The background features a cluster of colorful, overlapping 3D geometric shapes on the left side, primarily in shades of red, blue, green, and yellow, which gradually transition into a solid light blue on the right side.

Конструирование программного обеспечения

План лекции № 13

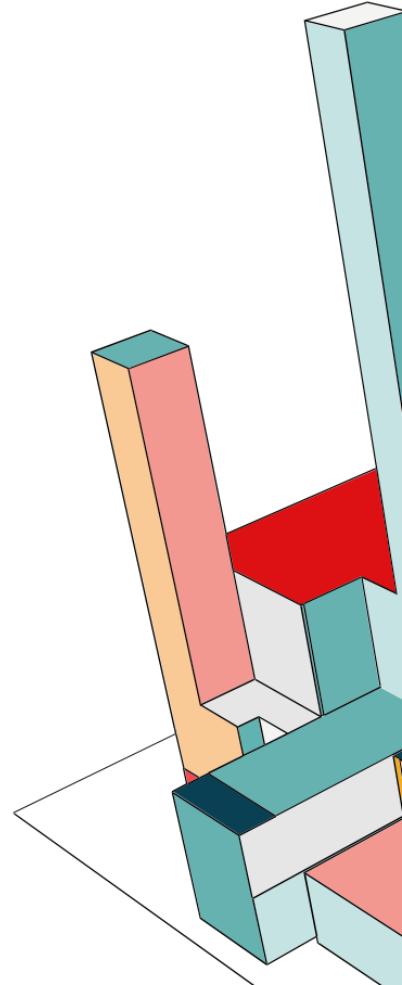
Паттерны работы с данными для высокой производительности

Кэширование

1. Зачем кэш нужен микросервисам.
2. Обзор основных паттернов кэширования.
3. Инвалидация кэша.
4. Инструменты для реализации кэширования.
5. Лучшие практики для построения кэш-слоя.

CQRS

1. Архитектурный паттерн CQRS
2. Различные варианты реализации CQRS



Зачем знать историю?

Современные архитектуры сложные.
Ретроспективно их понять легче

Отбрасывать лишнее, добавлять
собственное.

Предсказывать будущее.



Что такое кэширование?

Кэширование — это процесс сохранения заранее вычисленных данных или результатов обработки запросов в быстро доступной памяти для ускорения их повторного использования

Основная цель кэширования — сократить время отклика приложения, уменьшить нагрузку на серверы и базы данных, а также повысить производительность системы

Главный принцип работы с кэшем: «Если вы можете обойтись без кэширования, то не используйте его!»



Как работает кэширование?



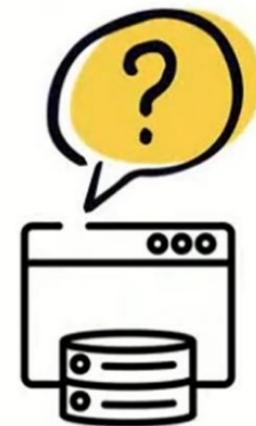
Метрики кэша

- Объем памяти*
- RPS чтения/записи*
- Количество элементов в кэше*
- Hit rate*
- Expired rate*
- Eviction rate*



Что можно кэшировать?

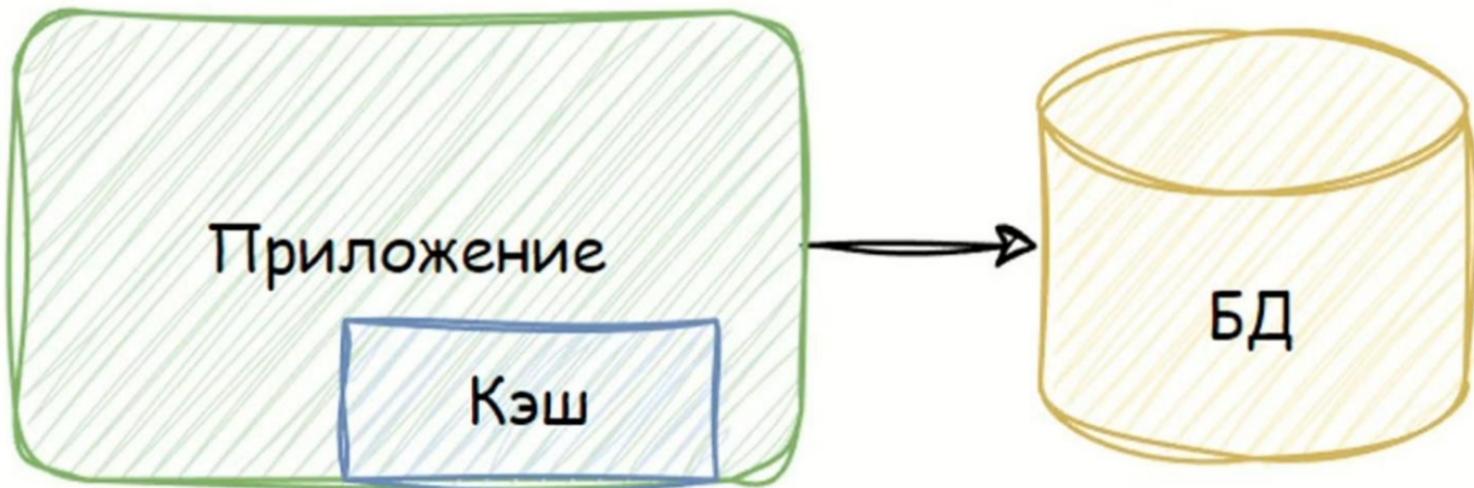
- ❑ Частота доступа (Hot vs Cold Data)
- ❑ Стоимость вычислений или запросов
- ❑ Состояние данных (Static vs Dynamic)
- ❑ Прогнозируемость запросов
- ❑ Трудность масштабирования
- ❑ Время жизни данных (TTL)
- ❑ Риски инвалидации



Caching



Типы кэшей: встроенный кэш (inline)



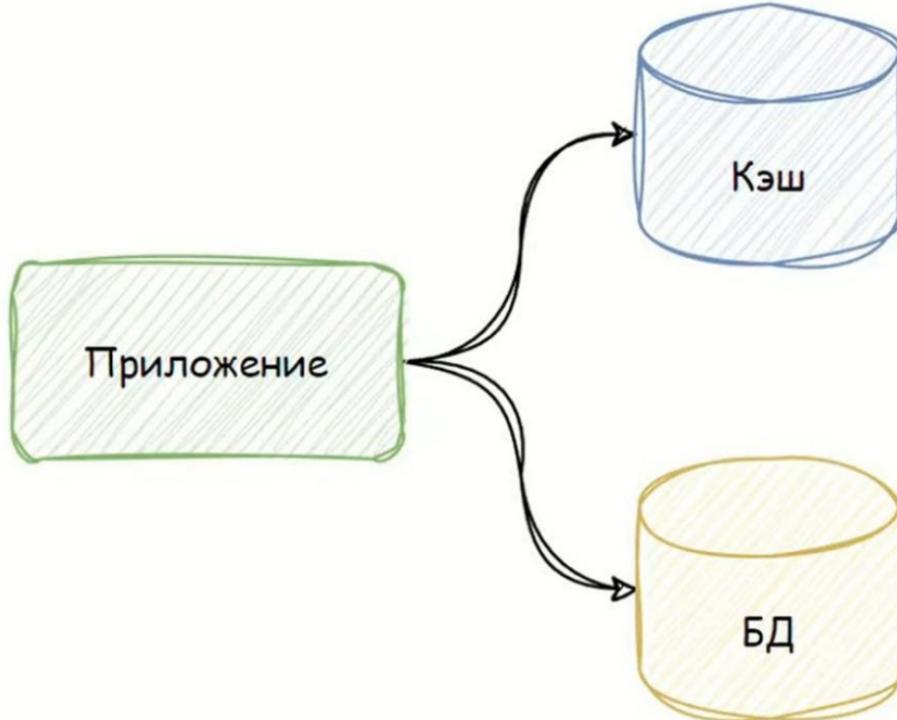
Типы кэшей: встроенный кэш (inline)

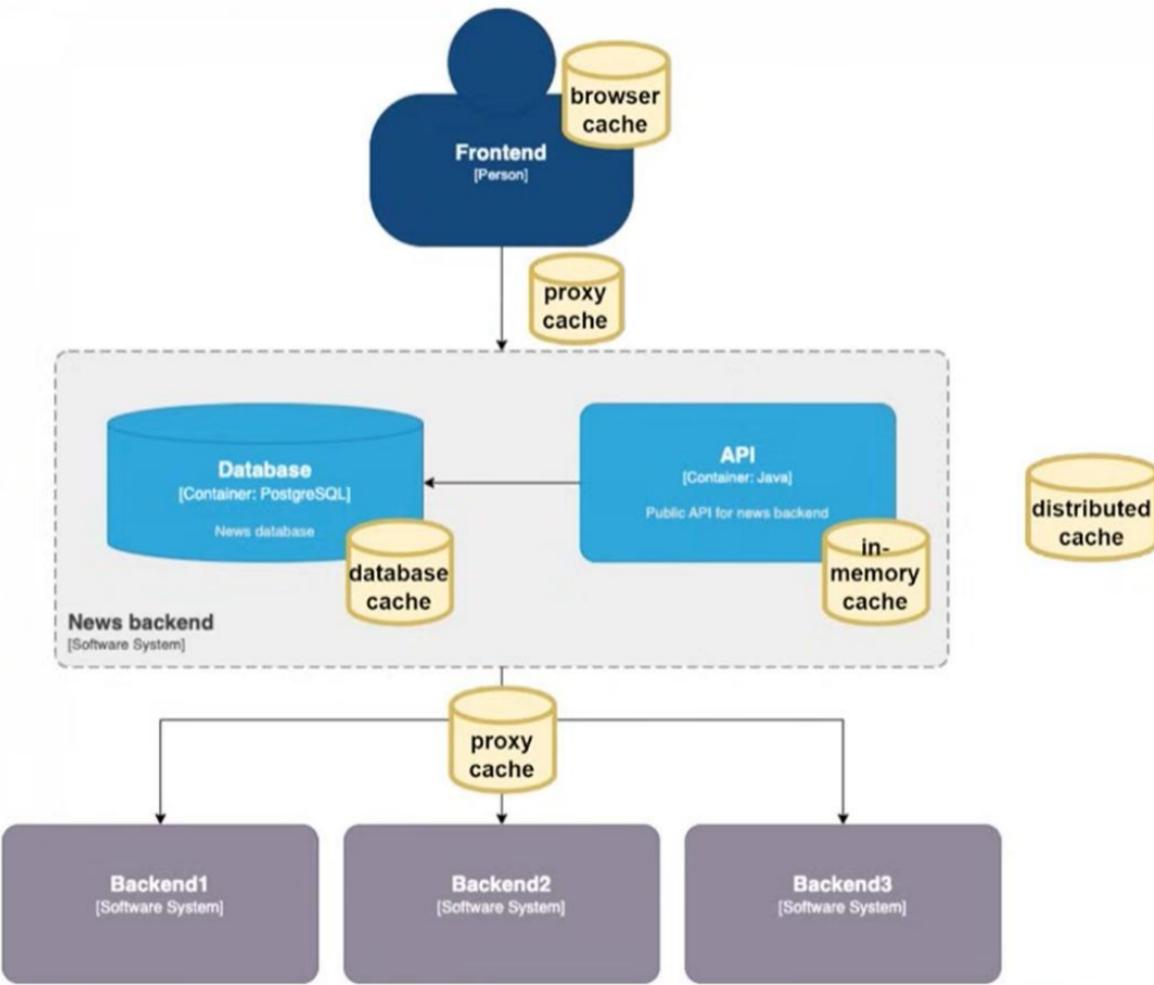
Варианты реализации:

- Использование **Map** для кеширования результатов
- Использование **WeakHashMap** для автоматического удаления элементов
- Использование Cache библиотеки: **Caffeine**, Ehcache, Guava Cache
- Использование аннотаций или Aspect-Oriented Programming (AOP)



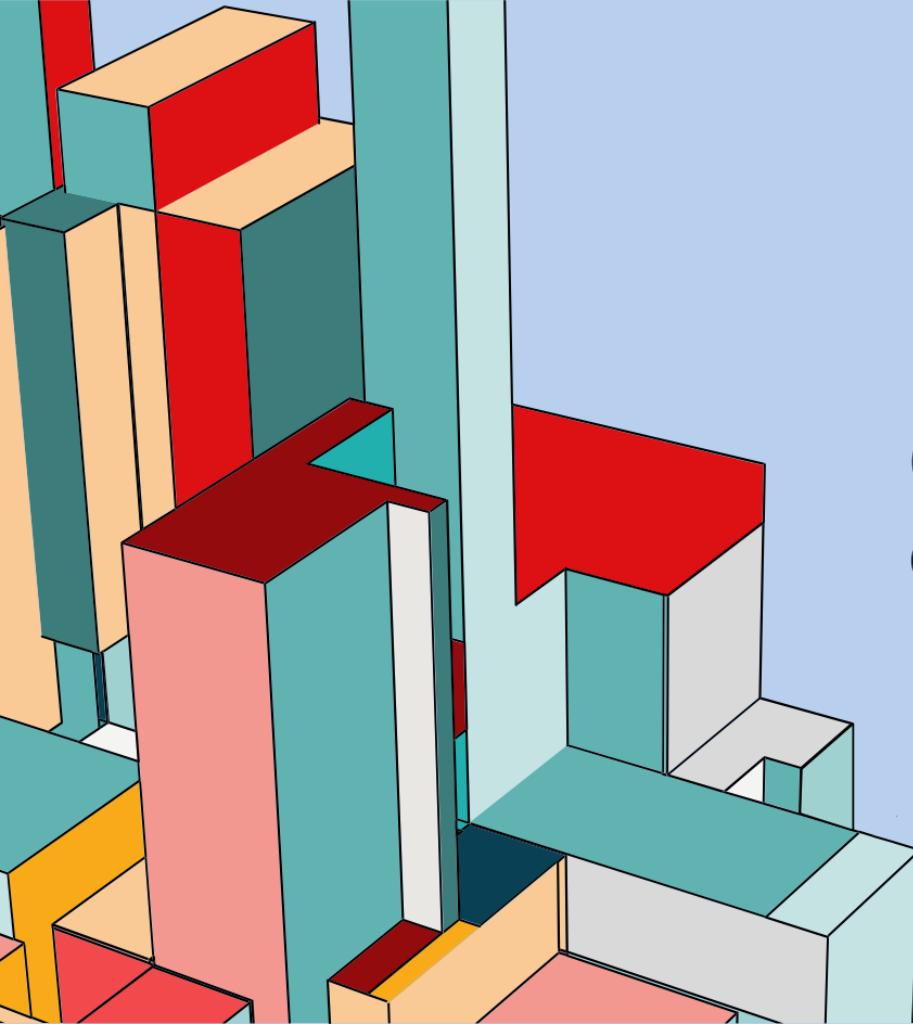
Типы кэшей: отдельные кэши (sidecar)





Сравнение кэшей

Характеристика	Встроенный кэш (inline)	Отдельные кэши (sidecar)
Скорость ответа	Выше: обращаемся напрямую к памяти	Ниже: есть сетевые вызовы, а также оверхед на работу самого хранилища
Память	Кэш и приложение делят одну область памяти, поэтому ее тратится меньше	Под кэш выделена отдельная память, поэтому ее тратится больше
Согласованность данных	Плохая: каждая копия приложения содержит только свои данные	Хорошая: все копии приложения обращаются к единому хранилищу
Масштабирование	Кэш нельзя масштабировать отдельно от приложения	Кэш и приложение можно масштабировать независимо друг от друга
Ресурсы (RAM, CPU)	Общие, поскольку "живут" в одном процессе	Выделенные, поскольку "живут" в разных процессах
Простота сопровождения	Проще: кэш – просто структура данных, предоставляемая библиотекой	Сложнее: кэш – отдельно разворачиваемый компонент, требующий отдельного мониторинга и экспертизы
Изолированность	Низкая: проводить отдельно работы над кэшем крайне сложно	Высокая: мы можем проводить любые работы над кэшем независимо от приложения
Горячий старт	Сложнее: приложение обычно не общается с диском напрямую	Проще: например, redis может периодически сбрасывать данные на диск и восстанавливать состояние после рестарта

The background features a cluster of colorful, overlapping 3D rectangular blocks on the left side. These blocks are rendered in various colors including red, orange, yellow, teal, light blue, and white, creating a sense of depth and volume. The right side of the slide is a solid light blue color.

