



# Microservices

.. T .. Systems ..

Saint Petersburg, 2018

- 1 ВИДЫ АРХИТЕКТУР
- 2 МОНОЛИТ
- 3 МИКРОСЕРВИСНАЯ АРХИТЕКТУРА
- 4 ХАРАКТЕРИСТИКИ МИКРОСЕРВИСОВ
- 5 ПРАВИЛА ПОСТРОЕНИЯ
- 6 ЛУЧШИЕ ПРАКТИКИ
- 7 ОГРАНИЧЕНИЯ
- 8 СОПРОВОЖДЕНИЕ

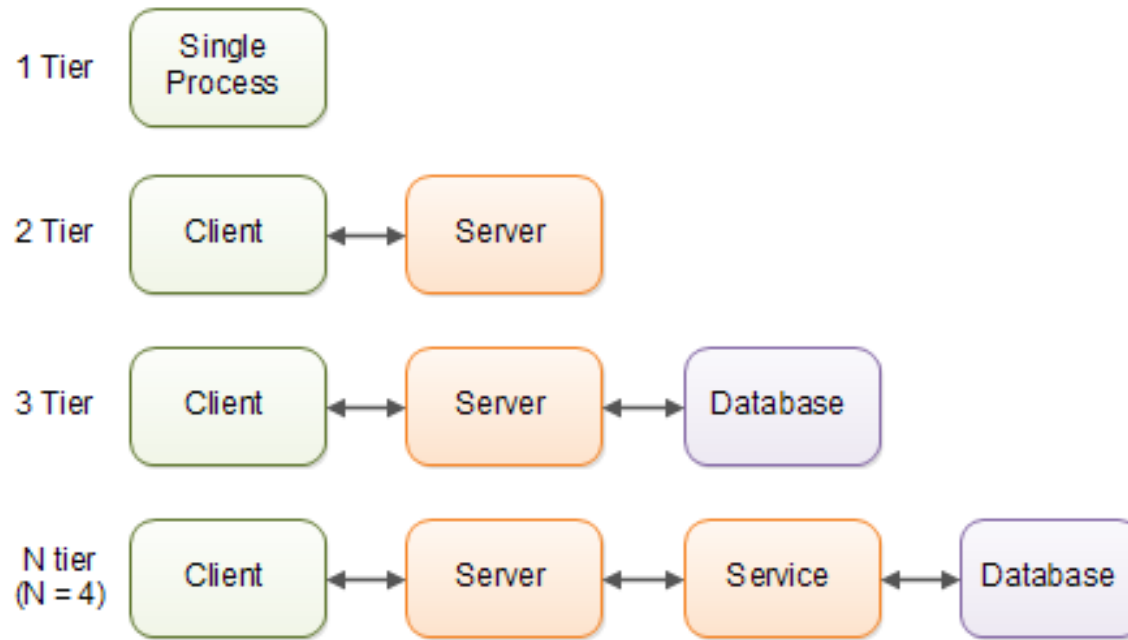
**Архитектура программного обеспечения** ([англ.](#) *software architecture*) — совокупность важнейших решений об организации программной системы.

Архитектура включает:

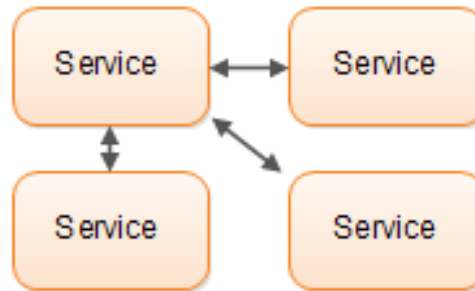
- выбор структурных элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов;
- соединение выбранных элементов структуры и поведения во всё более крупные системы;
- архитектурный стиль, который направляет всю организацию — все элементы, их интерфейсы, их сотрудничество и их соединение.

..T..Systems.....

