять, если придется  
(на каждую транзакцию заводим компенсирующую транзакцию)



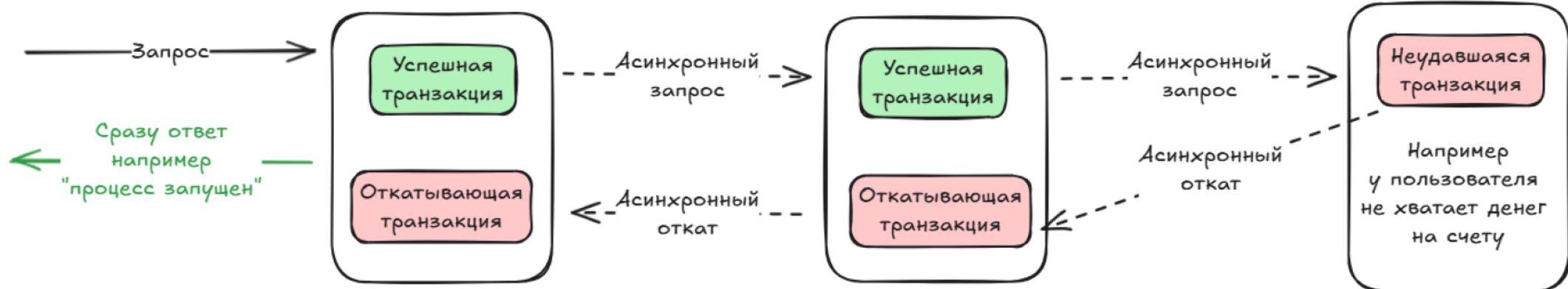
Два подхода:

- Хореографическая SAGA - сервисы сами разбираются как откатить транзакции
- Оркестрированная SAGA - есть сервис-оркестратор, который контролирует сагу

Вариант 2.1

## Хореографическая SAGA

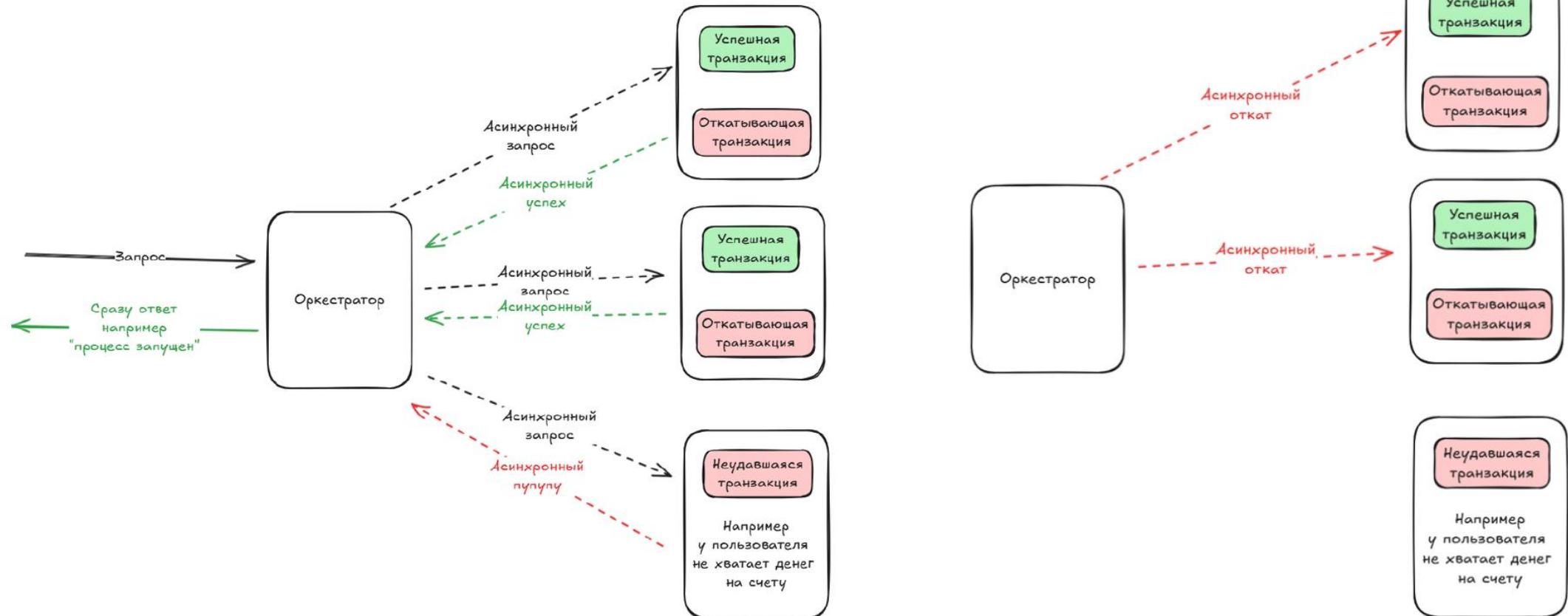
Сервисы сами разбираются как откатить транзакции  
(картинка взята с предыдущего слайда)