Стратегии работы с кэшем

Стратегии работы с кэшем

- Cache through (Сквозное кэширование)
- Write through (Сквозная запись)
- Cache aside (Кэширование на стороне): Read / Write
- Write aside (Запись на стороне)
- Cache ahead (Опережающие кэширование)

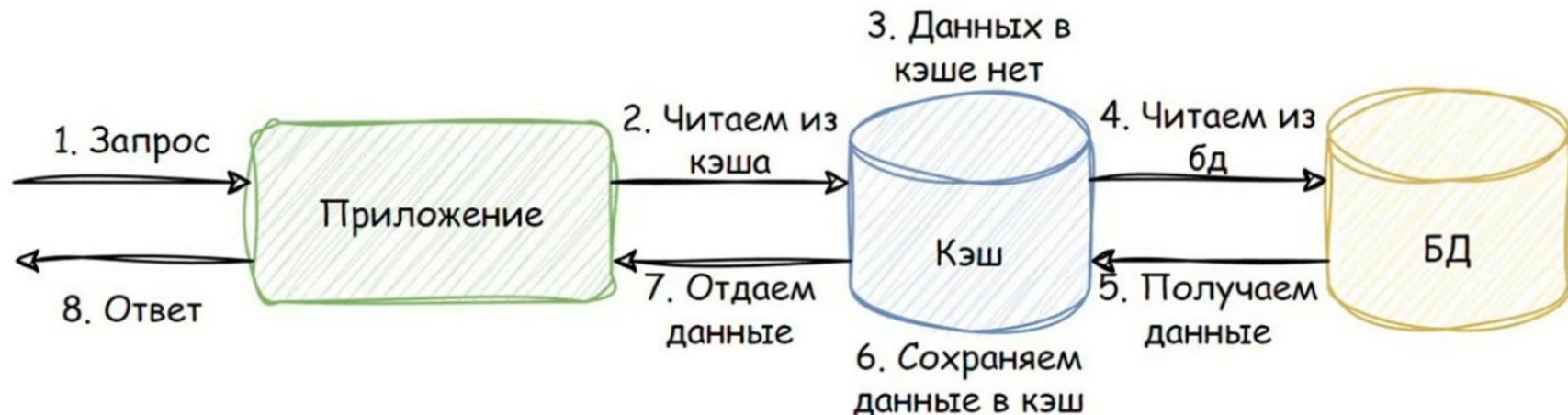


STRATEGY



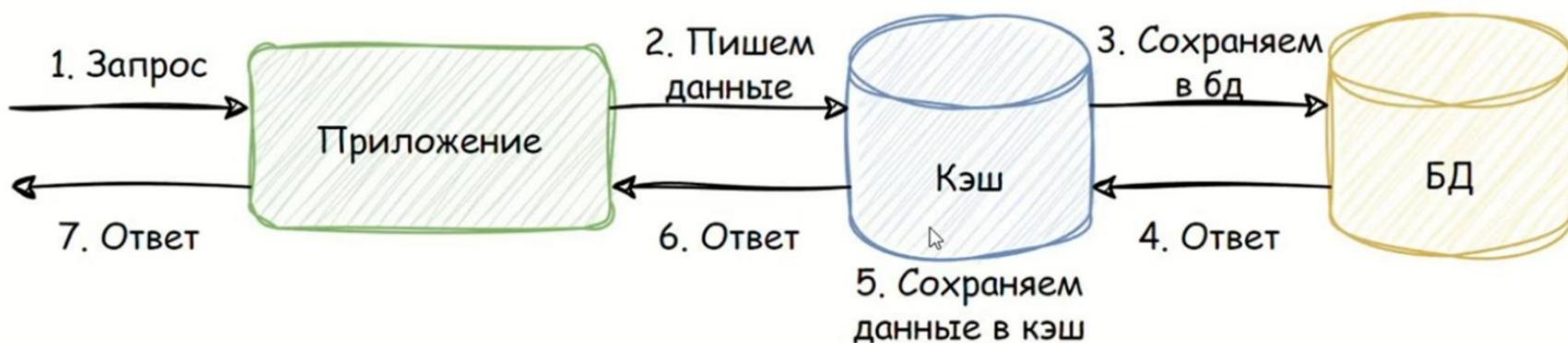
Cache through (Сквозное кэширование)

Read through (Сквозное чтение)

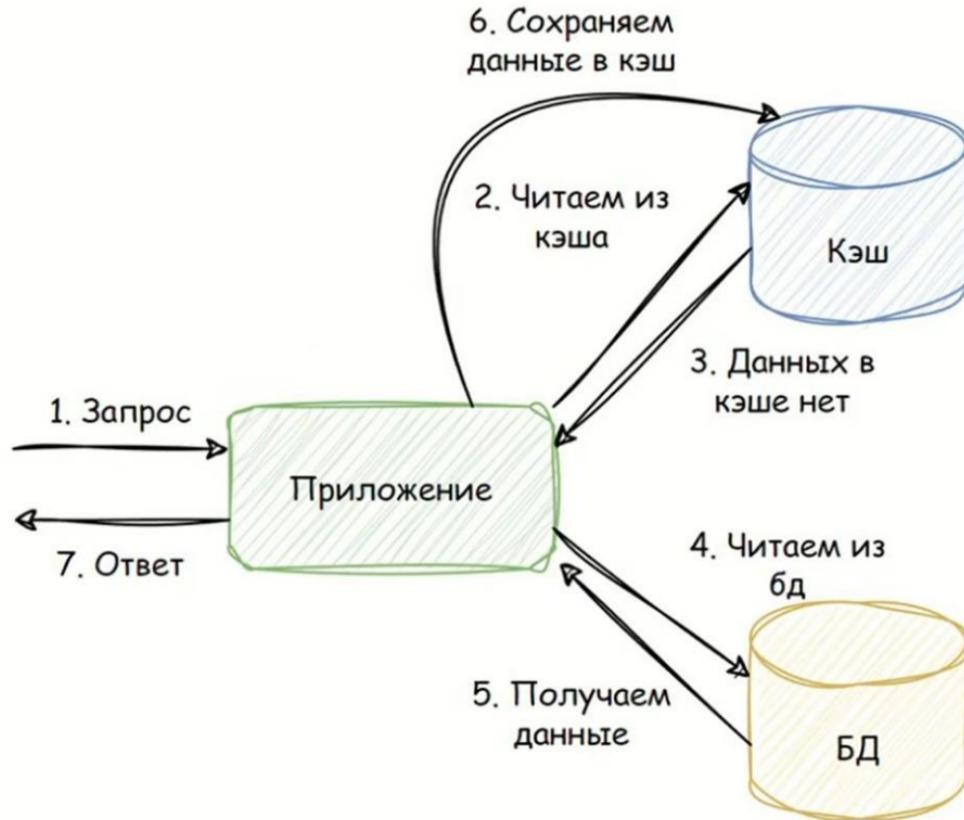


Cache through (Сквозное кэширование)

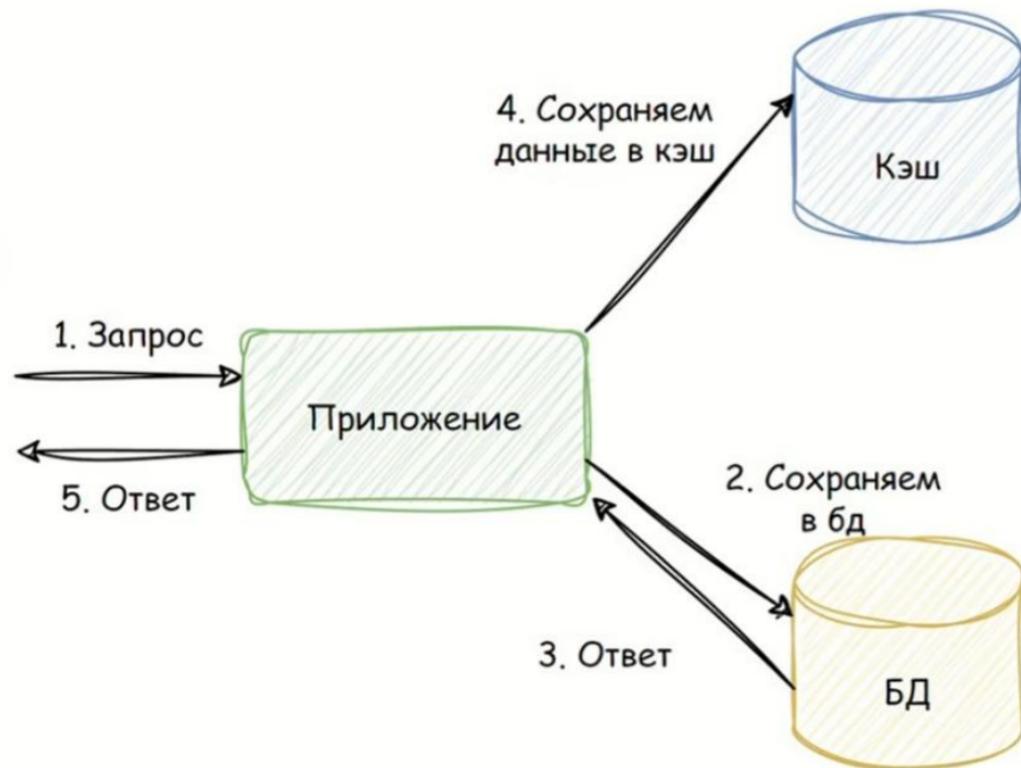
Write through (Сквозная запись)



Cache aside: Read aside (чтение на стороне)



Cache aside: Write aside (запись на стороне)



Cache ahead (Опережающие кэширование)

Чтение только из кэша и запись на опережение



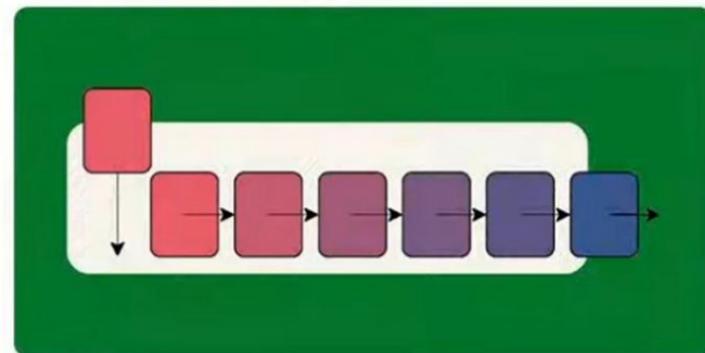
Инвалидация

Инвалидация – это процесс удаления данных из кэша / пометка их как недействительных

Цель инвалидации – для обесечения актуальности данных, с которыми работает приложение

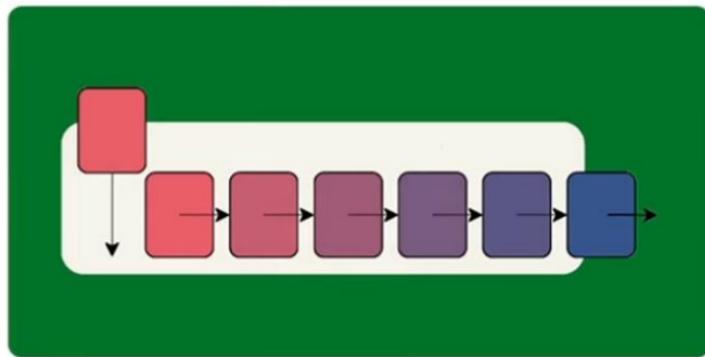
Стратегия инвалидации:

- Инвалидация по TTL
- Инвалидация по событию



Стратегии вытеснения

- Random
- TTL
- LRU



Стратегии вытеснения: TTL

- **TTL (Time To Live)** – это стратегия вытеснения данных из кэша, основанная на времени жизни данных. Каждому элементу в кэше присваивается время жизни, по истечении которого данные становятся устаревшими и подлежат удалению (вытеснению)



Стратегии вытеснения: LRU

- **LRU (Least Recently Used)** – это стратегия вытеснения в кэше, при которой из кэша удаляются данные, которые не использовались дольше всего. Этот алгоритм предполагает, что данные, которые использовались недавно, с большой вероятностью будут востребованы снова, в то время как данные, не использовавшиеся долго, вероятно, больше не понадобятся



Стратегии вытеснения: LFU

- **LFU (Least Frequently Used)** – это стратегия вытеснения в кэше, при которой удаляются данные, которые использовались реже всего. В отличие от LRU, которая основывается на времени последнего использования, LFU ориентируется на количество обращений к данным



Кэширование ошибок

Кэширование ошибок – это практика сохранения информации о произошедших ошибках в кэше для повторного использования или предотвращения повторных операций в будущем

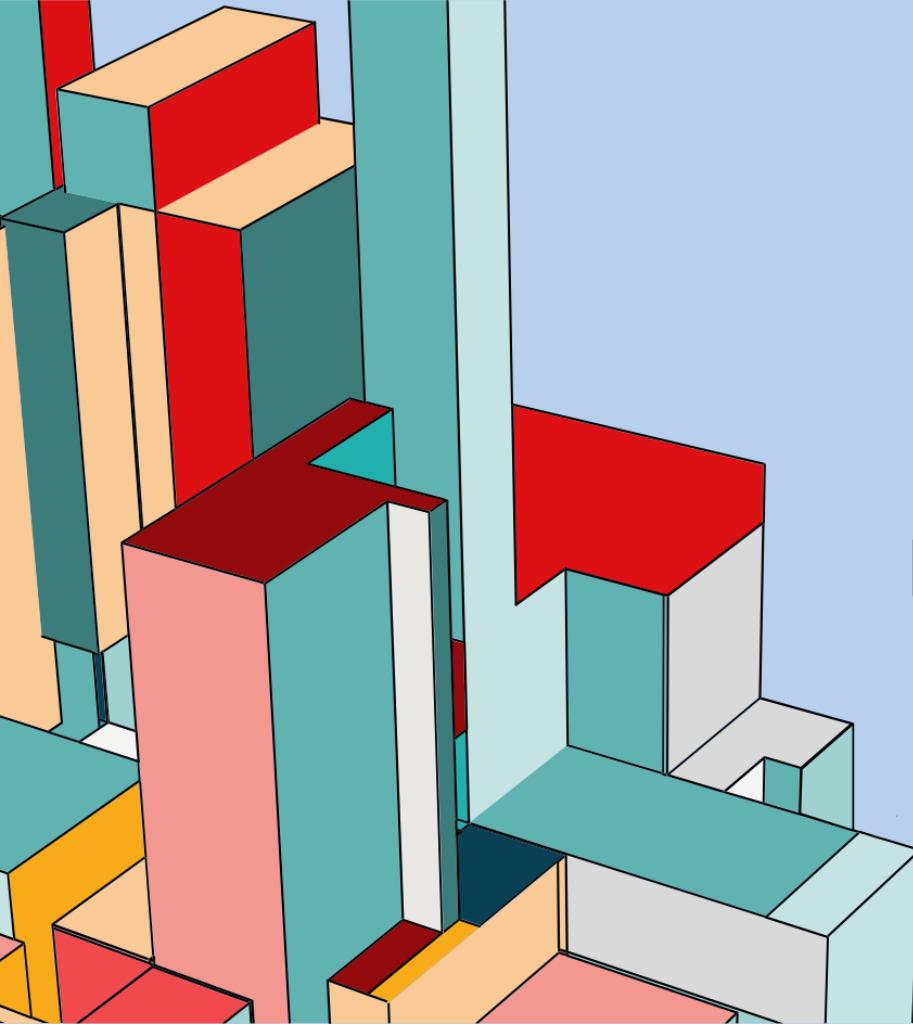
Когда стоит кэшировать ошибки:

- ✓ ошибки с одинаковыми последствиями
- ✓ защита от избыточных нагрузок

Когда не стоит кэшировать ошибки:

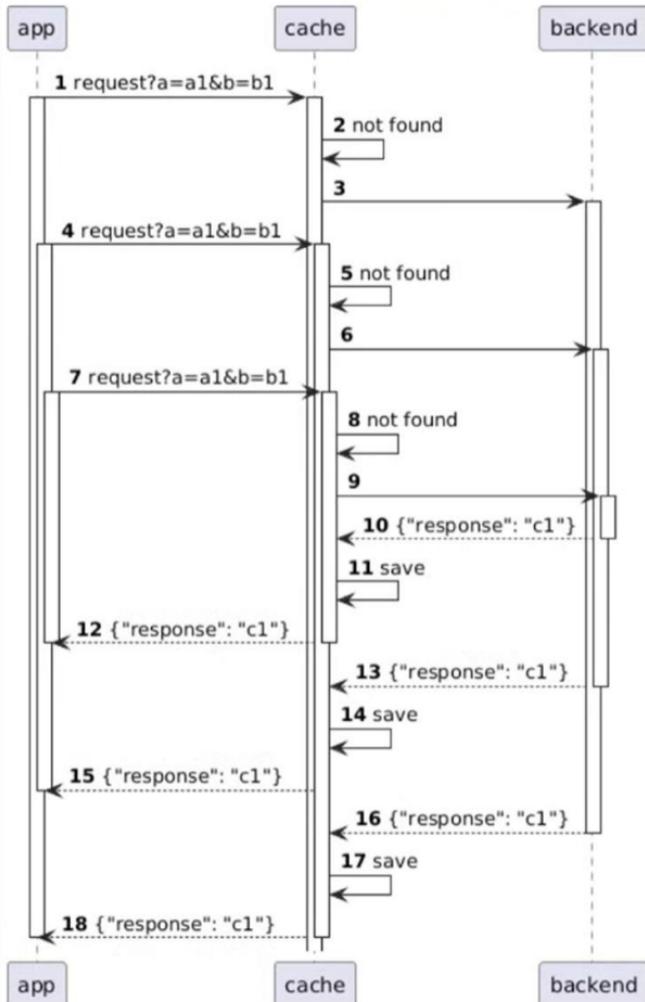
- ✓ ошибки, требующие немедленного исправления
- ✓ ошибки, которые могут изменяться
- ✓ негативное влияние на UX (User Experience)



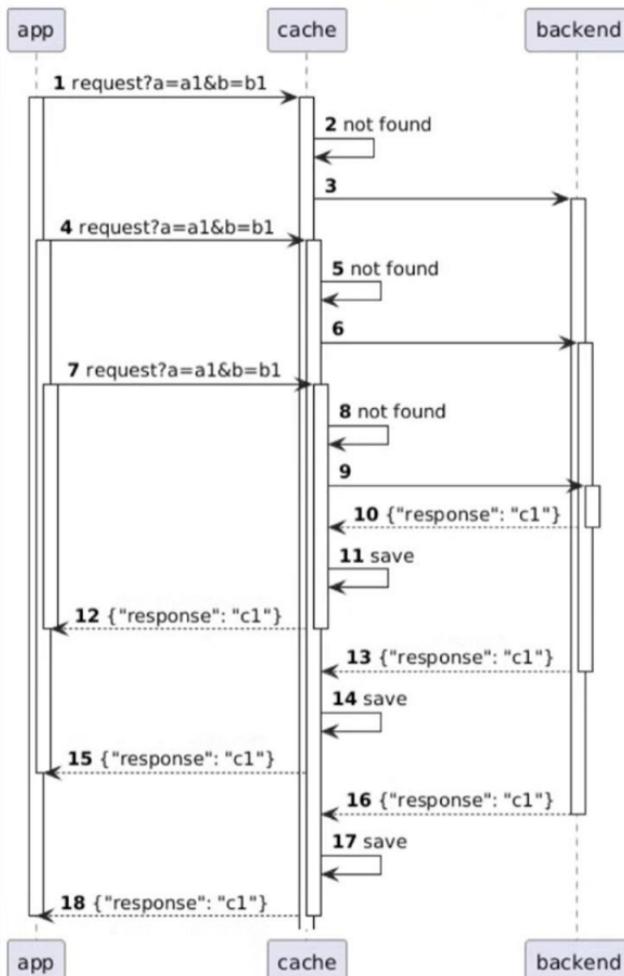
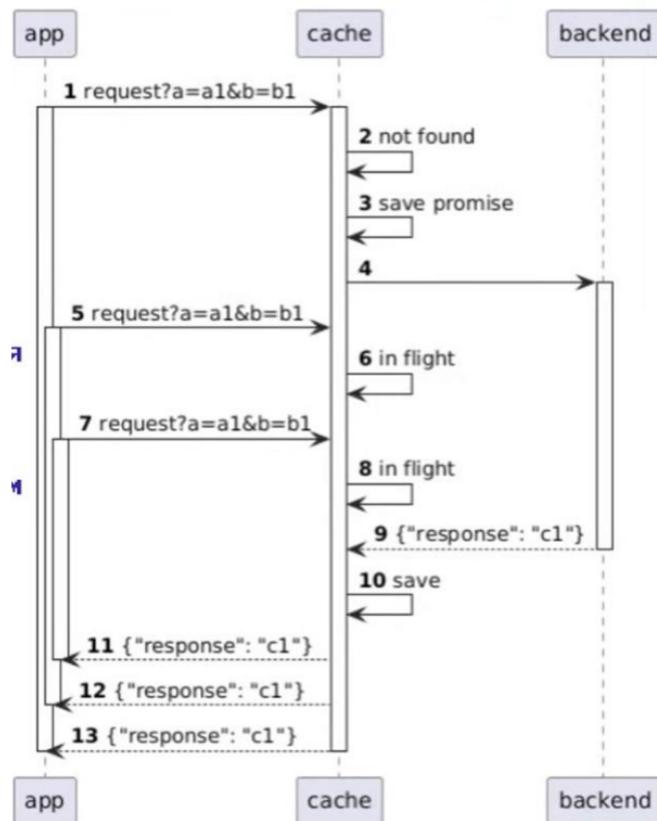
The background features a cluster of colorful, overlapping 3D rectangular blocks on the left side. These blocks are rendered in various colors including red, orange, yellow, teal, light blue, and white, creating a sense of depth and volume. The right side of the slide is a solid light blue color.