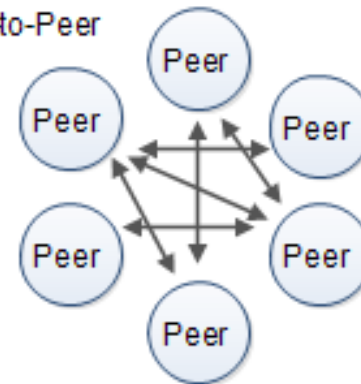
# ОСНОВНЫЕ ВИДЫ АРХИТЕКТУР



Service Oriented Architecture



Peer-to-Peer

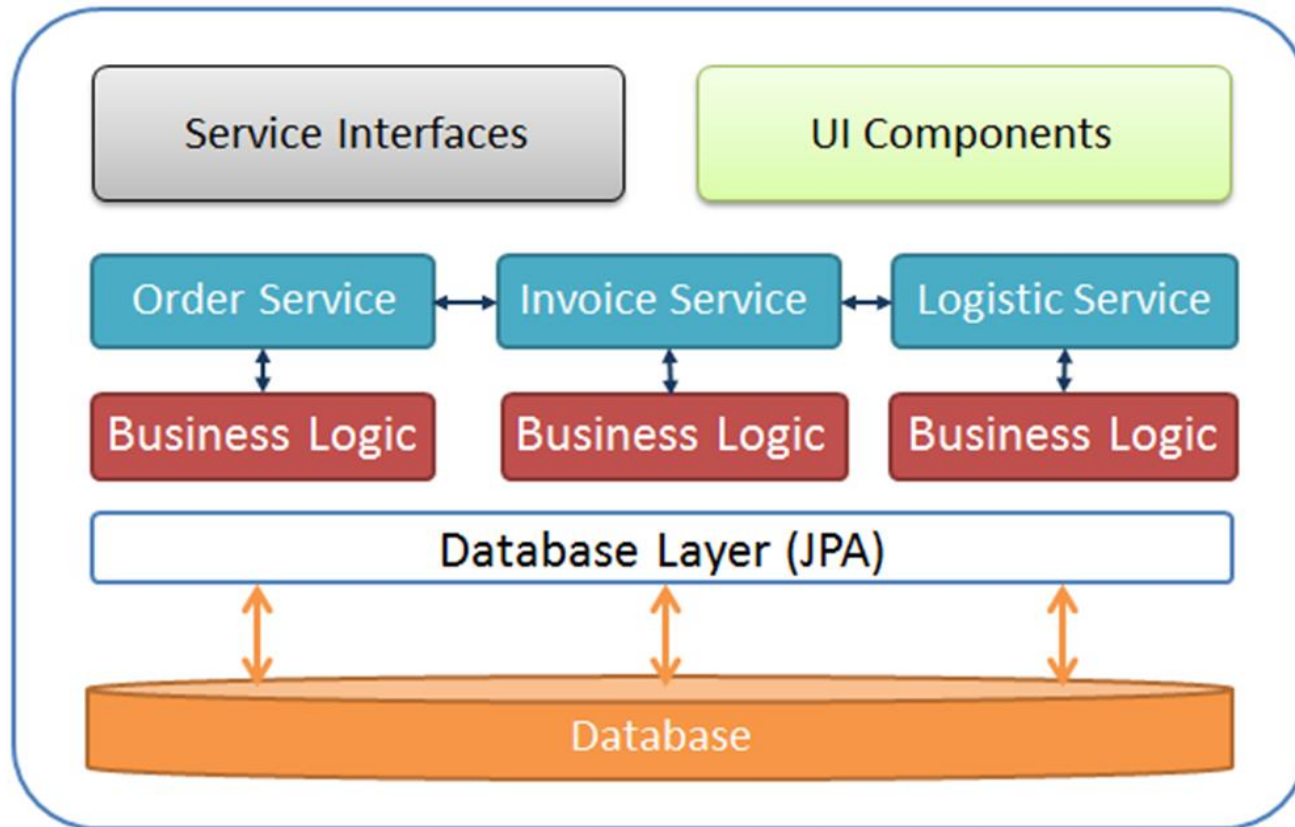


... T ... Systems ...

# Что такое монолитное приложение

**Монолит** – это архитектурный подход в проектировании приложений, при котором вся логика аккумулируется внутри одного сервера приложений.

## Monolithic Enterprise Application



... T ... Systems ...



# Характеристики монолитного приложения

---

- Разрабатывается несколькими командами
- Имеет очень большое количество кода
- Состоит из большого количества модулей
- Любые изменения требуют пересборки всего проекта
- При необходимости масштабирования, масштабировать приходится все приложение

...T...Systems.....

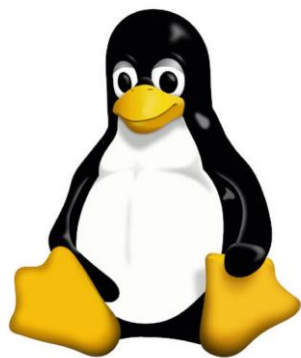
# Характеристики монолитного приложения

	
Практически нет затрат на персылку данных между компонентами	Сложность системы постоянно растет
Согласованность компонентов	Поддерживать ее все сложнее и сложнее
Простота развертывания и поддержки	Застревание на технологиях
	Дорого вносить изменения
	Плохая масштабируемость

.. T .. Systems ..



3.5 миллионов файлов  
50 миллионов строк кода  
Git репозиторий в 300 гигабайт.



2017

Ядро Linux 4.11.7    18 373 471  
строк кода



2 миллиарда строк кода

.. T .. Systems ..



# Монолит – это...

---

**Монолит** - это глыба, способная тебя раздавить, если упадет.



.. T .. Systems ..