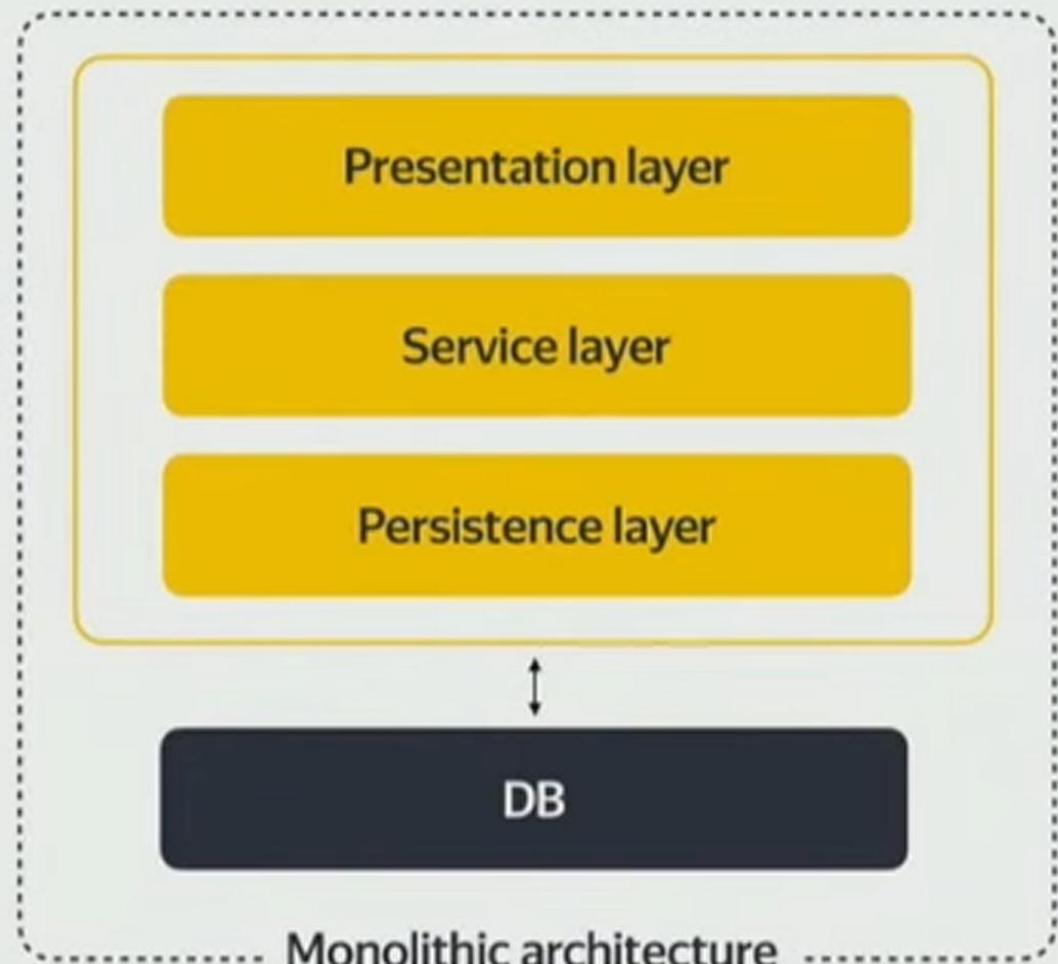
Вариант 2.2

# Оркестрированная SAGA

Есть сервис-оркестратор, который контролирует сагу



# КАК БЫЛО РАНЬШЕ



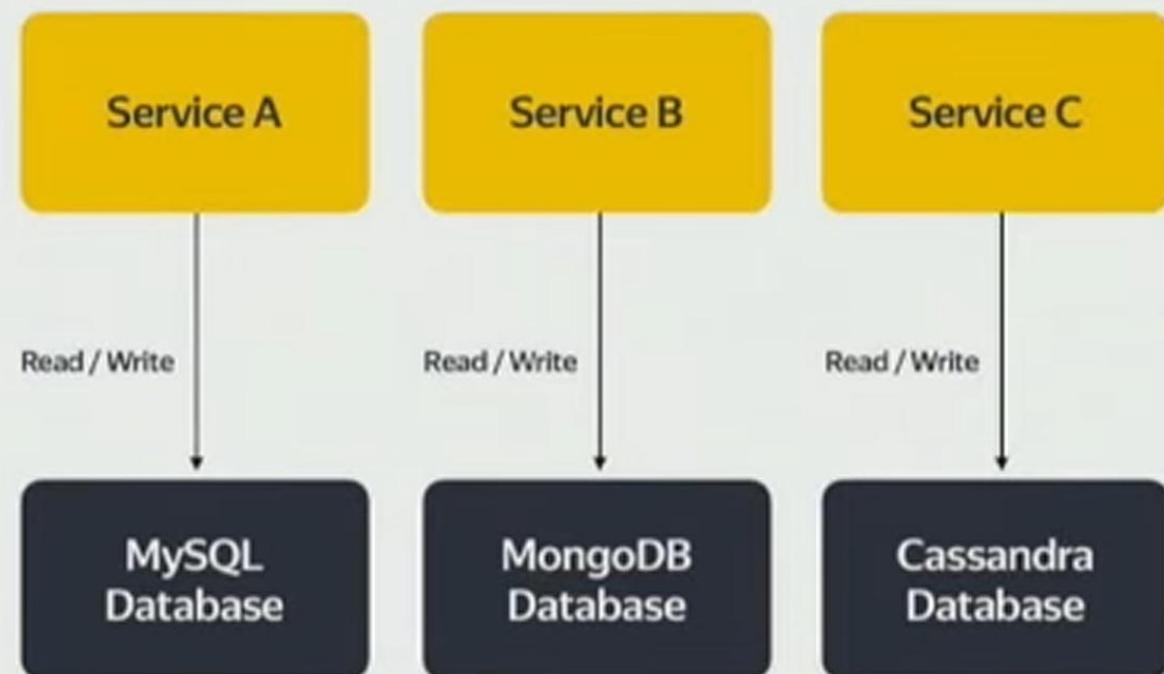
Сценарий: заказ путёвки в тур

- Авиабилет
- Номер в отеле
- Автомобиль в аренду

ACID-транзакции в помощь!

# DATABASE PER SERVICE PATTERN

Database per service



- Сервисы слабо связаны. Изменения хранилища сервиса не касается других
- Свой тип СУБД на сервис.