Проблемы реализации кэша

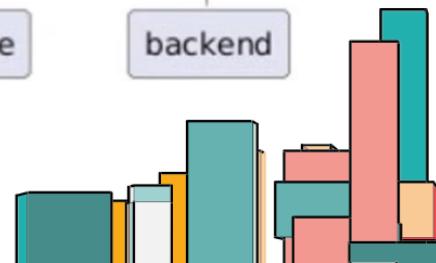
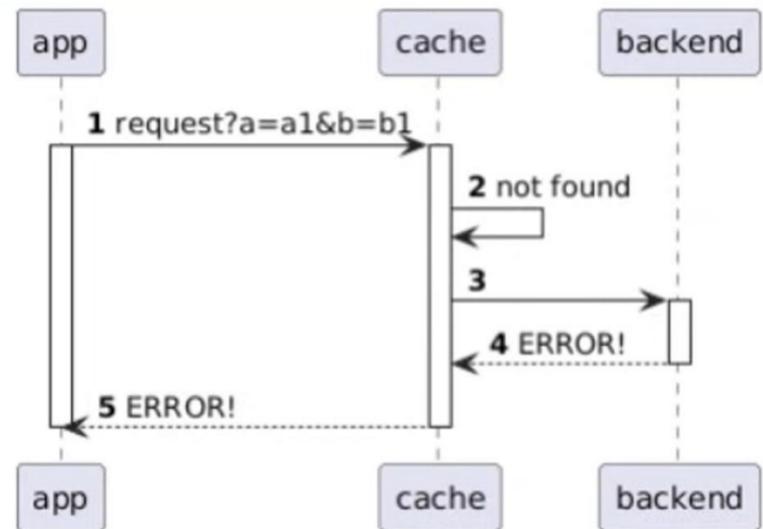
Проблема дублирования



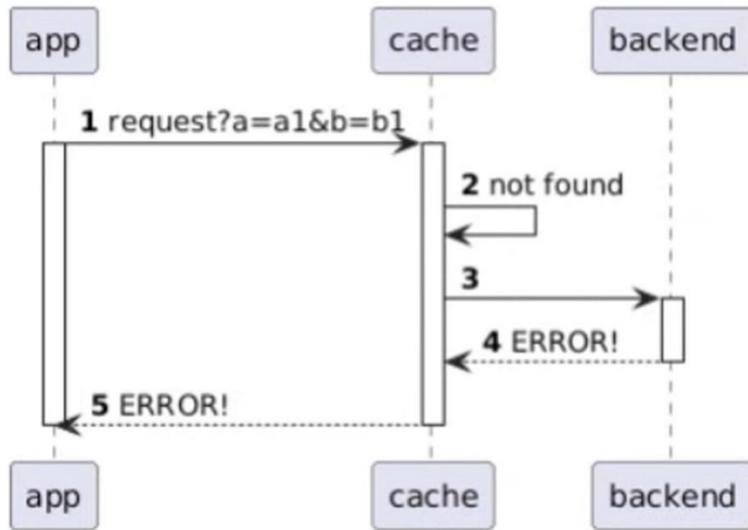
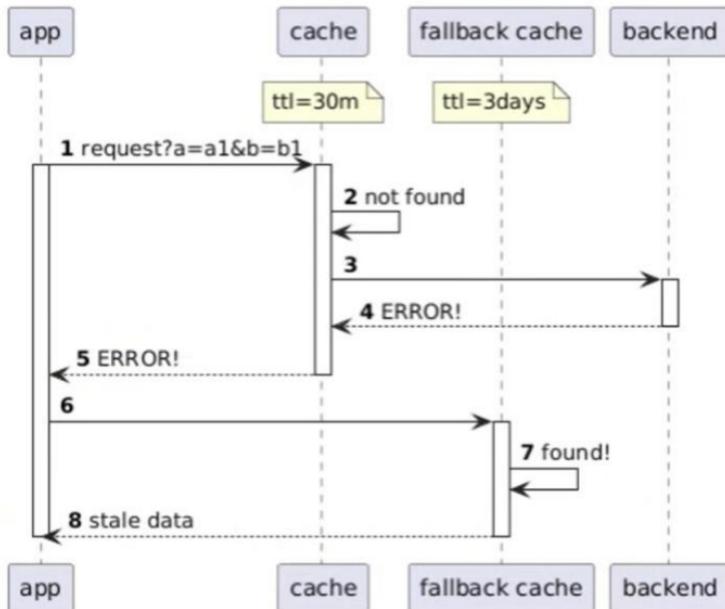
Паттерн Singleflight



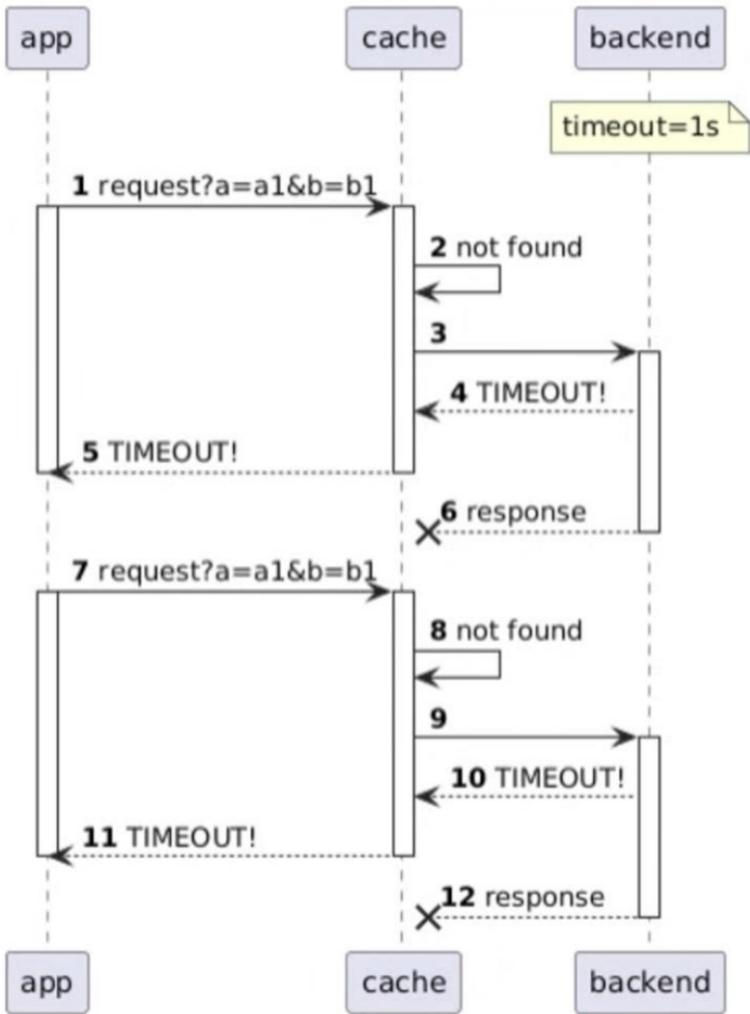
Проблема недоступность данных



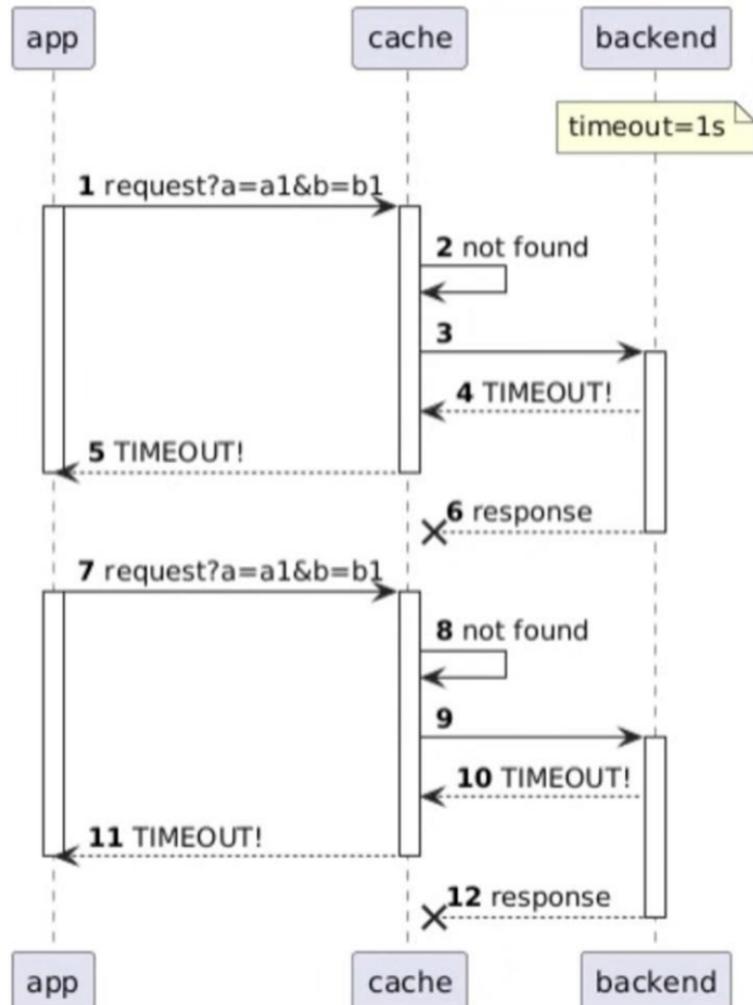
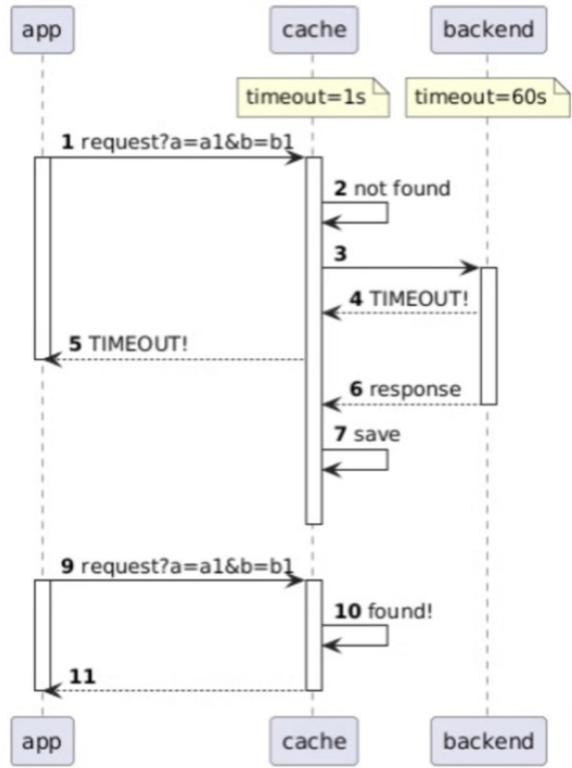
Паттерн Fallback

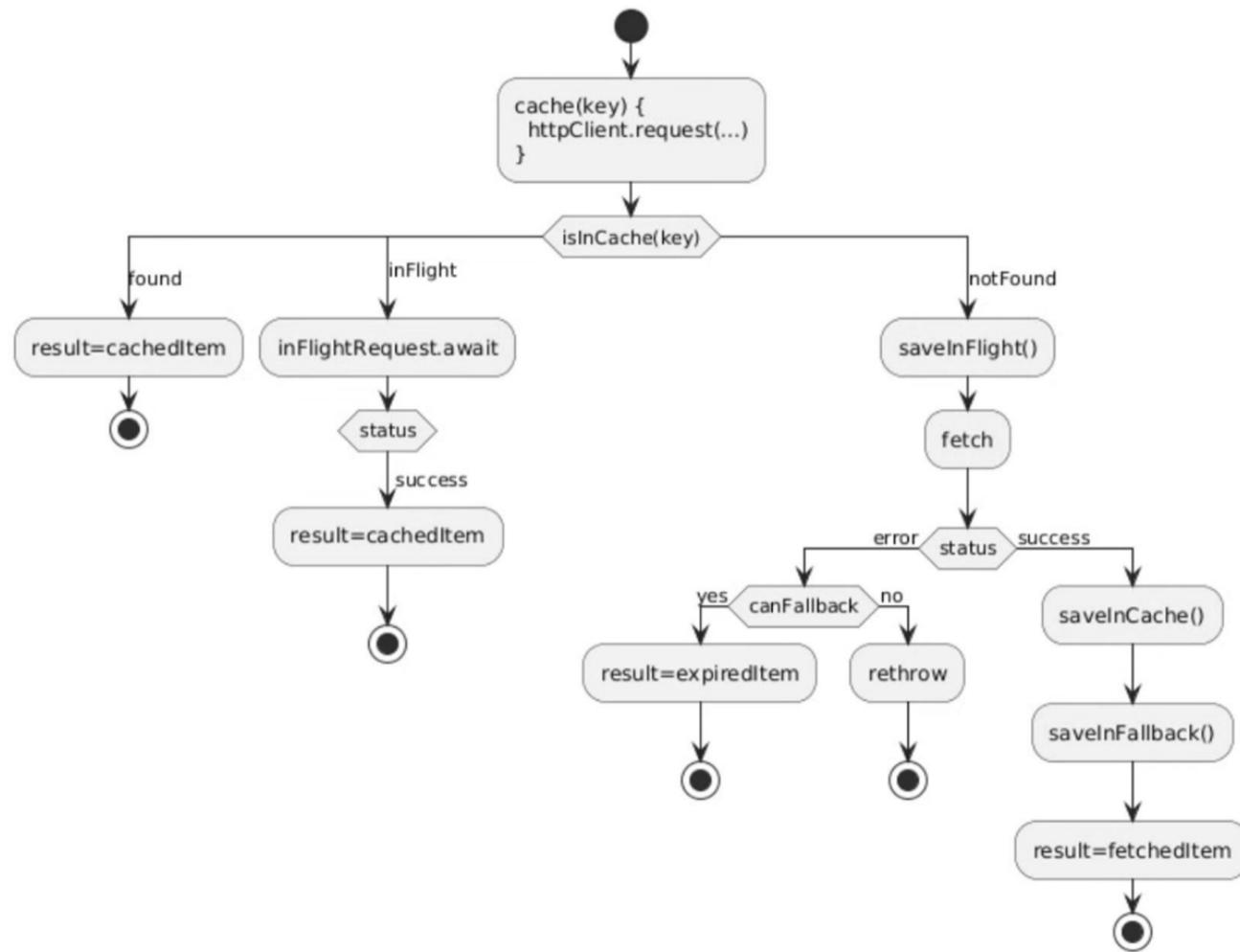


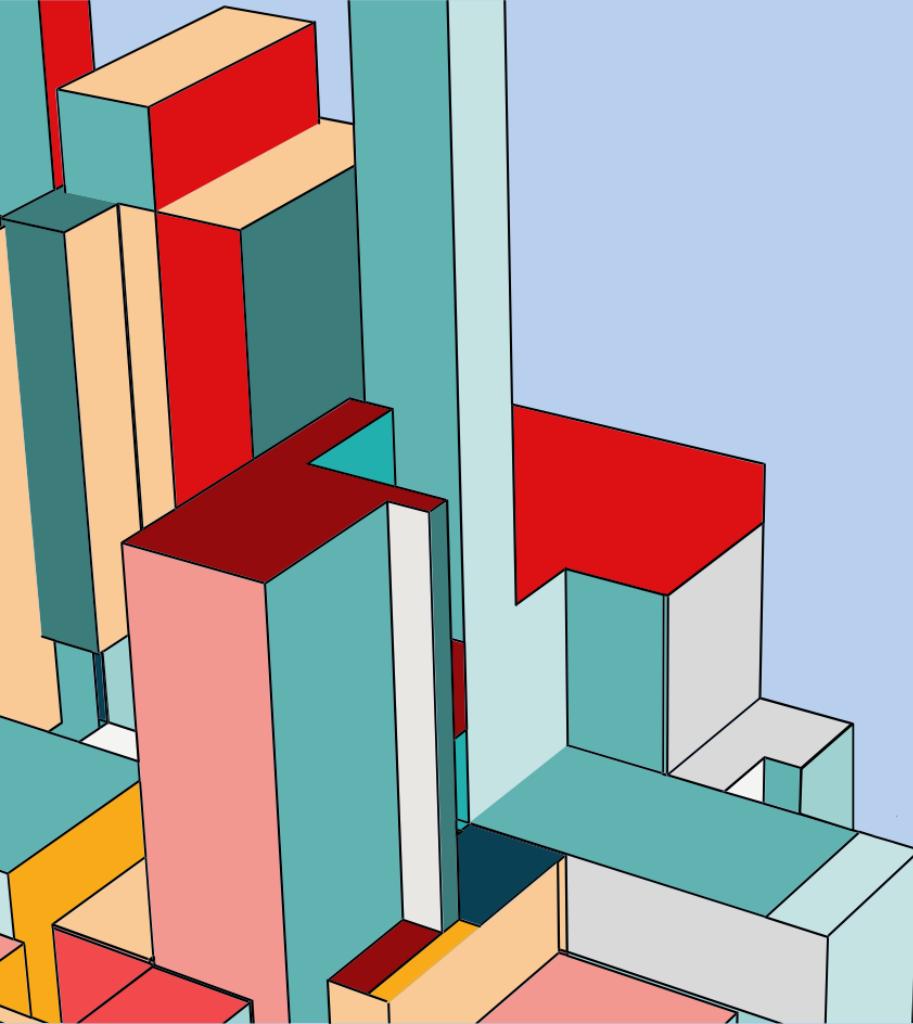
Таймауты мешают наполнению кэша



Таймауты мешают наполнению кэша







Прогрев кэша

Прогрев кэша

Прогрев кэша — это важная практика, особенно в высоконагруженных системах, где необходимо минимизировать задержки при доступе к данным

