# Меня хорошо слышно && видно?

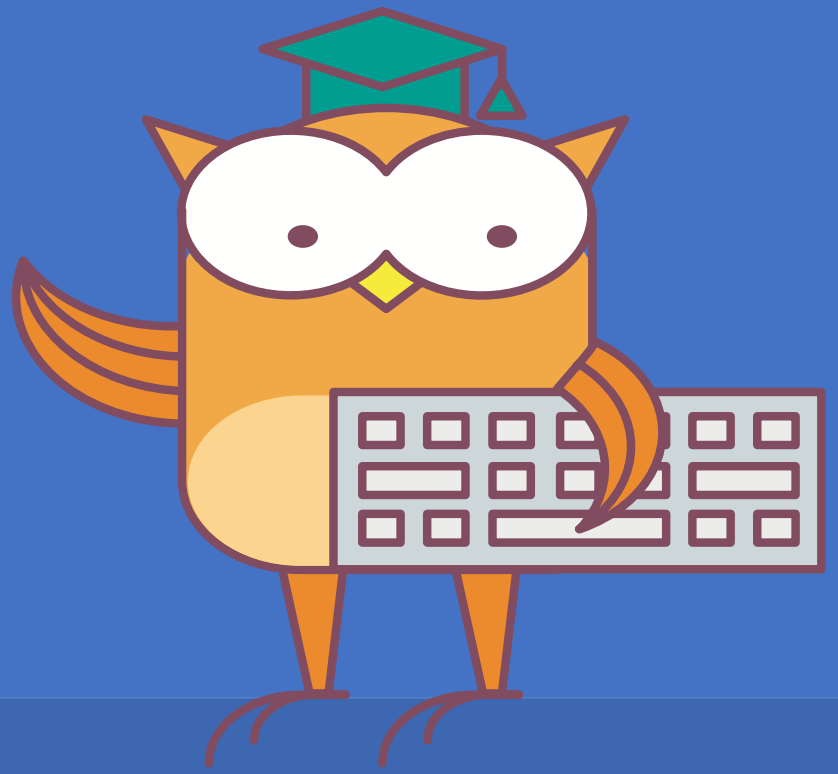
Напишите в чат, если есть проблемы!

Ставьте + если все хорошо

OTUS

# Tracing. Монолиты в микросервисы

Архитектор ПО



Щетинников Стас

# Преподаватель

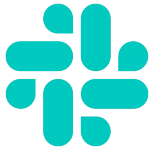
## Непомнящий Евгений

- 15 лет программировал контроллеры на C++ и руководил отделом разработки
- 3 года пишу на Java
- Последнее время пишу микросервисы на Java в Мвидео
- Телеграм @EvgeniyN

# Правила вебинара

Активно участвуем

Задаем вопросы в чат



Off-topic обсуждаем в Slack #канал группы или #general

Вопросы вижу в чате, могу ответить не сразу

## Карта курса

5 Архитектор

1 Инфраструктурные паттерны

**3 Распределенные системы**

2 Коммуникационные паттерны

4 Децентрализованные системы

Опрос по программе - каждый месяц в ЛК

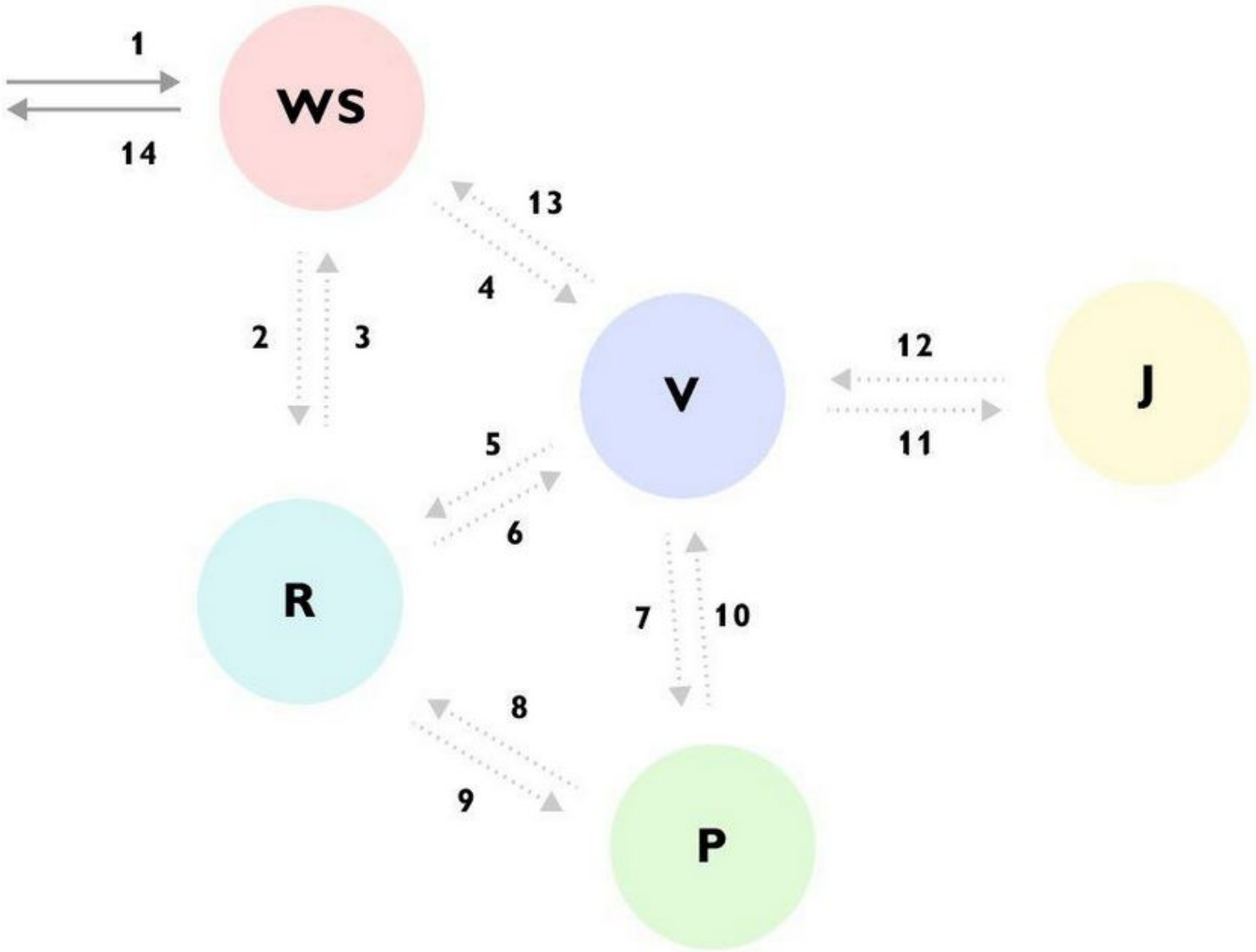
# Карта вебинара

- Distributed tracing
- Паттерны распиливания монолитов

# 01

# Трассировка (tracing)

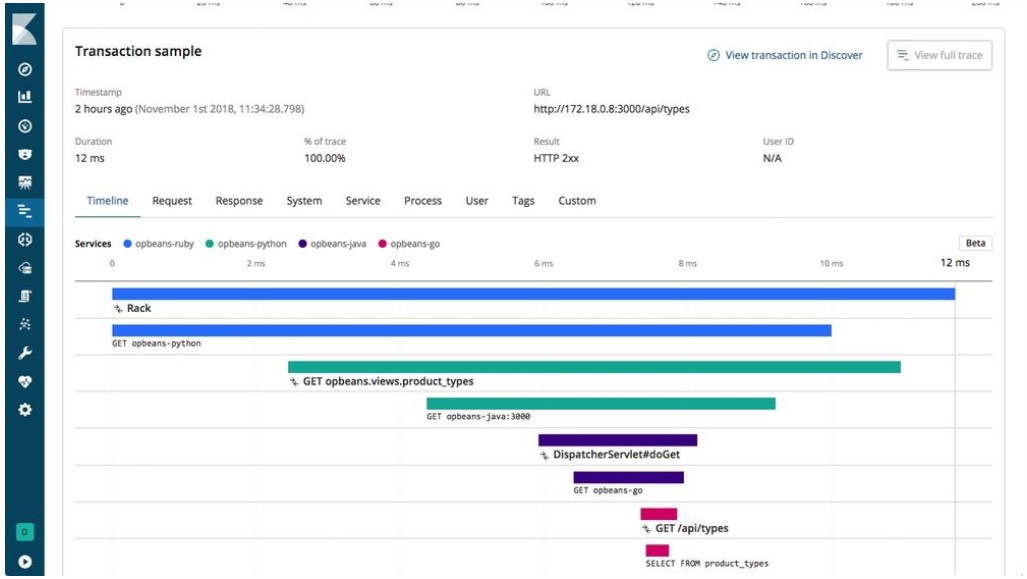
# Distrubuted tracing



# Distrubuted tracing

**Распределенная трассировка** – это путь прохождения запроса по разным сервисам.

- При трассировке, к каждому запросу добавляются метаданные о контексте этого запроса и эти метаданные сохраняются и передаются между компонентами, участвующими в обработке запроса
- В различных точках трассировки происходит сбор и запись событий вместе с дополнительной информацией (URL-запроса, идентификатор клиента, код запроса к БД)
- Информация о событиях сохраняется со всеми метаданными и контекстом и явным указанием причинно-следственных связей между событиями



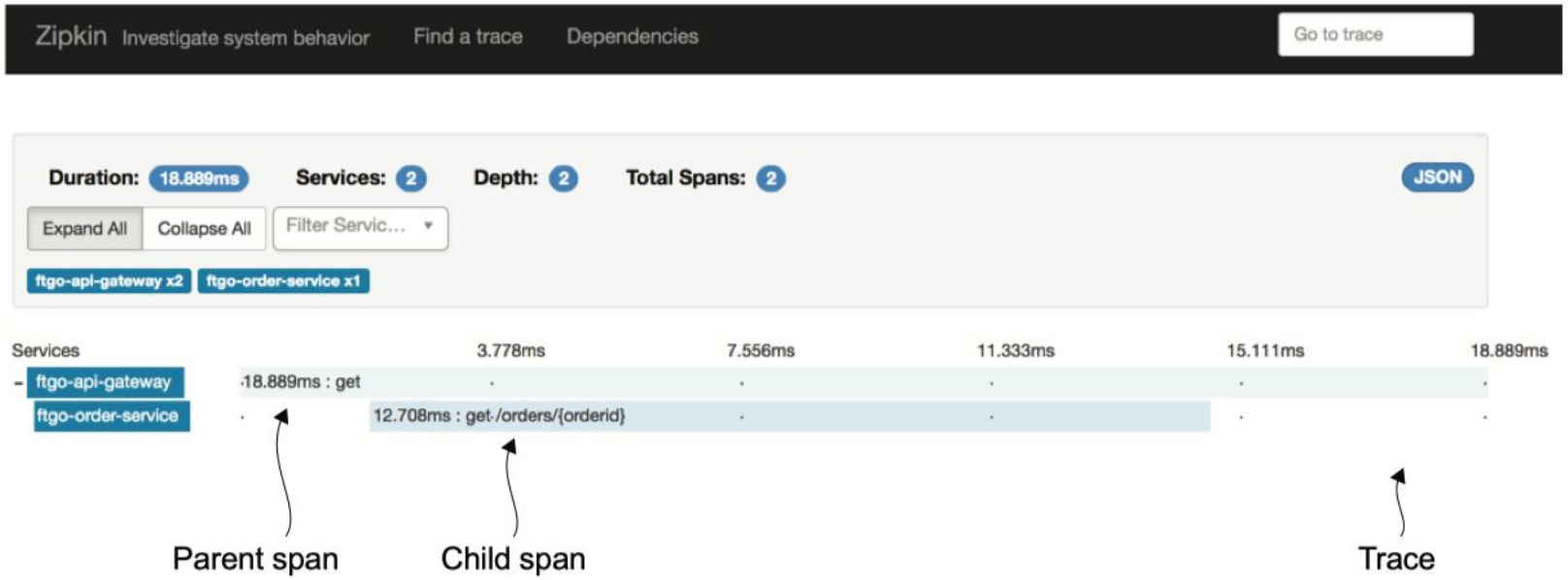
# Основая терминология

**Span -** запись об одной логической операции по обработке запроса (тайминги и метаданные).

- Каждый спан обязательно содержит ссылку на Trace-ID
- Каждый спан содержит свой уникальный идентификатор Span-ID

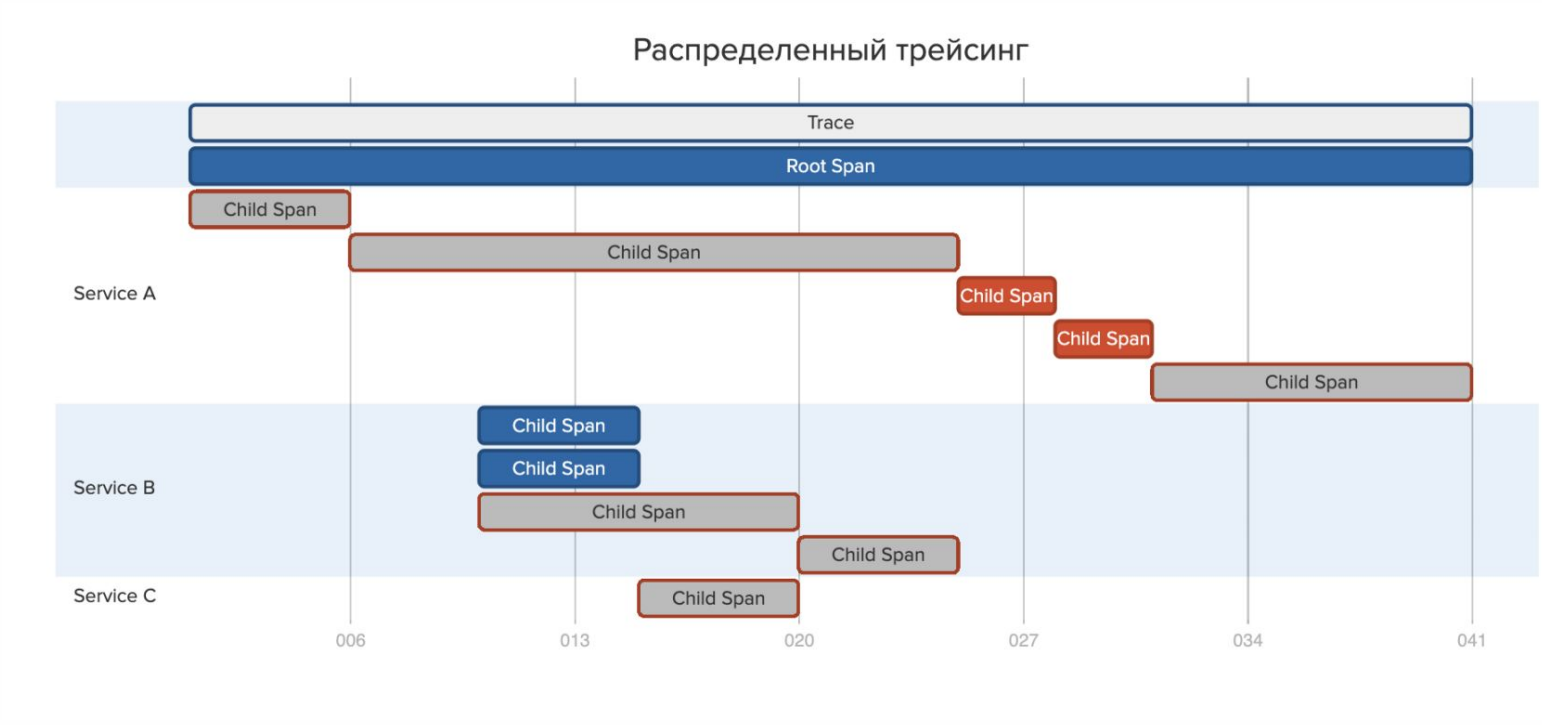
**Trace -** коллекция связанных записей (Spans), описывающая обработку одного запроса (end-to-end)

- каждый трейс имеет свой уникальный идентификатор - Trace ID

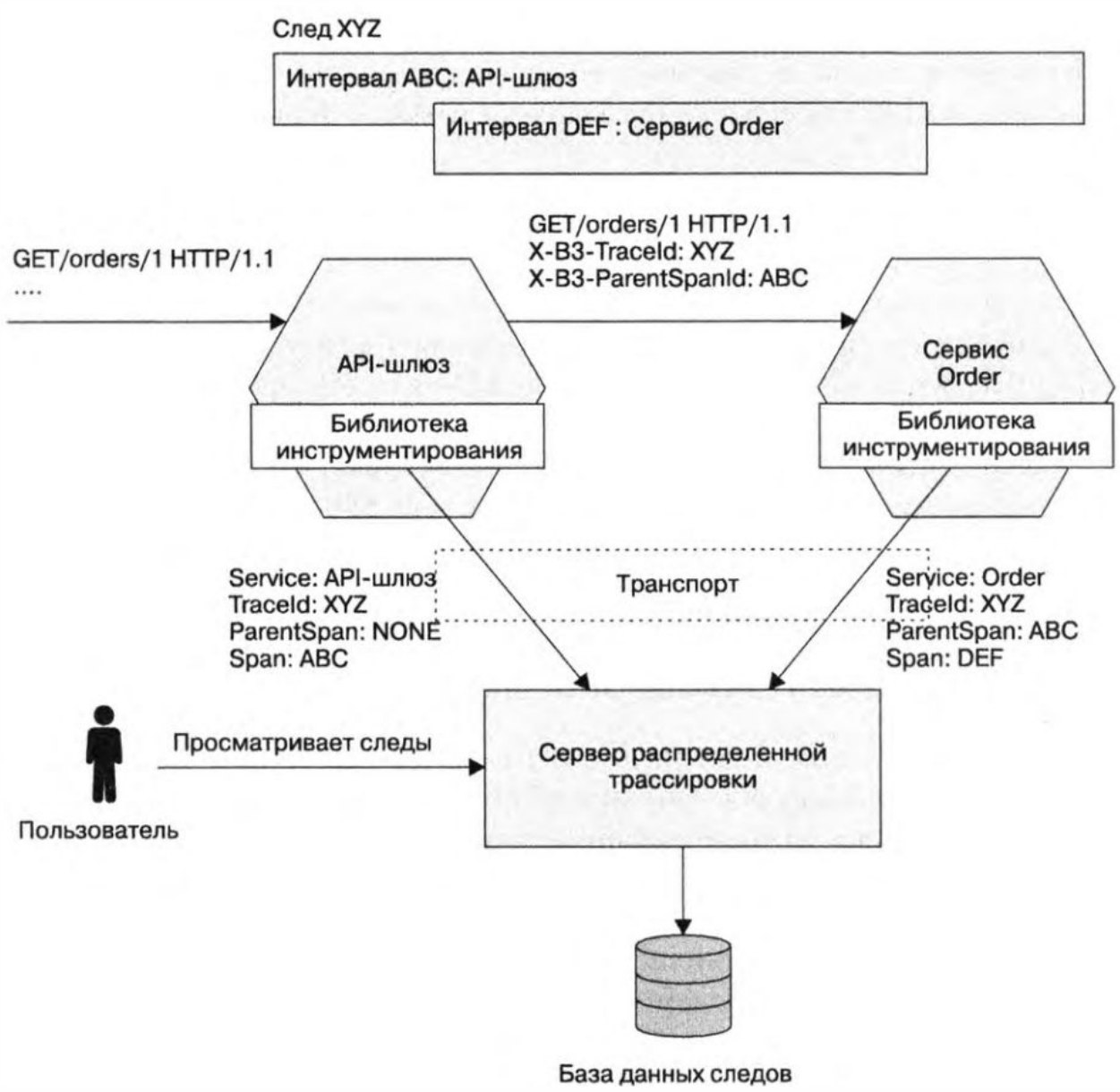


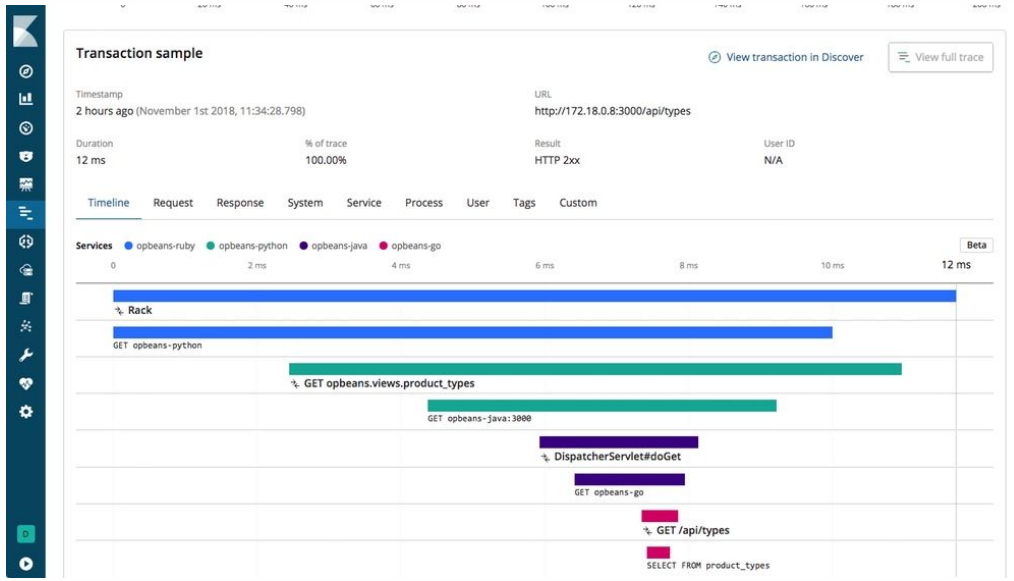
# Основная терминология

- **Root Span** - это спан, у которого нет ссылки на родительский спан (только Trace ID), он показывает общую длительность выполнения запроса