Теряем ACID-транзакции

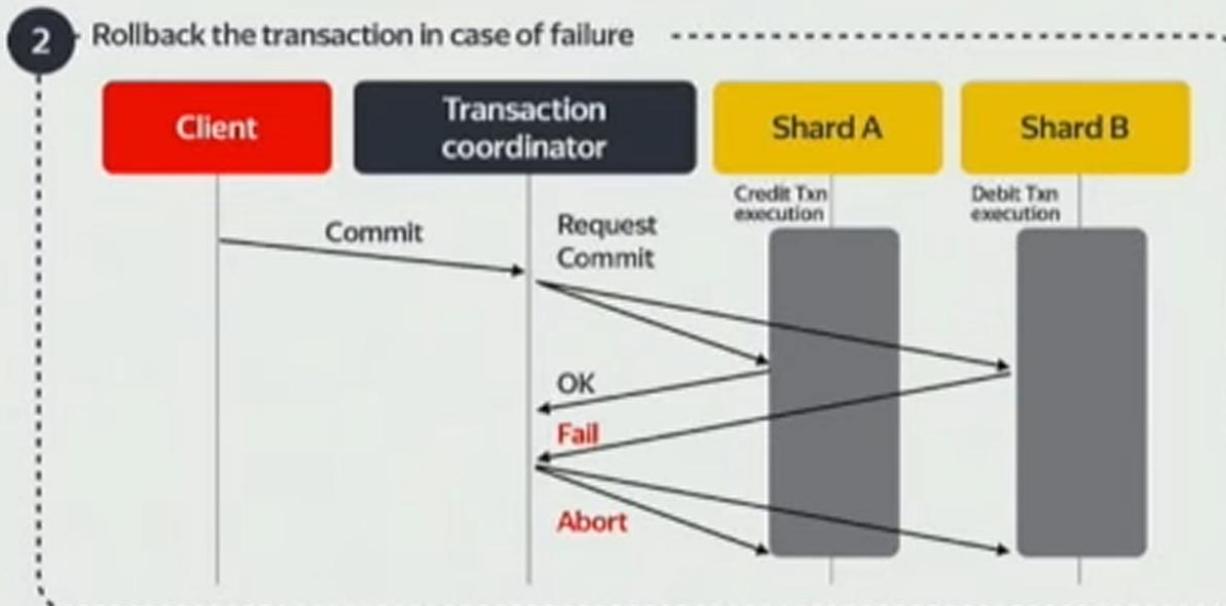
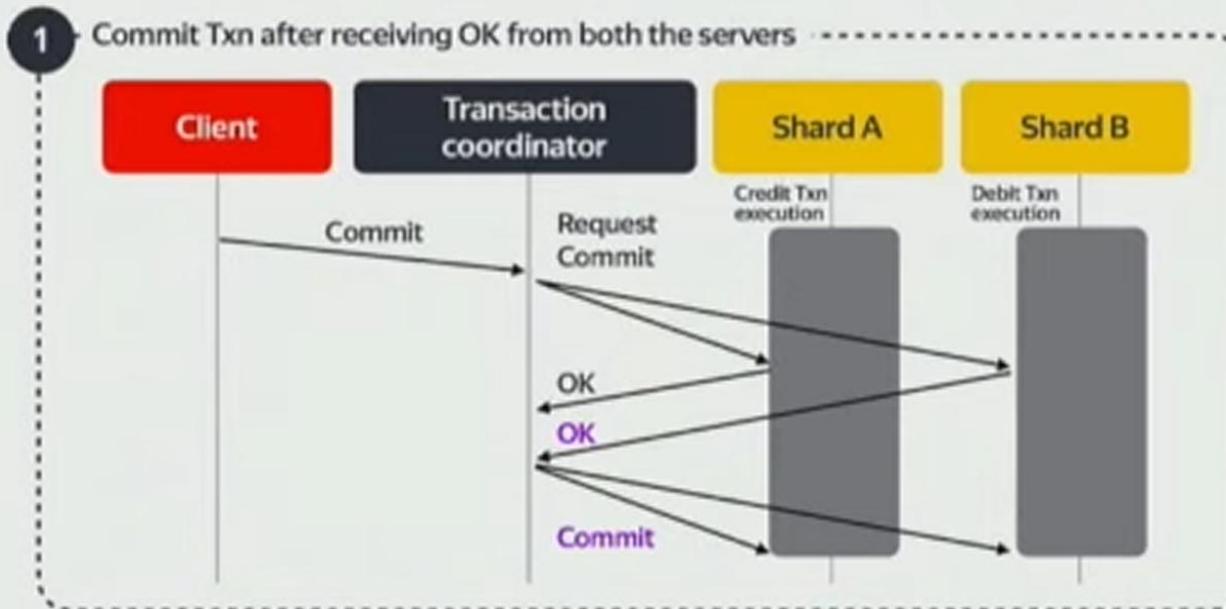
Имеем ACID-транзакции на каждый сервис

Как их согласовать?