Прогрев кэша предполагает предварительную загрузку часто используемых данных в кэш до того, как они будут запрошены пользователями

Это позволяет уменьшить время отклика и снизить нагрузку на основное хранилище данных (например, базу данных)



Основные методы прогрева кэша

- Ручной прогрев (Manual Warm-up)
- Автоматический прогрев по расписанию (Scheduled Warm-up)
- Прогрев на основе аналитики (Analytics-based Warm-up)
- Ленивый прогрев (Lazy Warm-up / On-demand)
- Прогрев по событиям (Event-driven Warm-up)
- Иерархический прогрев (Hierarchical Warm-up)



Ручной прогрев (Manual Warm-up)

- **Описание:** разработчики вручную загружают данные в кэш при запуске системы или после инвалидации кэша
- **Когда используется:** в системах с предсказуемыми и стабильными паттернами использования данных
- **Плюсы:** простота реализации, полный контроль
- **Минусы:** требует ручного вмешательства, не подходит для динамических данных



Автоматический прогрев по расписанию (Scheduled Warm-up)

- **Описание:** используются cron-задачи или планировщики для регулярной загрузки данных в кэш
- **Когда используется:** когда известно, когда данные становятся актуальными (например, статистика по утрам)
- **Плюсы:** автоматизация, предсказуемость
- **Минусы:** неэффективен при непредсказуемом поведении пользователей



Прогрев на основе аналитики (Analytics-based Warm-up)

- **Описание:** система анализирует логи, поведение пользователей и предсказывает, какие данные могут понадобиться
- **Когда используется:** в системах с большими объемами данных и сложными паттернами использования
- **Плюсы:** адаптивность, высокая эффективность
- **Минусы:** сложность реализации, требуется инфраструктура для анализа



Ленивый прогрев (Lazy Warm-up / On-demand)

- **Описание:** данные загружаются в кэш только при первом запросе, но затем остаются в кэше для последующих обращений
- **Когда используется:** при непредсказуемом поведении пользователей
- **Плюсы:** экономия ресурсов, автоматическая адаптация
- **Минусы:** первый запрос может быть медленным



Прогрев по событиям (Event-driven Warm-up)

- **Описание:** прогрев происходит в ответ на события (например, обновление данных, инвалидация кэша)
- **Когда используется:** в системах с событийной архитектурой (event-driven)
- **Плюсы:** гибкость, актуальность данных
- **Минусы:** требует инфраструктуры обработки событий



Иерархический прогрев (Hierarchical Warm-up)

- **Описание:** сначала прогреваются наиболее важные или часто используемые данные, затем — менее приоритетные
- **Когда используется:** в системах с ограниченными ресурсами кэша
- **Плюсы:** эффективное использование ресурсов
- **Минусы:** требуется приоритезация данных



Архитектурные практики

□ Интеграция с системой инвалидации кэша

- При инвалидации кэша (например, после обновления данных) можно сразу запускать прогрев, чтобы избежать ситуаций с холодным кэшем

□ Использование метрик и мониторинга

- Системы прогрева кэша должны быть интегрированы с мониторингом, чтобы отслеживать эффективность прогрева и при необходимости корректировать стратегию

□ Машинное обучение для прогнозирования

- В продвинутых системах используются ML-модели для предсказания популярных данных и их прогрева заранее



Рекомендации по выбору метода прогрева

№	Сценарий	Рекомендуемый метод
1	Стабильные паттерны	Ручной / по расписанию
2	Непредсказуемый трафик	Ленивый / аналитика
3	Событийная архитектура	Прогрев по событиям
4	Ограниченные ресурсы	Иерархический
5	Высокая нагрузка	Автоматический + аналитика



Примеры инструментов и технологий прогрева

- ❑ **Redis** — поддерживает TTL, инвалидацию и может использоваться с механизмами прогрева
- ❑ **Apache Kafka** — может использоваться для событийного прогрева
- ❑ **Prometheus + Grafana** — для мониторинга эффективности кэша
- ❑ **Elasticsearch** — для анализа логов и прогнозирования запросов
- ❑ **Celery, Quartz, Cron** — для планирования задач прогрева



Кэширование данных в MSA

Роль кеша в микросервисной архитектуре:

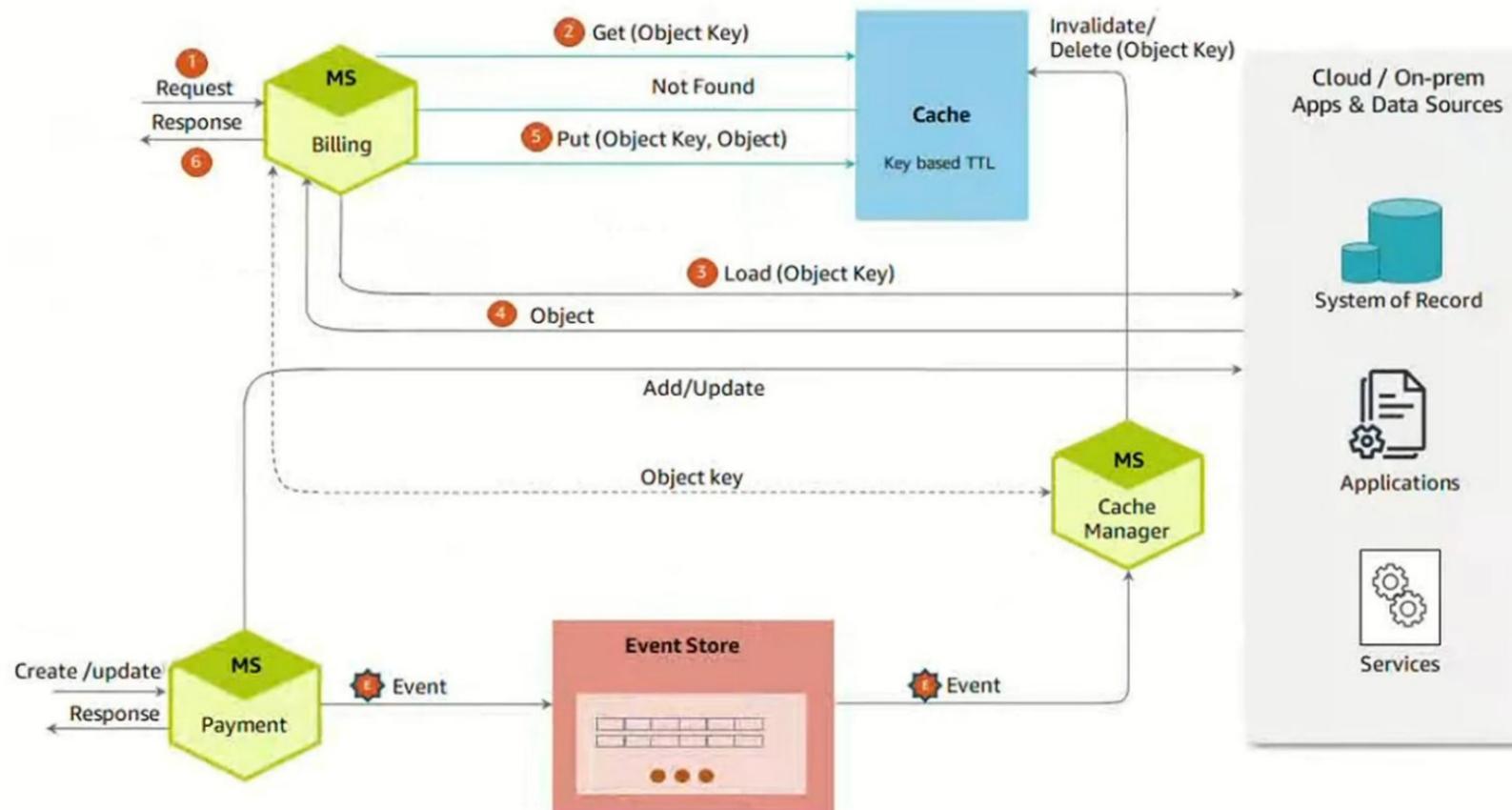
- ✓ Размещение кеша на уровне микросервисов снижает задержку
- ✓ Кеш позволяет реже обращаться к сервису
- ✓ Актуально для сценариев с высокой задержкой из-за объёмов данных и пропускной способности сети

Возможные подходы:

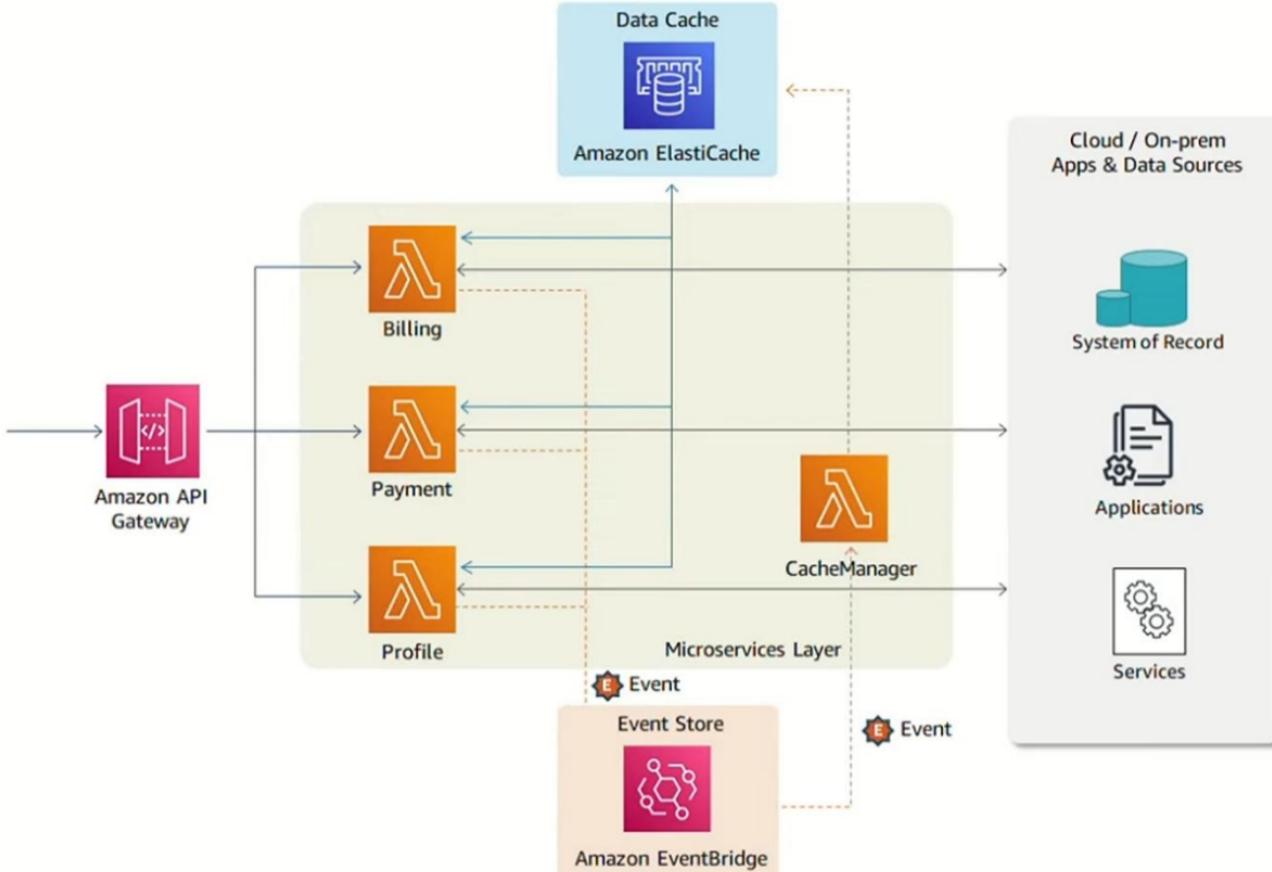
- кеш по требованию, чтобы сократить объём вызовов (On-Demand Caching)
- упреждающее кэширование больших объёмов данных (Preemptive Caching)



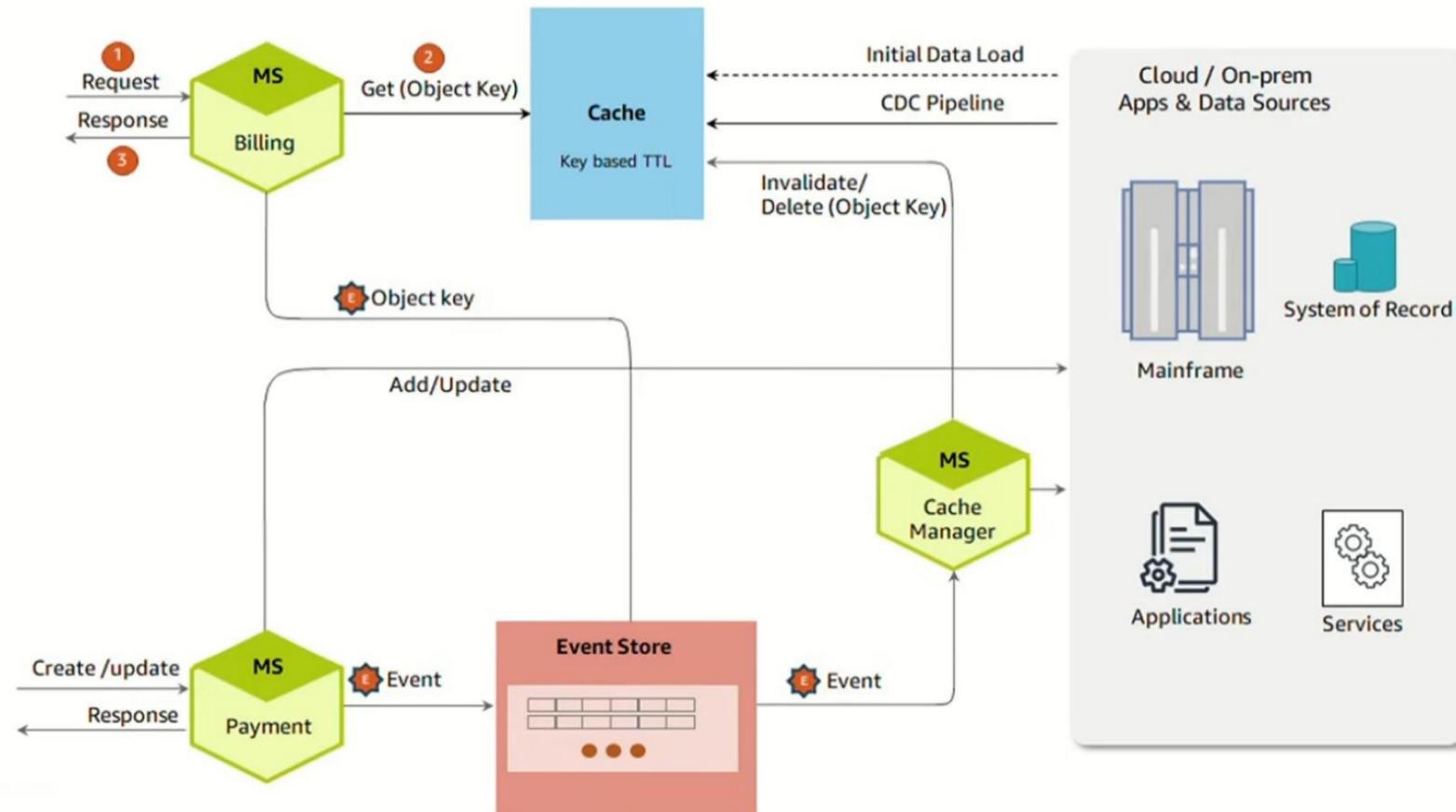
Кеш по требованию (On-Demand Caching)



Кеш по требованию (On-Demand Caching)



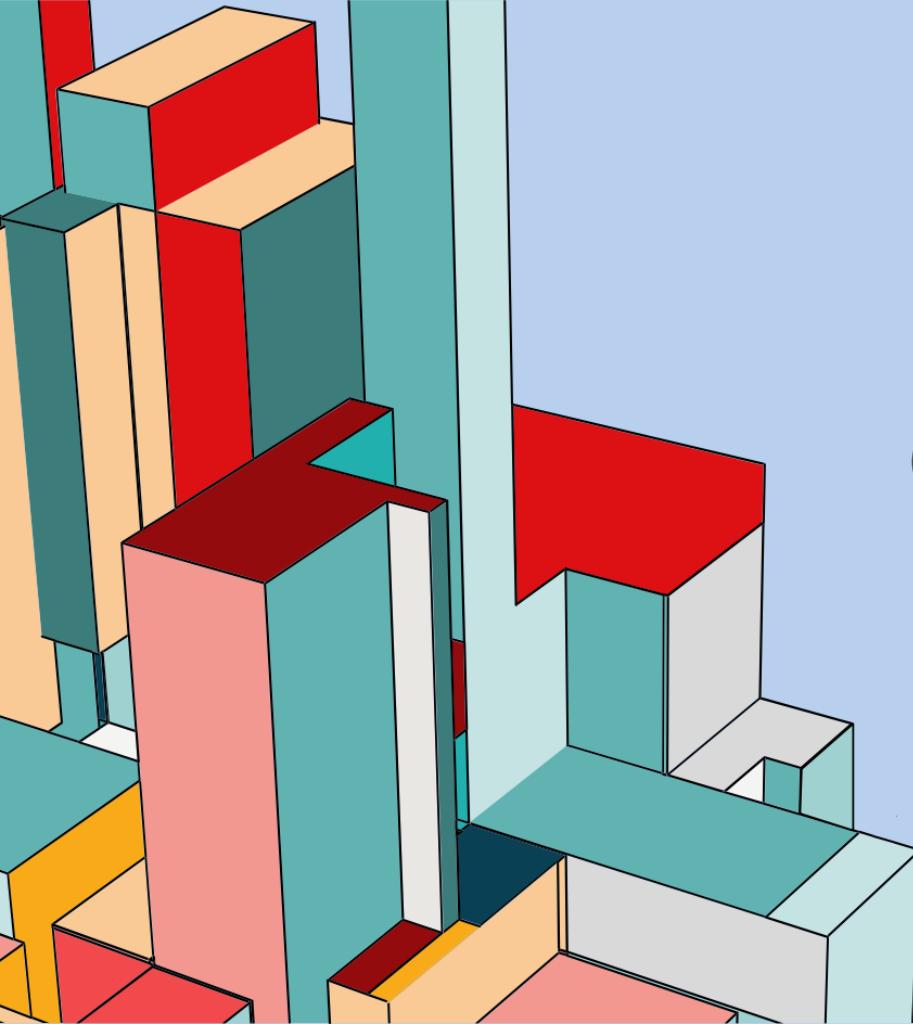
Упреждающее кеширование (Preemptive Caching)