# Distrubuted tracing



# Context Trace

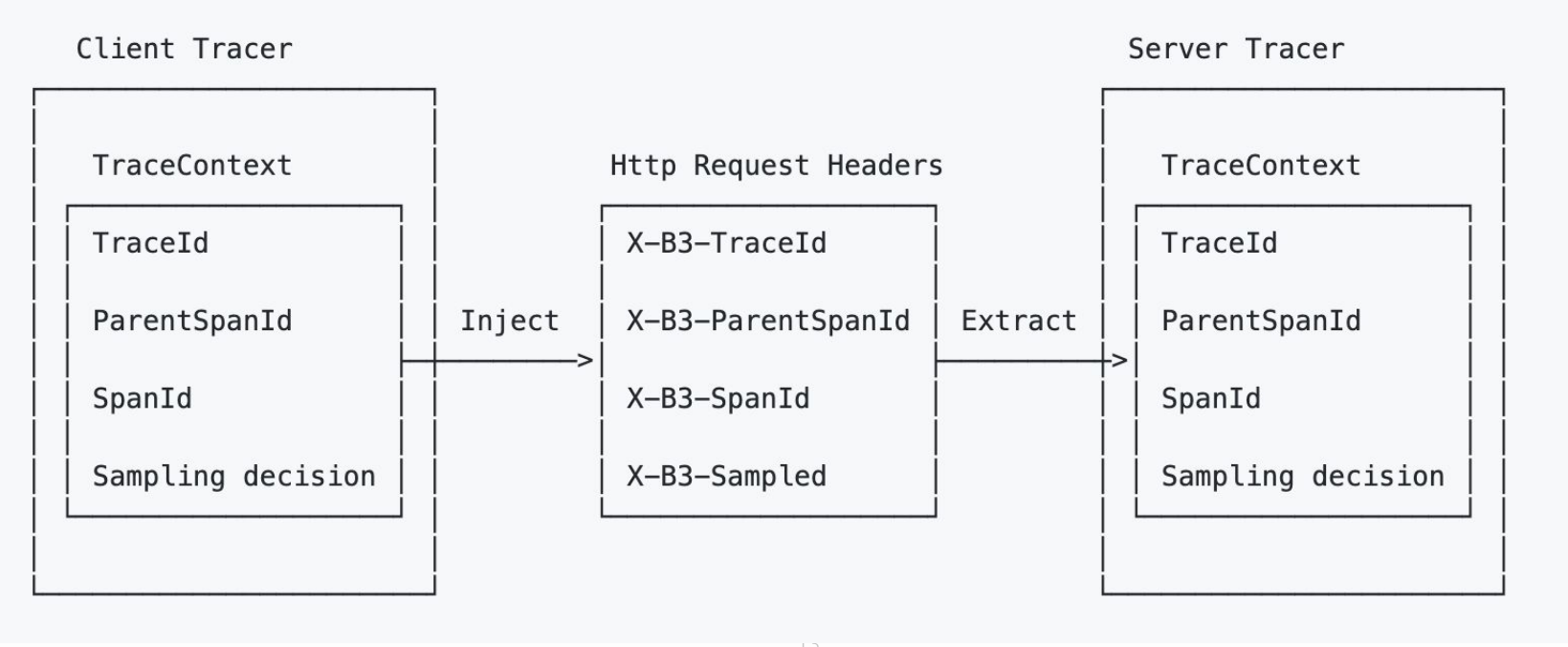
**Context trace** - каким образом мы прокидываем контекст трейса (trace-id, span-id, данные и т.д) от одного сервиса в другой.

Для HTTP есть рекомендации W3C Context Trace - https://www.w3.org/TR/trace-context-1/

Есть также свои реализации:

- Zipkin B3 протокол (https://github.com/openzipkin/b3-propagation )
- Uber протокол

И т.д.



# Для чего используется tracing?

- Упрощенное взаимодействие между командами - при регрессах можно скинуть TraceID, связать систему трэкинга ошибок с трейсами
- Оценка критического пути выполнения запроса и влияния разных факторов на время выполнения (сетевые проблемы, медленные запросы к БД)
- Графы зависимостей - с кем взаимодействует мой сервис, кого затронут изменения в нем?

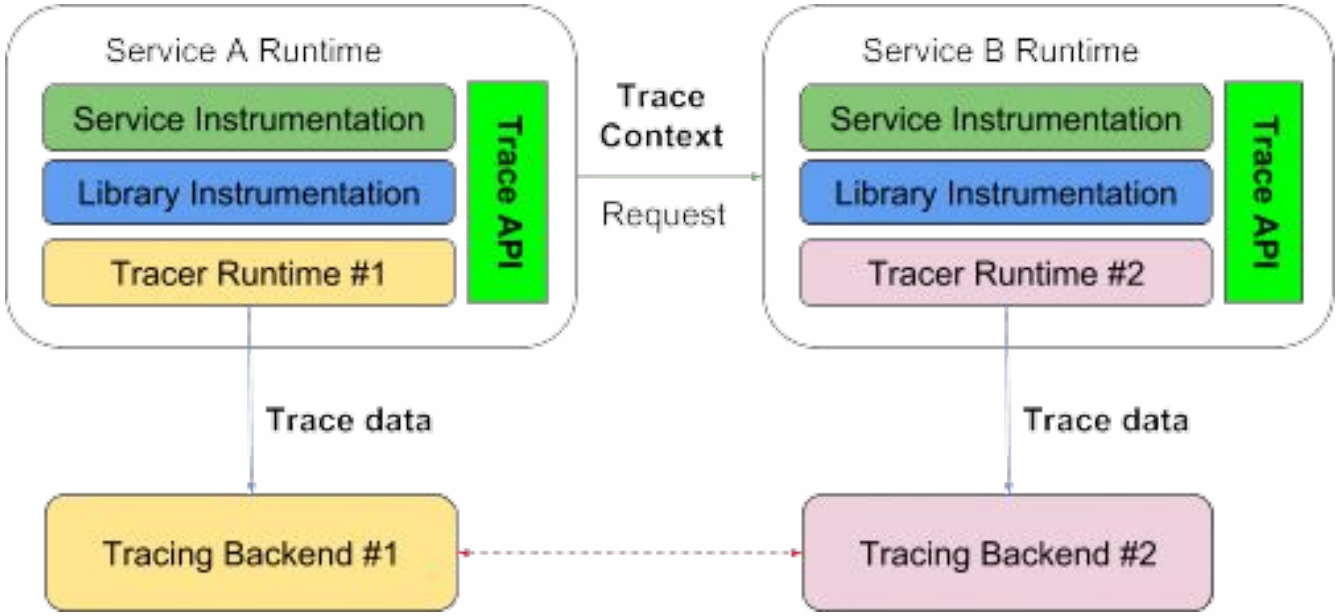
# Какие есть проблемы

- Не видны проблемы общей инфраструктуры (состояние очередей, IOPS и т. п.), "серые ошибки" в облаках
- В трассировках нет "низкоуровневых" данных - состояние ОС, ядра и т.п., то что добывается strace, ss и прочим
- Для протоколов, где нет метаданных, надо писать свои обвязки и прокидывать
- Надо выбирать нагрузку и частоту сэмплирования

# Инструментарий для трейсинга

Обычно разработчик сталкивается со следующими вопросами:

- Как систему для трейсинга использовать?
- А что если я захочу поменять трейсер? Мне придется переписывать весь код?
- А что будет с библиотеками, которые используют используют разные трейсеры?



# Opentracing

**Opentracing** – это спецификация на то, как должен выглядеть класс Tracer в разных языках программирования, чтобы его можно было поменять, не меняя код сервиса. https://opentracing.io/specification/

```
io.opentracing.Tracer tracer = ...; // GlobalTracer.get()

void DoSmth () {
   try (Scope scope = tracer.buildSpan("DoSmth").startActive(true)) {
      ...
   }
}
void DoOther () {
    Span span = tracer.buildSpan("someWork").start();
    try (Scope scope = tracer.scopeManager().activate(span, false)) {
        // Do things.
    } catch(Exception ex) {
        Tags.ERROR.set(span, true);
        span.log(Map.of(Fields.EVENT, "error", Fields.ERROR_OBJECT, ex, Fields.MESSAGE, ex.
    } finally {
        span.finish();
    }
}
```

# Инструменты для Tracing

- **Zipkin** (https://zipkin.io/ )

Много клиентских библиотек, свой протокол B3, который многими сторонними трейсерами поддерживается

- **Jaeger** (https://www.jaegertracing.io/ )

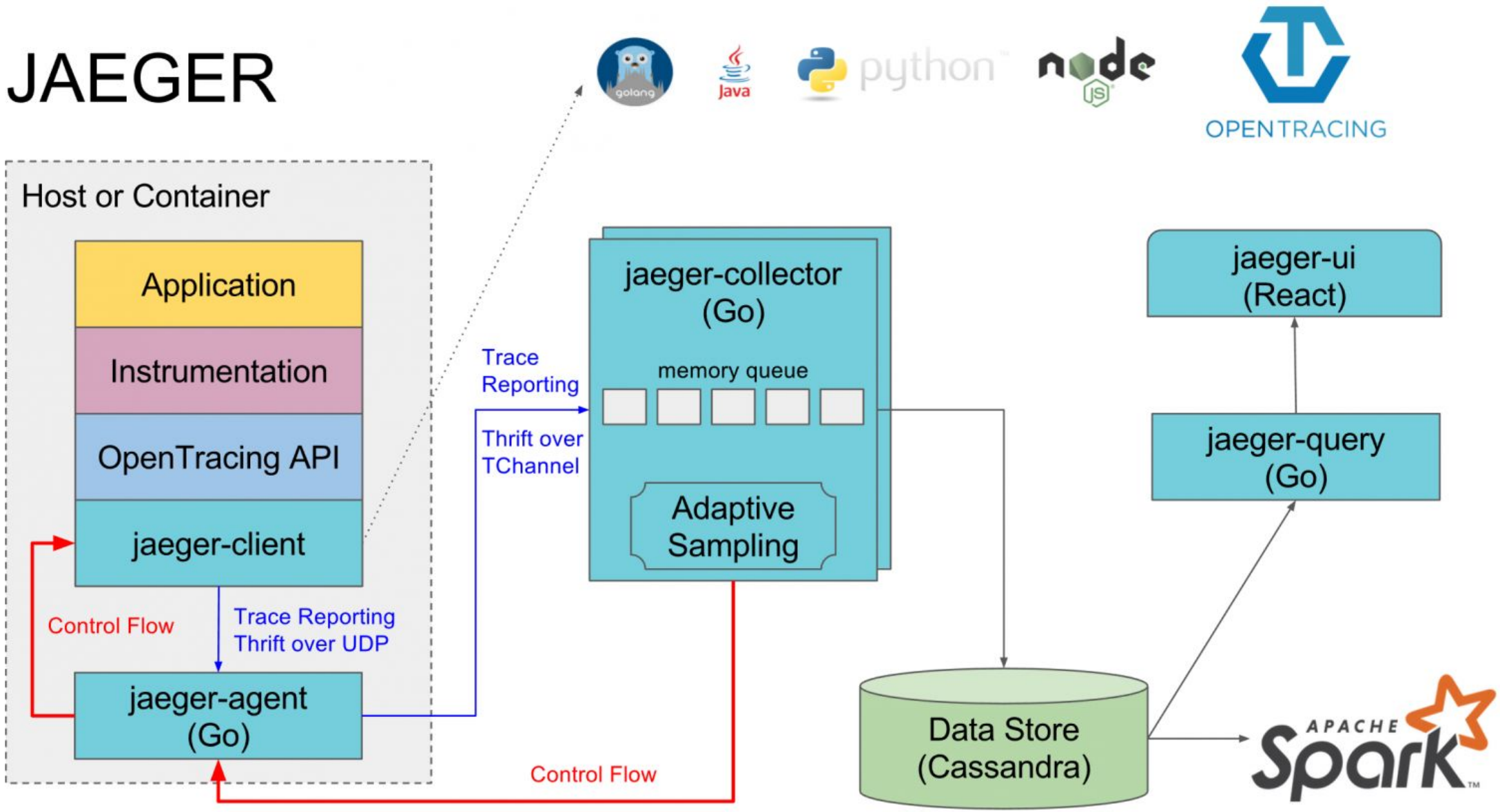
Много клиентских библиотек. Активно развивается, бежит в сторону поддержки OpenTracing. Пока w3c context trace не поддерживают, но поддерживает B3

Различные APM:

- Elastic APM, LightStep, DynaTrace, DataDog, Instana все бегут в сторону поддержки Opentracing-a и W3C Context Trace

- **OpenTelemetry** – дальнейшее развитие подхода OpenTracing с добавлением метрик Openmetrics протокола и логов

# Jaeger


JAEGER
golang
Java
python
node
OPENTRACING
Host or Container
Application
Instrumentation
OpenTracing API
jaeger-client
Control Flow
Trace Reporting
Thrift over UDP
jaeger-agent
(Go)
Trace
Reporting
Thrift over
TChannel
jaeger-collector
(Go)
memory queue
Adaptive
Sampling
Control Flow
jaeger-ui
(React)
jaeger-query
(Go)
Data Store
(Cassandra)
APACHE
Spark

# Demo

```
docker run -d --name jaeger -e COLLECTOR_ZIPKIN_HOST_PORT=:9411 -p
5775:5775/udp -p 6831:6831/udp -p 6832:6832/udp -p 5778:5778 -p 16686:16686 -p
14268:14268 -p 14250:14250 -p 9411:9411 jaegertracing/all-in-one:1.26
```

```
docker run -d --rm -it --link jaeger -p8080-8083:8080-8083 -e
JAEGER_AGENT_HOST="jaeger" jaegertracing/example-hotrod:1.26 all
```

# 02

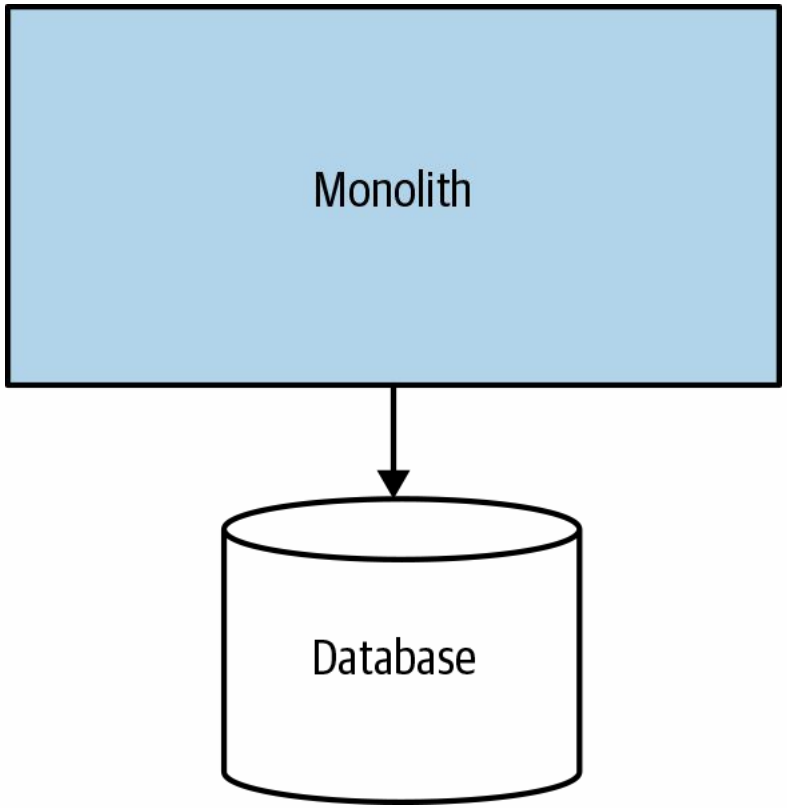
# Монолит в микросервисы

# Монолит в микросервисы

У нас есть монолит, как его распилить на микросервисы?

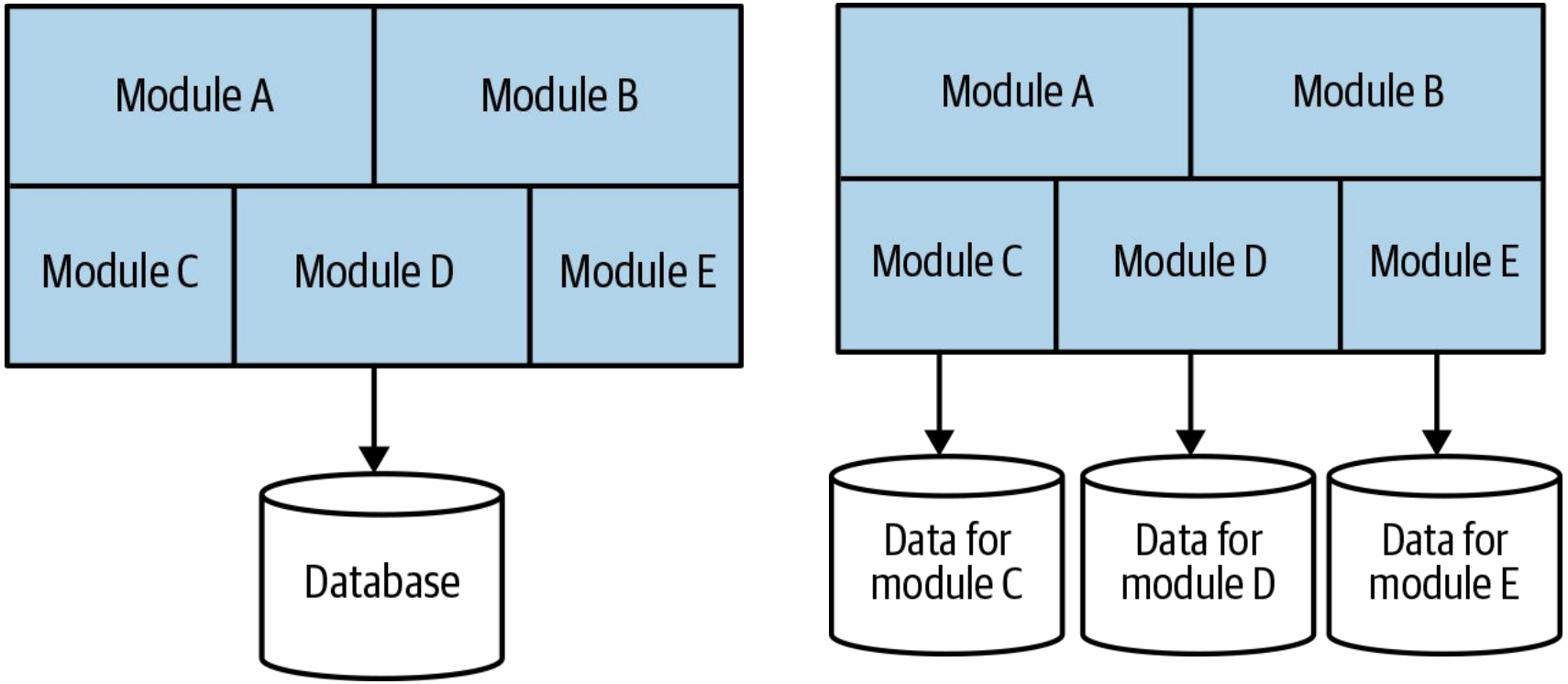
# Единый монолит

**Единый монолит** – монолит, который деплоится как единое целое, содержит в себе всю функциональность



# Модульный монолит

**Модульный монолит** – монолит, который деплоится как единое целое, содержит в себе всю функциональность, но у которого есть возможность деплоить только часть функциональности (некоторый набор модулей).



# Распределенный монолит

**Распределенный монолит** – монолит, который состоит из нескольких сервисов, но все-равно должен деплоится как единое целое, содержит в себе всю функциональность.