# TWO PHASE COMMIT

Можно, но лучше не надо :)

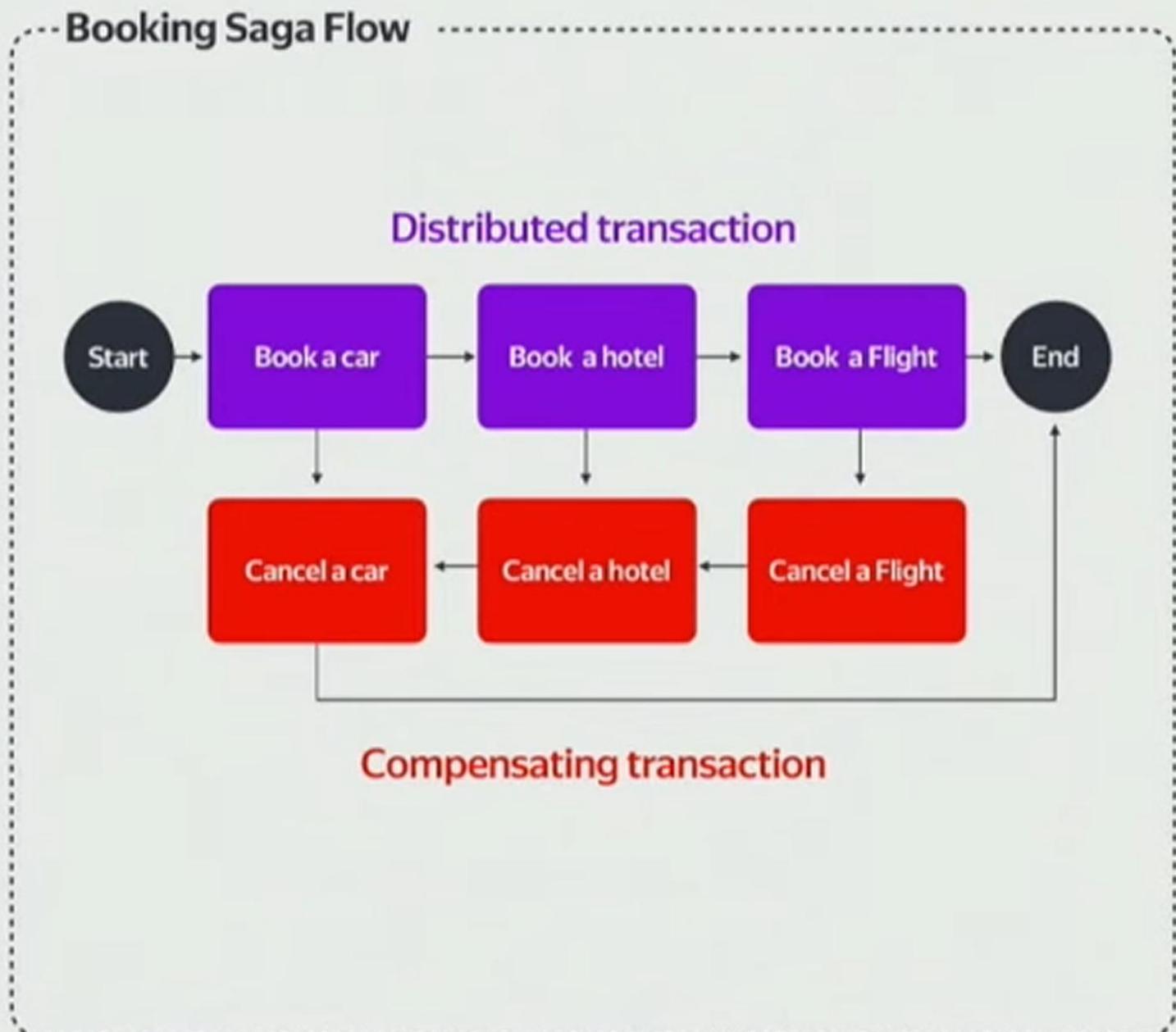
- Единая точка отказа  
(транзакции висят до его восстановления)
- Все ждут самого медленного
- 4 раунда обмена сообщениями



# SAGA ITSELF

## Решение

- Цельная транзакция → сага из локальных транзакций: update/abort
- Локальная транзакция обновляет базу → шлёт событие/триггер следующей
- Сбой локальной транзакции → выполняет свой abort
- Компенсационная транзакция



# СТЕПЕНЬ ПОДДЕРЖАНИЯ SAGA-ОЙ ПРИНЦИПОВ ACID

**Атомарность:** — Saga гарантирует, что либо все сервисы применяют локальные транзакции, либо, в случае сбоя, все уже выполненные локальные транзакции компенсируются, так что никакие изменения данных не применяются.