Инструменты кэширования

- Redis (REmote DIctionary Server in full)
- Memcached
- Apache Ignite
- Couchbase Server
- Hazelcast IMDG (In-Memory Data Grid)
- Mcrouter
- Varnish Cache
- Squid Caching Proxy
- NGINX (Engine-X)
- Apache Traffic Server



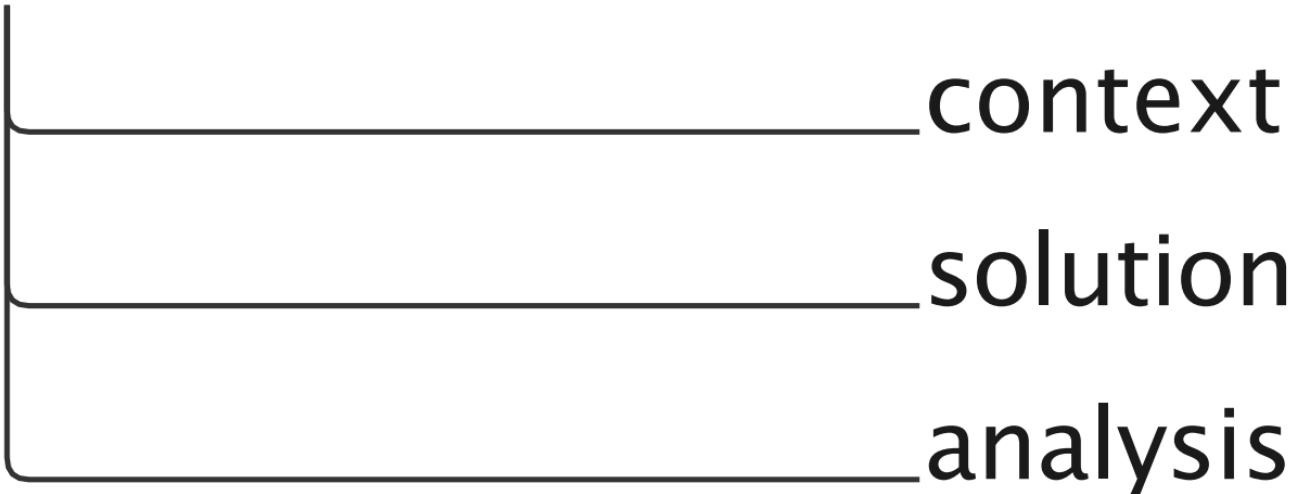


CQRS

(прагматично)

CQRS — это pattern?

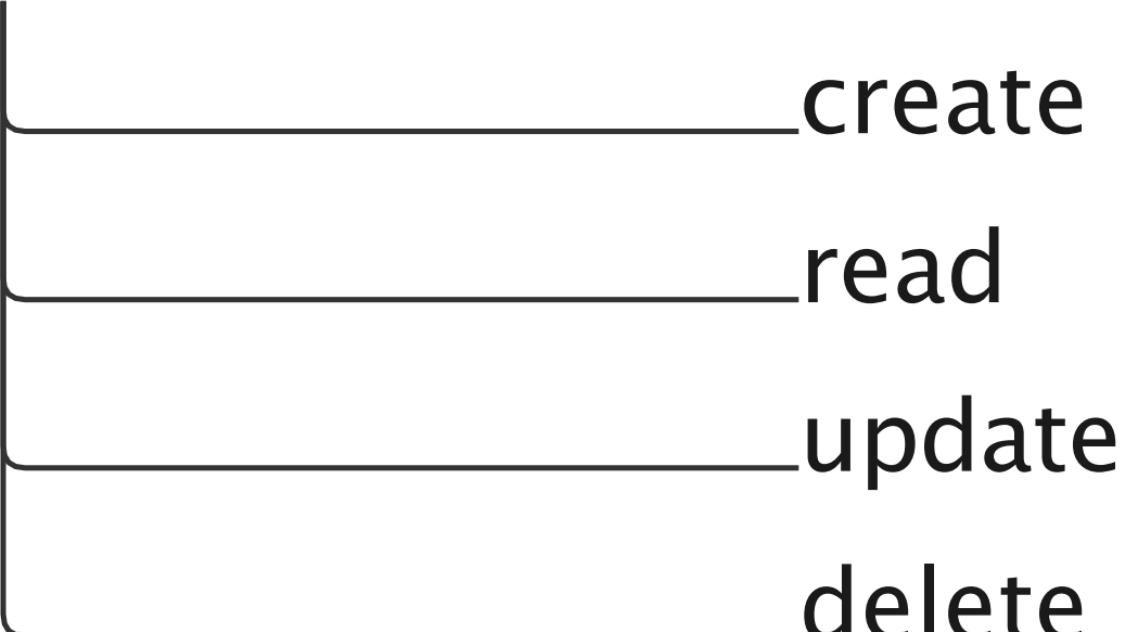
pattern



context

software =
data + operations

operations



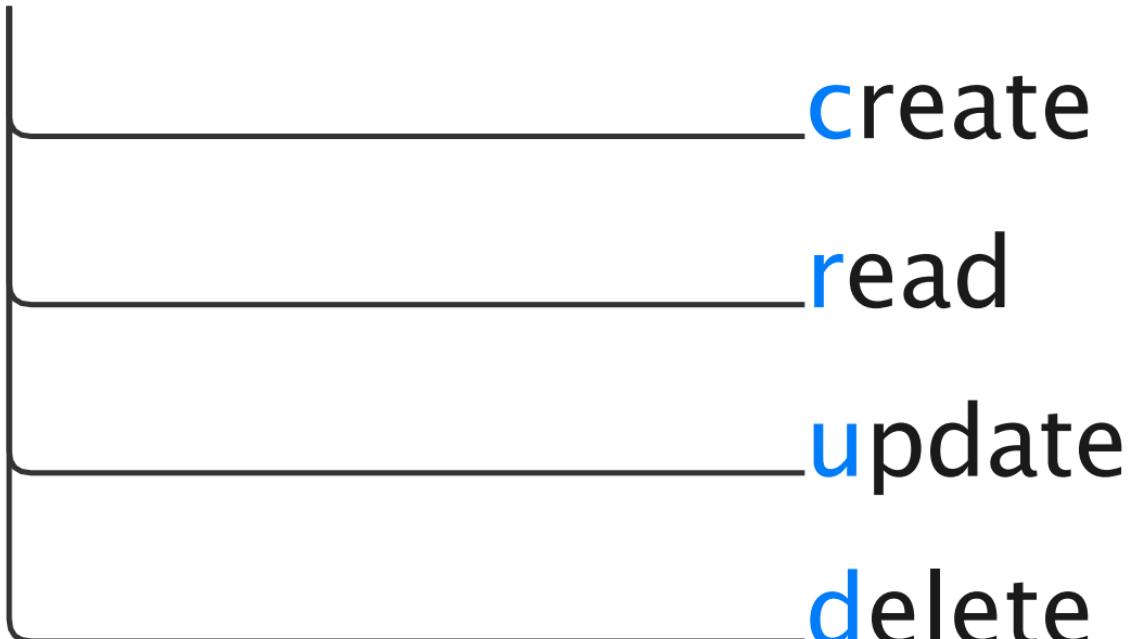
create

read

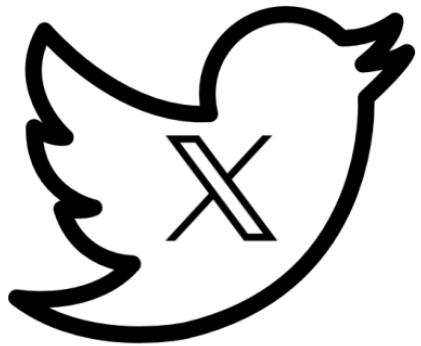
update

delete

operations



CRUD → cRud



~49% акаунтов
публикуют <5 твитов в
месяц

~30.4 минуты в день на
пользователя

eCom

READ

полистали каталог



READ

посмотрели товар



WRITE

добавили в корзину



CRUD → cRud

CUD

бизнес логика

работает в рамках aggregate / entity

нетривиальная валидация

транзакционность & корректность

R

несколько представлений
одних данных

пересекаем границы
aggregates & entities

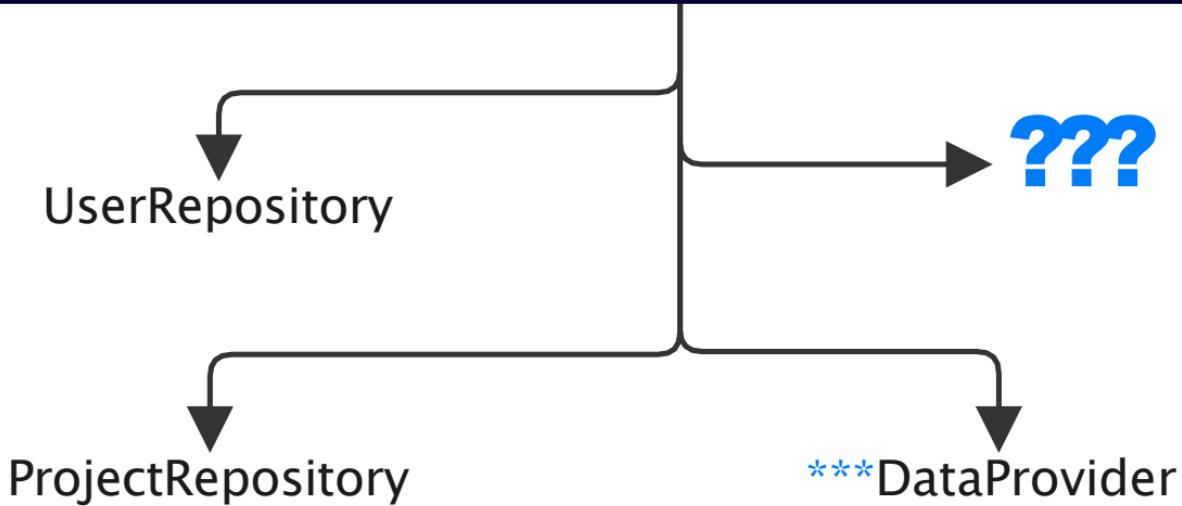
сложные запросы

производительность

CUD **vs** R

```
public class UserRepository: IUserRepository {  
    ...  
  
    public void Create(User user) {...}  
    public void Update(User user) {...}  
    public void Delete(User user) {...}  
  
    public User GetById(Guid id) {...}  
    public User GetRecentActive() {...}  
    public User? TryFind(string searchString) {...}  
    public User[] GetAllOnFreePlan() {...}  
    public User[] GetAllWithoutBillingInformation() {...}  
    ...  
    // 100500 methods later  
    public User[] GetWhoCommentedArticleInProject(Guid projectId) {...}  
}
```

```
public User[] GetWhoCommentedArticleInProject(Guid projectId)  
{  
    ...  
}
```

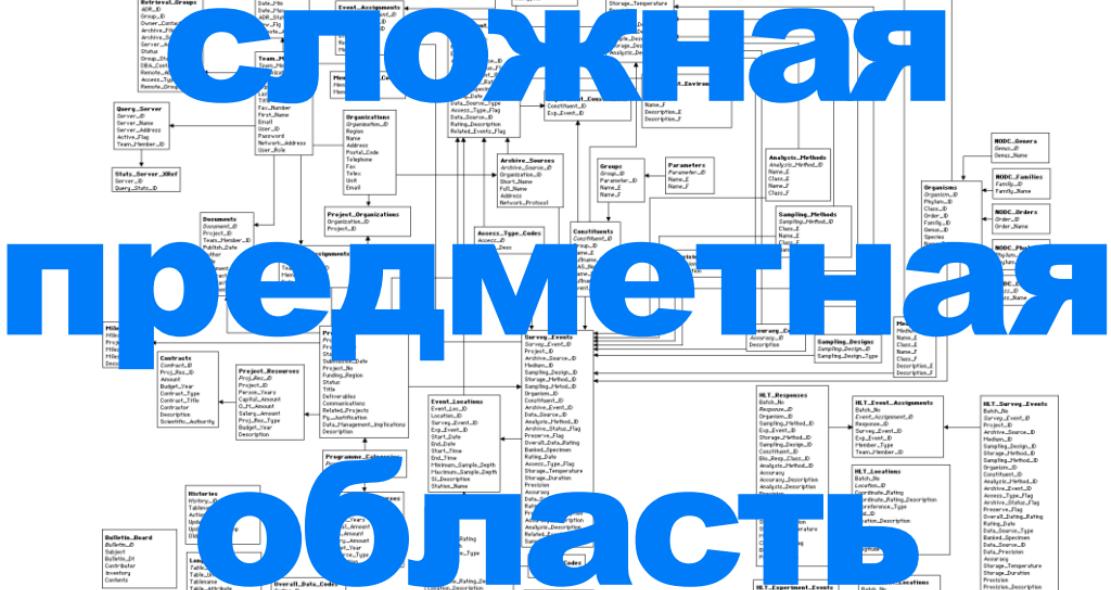


mapping

```
public class UserToShortViewMapper: IMapper<User, UserShortView>
{
    public UserShortView Map(User obj) { ... }
}
```

UserShortView, UserView, UserDetailView, ..., ***View

complexity



```
SELECT t.line_id AS LineId, ...
      sn.item_transaction_id AS SerialItemTransactionId, sn.serial_number AS SerialNumber,
      sl.id AS SaleShipmentNoteLineId, sl.item_id SaleShipmentNoteItemId,
      sh.id AS SaleShipmentNoteHeaderId, ...
      ol.id AS SaleOrderLineId, ...
      sc.lines AS ConfigurationsLines
  FROM unnest(@line_id, @link_id, @item_transaction_id, @link_serial_number,
             @link_item_id, @serial_number_item_transaction_id)
    AS t(line_id, link_id, item_transaction_id, link_serial_number,
          link_item_id, serial_number_item_transaction_id)
 LEFT JOIN read.serial_numbers AS sn
    ON (t.serial_number_item_transaction_id IS NOT NULL AND ... )
    OR (serial_number_item_transaction_id IS NULL AND ... )
 LEFT JOIN read.shipment_note_lines AS sl
    ON (t.serial_number_item_transaction_id IS NOT NULL AND ... )
    OR (t.serial_number_item_transaction_id IS NULL
        AND sn.item_transaction_id = sl.item_transaction_id
        AND t.link_item_id = sl.item_id)
 LEFT JOIN read.shipment_note_headers AS sh ON sl.header_id = sh.id
 LEFT JOIN read.order_lines AS ol ON sl.item_transaction_id = ol.item_transaction_id
 LEFT JOIN read.serial_number_configurations AS sc ON sl.item_transaction_id = sc.item_transaction_id
 LEFT JOIN ...
```

```
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    ...
);

CREATE TABLE order_lines (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    order_id UUID NOT NULL,
    ...
    CONSTRAINT fk_order
        FOREIGN KEY(order_id)
            REFERENCES orders(id)
);

```

дениормализация

cache hell



```
CREATE TABLE orders (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    lines JSONB,
    ...
);
```

complexity

performance

solution

CUD **vs** R

CUD

бизнес логика

работает в рамках aggregate / entity

нетривиальная валидация

транзакционность & корректность

R

несколько представлений
одних данных

пересекаем границы
aggregates & entities

сложные запросы

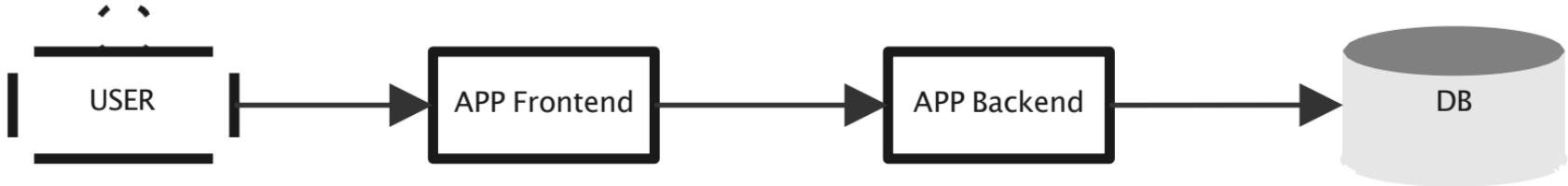
производительность

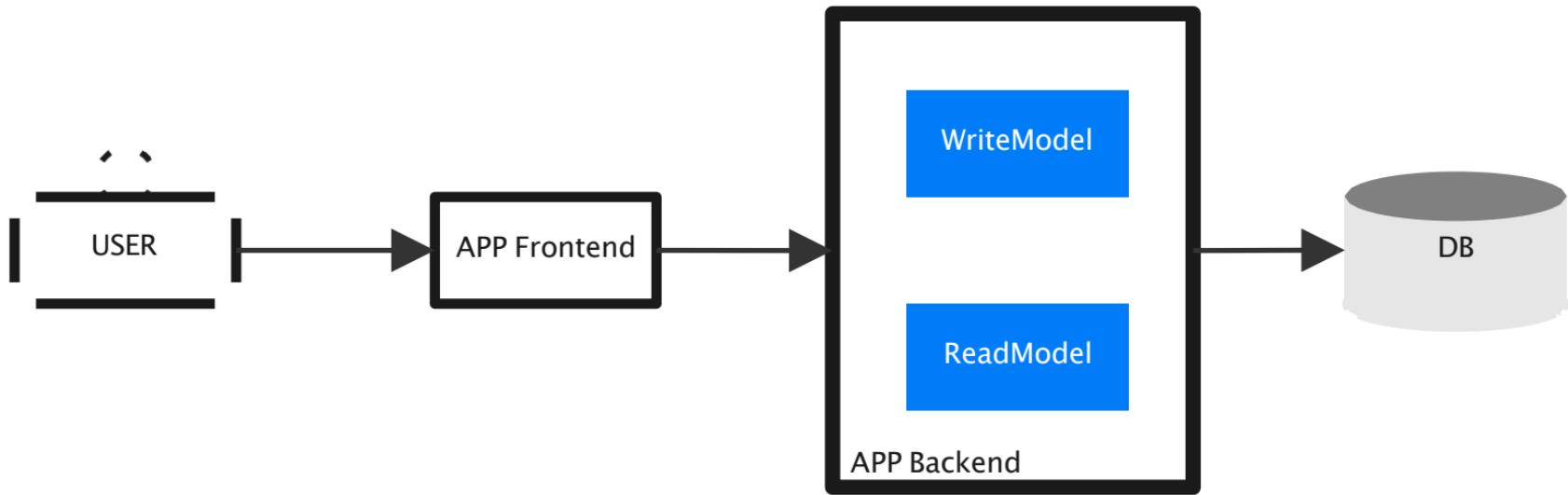
CUD vs R



CUD & R

RW Model →
R Model \cup **W Model**





CQRS v1.0 #1

Write Model

FluentValidator+MediatR+EF+AutoMapper

(не является рекомендацией)

clean architecture, DDD, rich domain, ...