# Зачем?

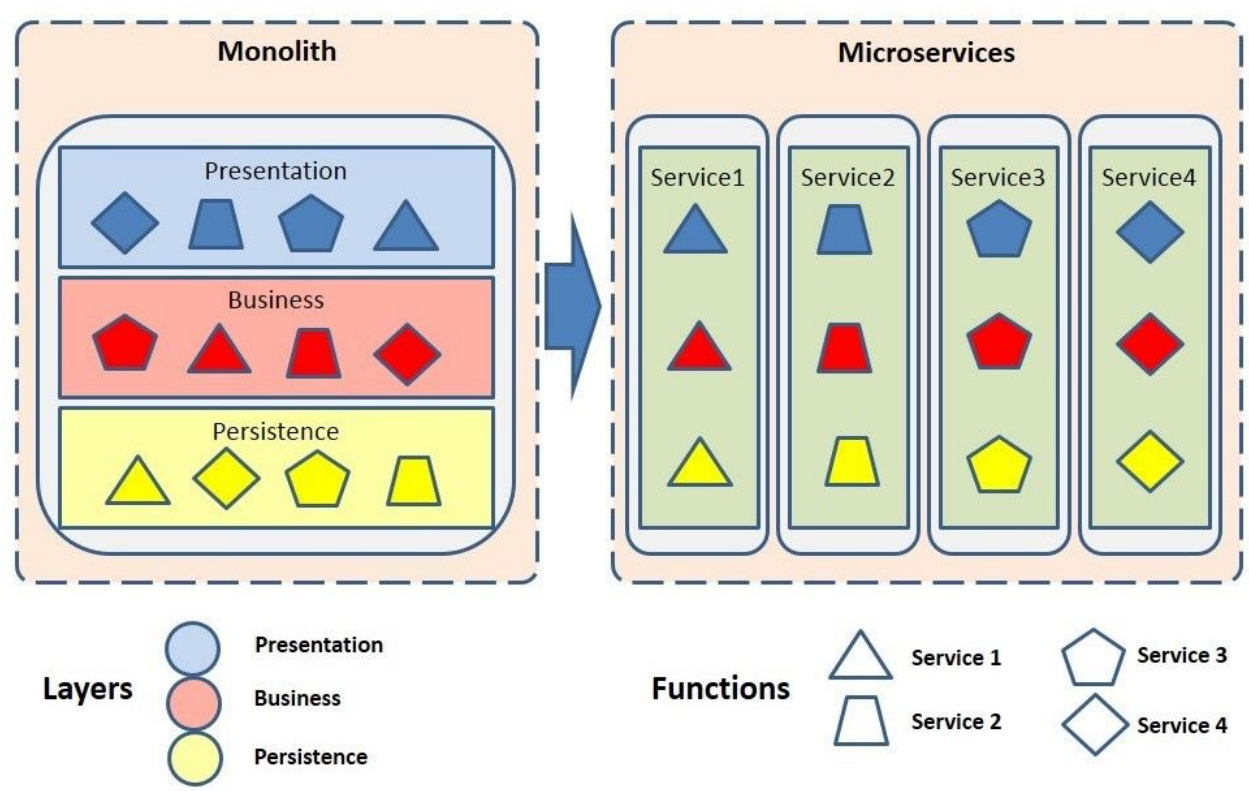
Прежде чем распиливать «монолит» в микросервисы нужно четкое понимание того, какие плюсы принесет микросервисная архитектура и сколько нужно будет потратить человеко-месяцев и лет, для того, чтобы перейти на новую архитектуру.

Если сделан осознанный выбор в сторону микросервисов, то нужно приготовится к долгому и планомерному переходу.

*Напишите в чат*

- **Медленная доставка** - сложно разобраться, сложно менять, сложно и долго тестировать
- **Обновления с ошибками** - из-за плохой тестируемости она недостаточна, из-за сложности понимания и изменений много ошибок
- **Плохая масштабируемость** - модули с разными требованиями к ресурсам

# Цель

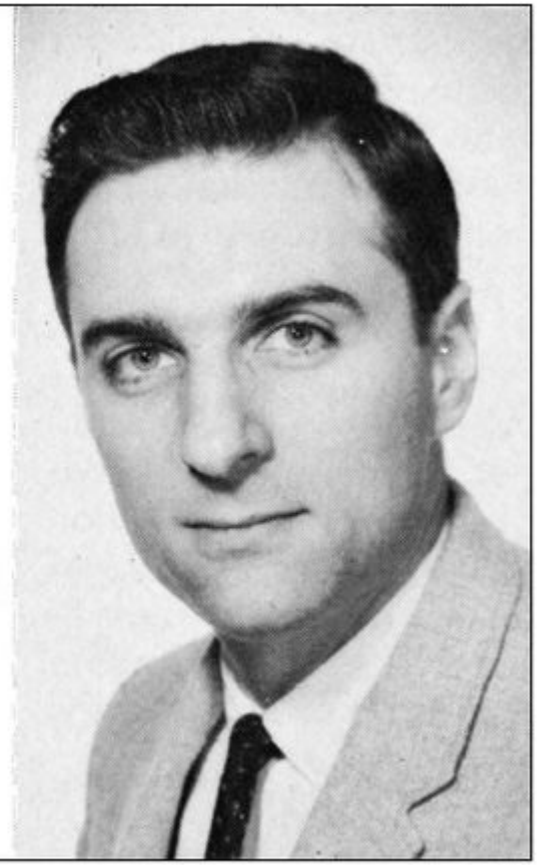



*С чего начать? Напишите в чат*

# Закон Конвея

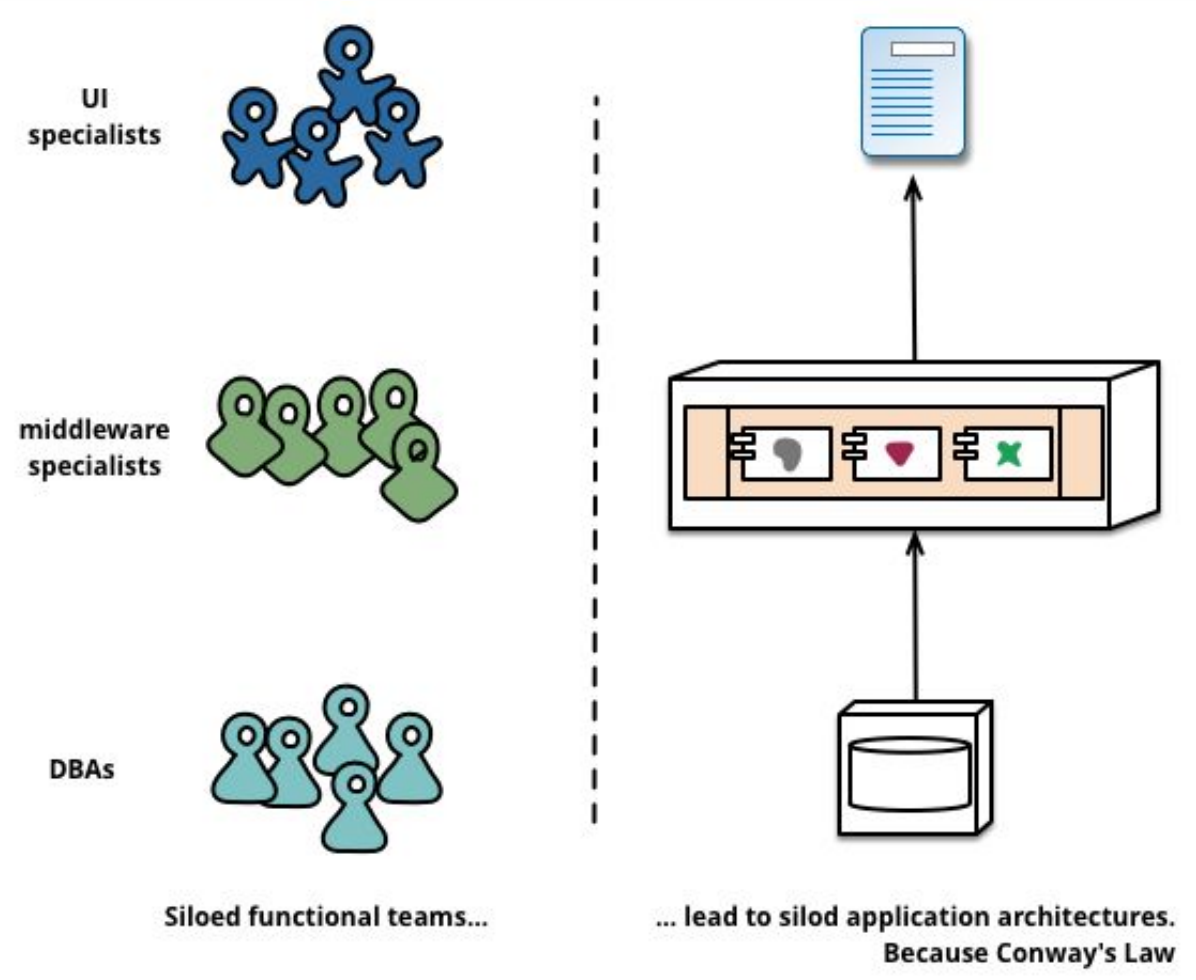
«Организации, проектирующие системы, ограничены дизайном, который копирует структуру коммуникации в этой организации»

*Мелвин Конвей*



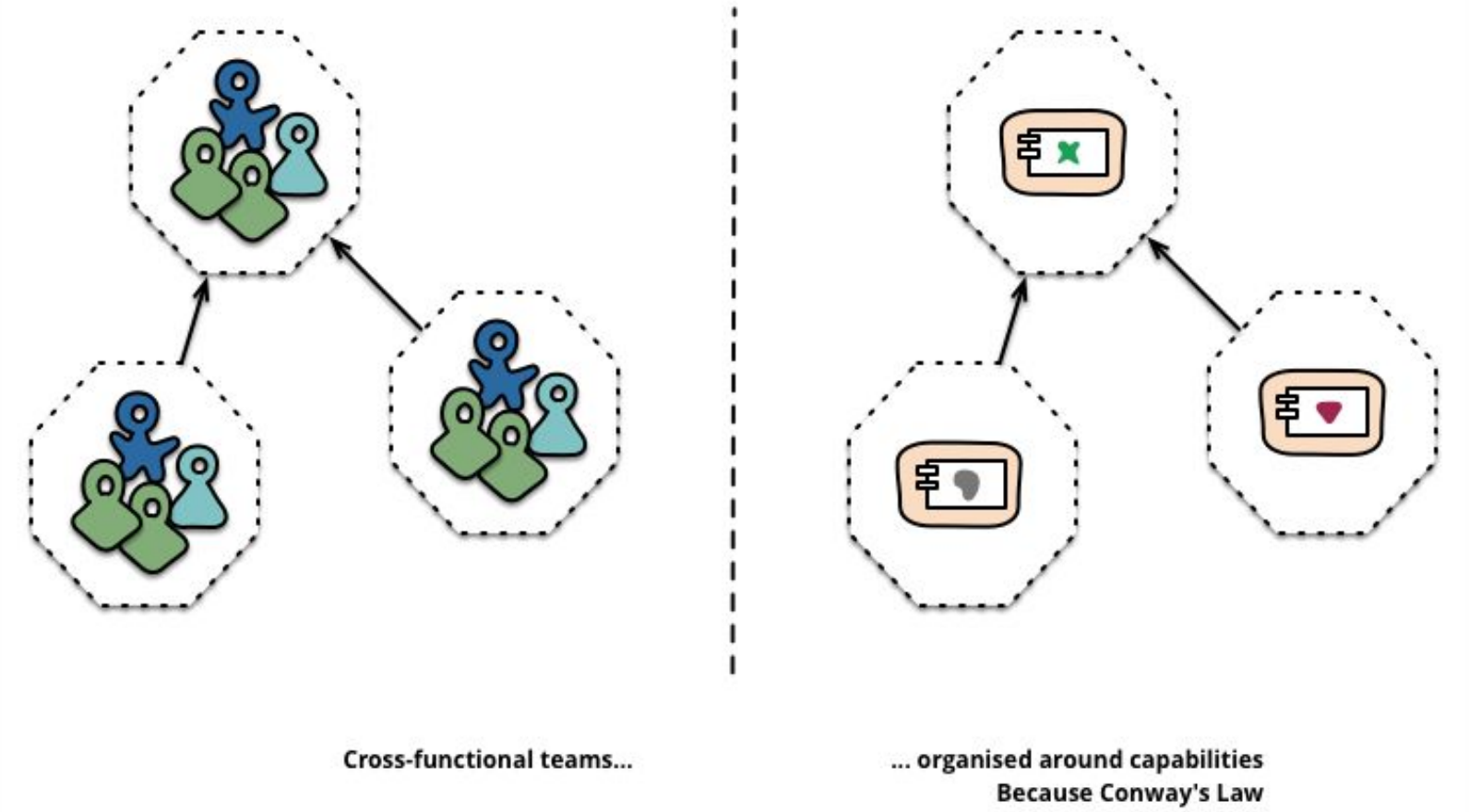
# Организационная структура при монолитах

В случае монолитов мы скорее всего будем иметь несколько «специализированных» по функциям команд – DBA, Backend, Frontend и т.д.



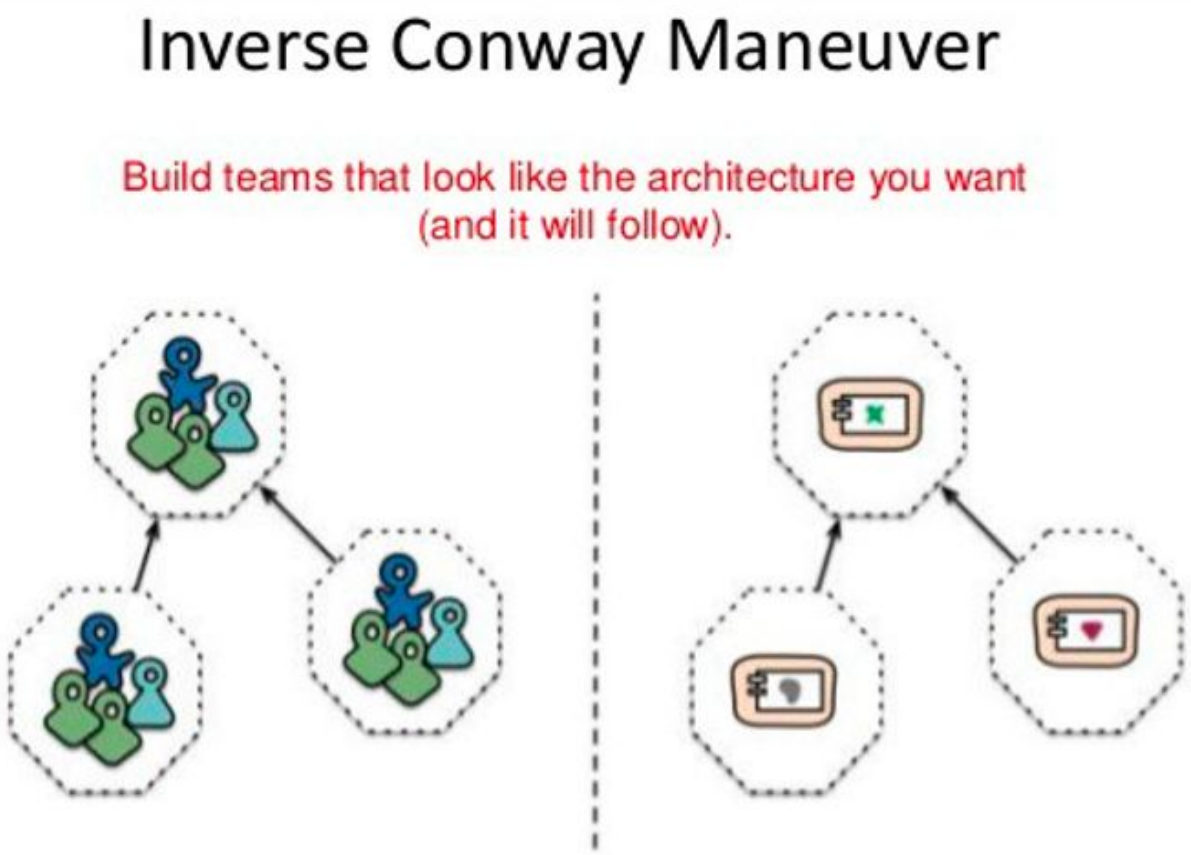
# Организационная структура при микросервисах

Микросервисы – это чаще всего самостоятельно деплоятся и границы микросервисов проходят по business capabilities (продукту). Поэтому чаще всего для микросервисной архитектуры характерны кросс-функциональные продуктовые команды.



# Начните с орг дизайна

Измените организационную структуру и архитектура сама постепенно станет повторять организационный дизайн.

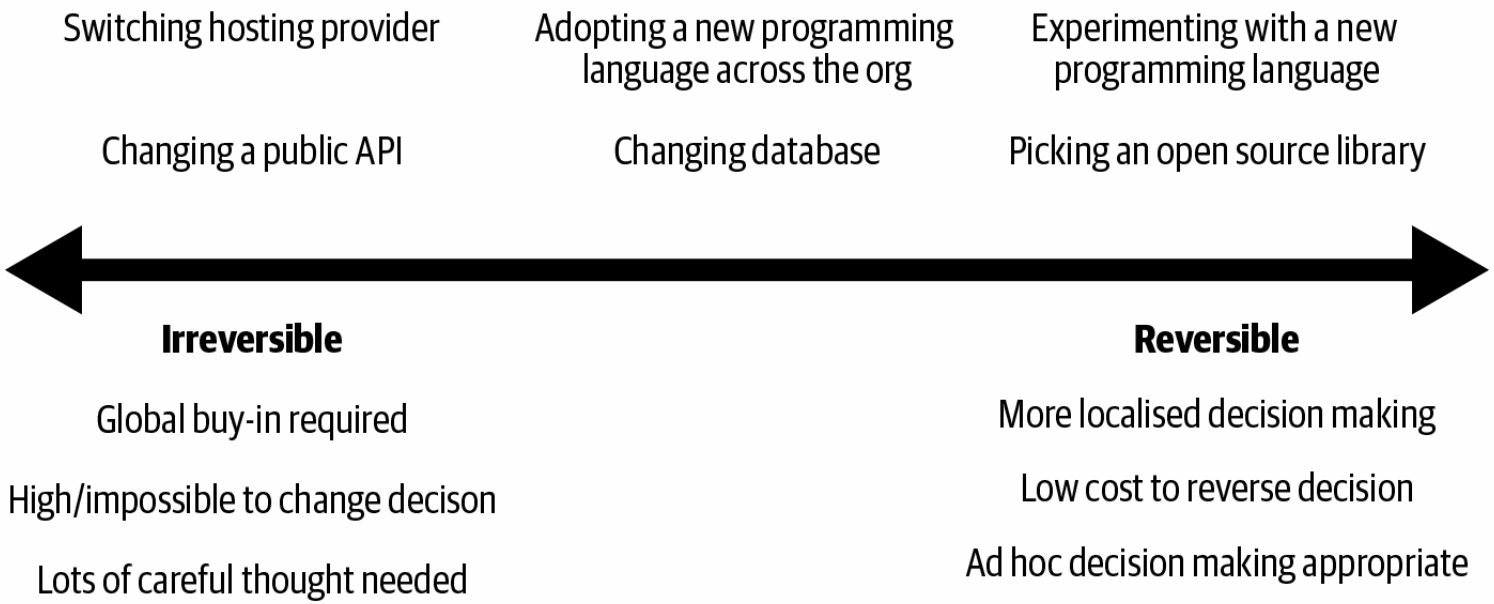


# Подходы

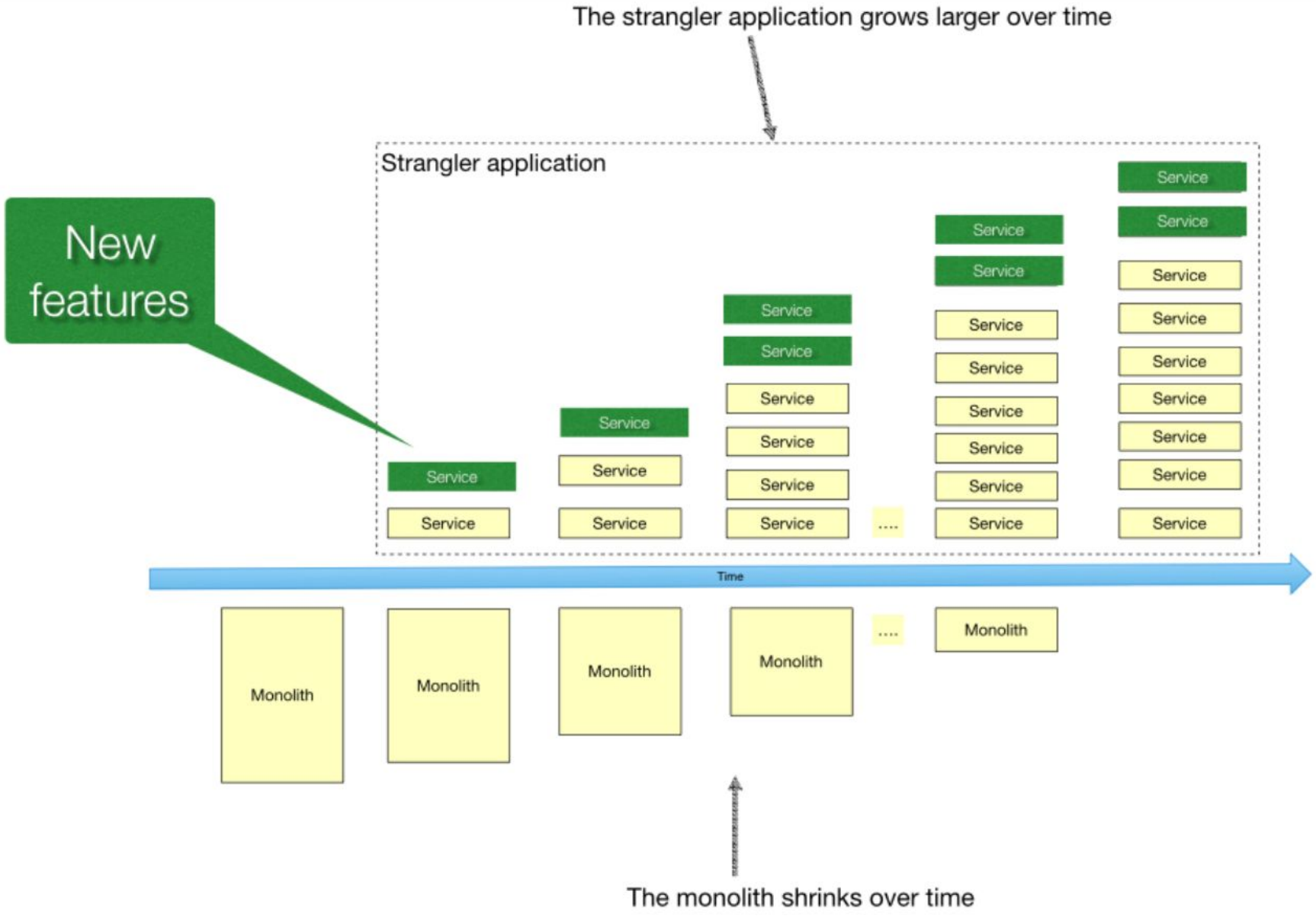
- Если вы в яме - перестаньте копать
- Переписать все
- Переписывать постепенно