**Согласованность:** — После успешного выполнения всех транзакций, составляющих сагу, все системы переводятся из одного согласованного состояния в другое.

**Изоляция:** — Поскольку локальные транзакции коммитятся во время выполнения Saga, их изменения уже видны другим параллельным транзакциям, несмотря на возможность того, что Saga в конечном итоге завершится сбоем, в результате чего все ранее примененные транзакции будут компенсированы. Т.е., с точки зрения общей Saga, уровень изоляции равен сравним с «read uncommitted»

**Надежность:** — После комита локальных транзакций Saga их изменения сохраняются даже в случае сбоя сервиса и перезапуска.

# ✗ ИЗОЛЯЦИИ – НЕ КОНЦ СВЕТА!

Обманутые транзакции. Виды: **False negative**

USER: Вася

```
for {  
    car <- selectCar(model = Cars.Maybach) // TxIO[CarIsBusyError, Car]  
        .orElse(selectCar(model = Cars.Aurus)) // TxIO[CarIsBusyError, Car]  
    _ <- markAsBusy(model = car.model) // Cars.Maybach  
} yield car
```

USER: Петя

```
for {  
    car <- selectCar(model = Cars.Maybach) // TxIO[CarIsBusyError, Car]  
        .orElse(selectCar(model = Cars.Aurus)) // TxIO[CarIsBusyError, Car]  
    _ <- markAsBusy(model = car.model) // Cars.Aurus  
} yield car
```

```
for {  
    _ <- releaseCar(model = Cars.Maybach) // ZIO[Any, Nothing, Unit]  
} yield ()
```

# ✗ ИЗОЛЯЦИИ – НЕ КОНЕЦ СВЕТА!

Обманутые транзакции. Виды: **False positive**

USER: Вася

```
// Read committed isolation level
for {
    car <- selectCar(model = Cars.Maybach) // TxIO[CarIsBusyError, Car]
        .orElse(selectCar(model = Cars.Aurus)) // TxIO[CarIsBusyError, Car]
    // time going..
    _ <- markAsBusy(model = car.model) // Cars.Maybach
    // time going..
} yield car
```

USER: Петя

```
// Read committed isolation level
for {
    // time going..
    // time going..
    car <- selectCar(model = Cars.Maybach) // TxIO[CarIsBusyError, Car]
        .orElse(selectCar(model = Cars.Aurus)) // TxIO[CarIsBusyError, Car]
    // time going..
    _ <- markAsBusy(model = car.model) // Cars.Maybach
    // time going..
} yield car
```

# ИЗОЛЯЦИИ – НЕ КОНЕЦ СВЕТА!

## Выводы

01

Свести логику  
приложения к false  
negative

02

Получаем по большей  
мере целевой,  
по меньшей мере  
допустимый результат

03

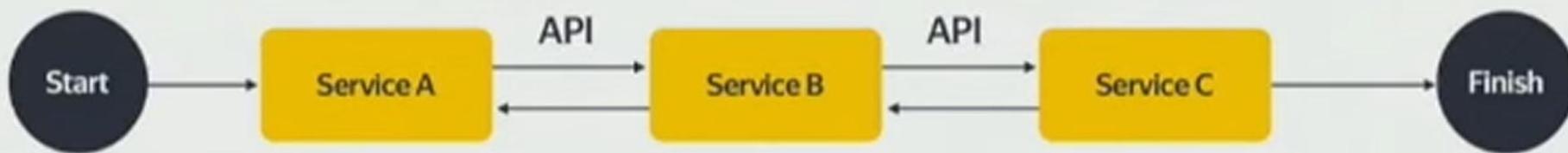
С SAGA можно жить,  
хоть и без I из ACID