DAL: repository & co

Read Model

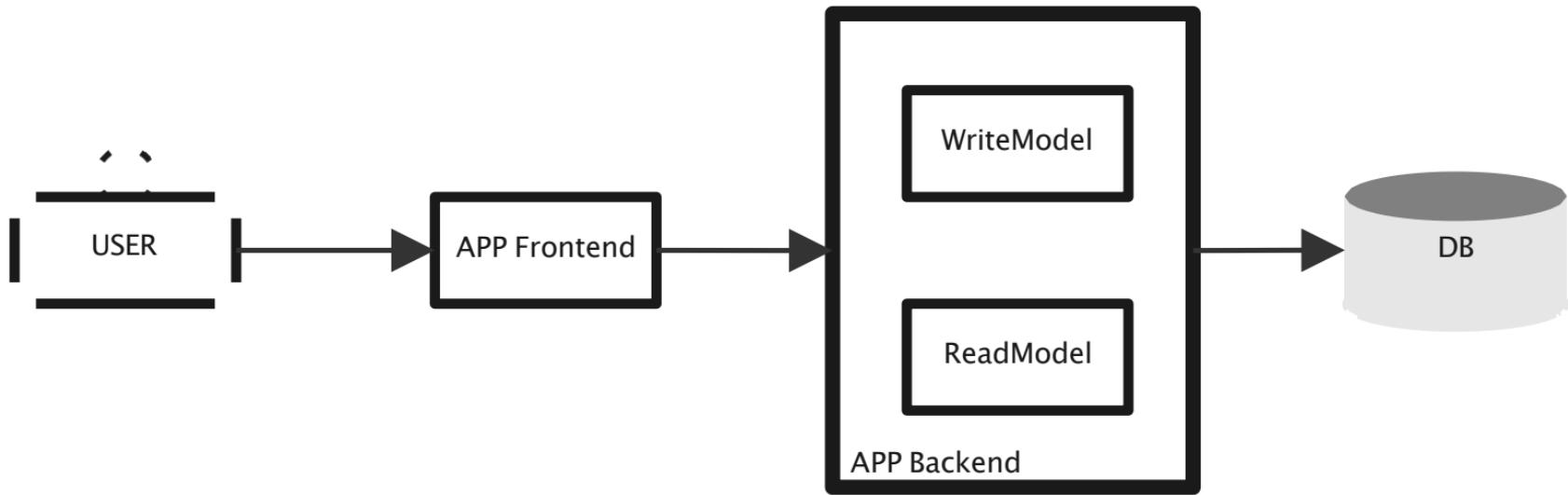
Dapper +SqlKata | Linq2DB

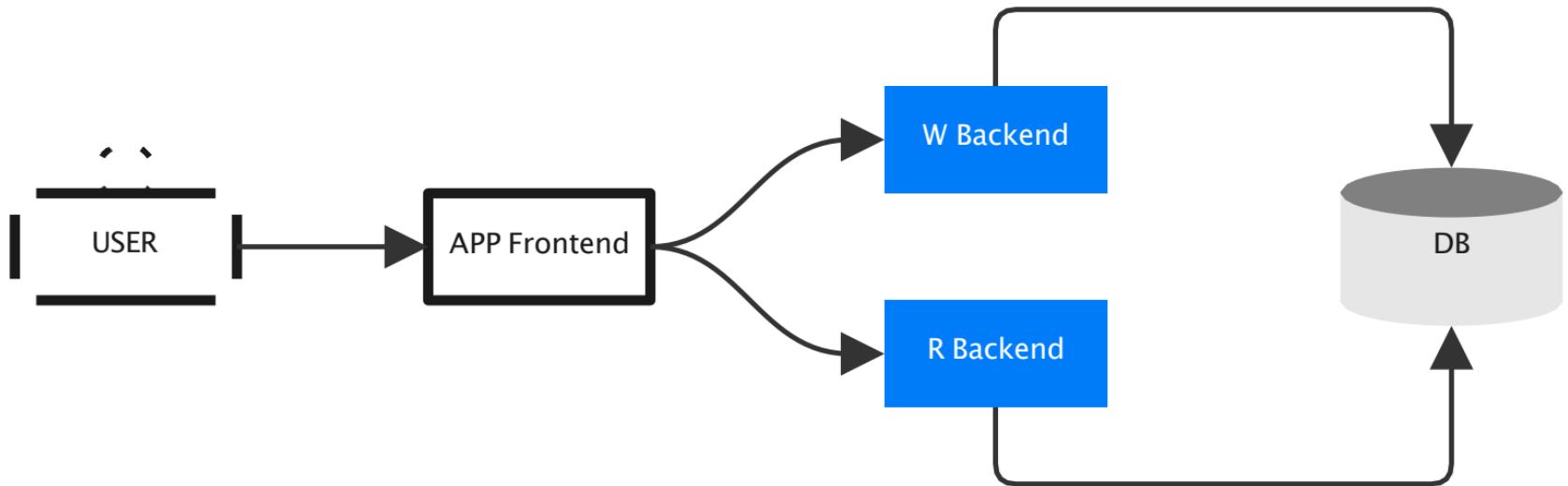
minimum abstraction, anemic model,

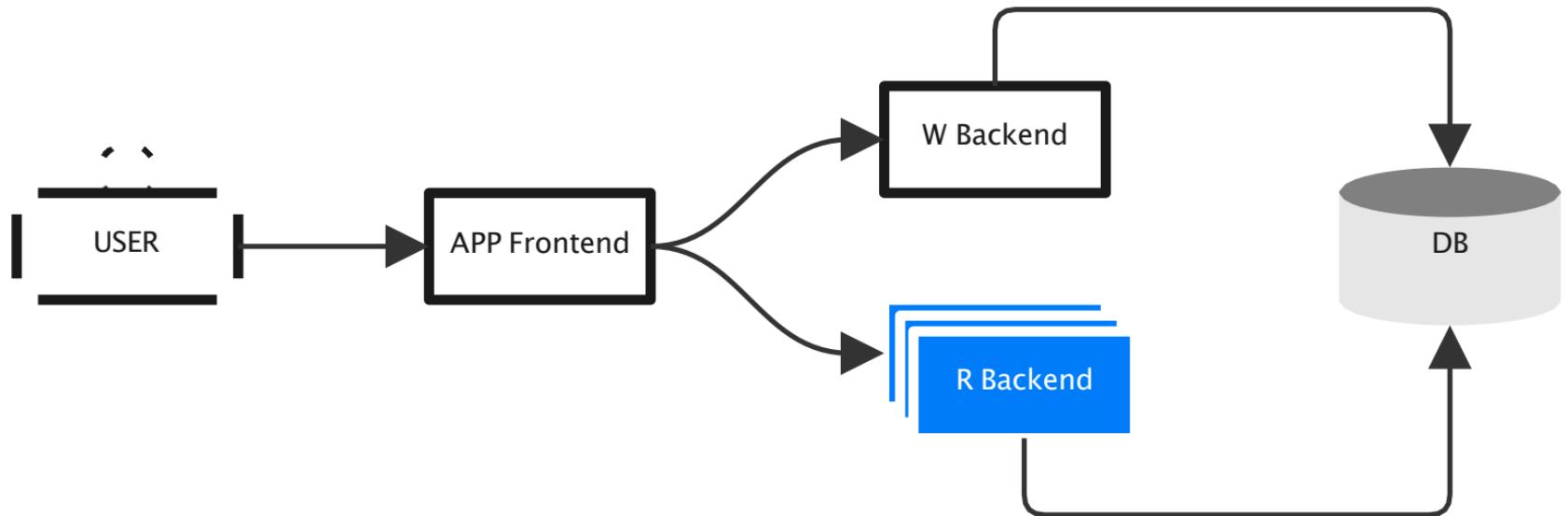
no DAL, ...