# Стоимость изменений

Как и любое архитектурное решение, переход от монолита к микросервисам должен происходить постепенно. Решения должны быть как можно менее рискованным (возвратными). При этом сам переход займет больше времени, но будет более предсказуемым и вероятность успеха будет больше

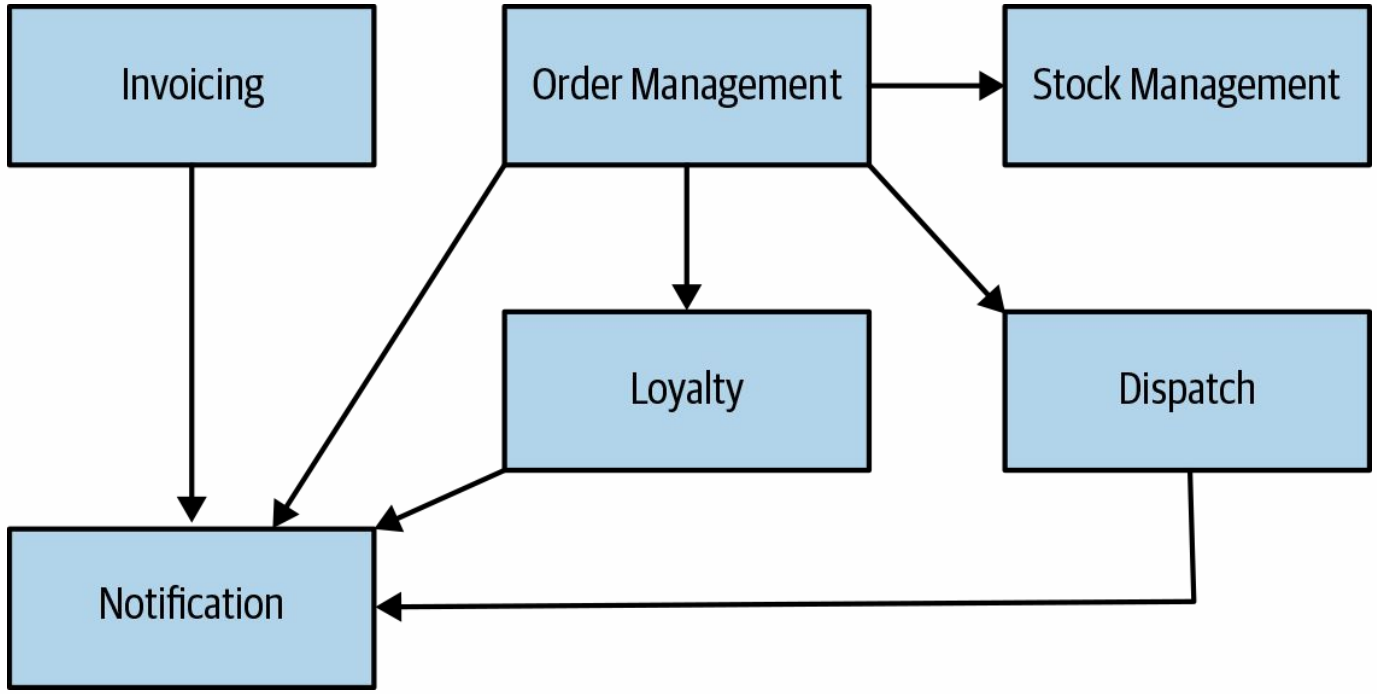


# Strangler pattern



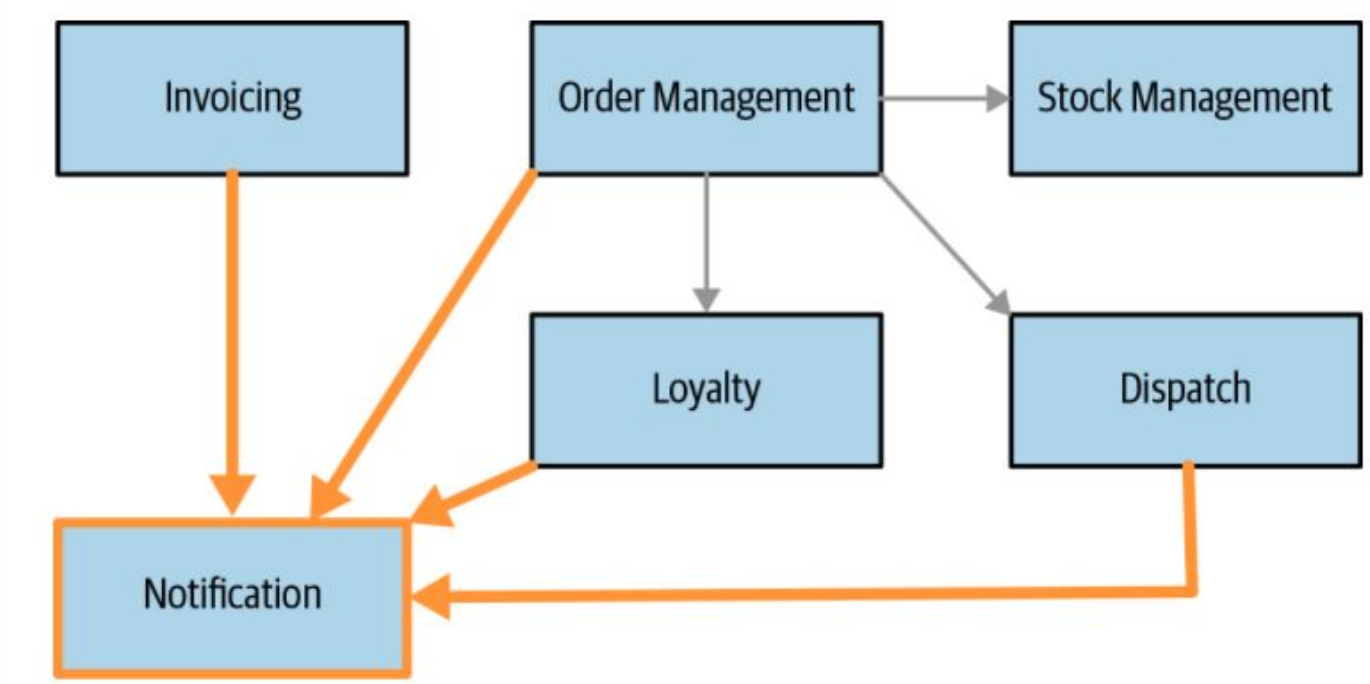
# Какие сервисы вытаскивать в первую очередь?

Предположим, что в результате практик декомпозиции сервисов мы приняли решение разделить сервисы следующим образом.



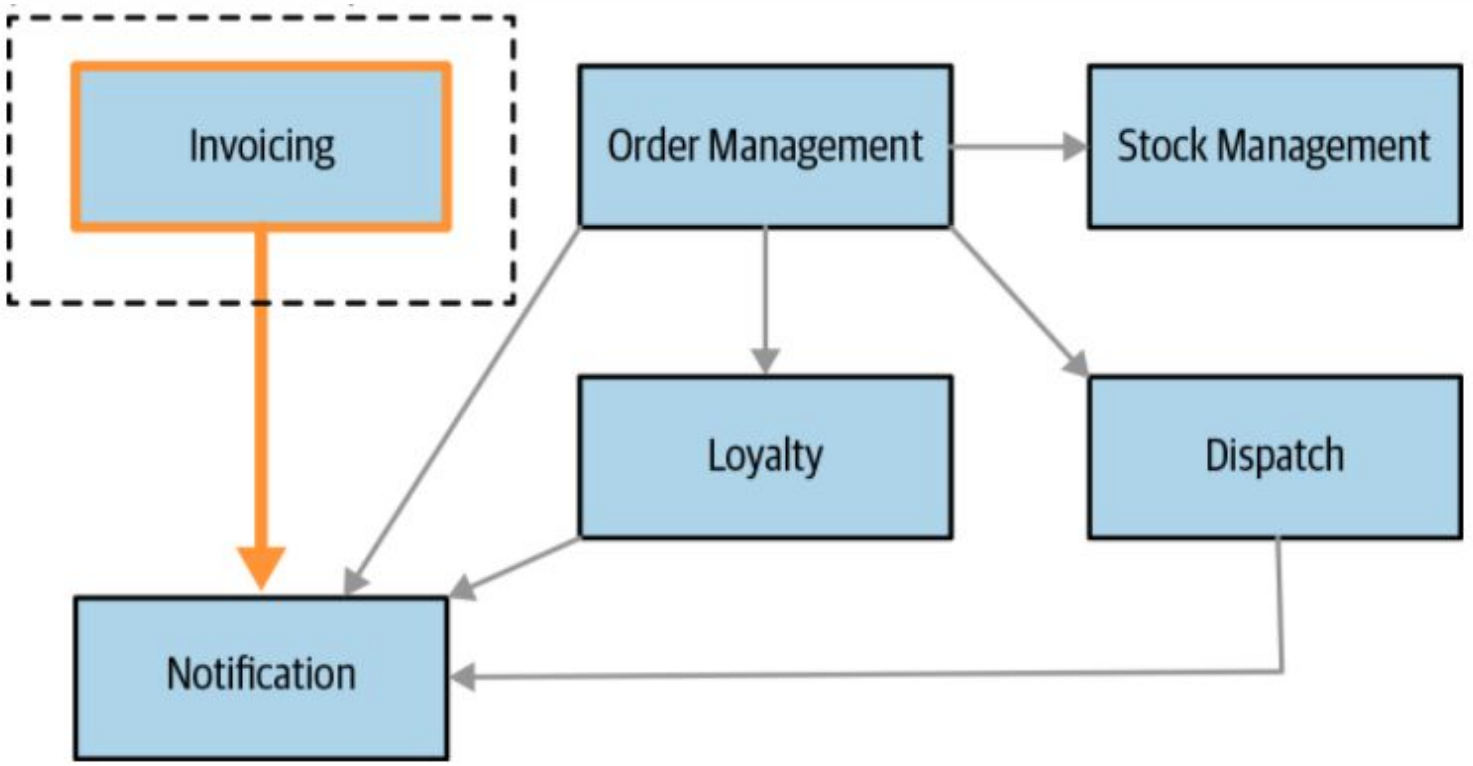
# Какие сервисы вытаскивать в первую очередь?

Чем больше зависимостей требует будущий сервис, тем больше придется переписывать и тем сложнее будет миграция.



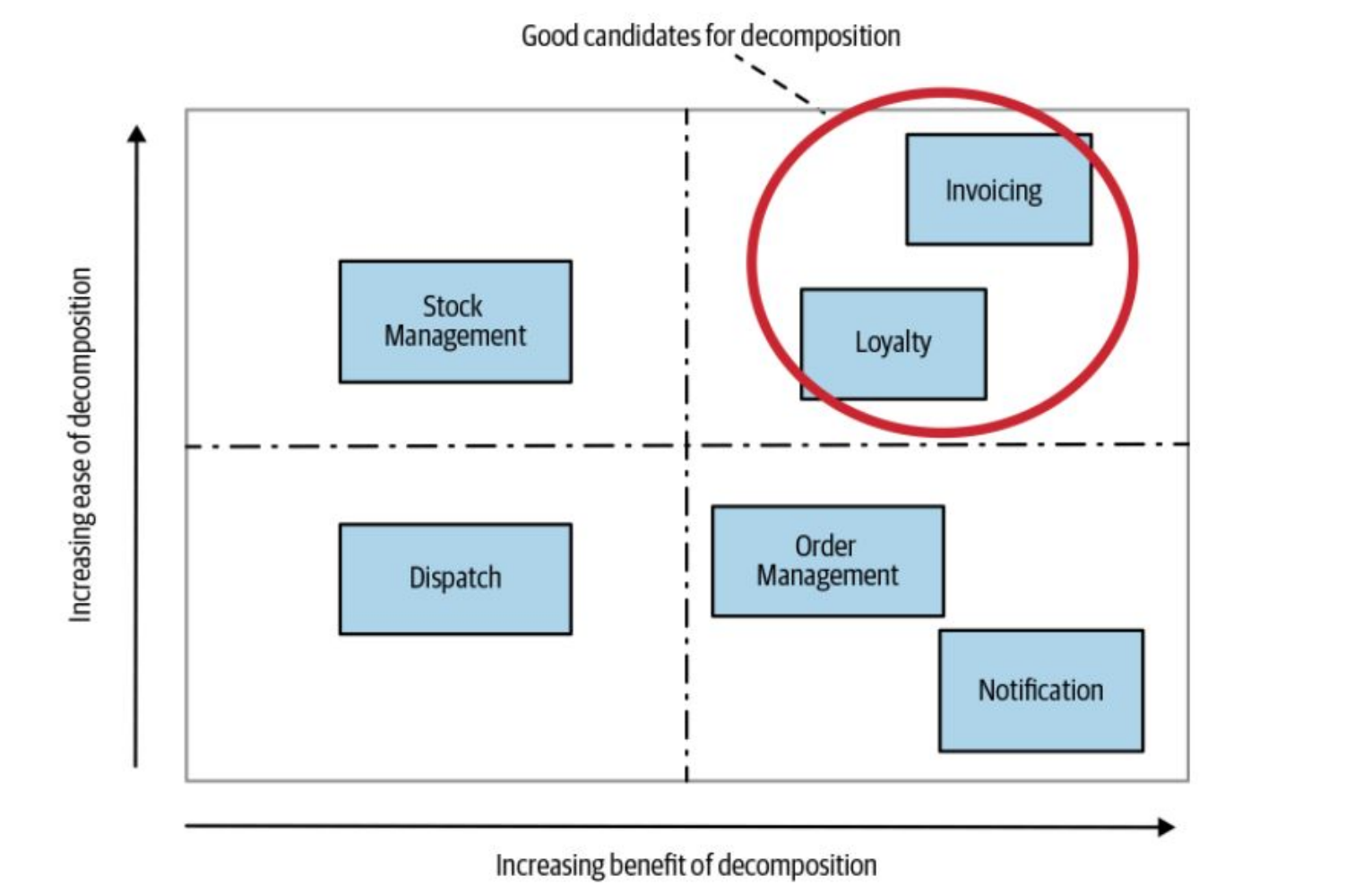
# Какие сервисы вытаскивать в первую очередь?

Чем меньше зависимостей требует будущий сервис, тем меньше придется переписывать и тем проще будет миграция.



# Какие сервисы вытаскивать в первую очередь?

Лучше вытаскивать те сервисы, которые принесут больше всего профита и которые легче всего вытащить .

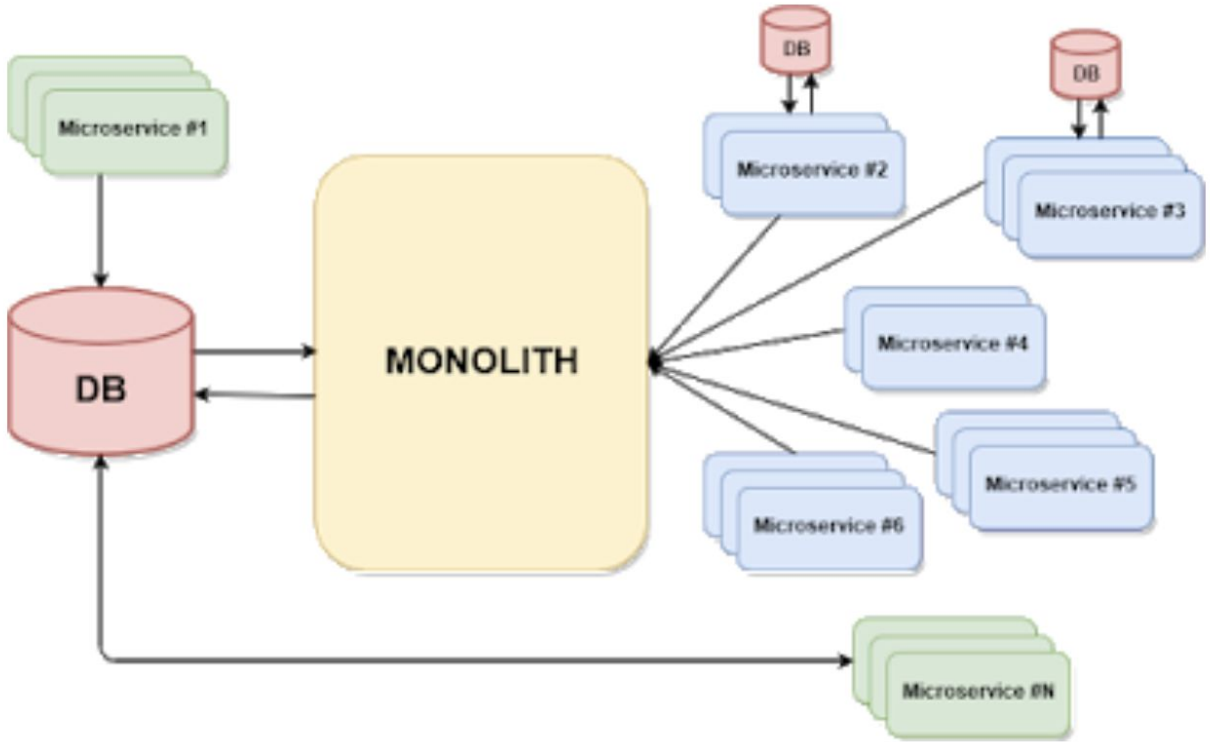


# Какие сервисы вытаскивать в первую очередь?

- **Ускорение разработки**. Если согласно плану какая-то часть вашего приложения будет активно развиваться на протяжении следующего года, разработку можно ускорить путем извлечения ее в сервис.
- **Решение проблем с производительностью, масштабируемостью и надежностью**. Если определенная часть вашего приложения ненадежна или имеет проблемы с производительностью или масштабируемостью, будет полезно преобразовать ее в сервис.
- **Возможность извлечь какие-то другие сервисы**. Иногда из-за зависимостей между модулями извлечение одного сервиса упрощает извлечение другого.

# Микромонолит

Типичной ошибкой является вытаскивать те сервисы, которые проще всего вытащить, но которые не приносят существенных плюсов. Тогда получается микромонолит – монолит, в котором находится основная бизнес логика + набор (чаще всего инфраструктурных) микросервисов, которые не часто меняются.