# Да придет спаситель

Martin Fowler James Lewis Sam Newman

## MICROSERVICES ДА ПРИДЕТ СПАСИТЕЛЬ

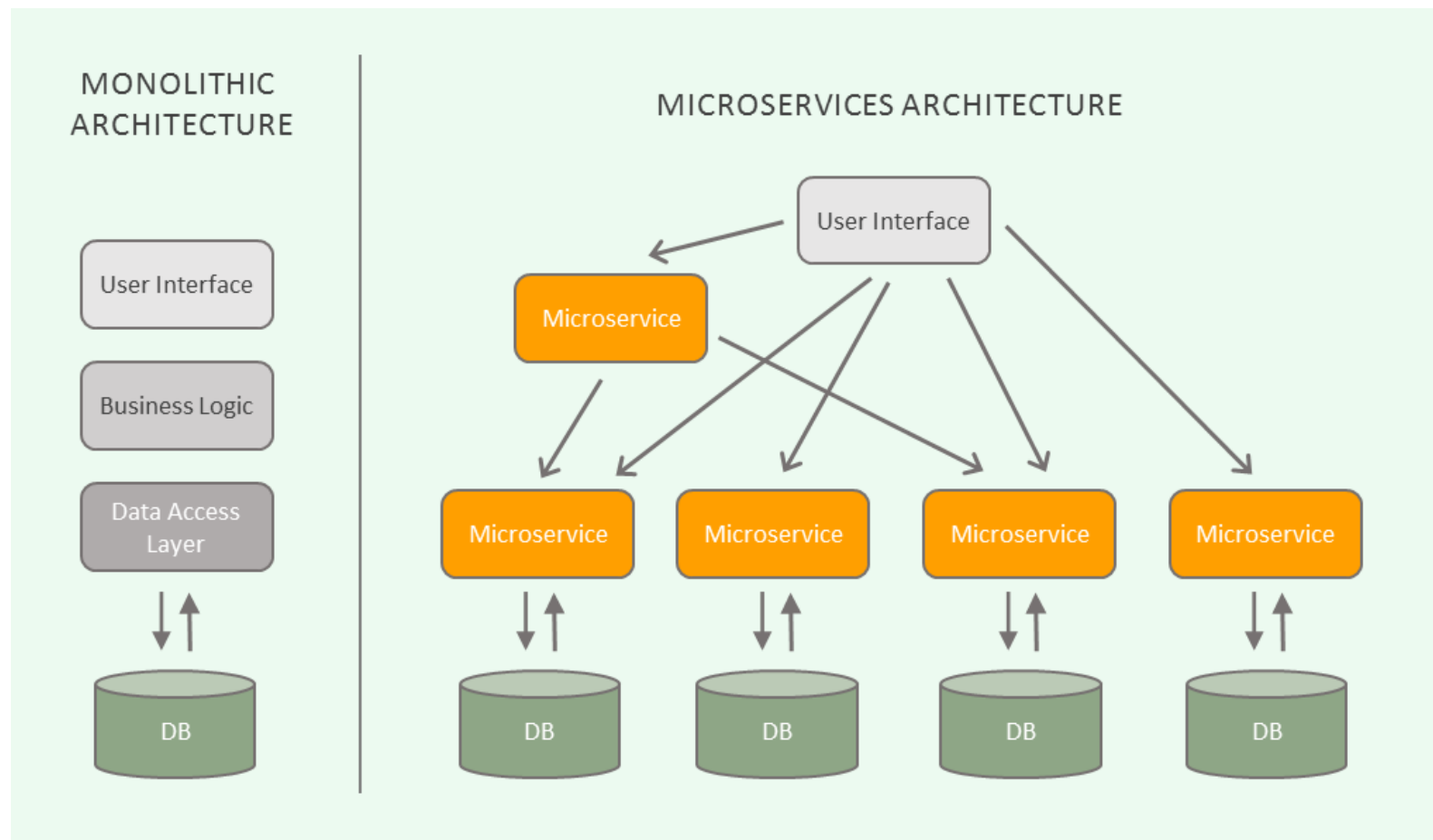


..T..Systems.....

**Микросервисная архитектура** — это подход, при котором единое приложение строится как набор небольших сервисов, каждый из которых работает в собственном процессе и коммуницирует с остальными используя легковесные механизмы, как правило HTTP. Эти сервисы построены вокруг бизнес-потребностей и развертываются независимо с использованием полностью автоматизированной среды

# Микросервисы

**Микросервисы** – это архитектурный подход, в котором большие комплексные системы состоят из одного или более компонентов.



...T...Systems...

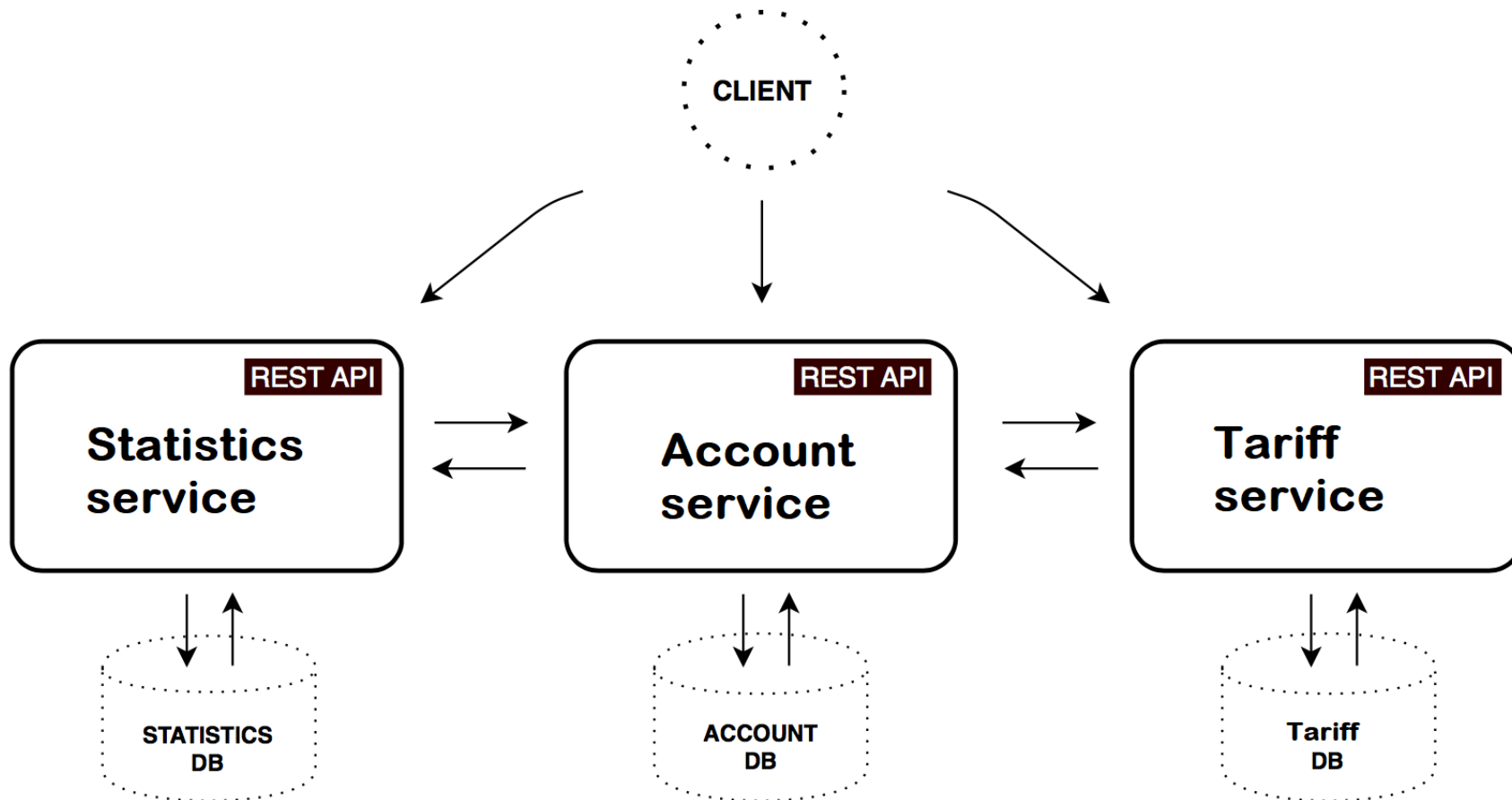
# Характеристики микросервисов

---

## Основные характеристики:

- Разделение на компоненты (сервисы)
- Группировка по бизнес-задачам
- Динамический технологический стек
- Небольшие команды разработчиков
- Децентрализованное управление
- Слабая связанность

... T ... Systems ...