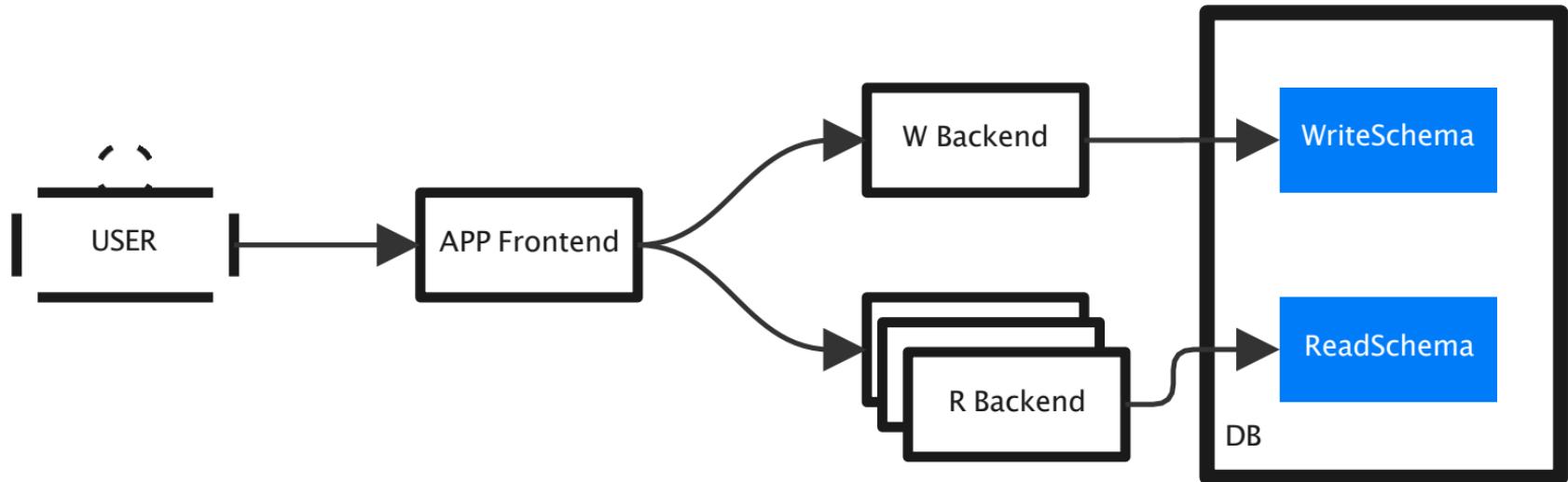
data driven architecture

~~complexity~~
performance



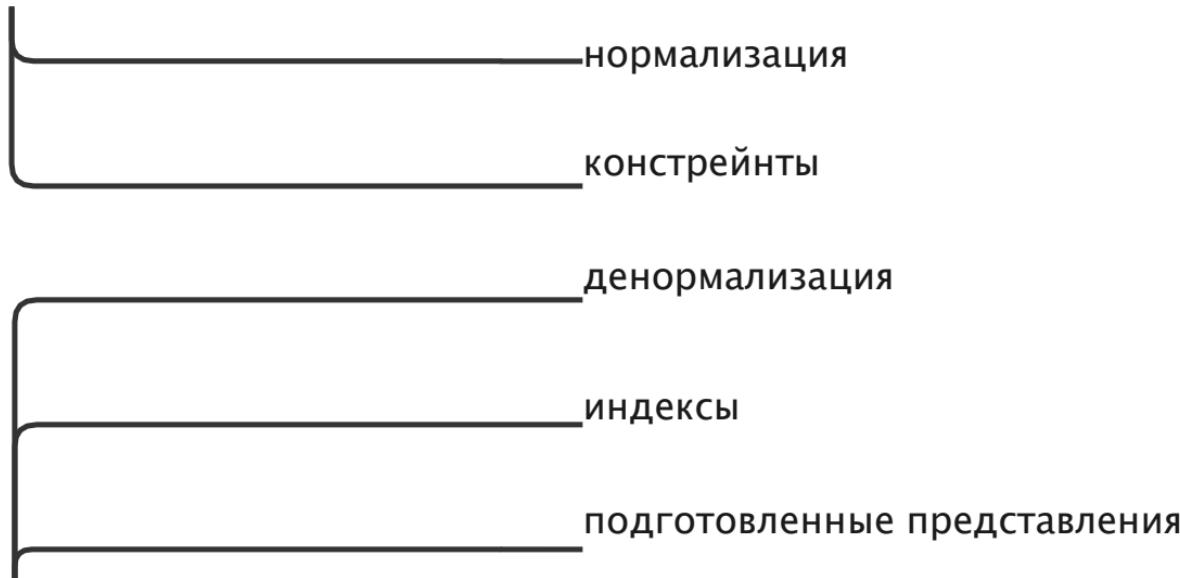




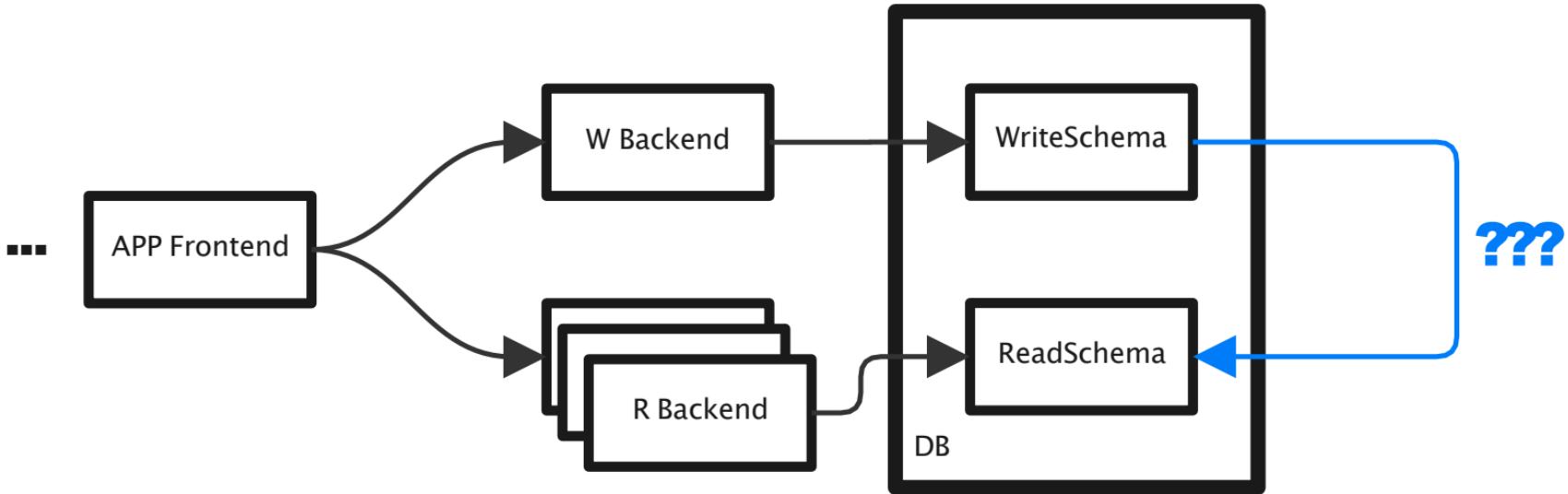


CQRS #2

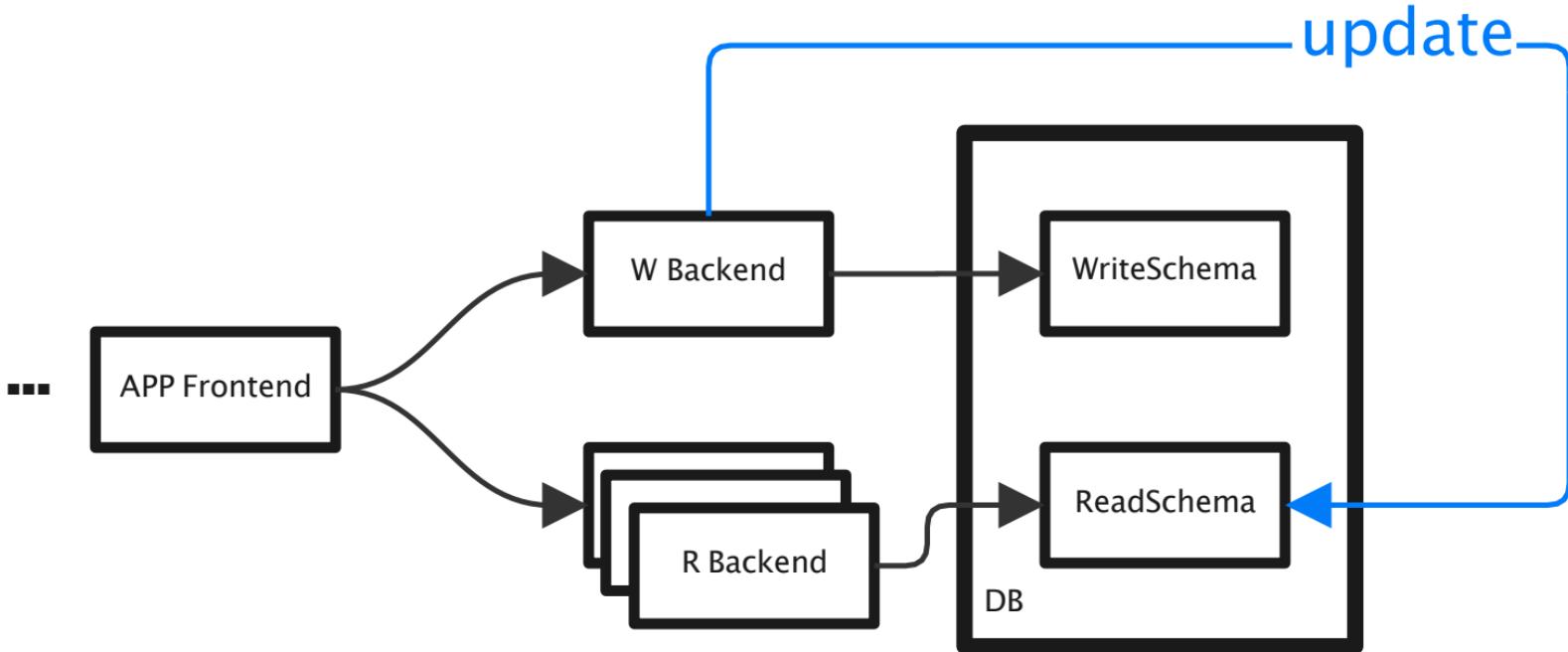
Write Schema



Read Schema



sync *vs* async



```
public sealed class ChangeUserNameCommandHandler : ICommandHandler<ChangeUserNameCommand, Result>
{
    private readonly IUserRepository _userRepository;
    private readonly IChangeNotifier _changeNotifier;

    ...

    public Result Handle(ChangeUserNameCommand command)
    {
        var user = _userRepository.GetById(command.UserId);

        if (!user.CanChangeName() || !IsValidNewName(command.NewName))
        {
            return Result.Error(...);
        }

        user.Name = command.NewName;
        _changeNotifier.Notify(new UserChangeNotification(command.UserId));

        return Result.Ok();
    }
}
```

```
public sealed class ReadModelUpdateDecorator<TCommand, TResult> : ICommandHandler<TCommand, TResult>
{
    private readonly ICommandHandler<TCommand, TResult> _decoratedHandler;
    private readonly IChangeNotificationsProvider _notificationsProvider;
    private readonly IChangeNotificationDispatcher _dispatcher;

    ...

    public TResult Handle(TCommand command)
    {
        using var transaction = new TransactionScope();

        var result = _decoratedHandler.Handle(command);
        foreach (var notification in _notificationsProvider.GetNotifications())
        {
            _dispatcher.Handle(notification);
        }

        transaction.Complete();
        return result;
    }
}
```

```
public sealed class UserShortViewUpdater : IChangeNotificationHandler<UserChangeNotification>
{
    ...

    public void Handle(UserChangeNotification notification)
    {
        // update read model here
        ...
    }
}
```

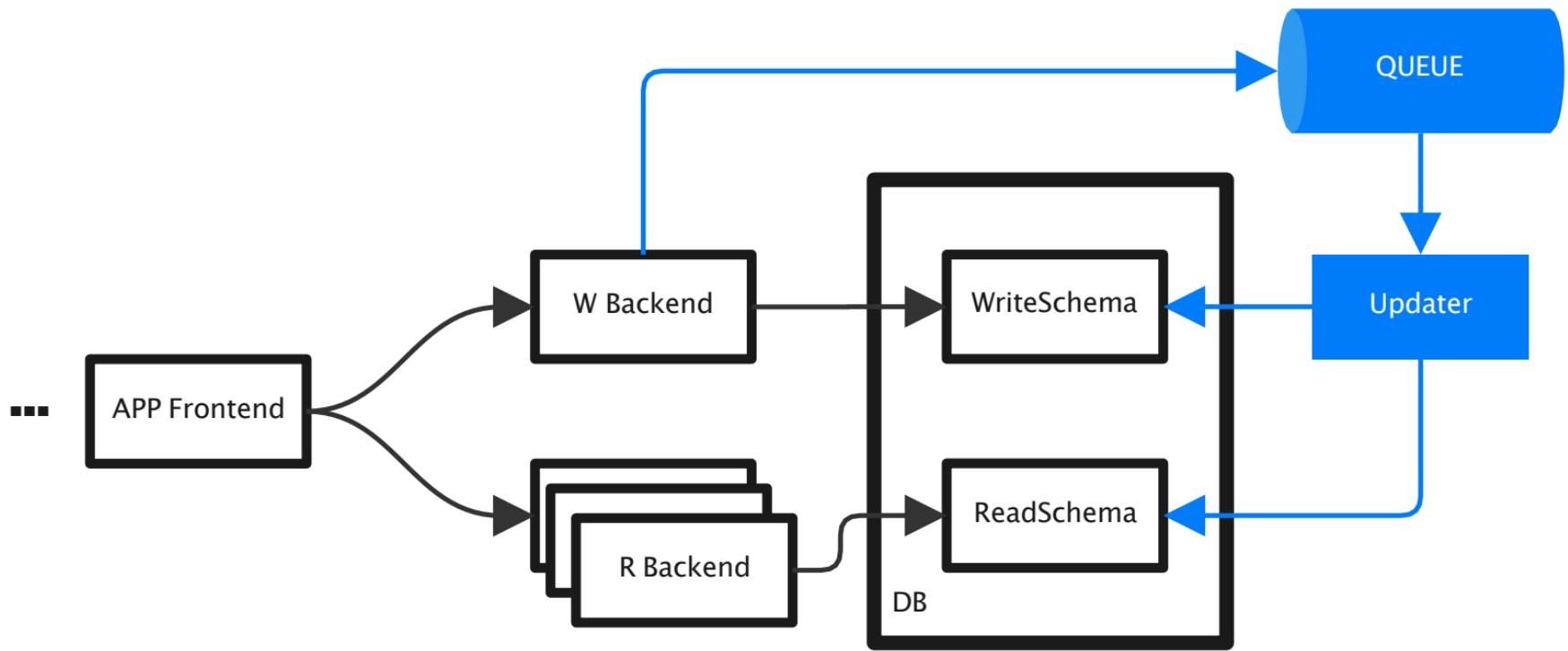
```
_dbContext
    .ChangeTracker
    .Entries()
    .Where(e => e is IAggregate)
    .Select(e => (e as IAggregate).ToChangedNotification())
    .ToList()
    .ForEach(cn => _dispatcher.Handle(cn));
```

sync

write операции становятся медленнее

дополнительная сложность во write model

write & read модели консистентны



async

write операции не теряют в производительности
значительно

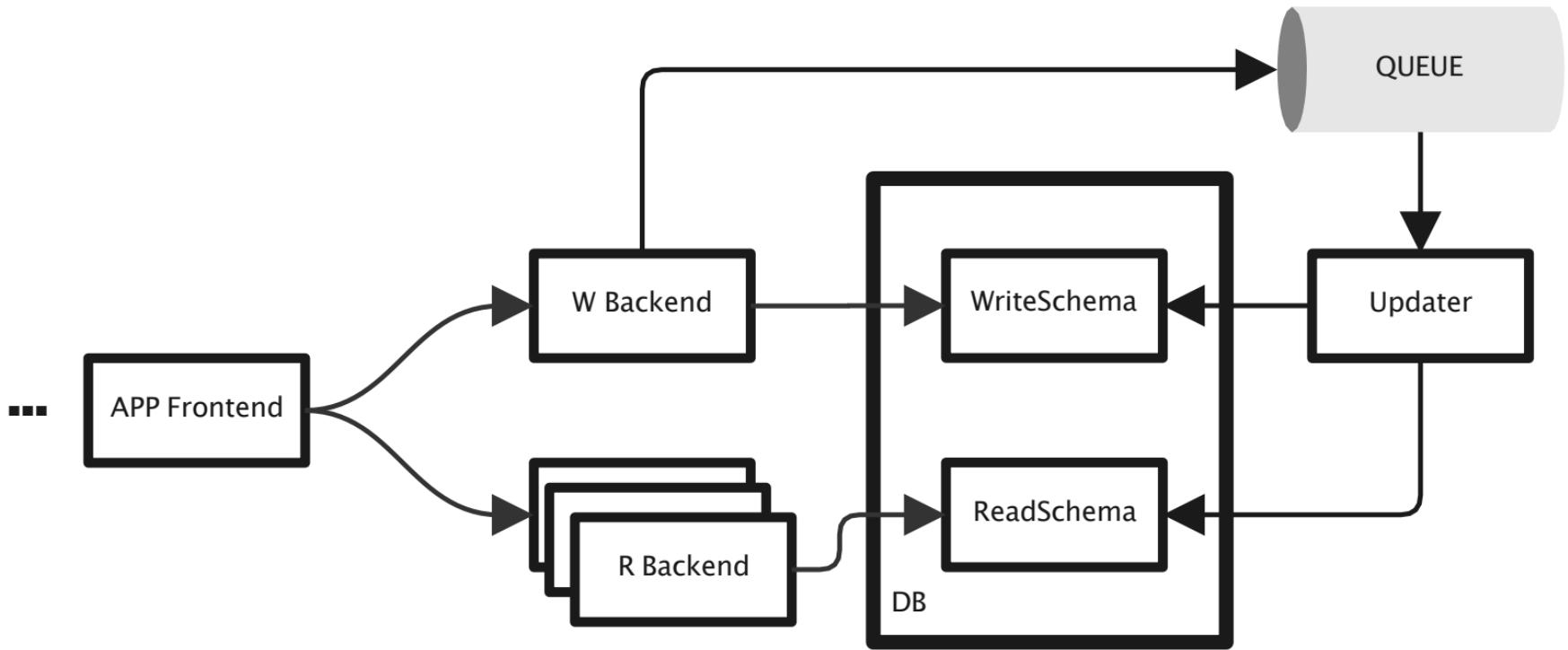
дополнительная сложность во write model

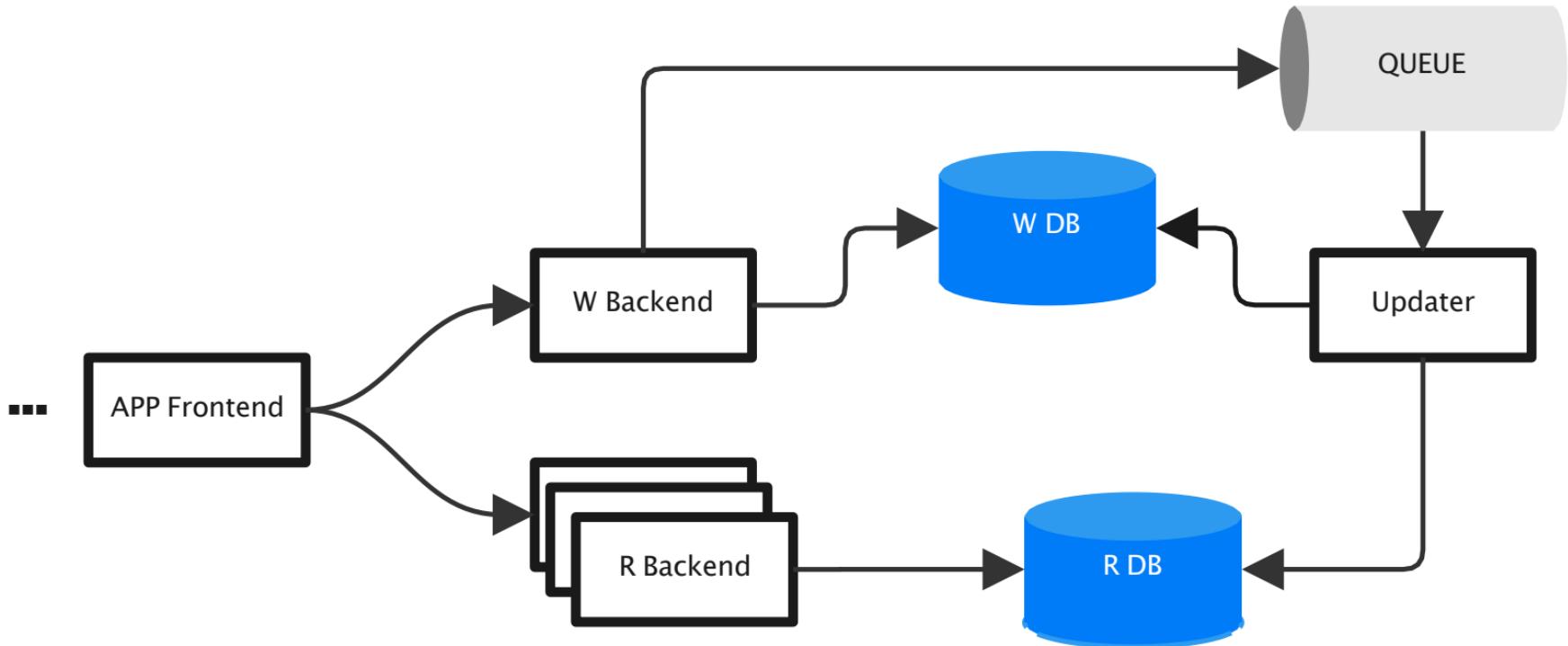
дополнительная архитектурная сложность и новые
точки отказа