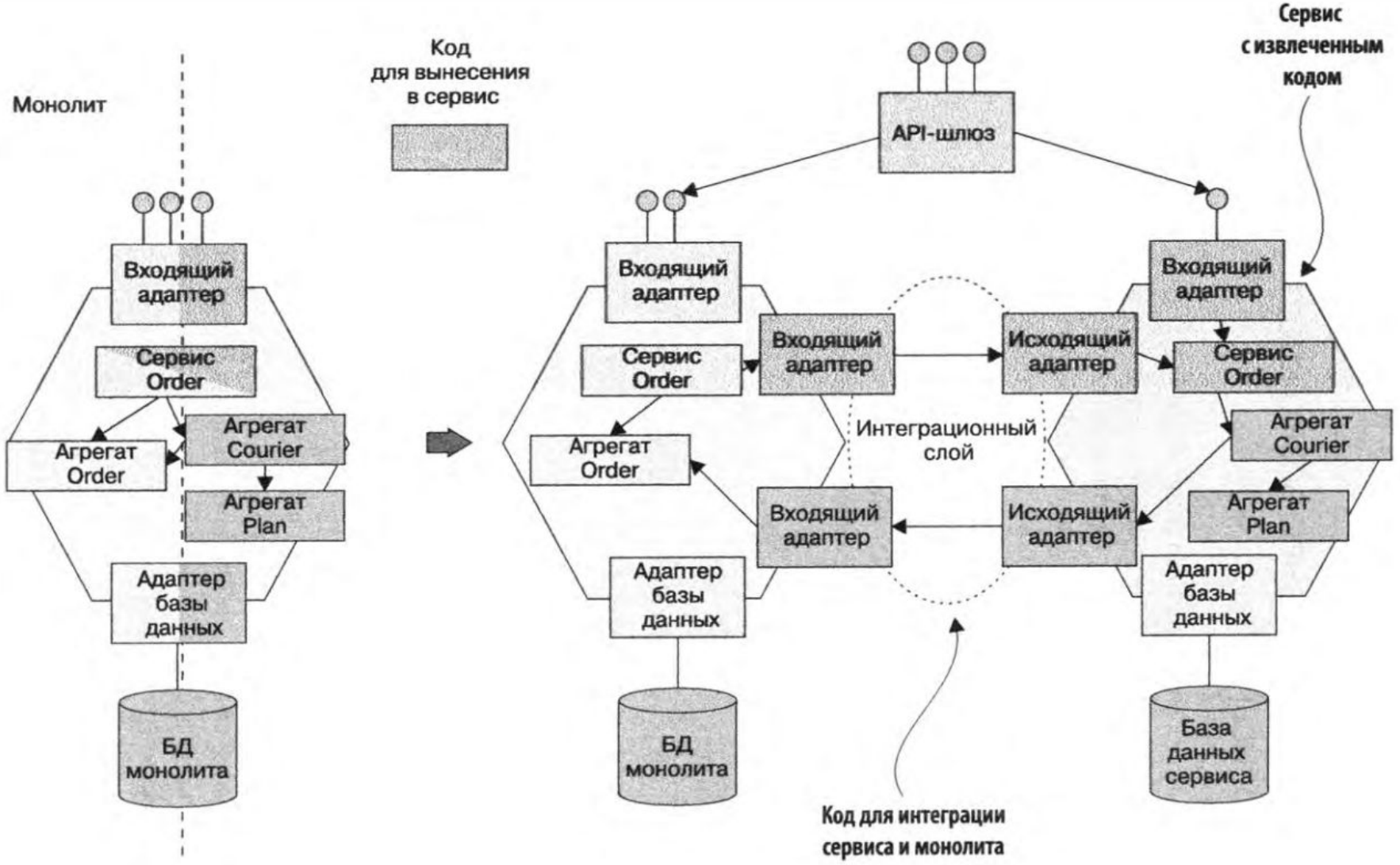
# Извлечение функционала в сервис - транзакции

1. Проверить детали заказа
2. Клиент может размещать заказ?
3. Авторизовать карту
4. Создать заказ

1. В монолите
    a. Клиент может размещать заказ?
    b. Создать заказ в состоянии PENDING
2. В сервисе
    a. Проверить детали
    b. Создать заявку в статусе PENDING
3. В монолите
    a. Авторизовать карту
    b. Изменить статус на APPROVED
4. В сервисе - изменить статус на APPROVED

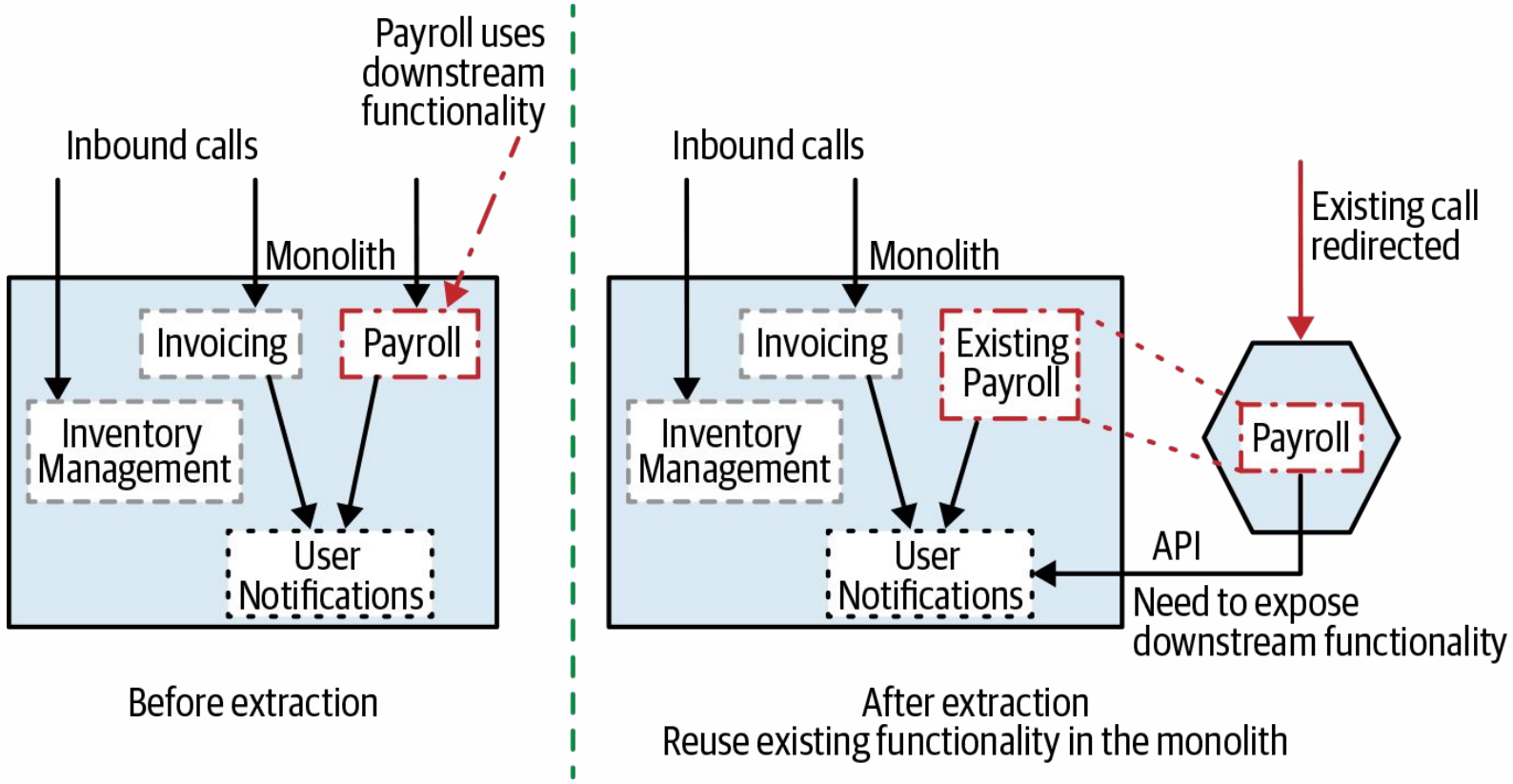
# Извлечение функционала в сервис

- Создаем новый микросервис, который повторяет функциональность из монолита
- Переводит внешнием запросы на запросы из монолита в микросервис



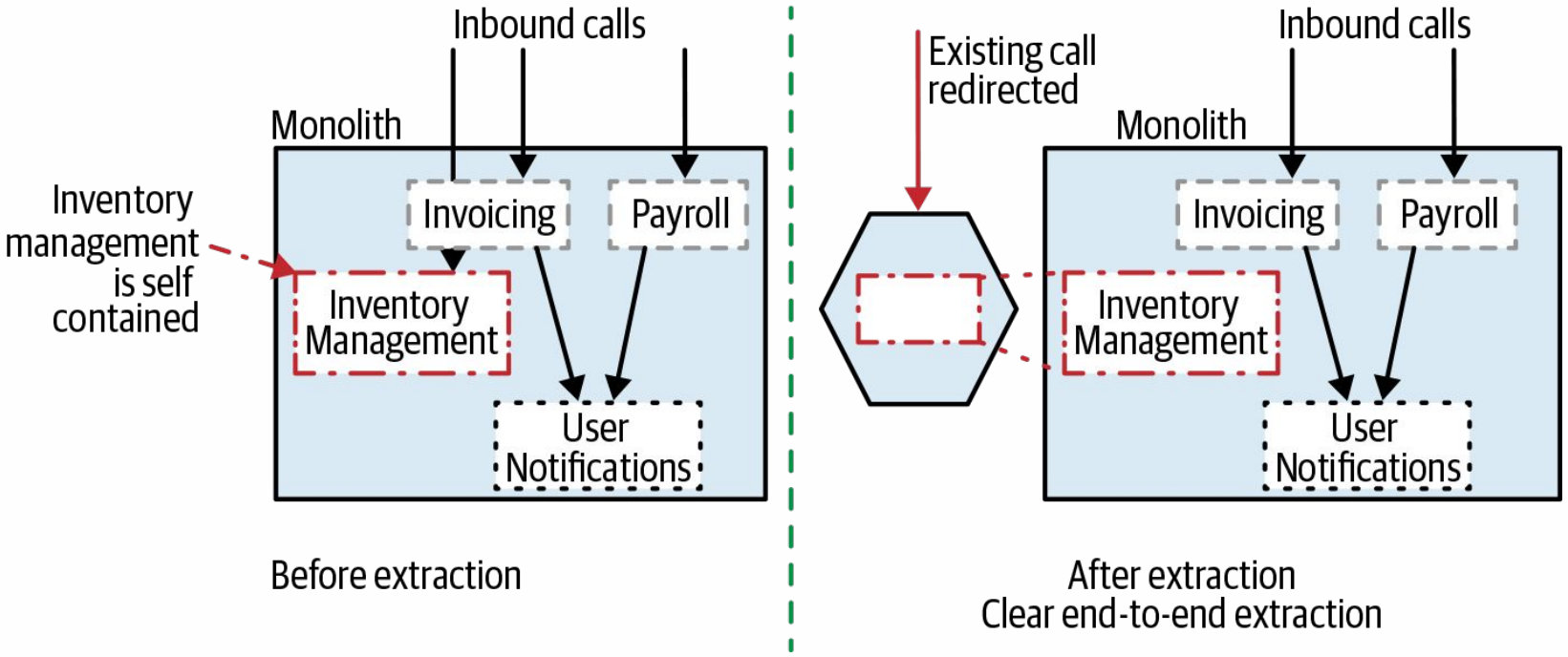
# Strangler pattern

Иногда хочется вытащить сервис, который имеет зависимости внутри монолита. В этом случае придется делать интеграции, которые потом придется рефакторить



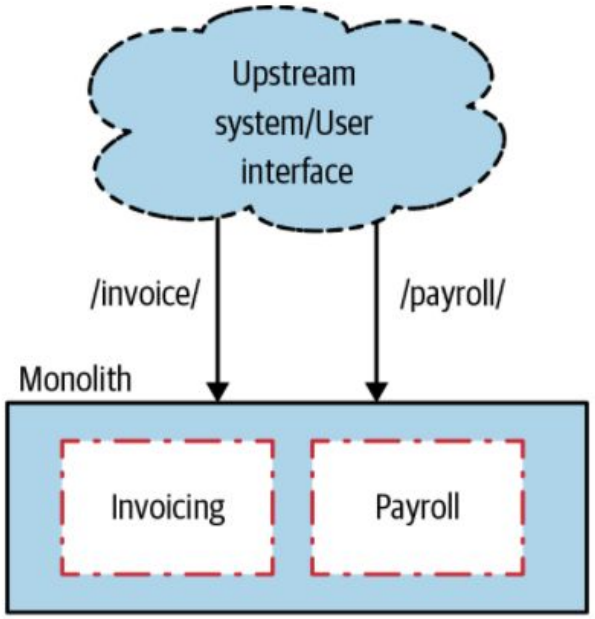
# Strangler pattern

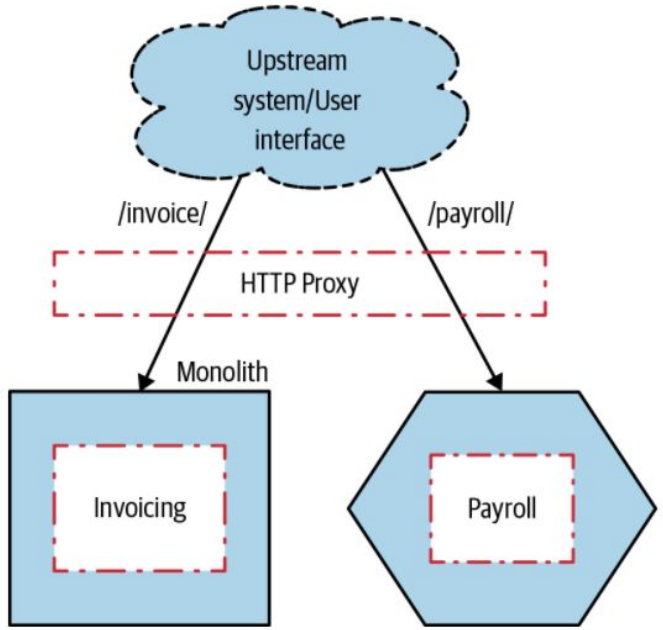
Таким образом можно вытаскивать сервисы, которые имеют не очень много зависимостей и представляют собой «цельный» сервис.



# Strangler pattern

Для того, чтобы управлять трафиком, скорее всего придется настраивать балансер (прокси), либо его в явном виде добавлять



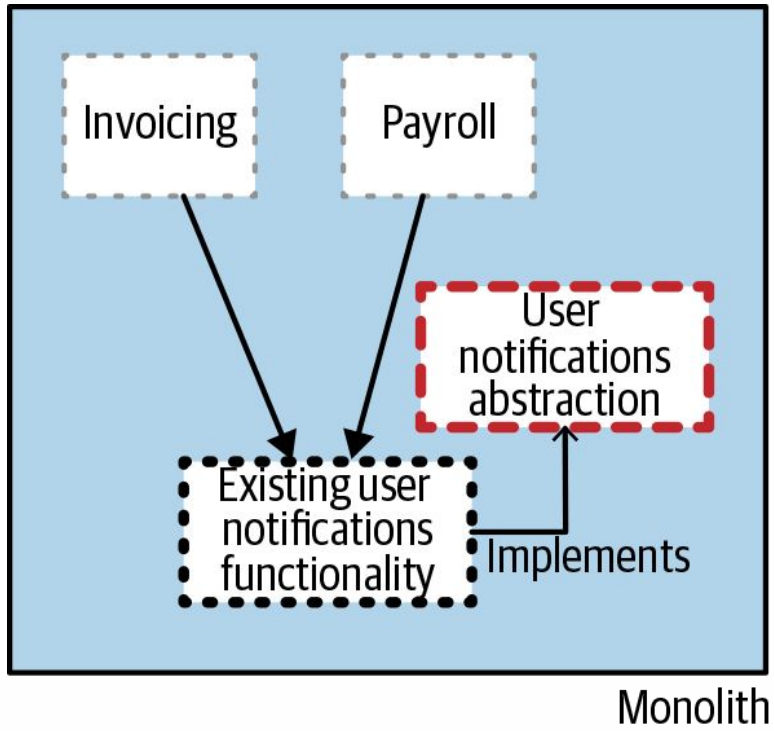



# Branch by abstraction

В случае, если мы хотим декомпозировать сервис, который находится “внутри” и внешний трафик до которого не доходит. Можно воспользоваться другим паттерном - Branch by abstraction

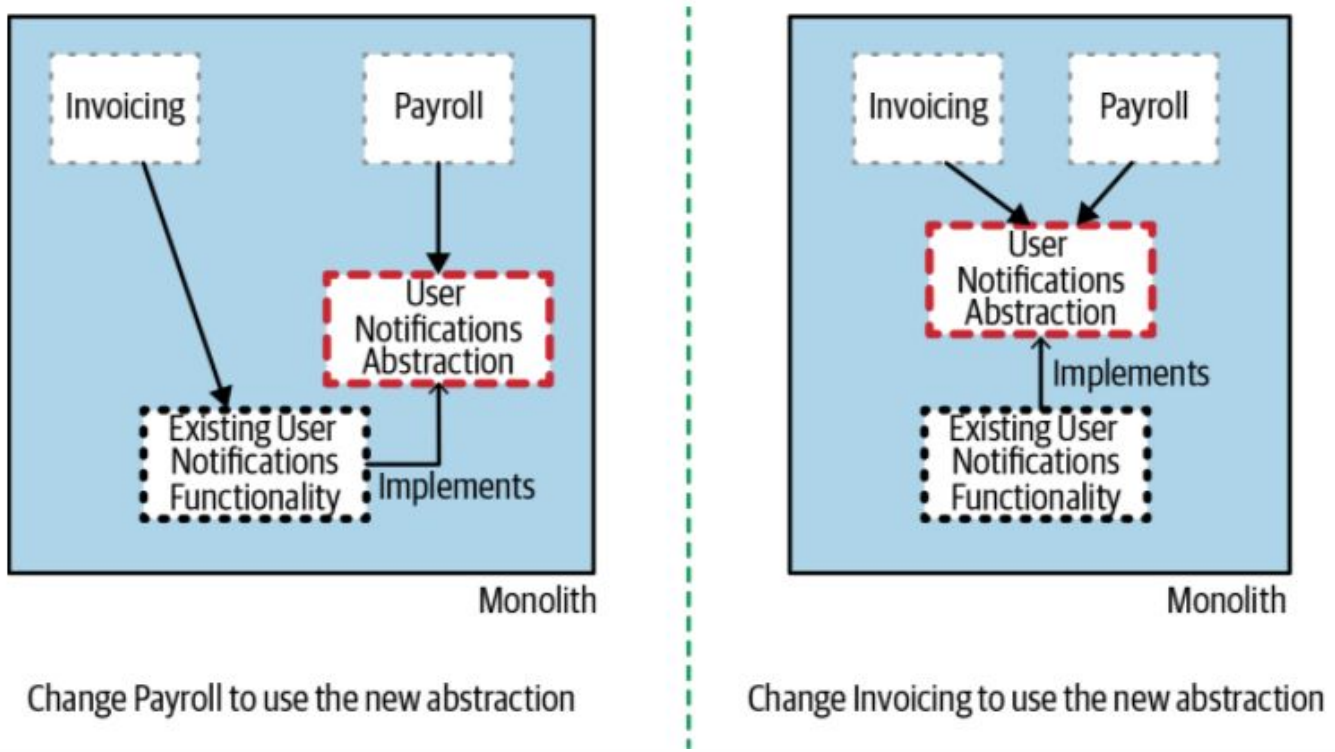
# Branch by abstraction

- Выделяем абстракцию (интерфейс).



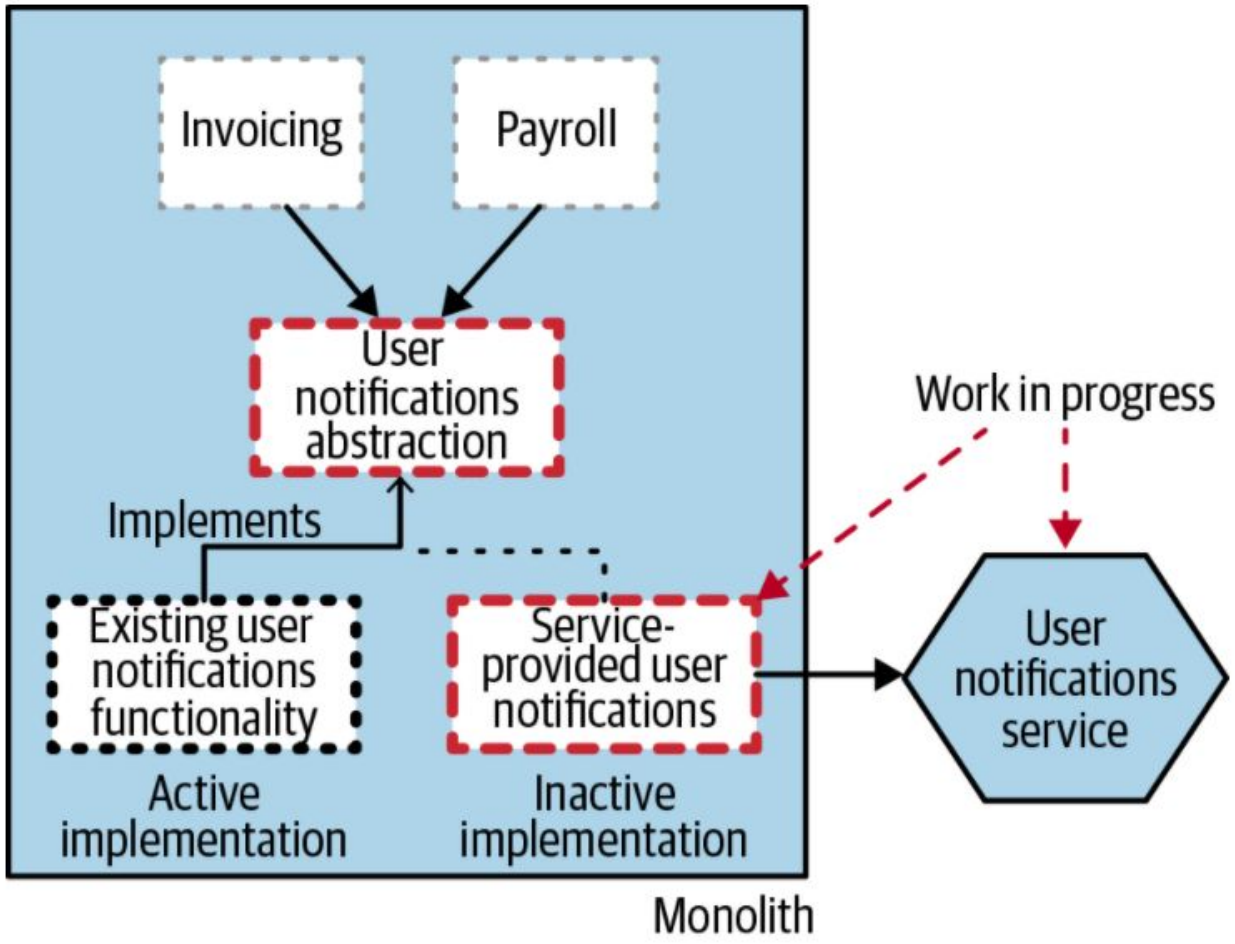
# Branch by abstraction

- Используем «абстрактный» интерфейс во всех местах.