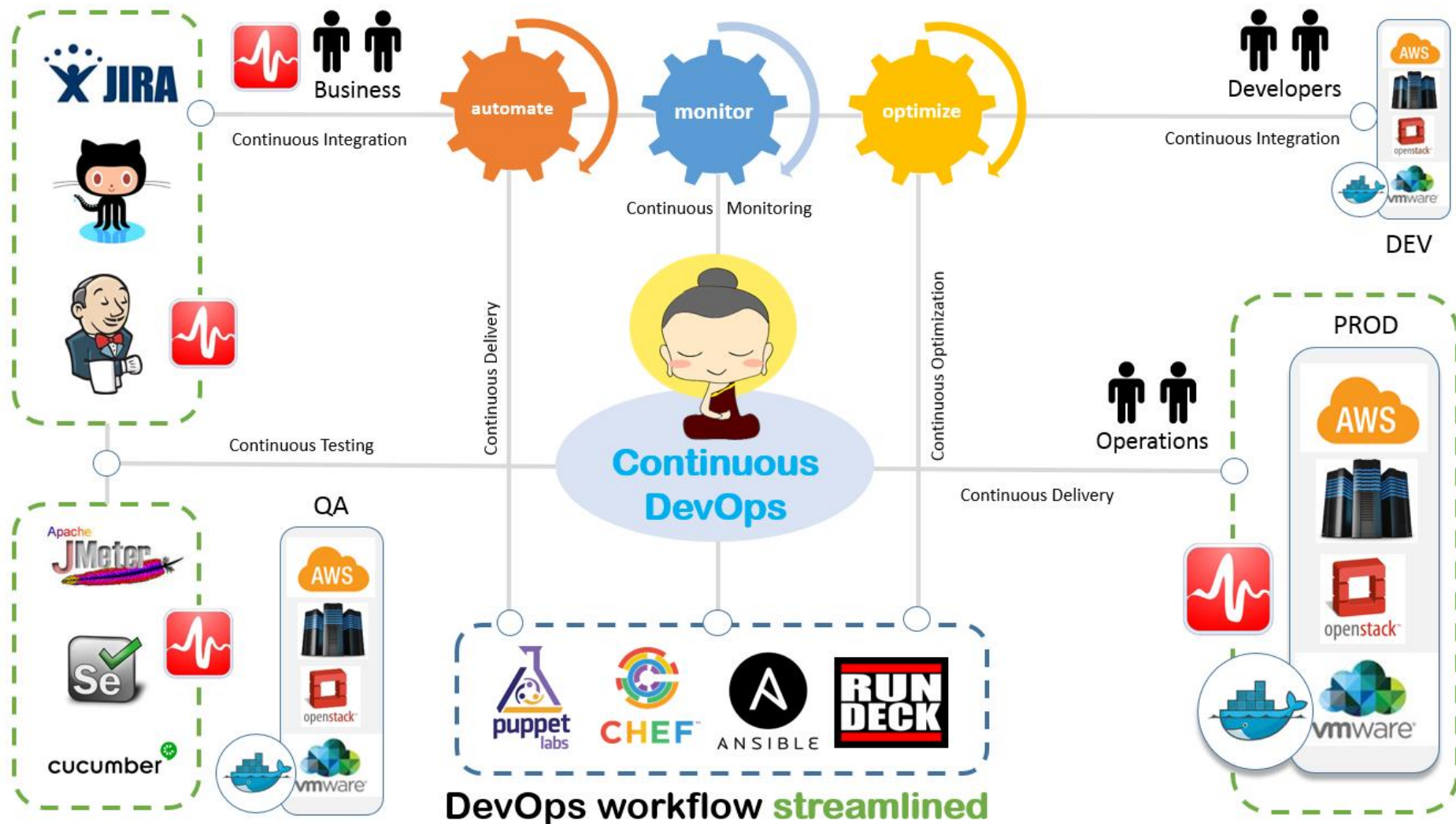
... T ... Systems ...





... T ... Systems ...

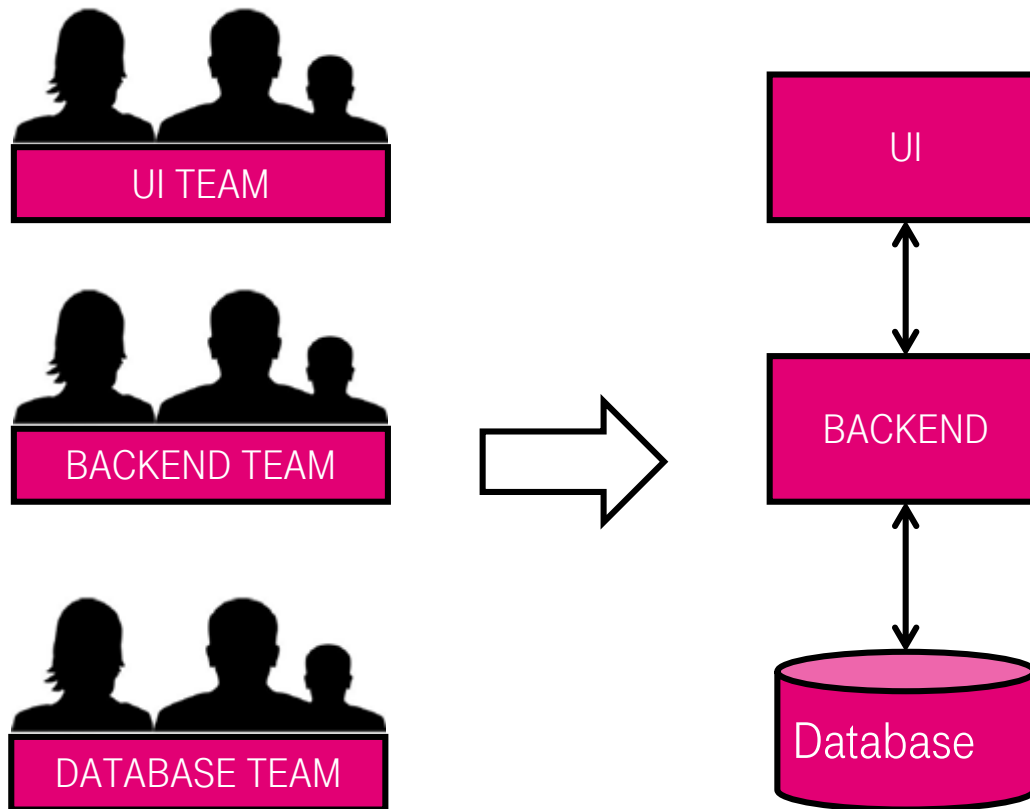
# Как это работает



..T..Systems..