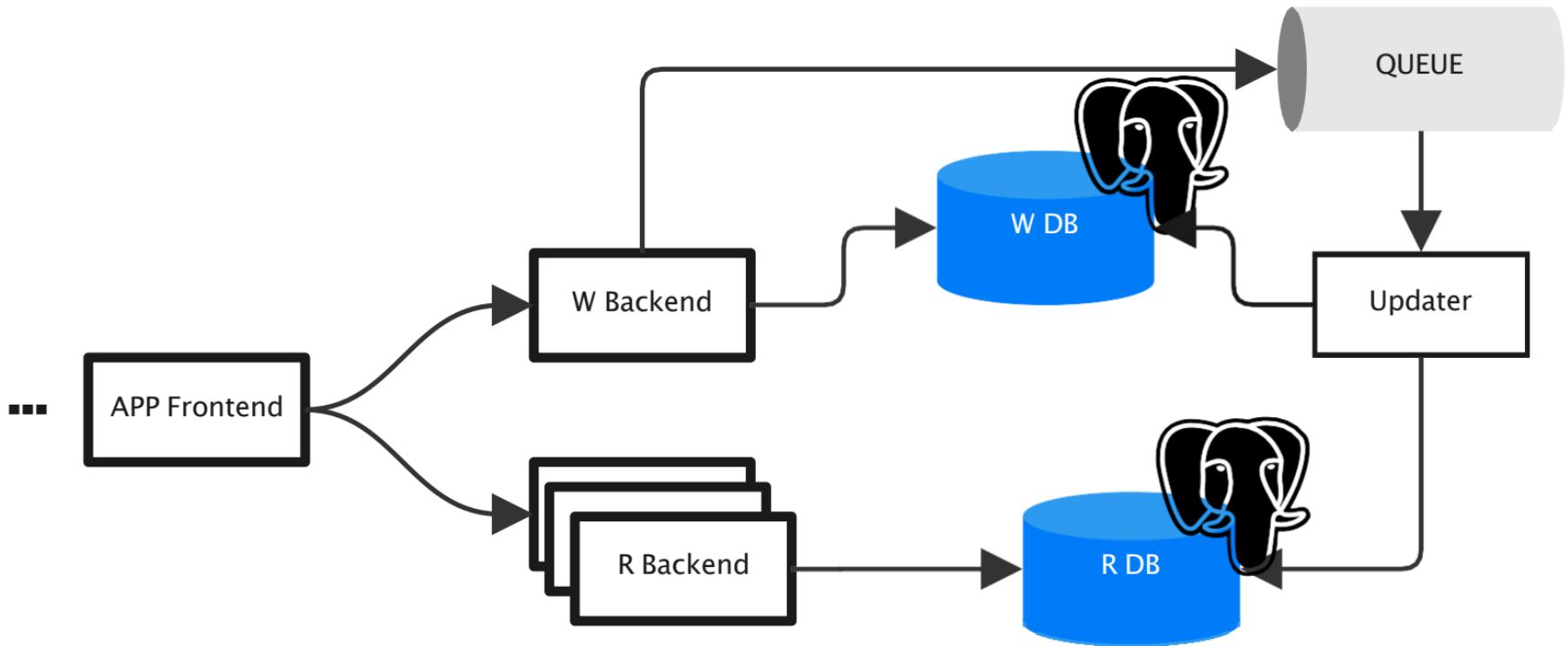
eventually consistency

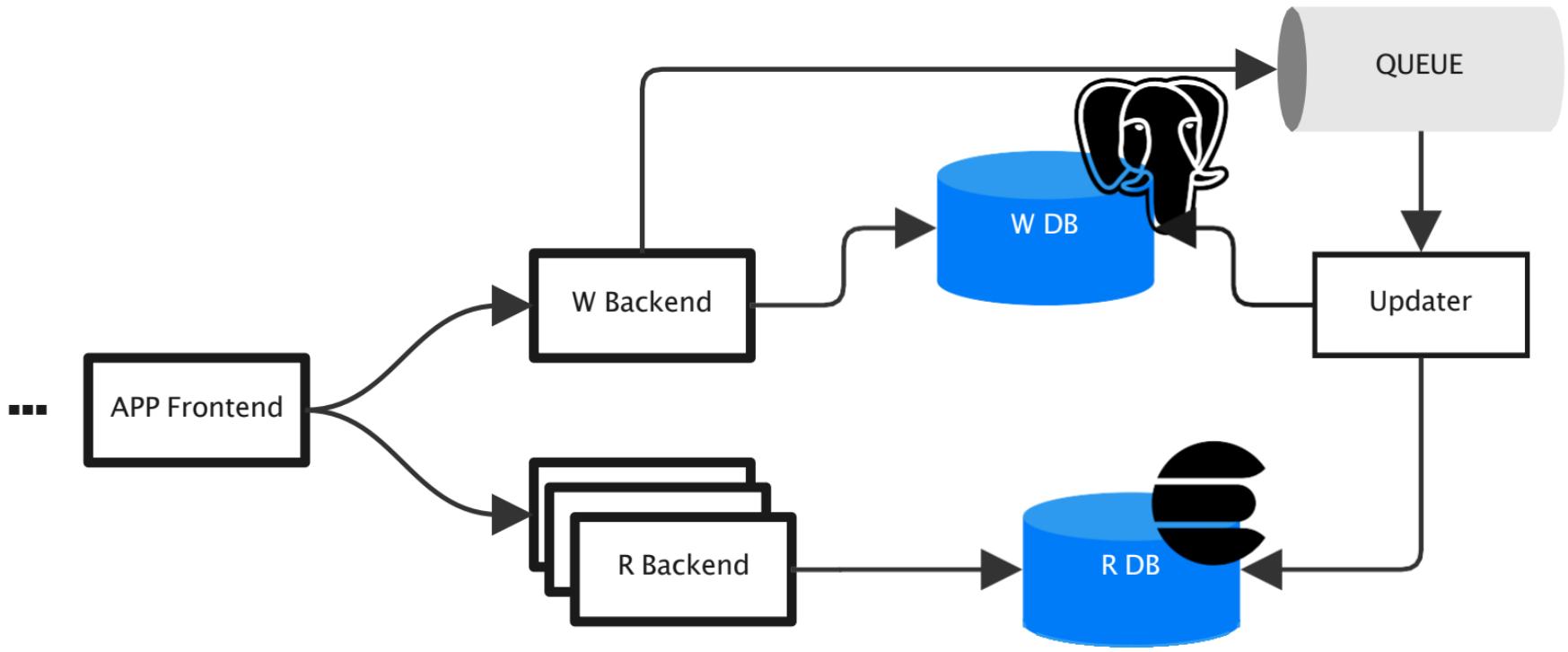
~~complexity~~
~~performance~~

нужно больше
performance









CQRS #3

sync –фантастический вариант
eventually consistency –данность

~~complexity~~
~~performance~~

command

& query?

```
public sealed record ChangeUserNameCommand(Guid UserId, string NewName) : ICommand;
...
public sealed class ChangeUserNameCommandHandler : ICommandHandler<ChangeUserNameCommand, Result>
{
    ...
    public Result Handle(ChangeUserNameCommand command)
    {
        var user = _userRepository.GetById(command.UserId);

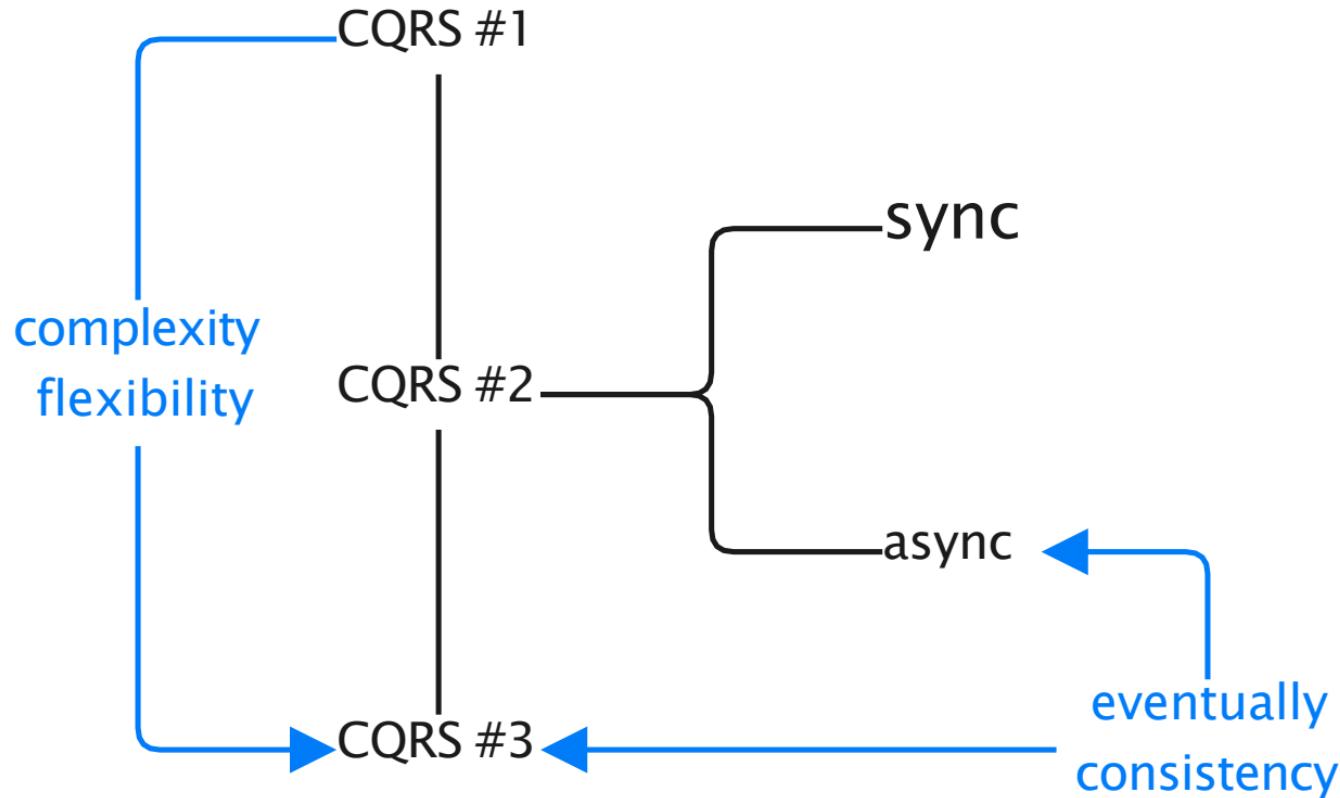
        if (!user.CanChangeName() || !isValidNewName(command.NewName))
        {
            return Result.Error(...);
        }

        user.Name = command.NewName;
        _changeNotifier.Notify(new UserChangeNotification(command.UserId));

        return Result.Ok();
    }
}
```

- один сценарий = один handler
- единство ответственности
- удобное управление зависимостями
- удобно реализовывать сквозную логику
- управление сложностью
- это не главное в CQRS

analysis



$\text{fcmp}(\text{rwmodel}) < \text{fcmp}(\text{rmodel}) \cup \text{fcmp}(\text{wmodel})$

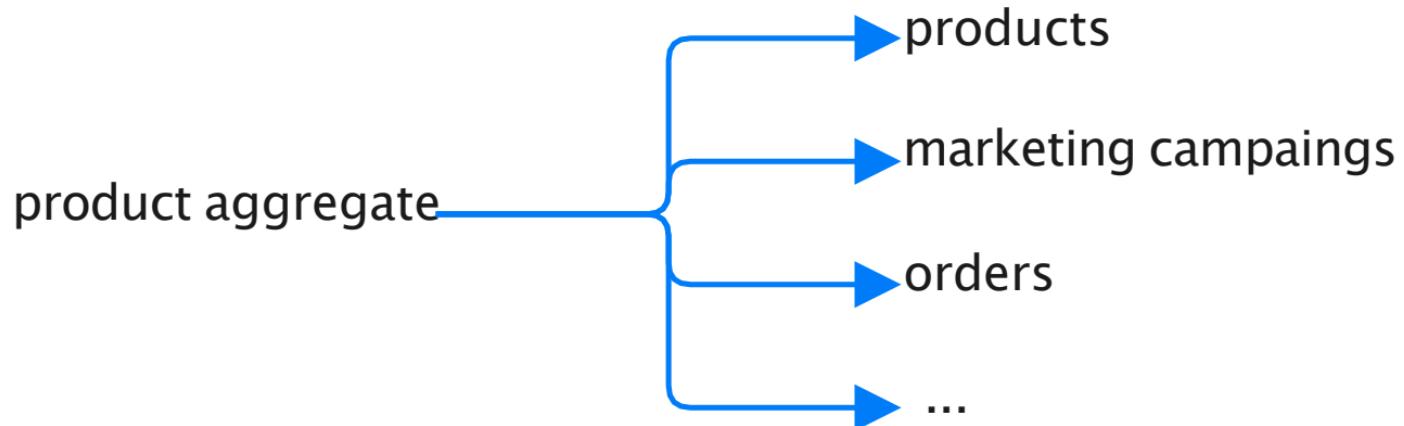
$\text{fcmp}(\text{rwmodel}) > \text{fcmp}(\text{rmodel})$

$\text{fcmp}(\text{rwmodel}) > \text{fcmp}(\text{wmodel})$

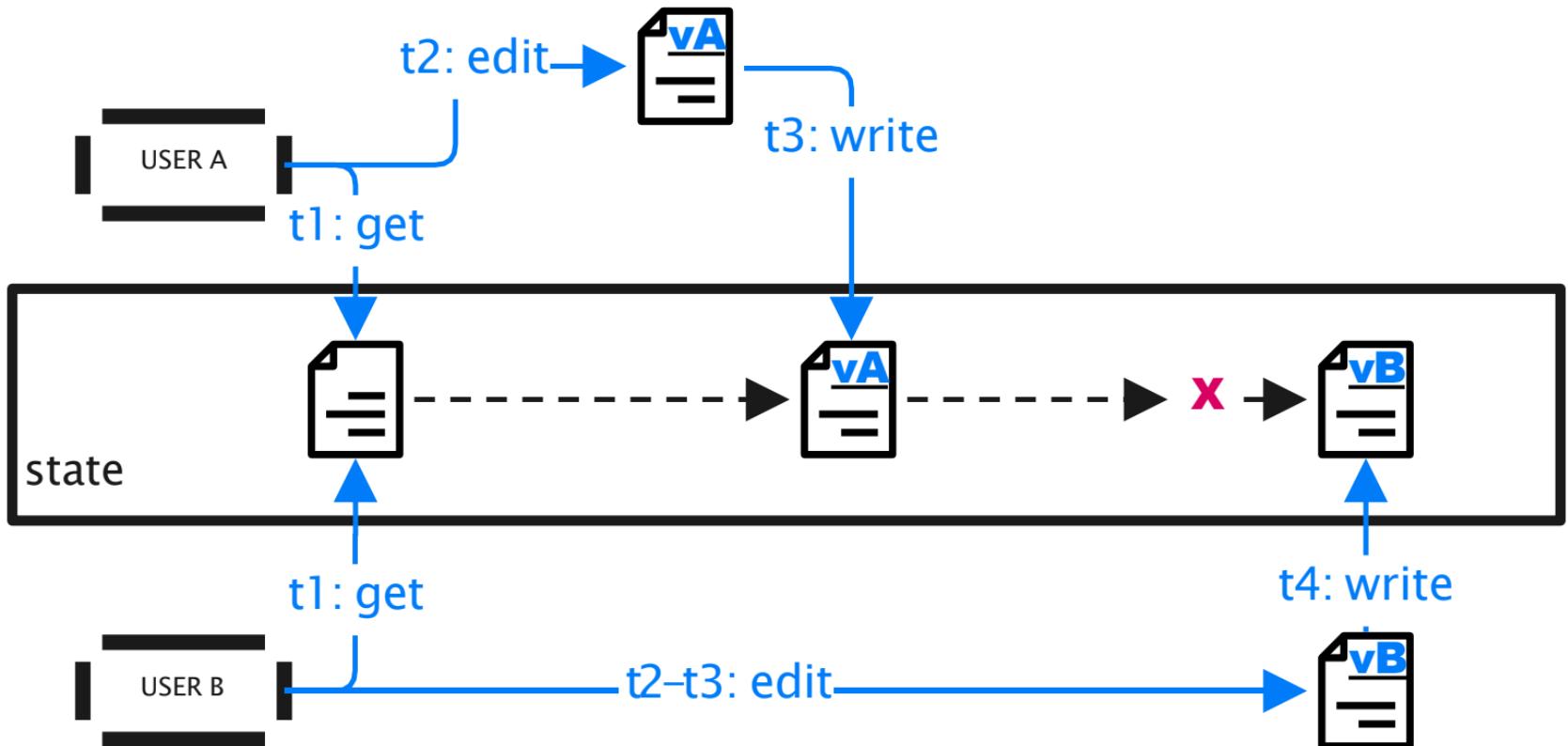
fcmp -complexity

```
public class UserRepository: IUserRepository {  
    ...  
  
    public void Create(User user) {...}  
    public void Update(User user) {...}  
    public void Delete(User user) {...}  
  
    public User GetById(Guid id) {...}  
    public User GetRecentActive() {...}  
    public User? TryFind(string searchString) {...}  
    public User[] GetAllOnFreePlan() {...}  
    public User[] GetAllWithoutBillingInformation() {...}  
    ...  
    // 100500 methods later  
    public User[] GetWhoCommentedArticleInProject(Guid projectId) {...}  
}
```

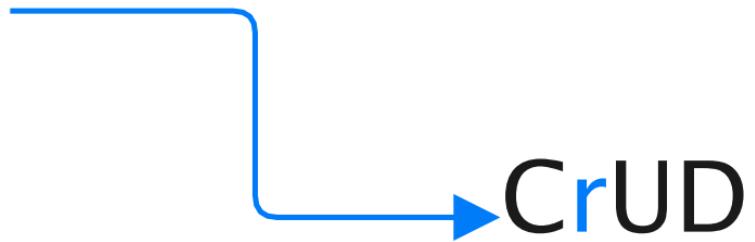
обновление read model может быть далеко не простым



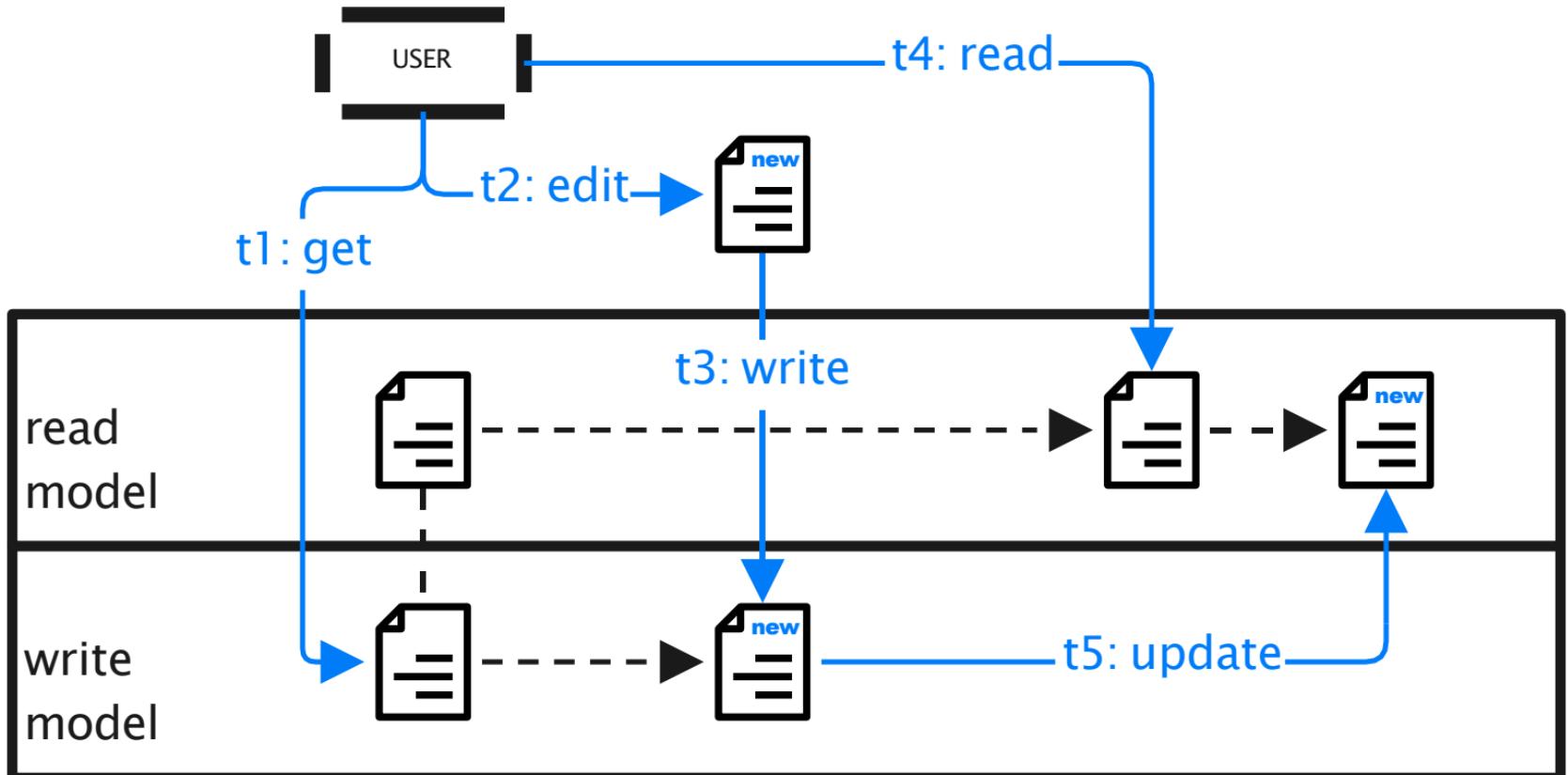
eventually consistency

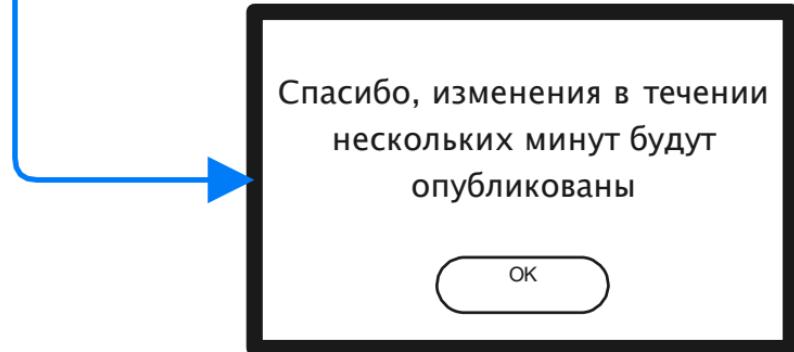
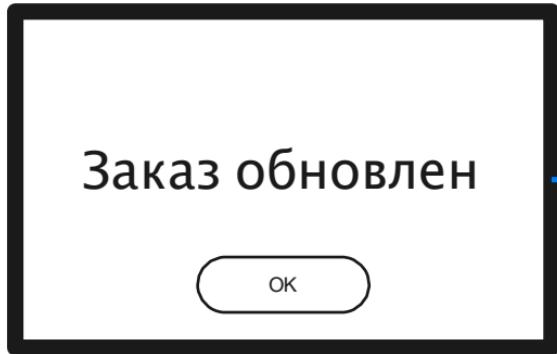


write model: CUD



versioning check & (merge or conflict solving)





bussines solution



CQRS полезно,
но осторожно