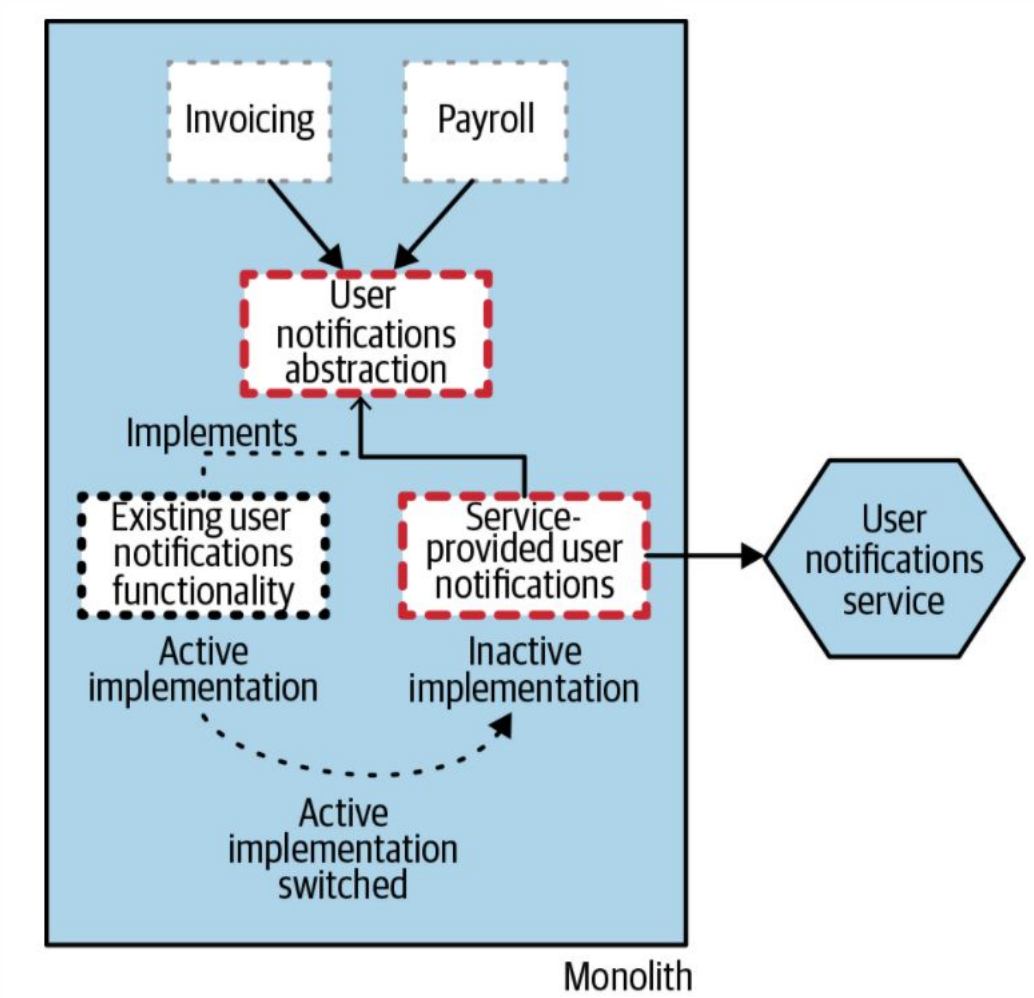
# Branch by abstraction

- Создаем сервисную реализацию интерфейса



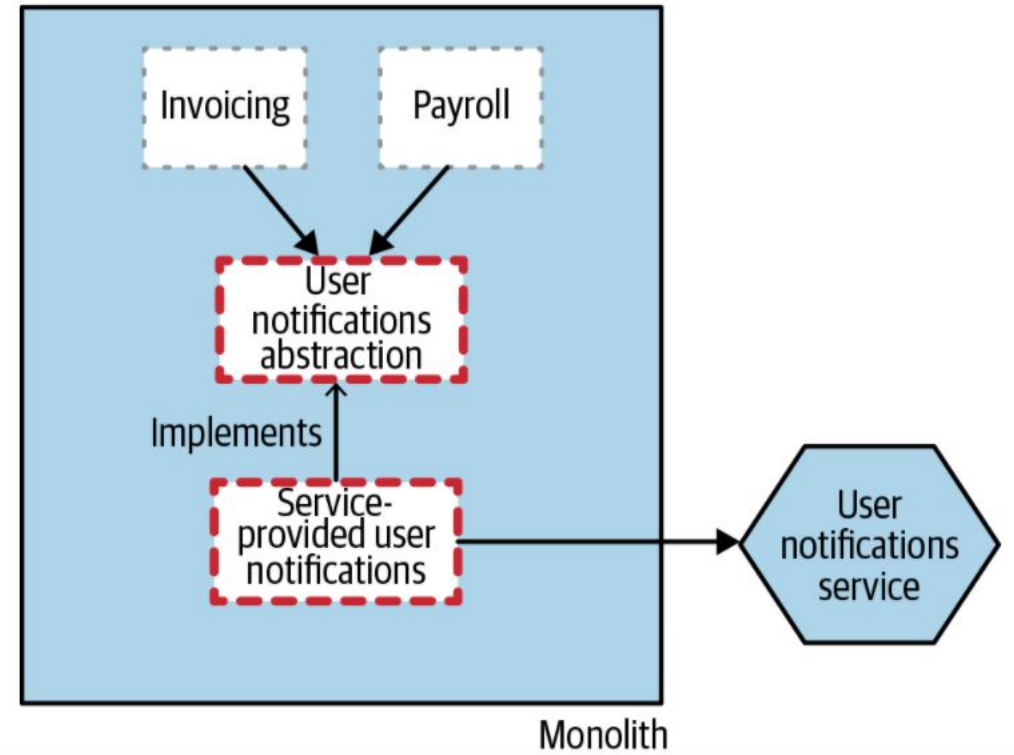
# Branch by abstraction

- Переключаем на сервисную реализацию

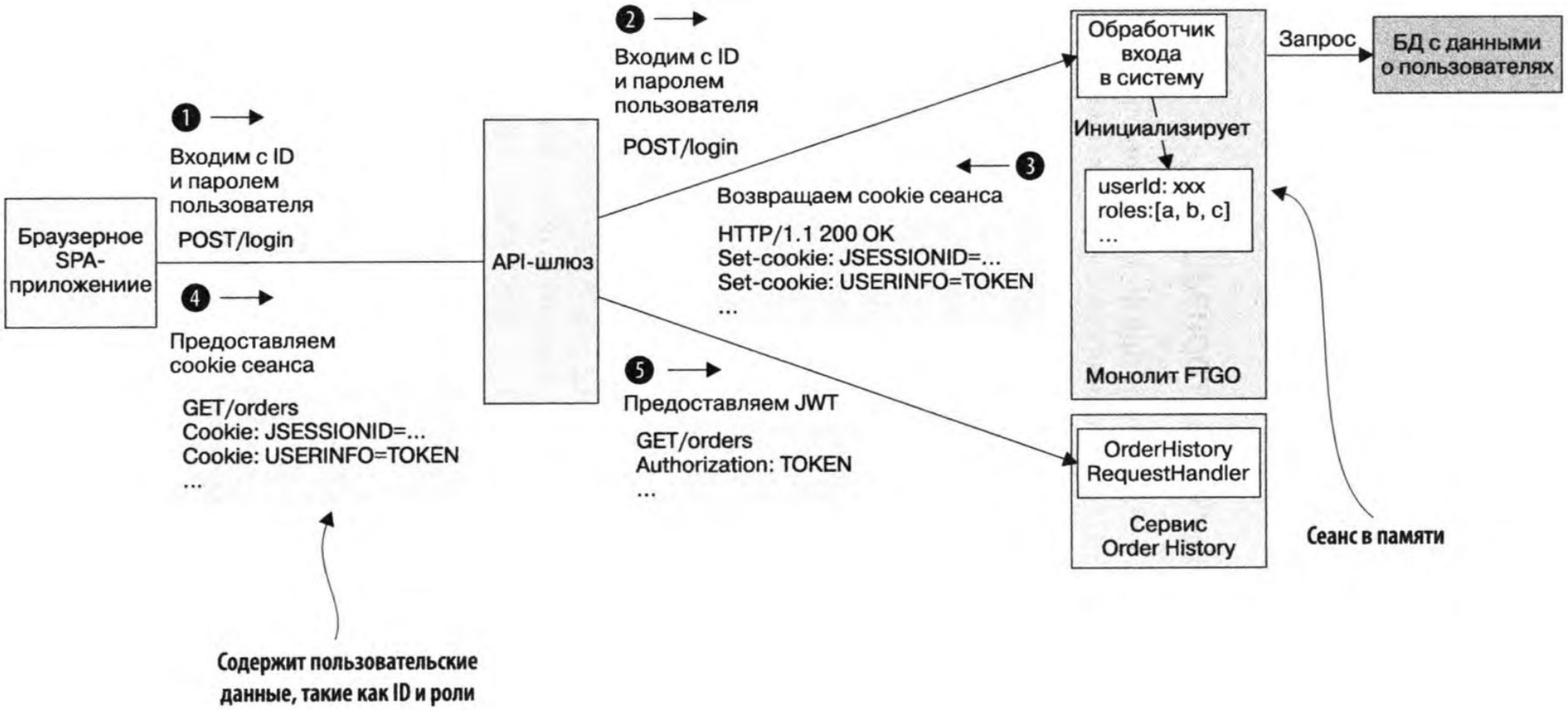


# Branch by abstraction

- Очищаем код от старой реализации



# Аутентификация и авторизация



# 03

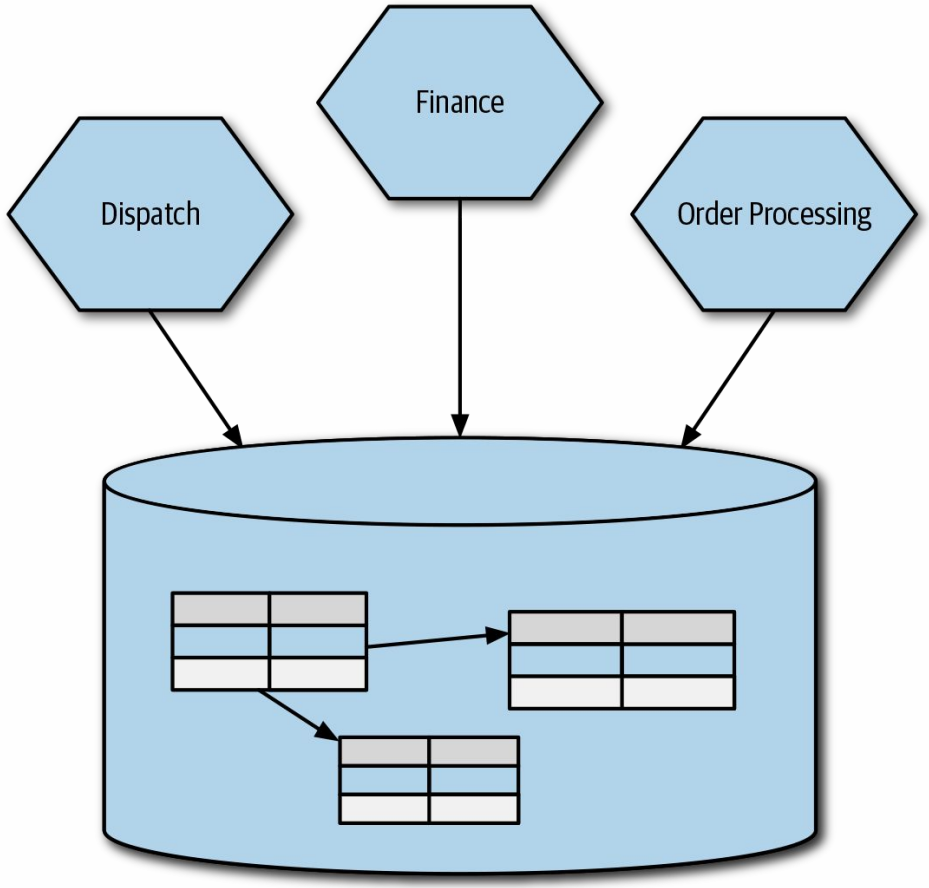
# Переход от БД монолита

# Декомпозиция БД

- В монолите БД (и схемы) одна. При выделении сервиса мы также должны решить, как мы будем вытаскивать данные из монолита.

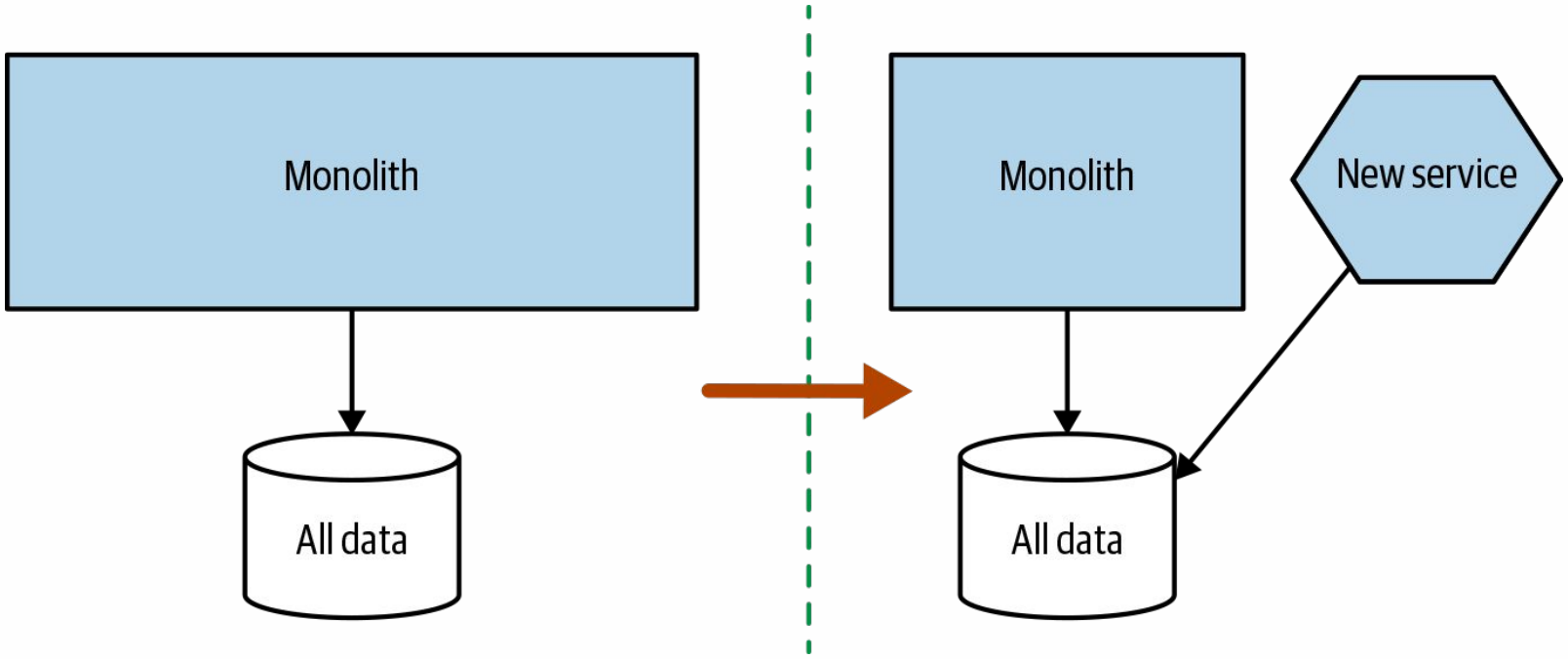
# Одна БД для всех сервисов

- В принципе мы можем оставить как есть и оставить одну БД для всех сервисов.



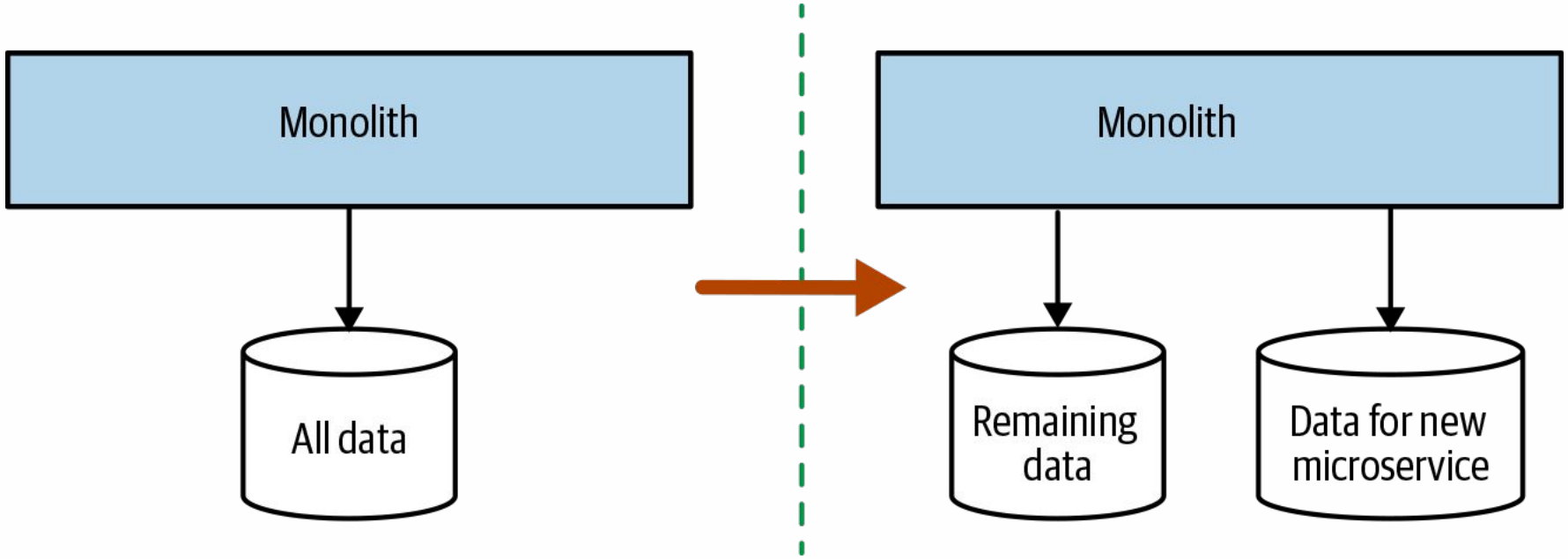
# Выделяем сервис, потом  данные

Сначала выделяем сервис, потом отделяем данные.
Основная проблема в том, что шаг 2 часто не делают и остаются с общей БД.

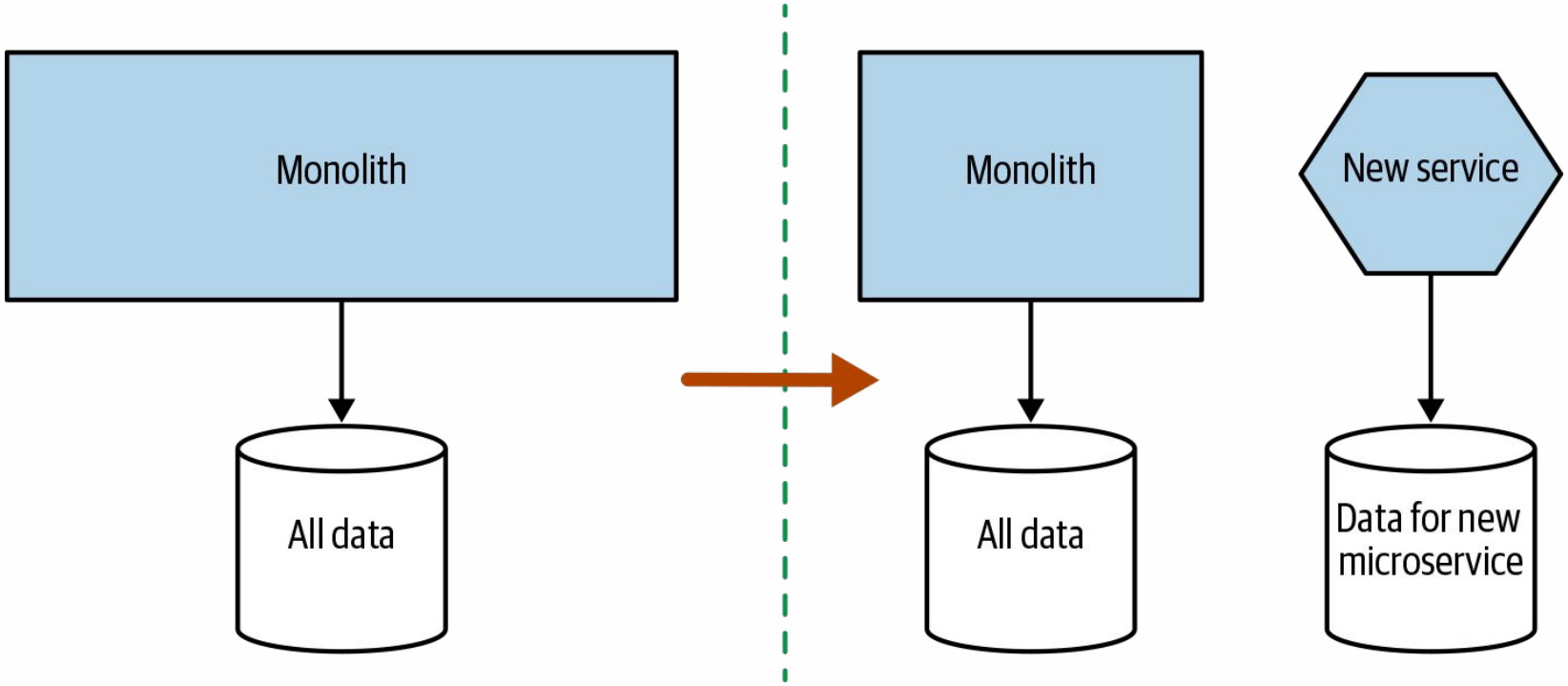


# Выделяем данные, потом сервис

Сначала отделяем данные, потом выделяем сервис.