04

В общем случае  
эффективнее 2PC

# КАК СЕРВИСАМ ОТЧИТЬВАТЬСЯ ДРУГ ПЕРЕД ДРУГОМ?

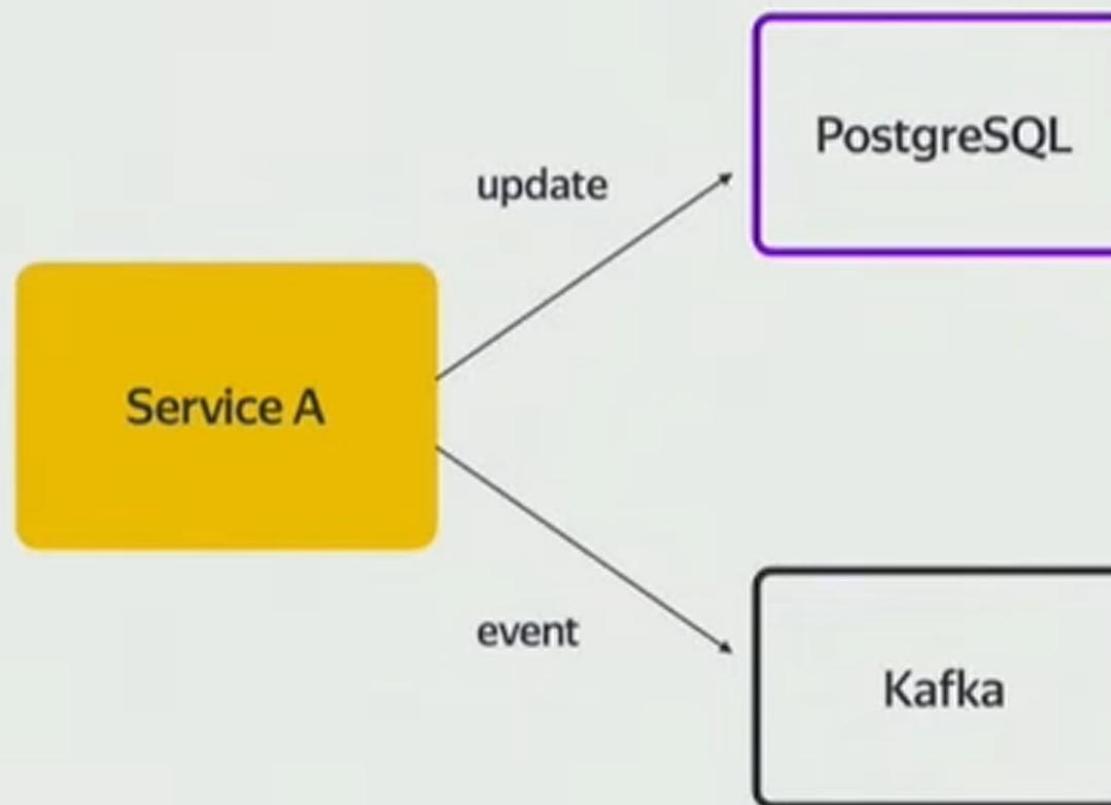
## 1 Синхронно



## 2 Асинхронно — нам это нравится!



# ZOOM-IN НА ФЛОУ ОДНОГО СЕРВИСА

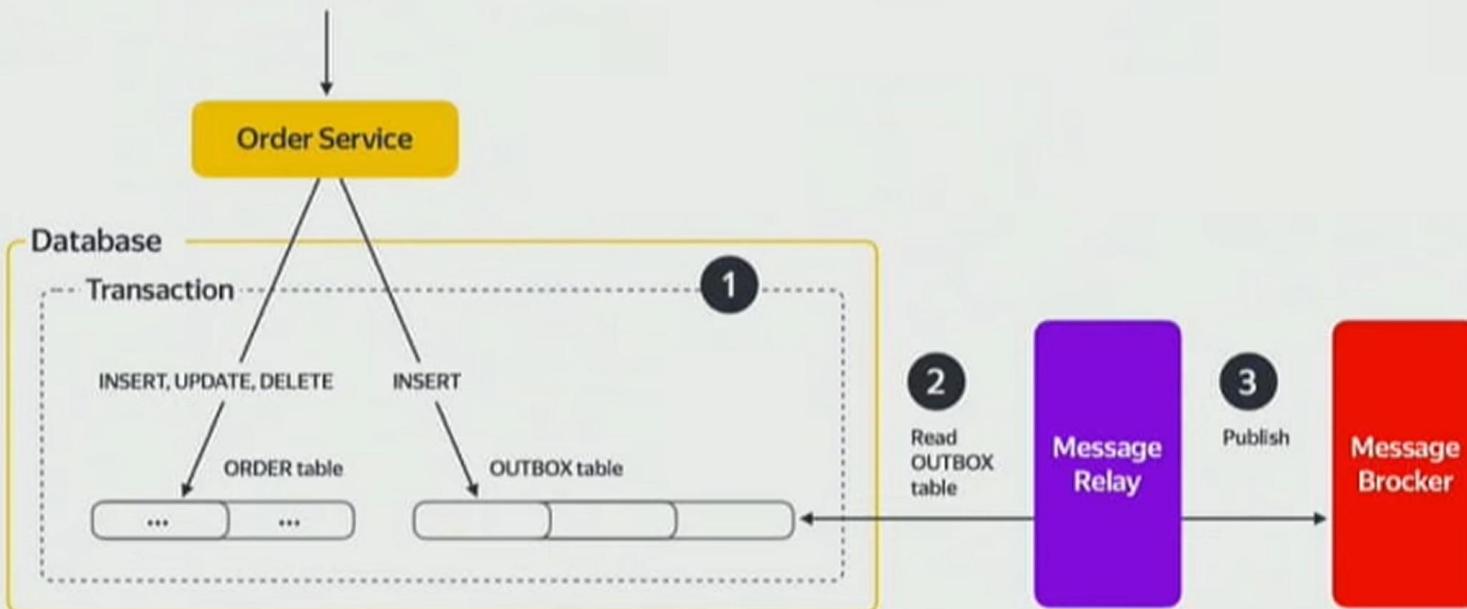


**Что может пойти не так?**

- updateTx() - OK, publish() - OK ✓
- updateTx() - FAIL ✓
- updateTx() - OK, publish() - FAIL ✗

# РЕШЕНИЕ: TRANSACTIONAL OUTBOX

## Внедрение outbox



- Сообщение пишется в таблицу
- Соблюдается атомарность
- Отдельный процесс = отправка сообщений

```
for {  
    txRunner <- db.createTransaction() // ZIO[Any, ConnectionIOException, TxRunner]  
    placeOrderTx = orderDao.insertOrder(order.id) // TxIO[TxError, PlacedOrder]  
    insertToOutboxTx = outboxDao.insertMessage(order.id) // TxIO[TxError, Event]  
    _ <- txRunner.execute(placeOrderTx, insertToOutboxTx) // ZIO[Any, DatabaseError, Unit]  
} yield
```

# ИТОГ ПО РАСПРЕДЕЛЕННЫМ ТРАНЗАКЦИЯМ

**Являются ли они ACID транзакциями? Не совсем.**

- **Atomicity** - “либо применяется целиком, либо нет” - eventually да - ставим
- **Consistency** - “система останется консистентной” - eventually да - ставим
- **Isolation** - “транзакции не влияют друг на друга” - нет
- **Durability** - “если что-то сохранили - оно сохранилось” - да



# ПРИМЕР ПРОБЛЕМЫ ИЗОЛЯЦИИ ТРАНЗАКЦИЙ

Пример нарушения **Isolation**

- Транзакция 1 началась и сохранилась в нескольких сервисах
- Транзакция 2 началась и прочитала данные транзакции 1
- Транзакция 1 отменилась

Сценарии проблемы:

1. **False negative**: транзакция 2 прочитала, что у пользователя недостаточно денег на счету и не продала ему кепку, хотя транзакция 1 откатилась и денег хватает
2. **False positive**: Транзакция 2 прочитала, что у пользователя достаточно денег на счету и продала ему кепку, хотя транзакция 1 откатилась и денег теперь не хватает

+ - терпимо

чревато  
последствиями



# СПАСИБО!

Виденин Сергей

@videninserg