# Закон Конвея - Пример #1



... **T** ... **Systems** ...



amazon

The Amazon logo is centered in the upper half of the slide. It features the word "amazon" in a bold, black, sans-serif font. Below the text is the iconic orange curved arrow that starts under the 'a' and ends under the 'z'. The background of the entire slide is a stylized illustration of a city skyline at dusk or dawn, with various building silhouettes in shades of purple, blue, and pink, and a bright yellow sun or moon on the right side.

theguardian

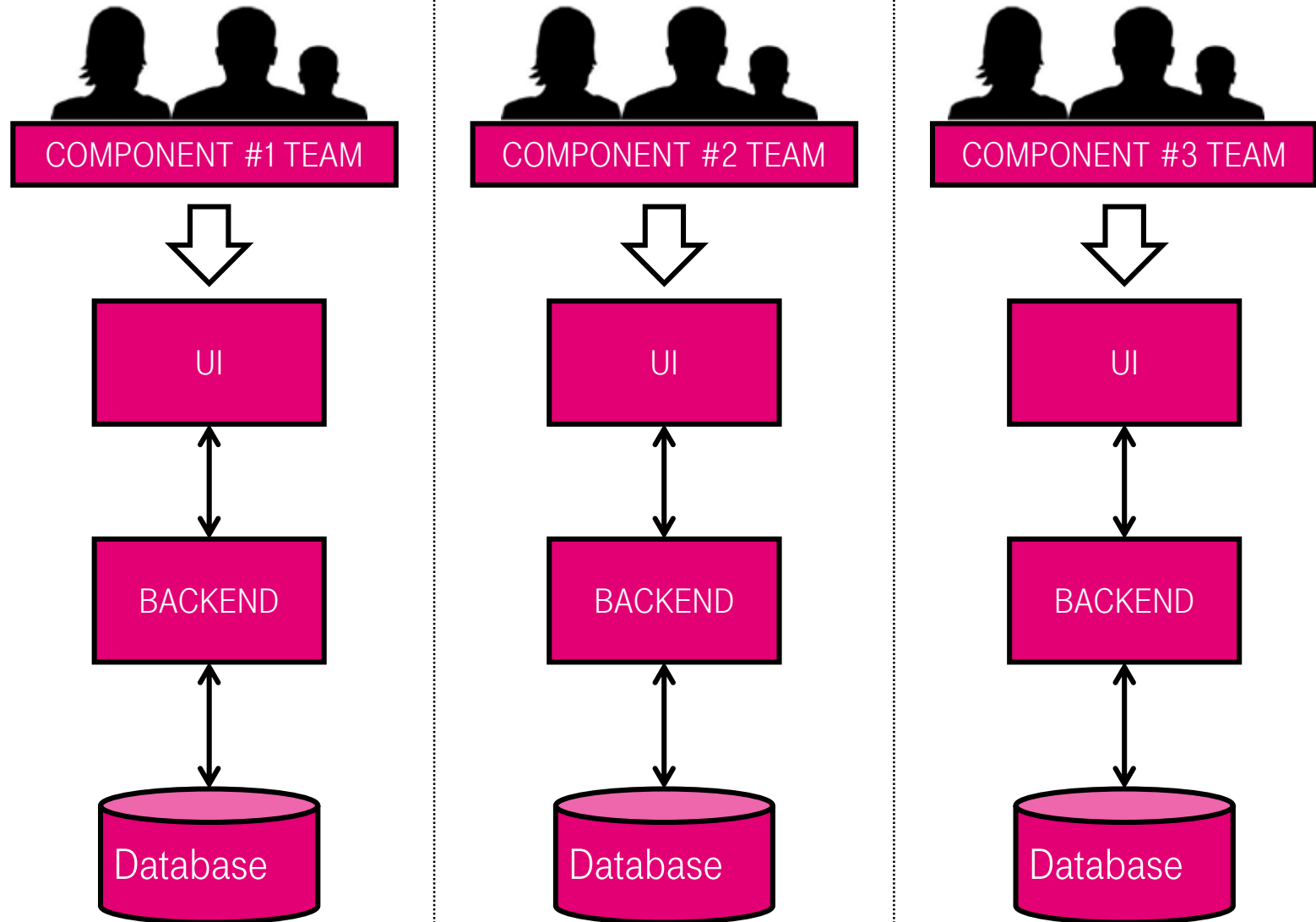
The logo for The Guardian is centered in the middle section of the slide. It consists of the words "theguardian" in a blue, lowercase, sans-serif font. The background is a solid light blue.

NETFLIX

The Netflix logo is centered in the lower right section of the slide. It features the word "NETFLIX" in a bold, red, sans-serif font. The background is a solid black.

.. T .. Systems ..

# Закон Конвея - Пример #2



...T...Systems...

Организации, которые разрабатывают системы, вынуждены создавать продукты, которые по структуре повторяют структуру коммуникаций внутри данных организаций

...T...Systems.....

## Правила построения:

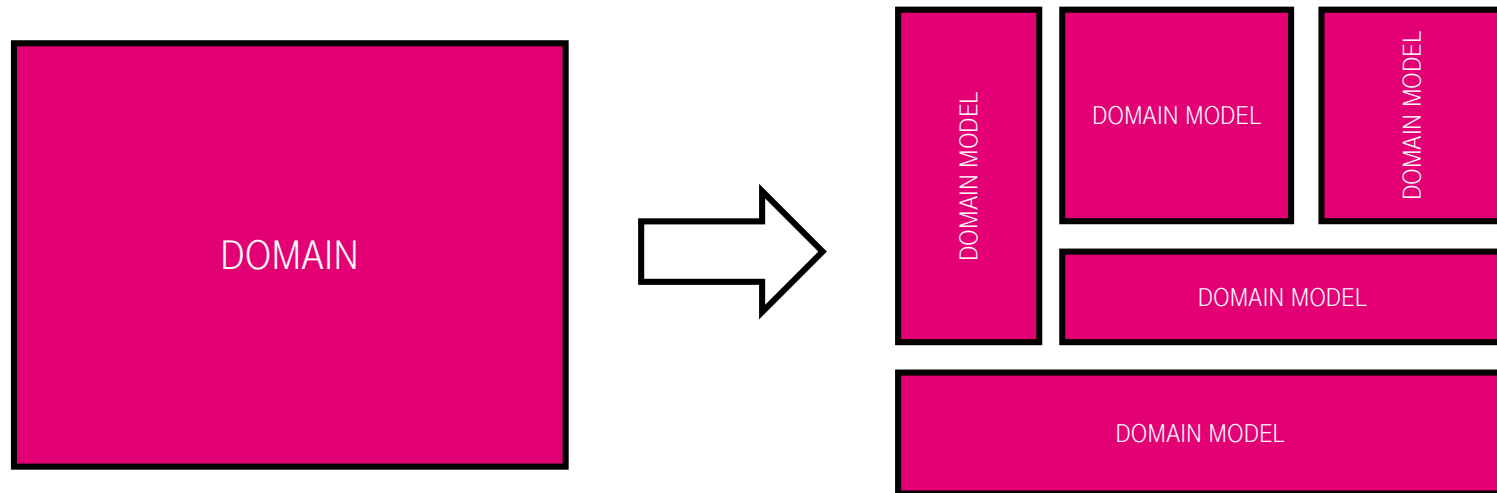
- Фокусируется на выполнении только одной задачи
- Технологическая свобода
- Относительно небольшое количество кода
- Легко заменить
- Слабая связность между микросервисами
- Децентрализованное управление данными
- Обладает собственными: репозиторием, процессами сборки и развертывания