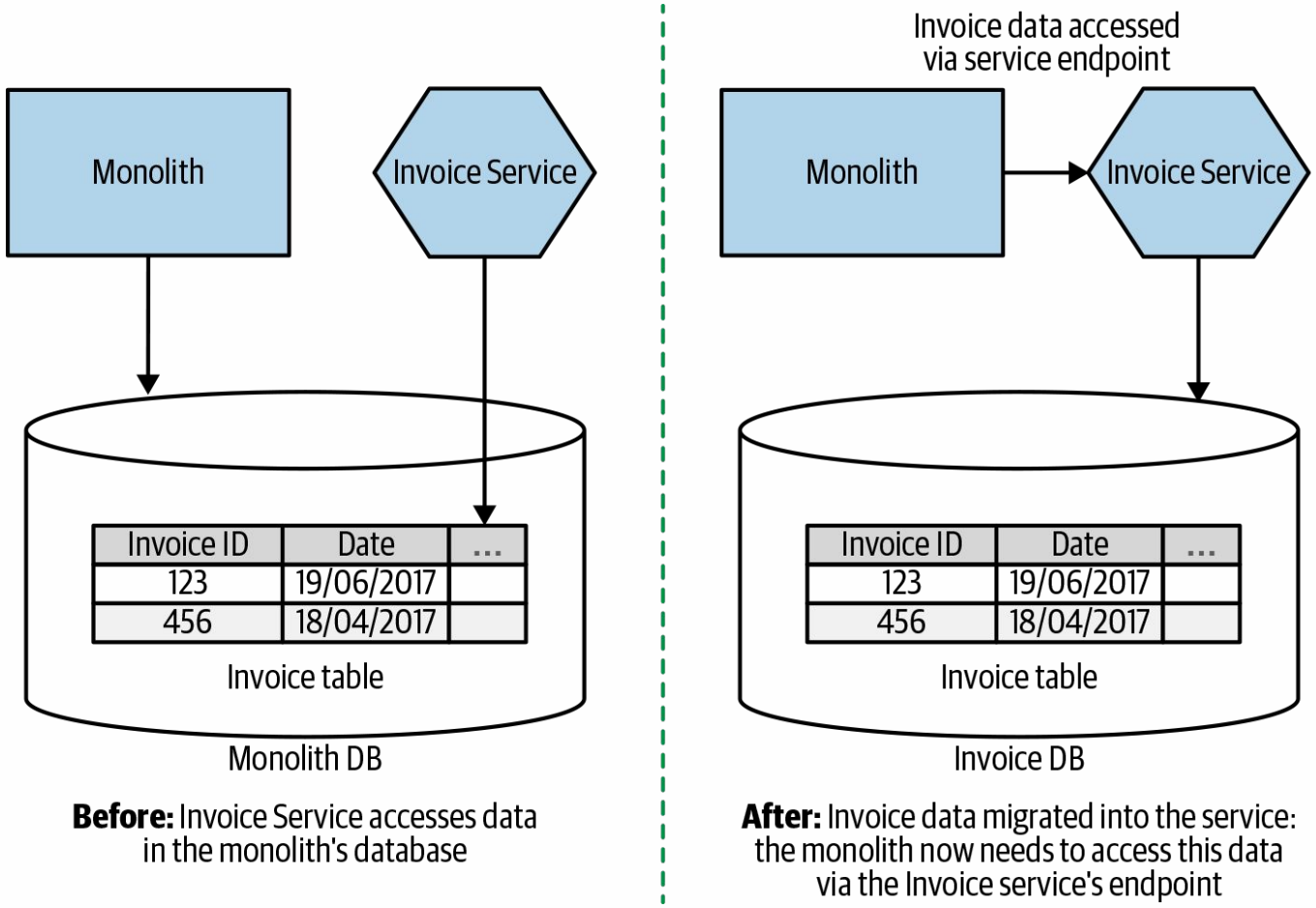


# Выделяем данные и сервис одновременно



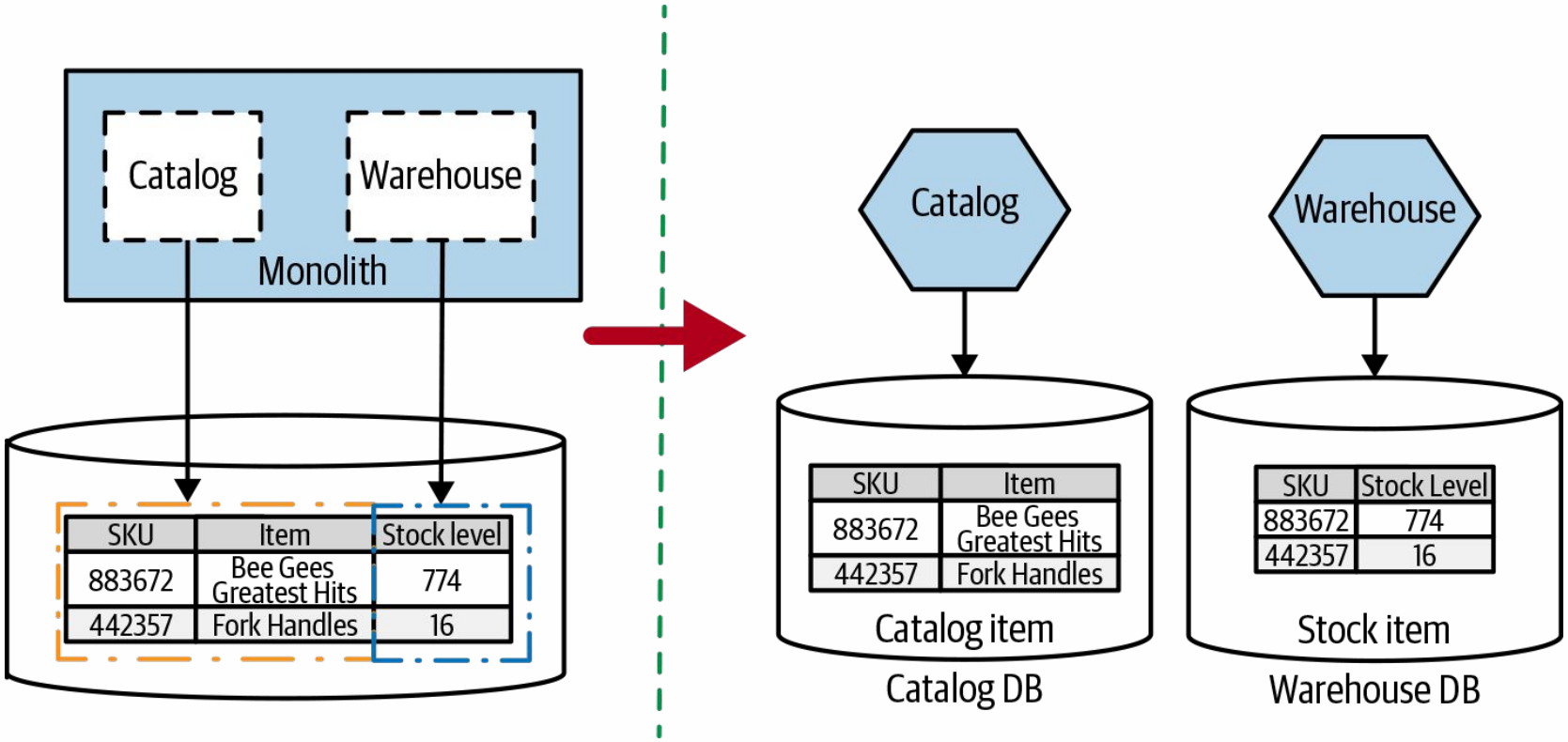
# Меняем ответственность данных

Доступ к данным (даже в рамках одной БД) осуществляется через сервис



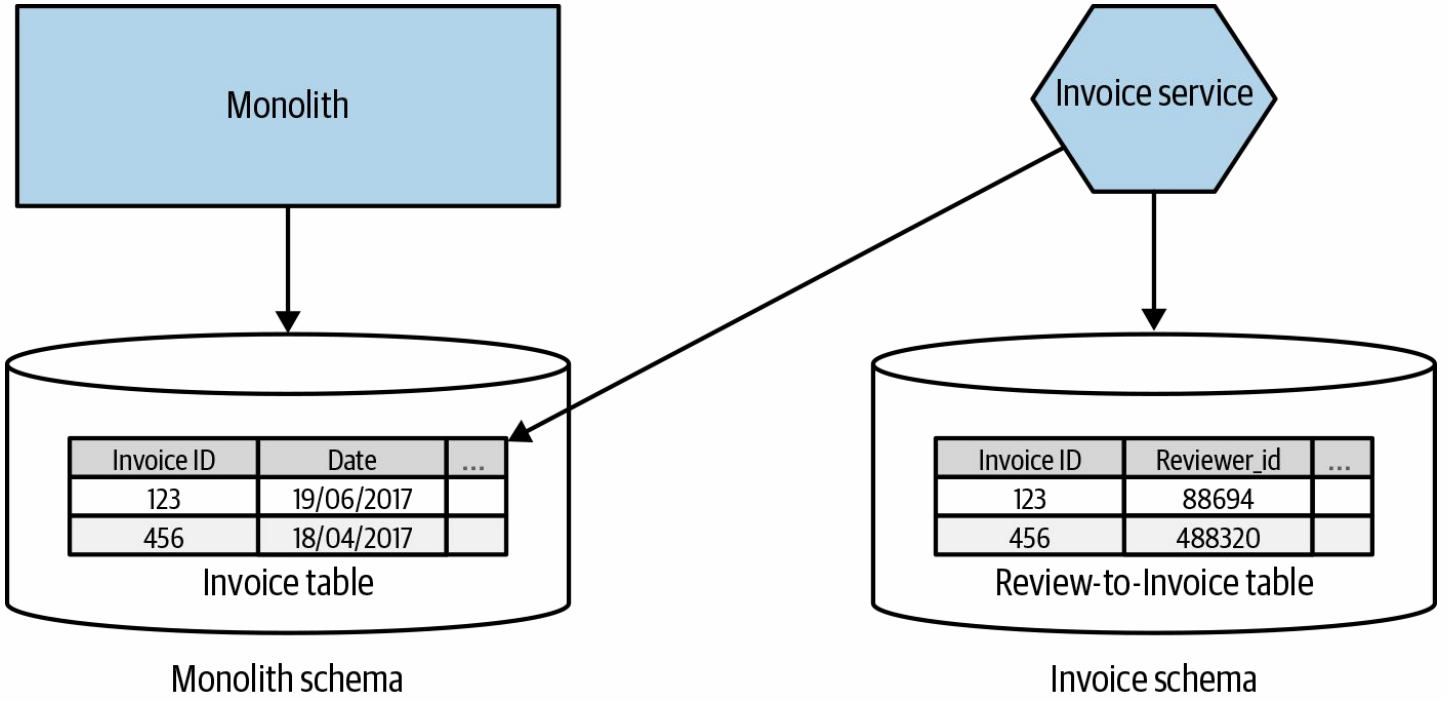
# Разделение табличек

- Иногда даже можно начать с разделения табличек в рамках одной БД (схемы), которые потом будут разнесены по разным сервисам



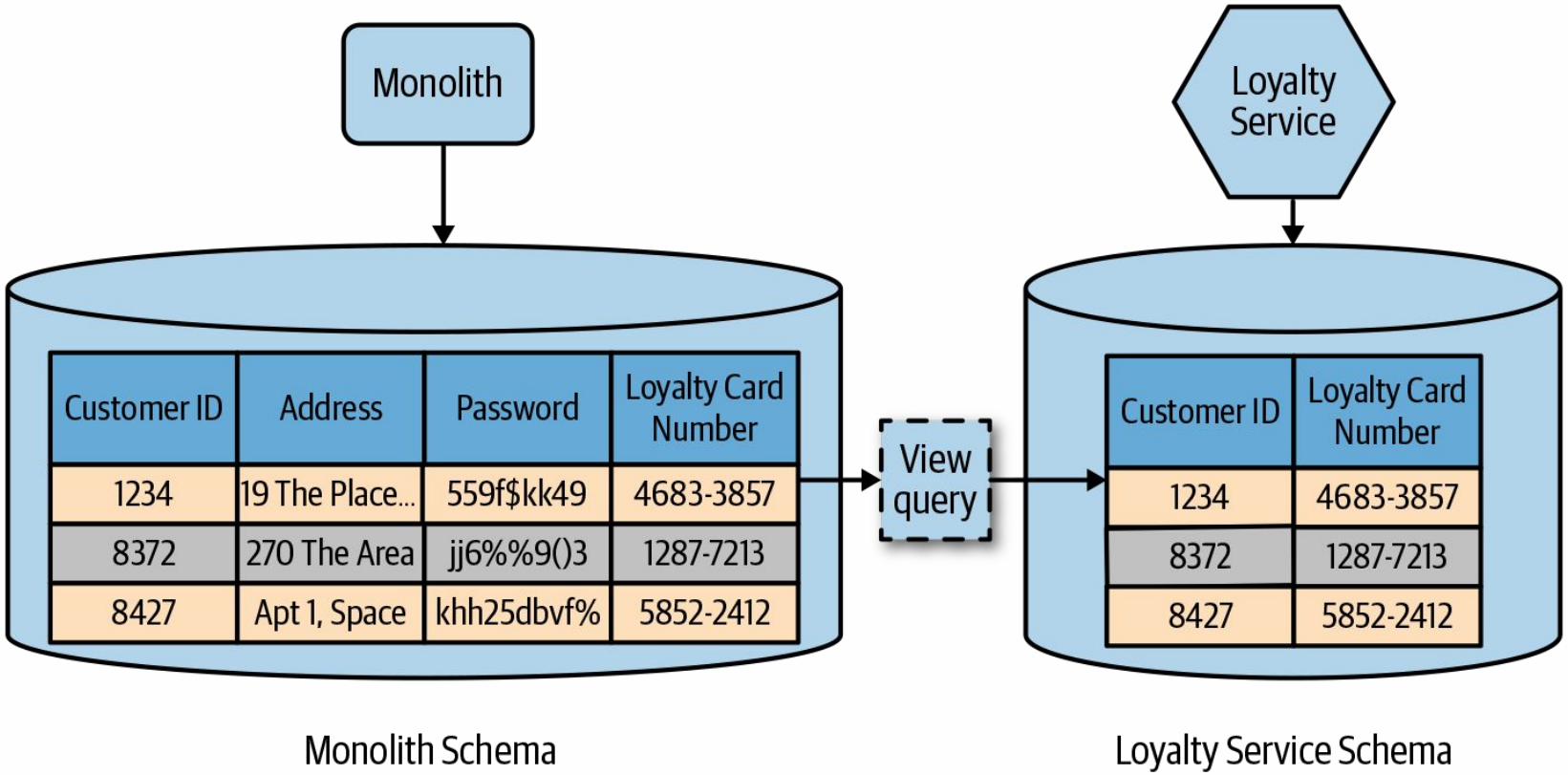
# Данные разделенные по схемам

А иногда можно часть данных унести в схему нового сервиса, а часть оставить в монолите.



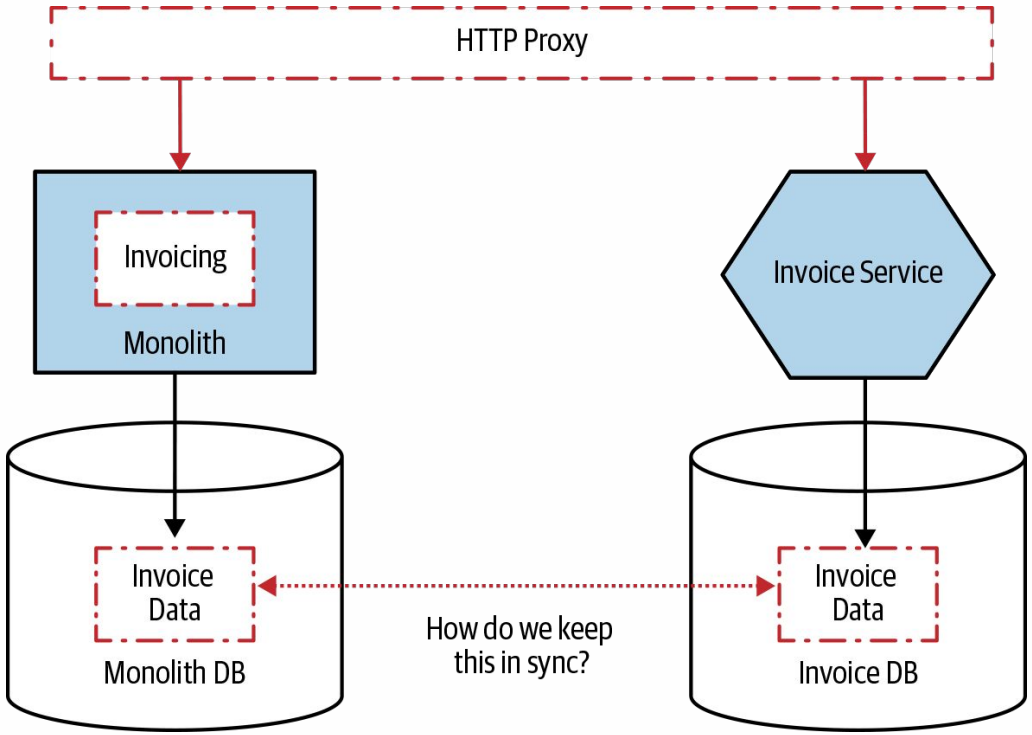
# Предоставляем доступ к данным через view

Можно положить данные не в разные физические БД, а в разные схемы на одном сервере, тогда, можно, например, в монолите сделать view из схемы другого сервиса



# Как обеспечить консистентность данных в разных БД?

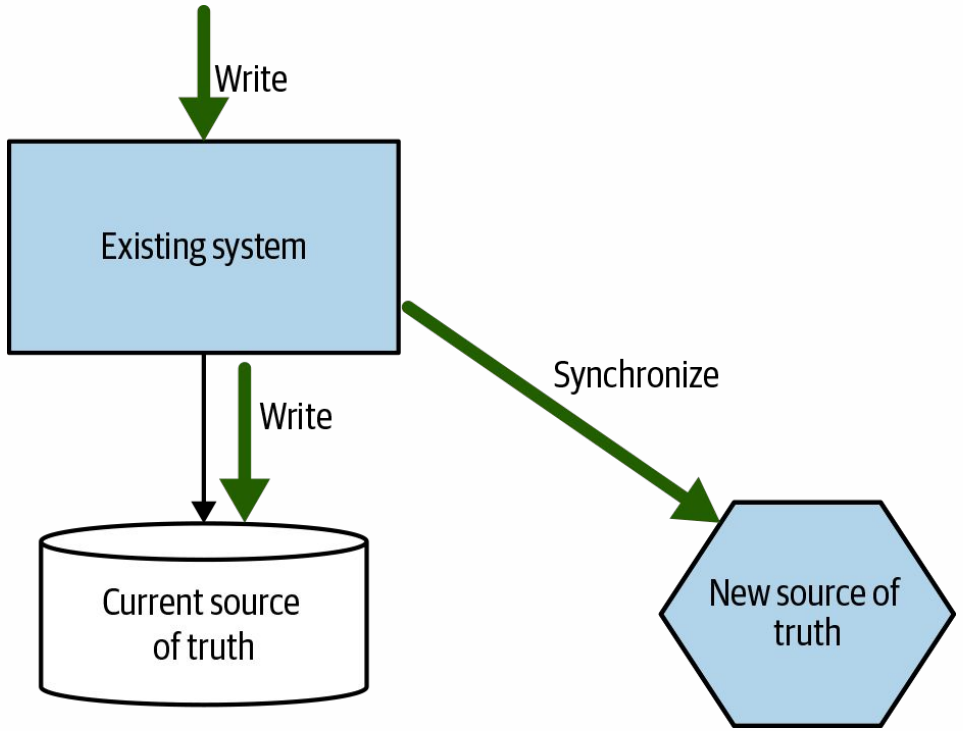
Если же все же данные у нас находятся в разных БД, каким образом обеспечить консистентность этих данных, если предположить, что эти данные все еще нужны.



# Синхронизация данных из приложения

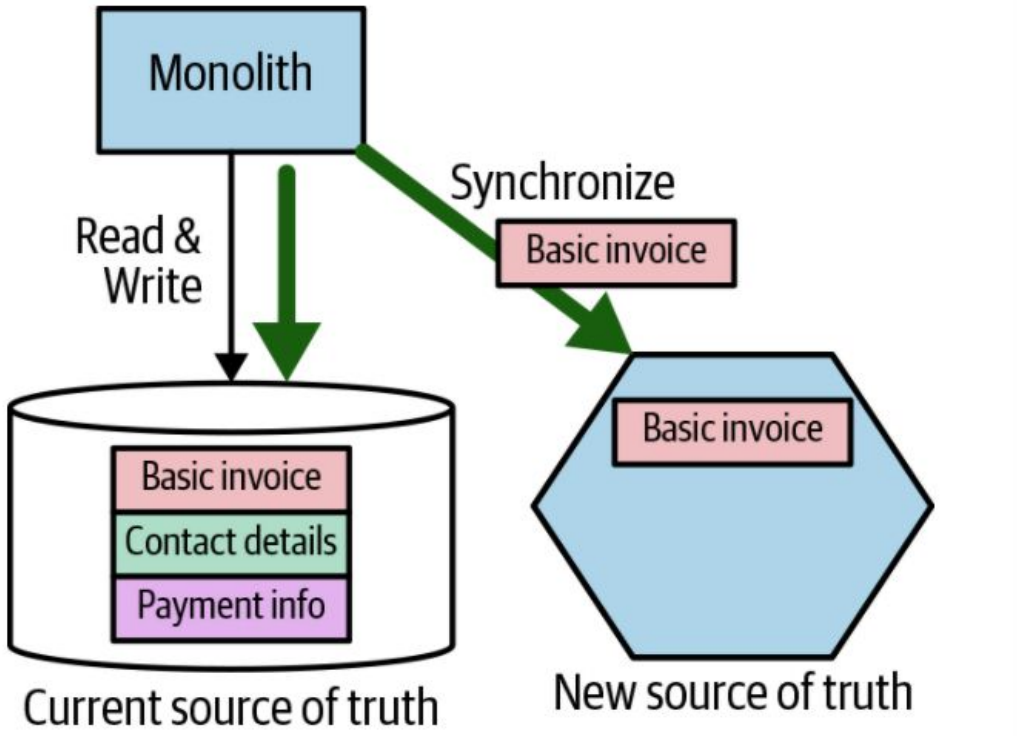
Текущая система (монолит) должна будет писать сразу в 2 источника (на время миграции) в старый источник данных и в новый.

С течением времени потребители должны мигрировать в новый источник. После окончательной миграции можно старый источник удалять.



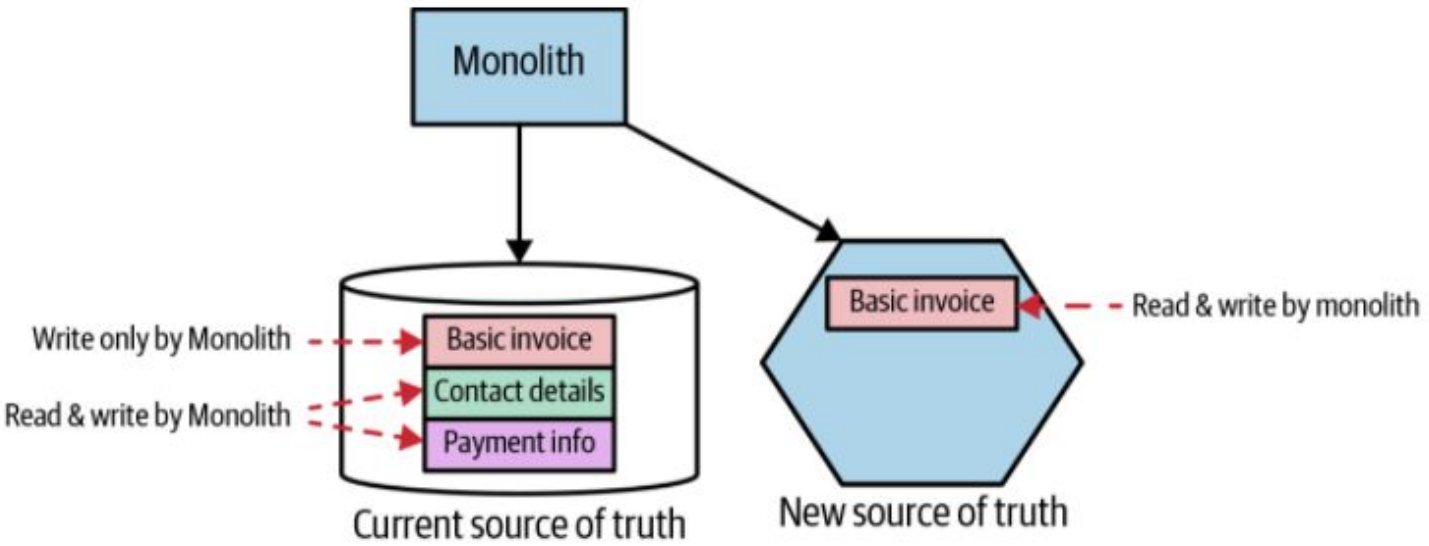
# Синхронизация данных из приложения

Можно инкрементально добавлять в новый источник данных  данные.



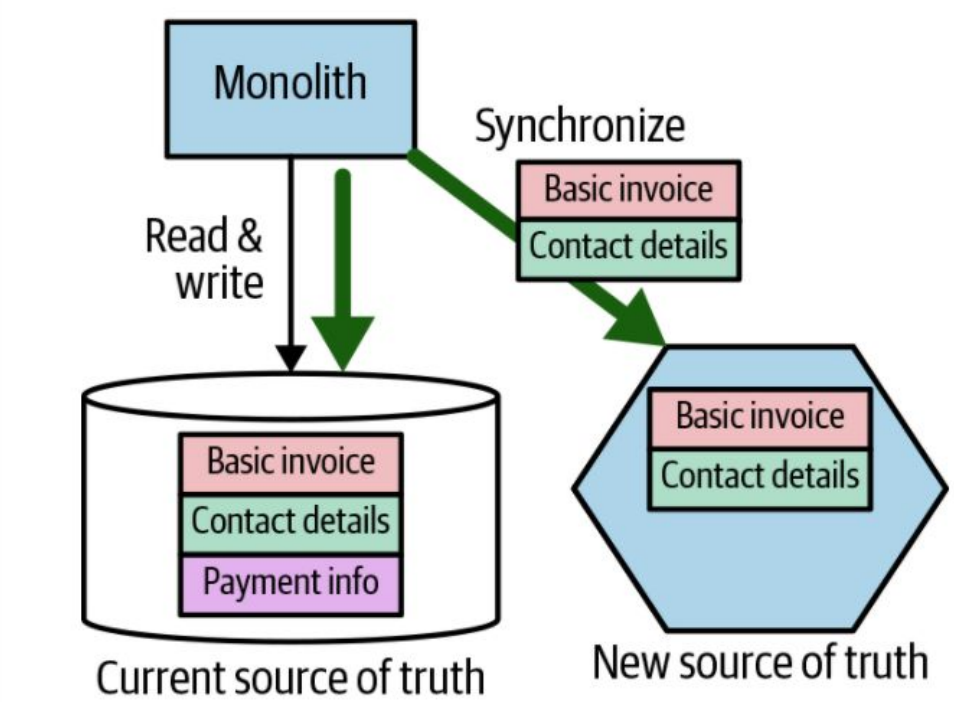
# Синхронизация данных из приложения

Можно инкрементально добавлять в новый источник данных  данные.



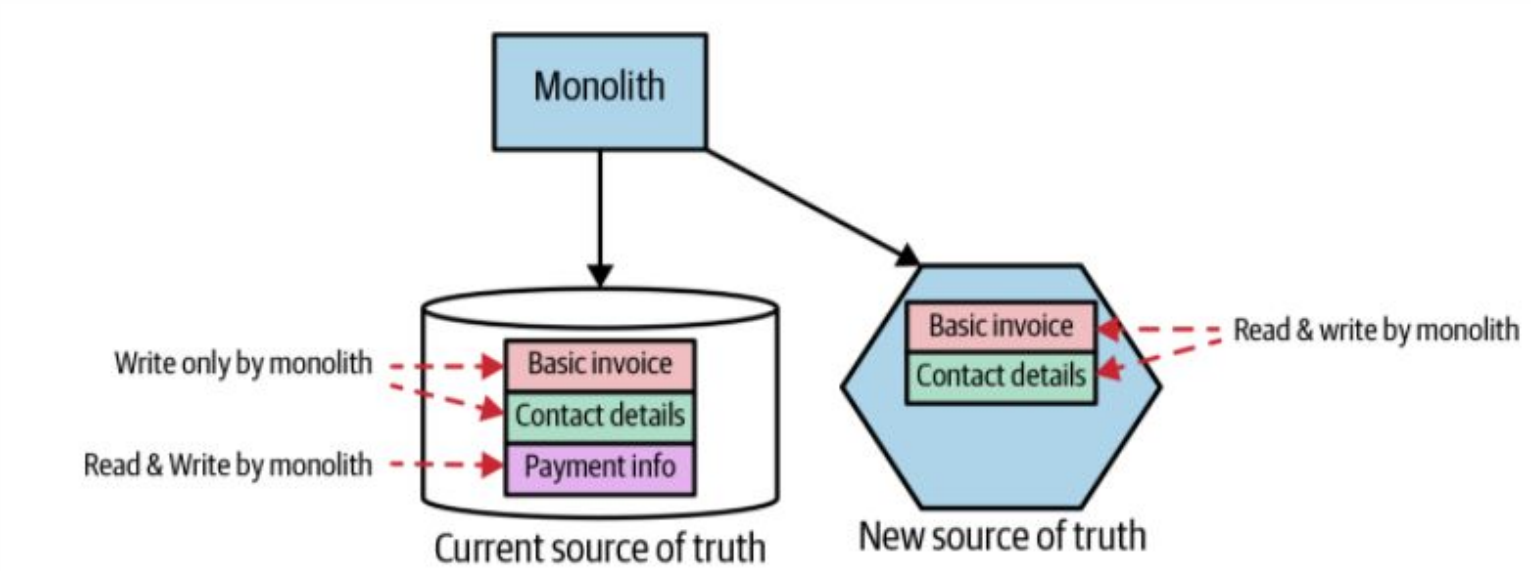
# Синхронизация данных из приложения

Можно инкрементально добавлять в новый источник данных  данные.



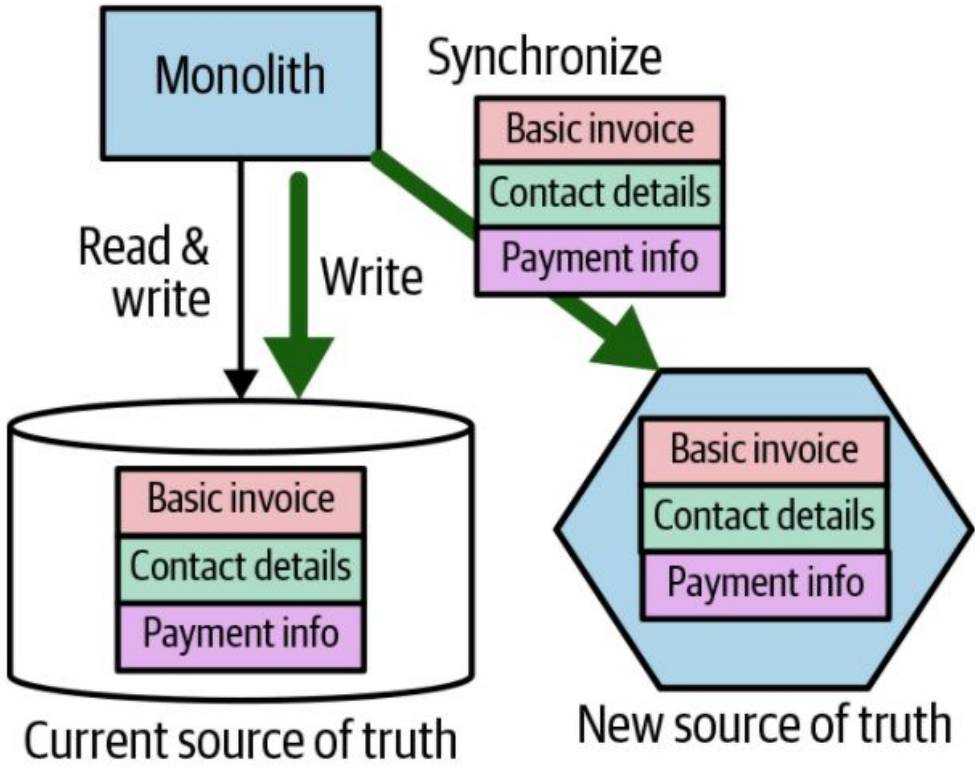
# Синхронизация данных из приложения

Можно инкрементально добавлять в новый источник данных  данные.

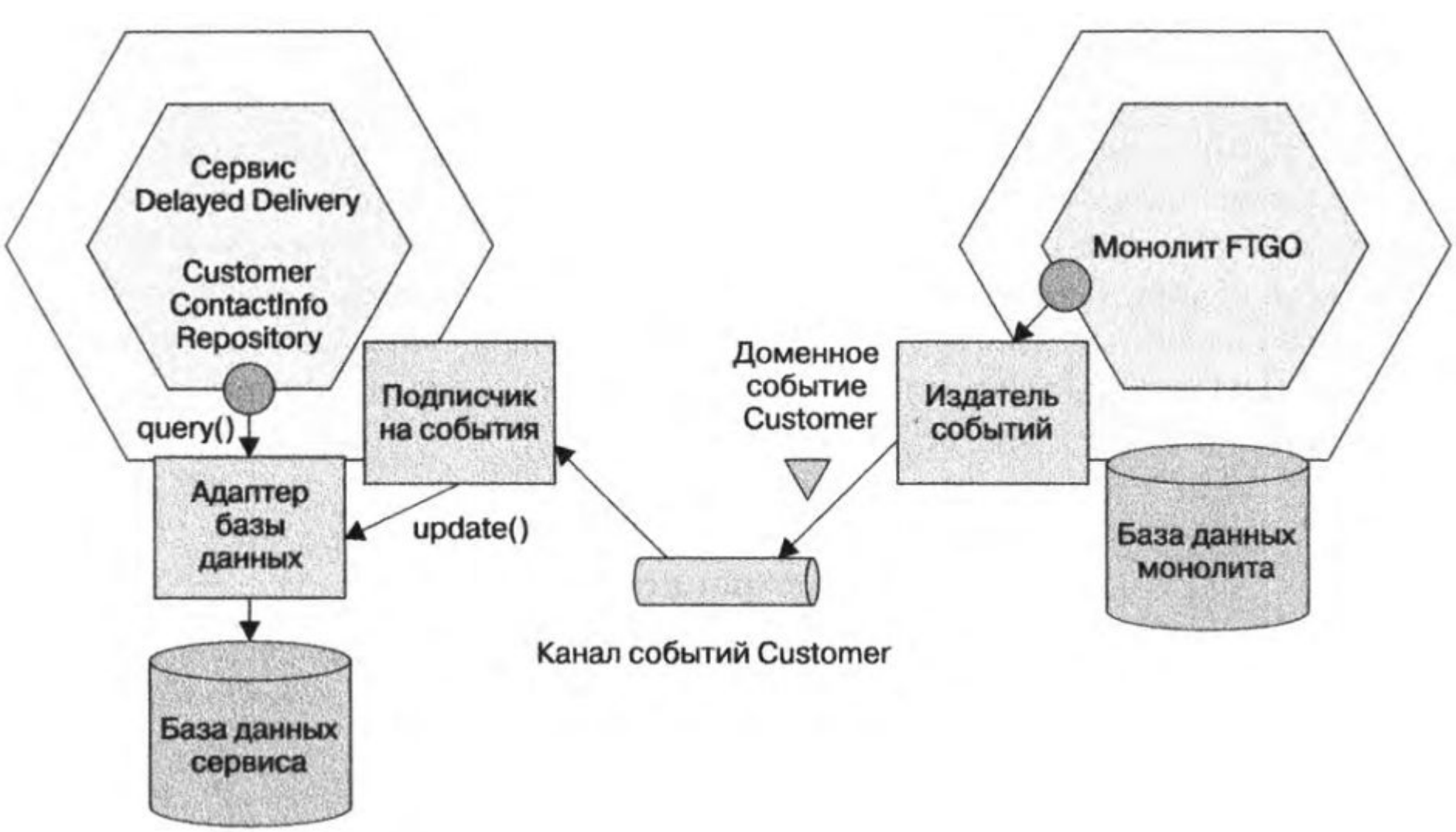


# Синхронизация данных из приложения

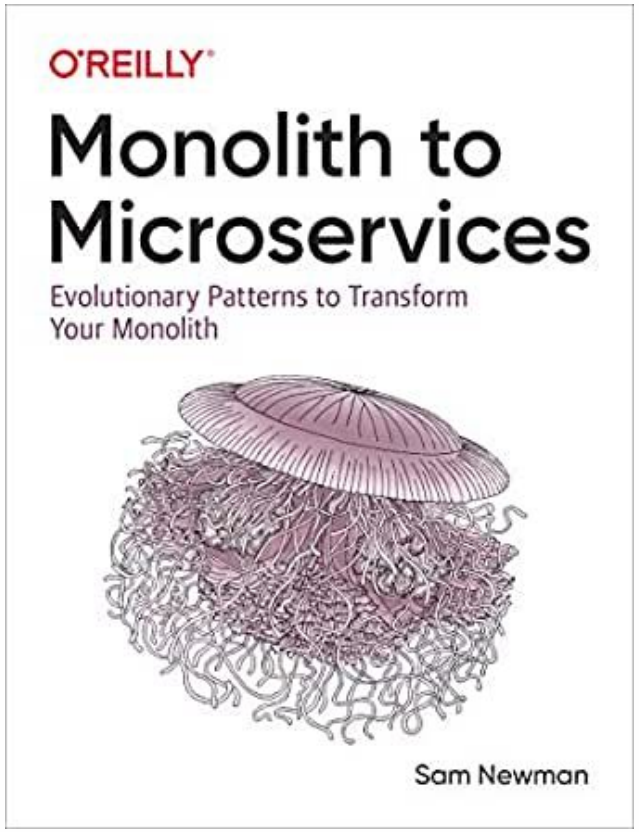
Можно инкрементально добавлять в новый источник данных  данные.

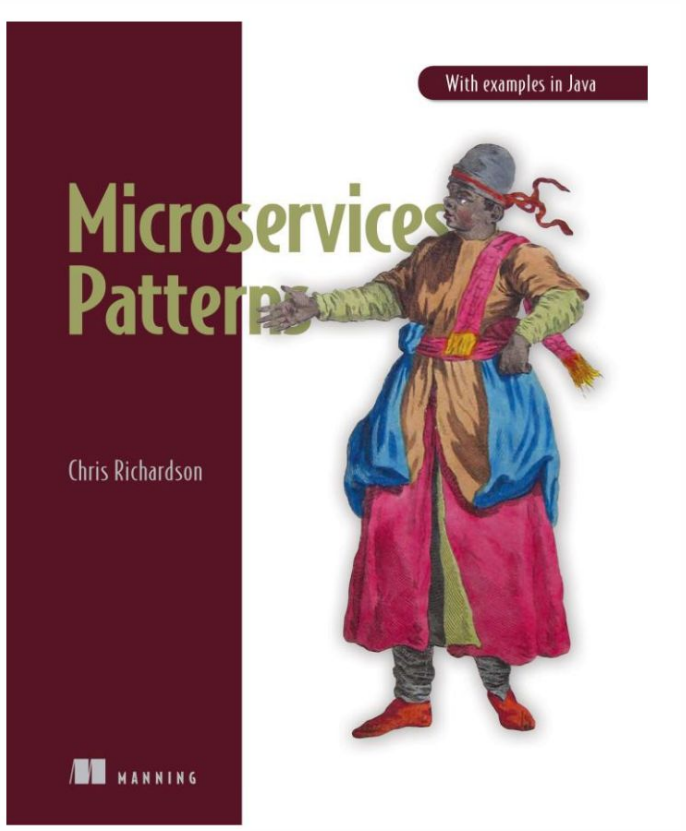


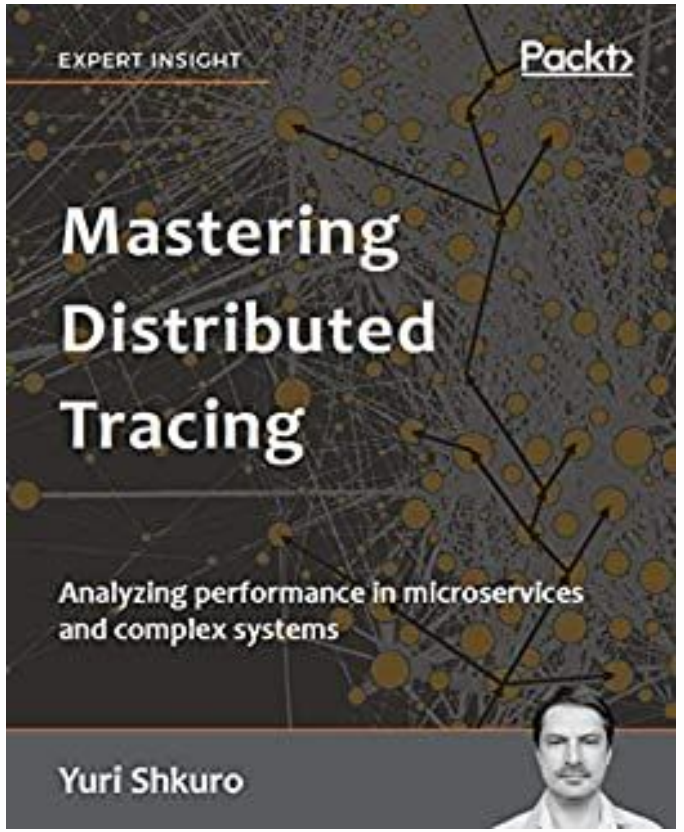
# Синхронизация данных - доменные события



# Книги


O'REILLY
Monolith to Microservices
Evolutionary Patterns to Transform Your Monolith
Sam Newman


With examples in Java
Microservices Patterns
Chris Richardson
MANNING


EXPERT INSIGHT
Packt
Mastering Distributed Tracing
Analyzing performance in microservices and complex systems
Yuri Shkuro

# Опрос

https://otus.ru/polls/36008/

# Спасибо
# за внимание!