# Domain-Driven Design

---

**Предметно-ориентированное проектирование** – это набор принципов и схем, направленных на создание оптимальных систем объектов. И сводится к созданию программных абстракций, которые называются моделями предметных областей (Domain Model).

**Domain Model** – это структурированные знания, которые связаны с конкретной проблемой.



# Bounded Context

---

**Bounded Context** – это центральный паттерн в предметно-ориентированном проектировании (Domain-Driven Design, DDD).

⇒ Позволяет четко задать границы для модели предметной области и не допустить какого-либо на нее влияния из внешнего мира.

Это позволяет:

- Моделировать различные аспекты проблемы не контактируя с другими частями бизнеса.
- Использовать терминологию с единственным, четким определением, которое точно описывает конкретную проблему.

.. T .. Systems ..

# Bounded Context в Микросервисах

---

Применительно к микросервисной архитектуре, **Bounded Context** подразумевает, что определенный микросервис ничего не знает об особенностях реализации окружающих его микросервисов и имеет одну четко поставленную задачу.

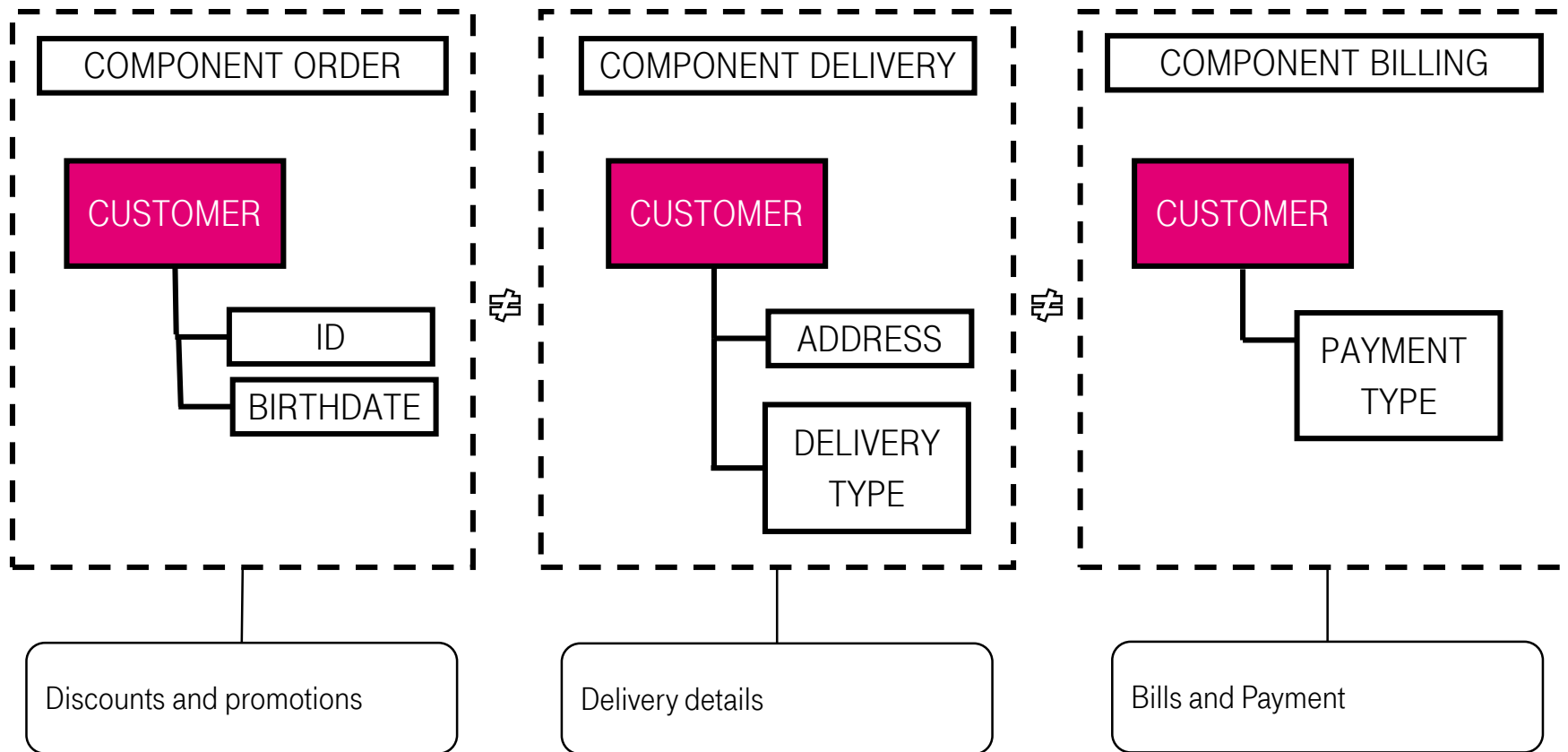
**! Каждый микросервис должен иметь Bounded Context**

!Микросервисы придерживаются подхода Shared Nothing и приветствуют избыточность кода

...T...Systems.....



# Bounded Context в Микросервисах: Пример



...T...Systems...

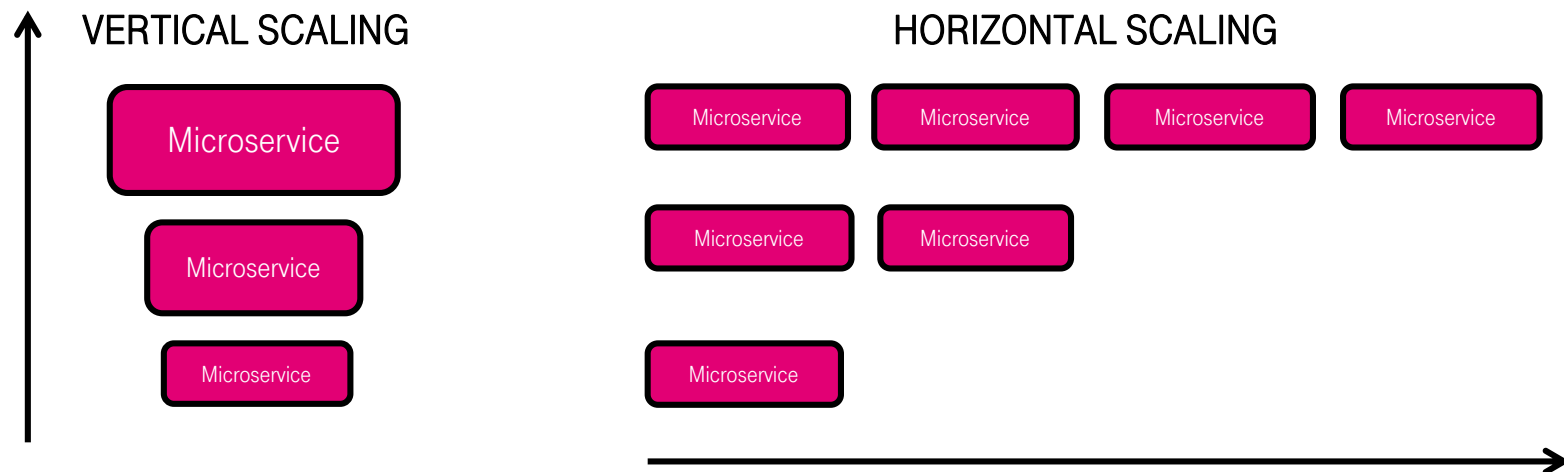
# Типы масштабируемости приложений

Проектируемая система должна быть способна обрабатывать тем большую нагрузку, чем больше ресурсов она получит.

**! Микросервис должен быть масштабируемым**

Типы масштабируемости:

- **Горизонтальная.** Означает, что чем большее количество ресурсов имеется, тем большая нагрузка снимается, каждым из них.
- **Вертикальная.** Означает, что используется более производительный ресурс, способный в одиночку справляться с заданной нагрузкой.

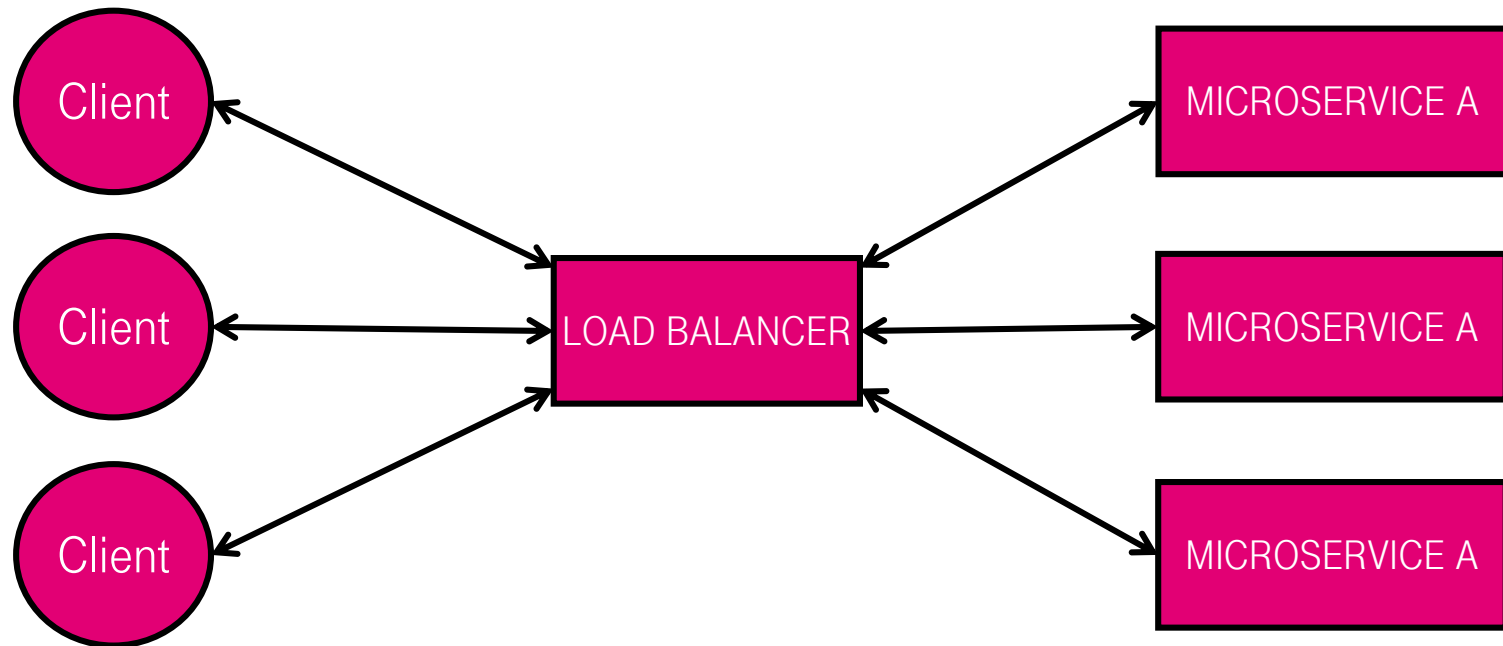


... T ... Systems ...

# Горизонтальное масштабирование - Load Balancing

**Load balancer** – это приложение, которое выполняет 2 основные роли:

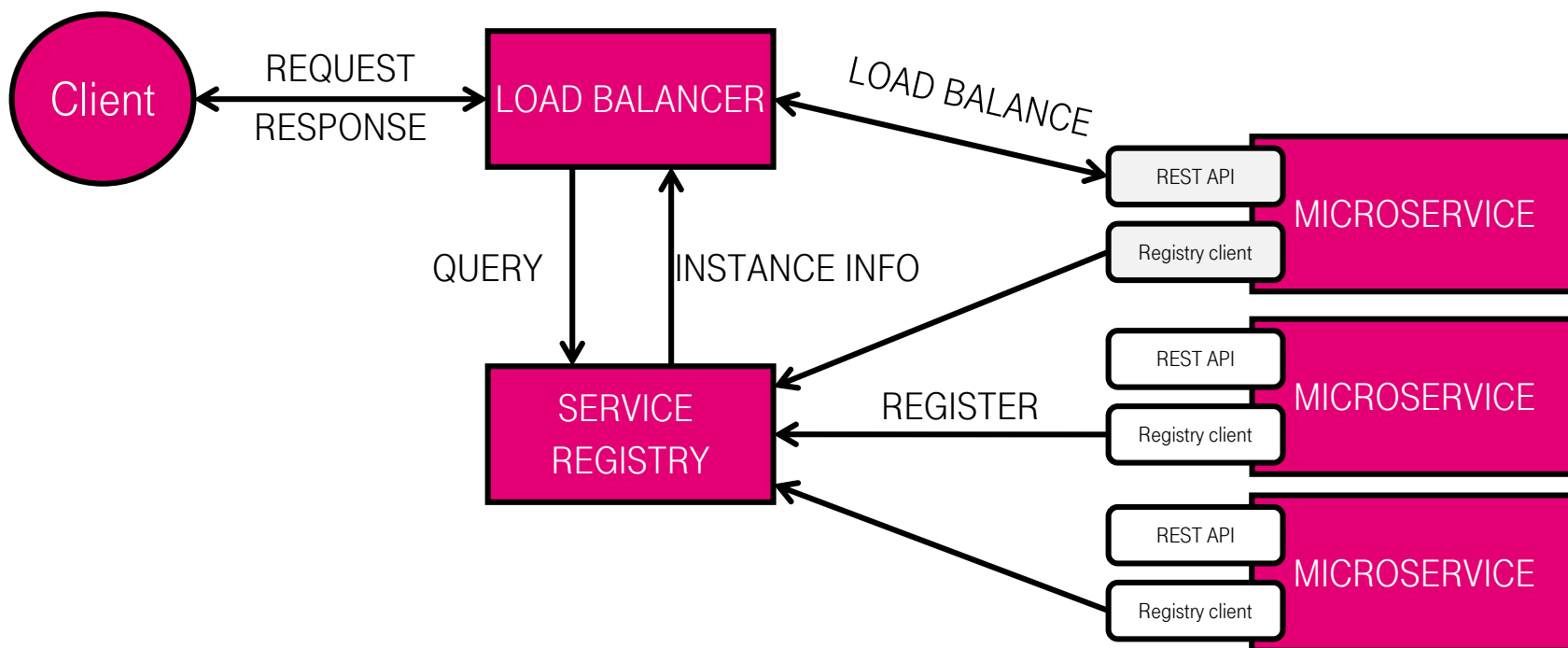
- Играет роль фасада (service endpoint), к которому обращаются сторонние сервисы.
- Перенаправляет запросы на множественные экземпляры микросервиса, тем самым распределяя нагрузку.



...T...Systems.....

# Горизонтальное масштабирование - Service Discovery

**Service Discovery** – это инфраструктурный паттерн, направленный на обеспечение целостности связей между приложениями и сервисами. Создает единую точку доступа к сервисам.



## Преимущества микросервисного подхода:

- Позволяет избежать большого количества кода, тем самым облегчая поддержку и внесение изменений
- Позволяет использовать те навыки разработчиков, которыми они уже обладают, либо использовать наиболее оптимальную технологию
- Значительно ускоряет развертывание приложения и открытия проекта в IDE
- Позволяет работать командам более независимо друг от друга
- Становится намного проще масштабировать наиболее загруженные сервисы, с целью нивелирования эффекта «бутылочного горлышка»

# COFFEE

---

```
$git clone https://github.com/parkito/LearnMicro
```

```
$cd git checkout lvl5
```

```
$cd POST_SERVICE
```

```
$mvn install
```



- Resilience
- Stateless
- Timeout
- Fail-Fast
- Circuit Breaker
- Bulkheads
- Logging
- Monitoring

...T...Systems.....

# Patterns of Resilience

---

**Resilience** – свойство системы справляться с непредвиденными ситуациями.

- Системы построенные на основе микросервисов являются распределенными
- Распределенные системы ненадежны
- Каждый микросервис должен быть защищен от отказа других микросервисов

**!Влияние вышедшего из строя микросервиса должно быть минимальным либо отсутствовать вовсе**

.. **T** .. **Systems** ..



# Patterns of Resilience: Stateless

---

**Stateless** – данный паттерн является фундаментальным и заключается в обеспечении горизонтальной масштабируемости разрабатываемой системы.

## Характеристики:

- Поведение системы не должно зависеть от ее состояния
- Результат работы определяется исключительно входными данными

...T...Systems.....

# Patterns of Resilience: Timeout

---

**Timeout** – этот паттерн предназначен для реализации возможности обнаружения недоступности внешней системы при обращении к ней.

## Характеристики:

- Измеряет время ответа при обращении к внешней системе
- Останавливает ожидание после заданного периода времени
- Выполняет альтернативный набор операций, если заданный период времени истек

... T ... Systems ...

# Patterns of Resilience: Fail-Fast

---

**Fail-Fast** – задача паттерна заключается в том, что система должна мочь распознавать ошибки настолько быстро, насколько это возможно, и реагировать на них незамедлительно, не позволяя ошибкам уйти в более глубокие слои логики системы.

## Характеристики:

- Позволяет избежать некорректное поведение, если его можно предвидеть
- Используется как проверка перед дорогостоящими операциями
- Уменьшает шансы отказа системы

...T...Systems.....

# Patterns of Resilience: Circuit Breaker

---

**Circuit Breaker** – данный паттерн позволяет реализовать систему с гарантией, что если имеется отказавший сервис, то это не повлияет на всю систему.

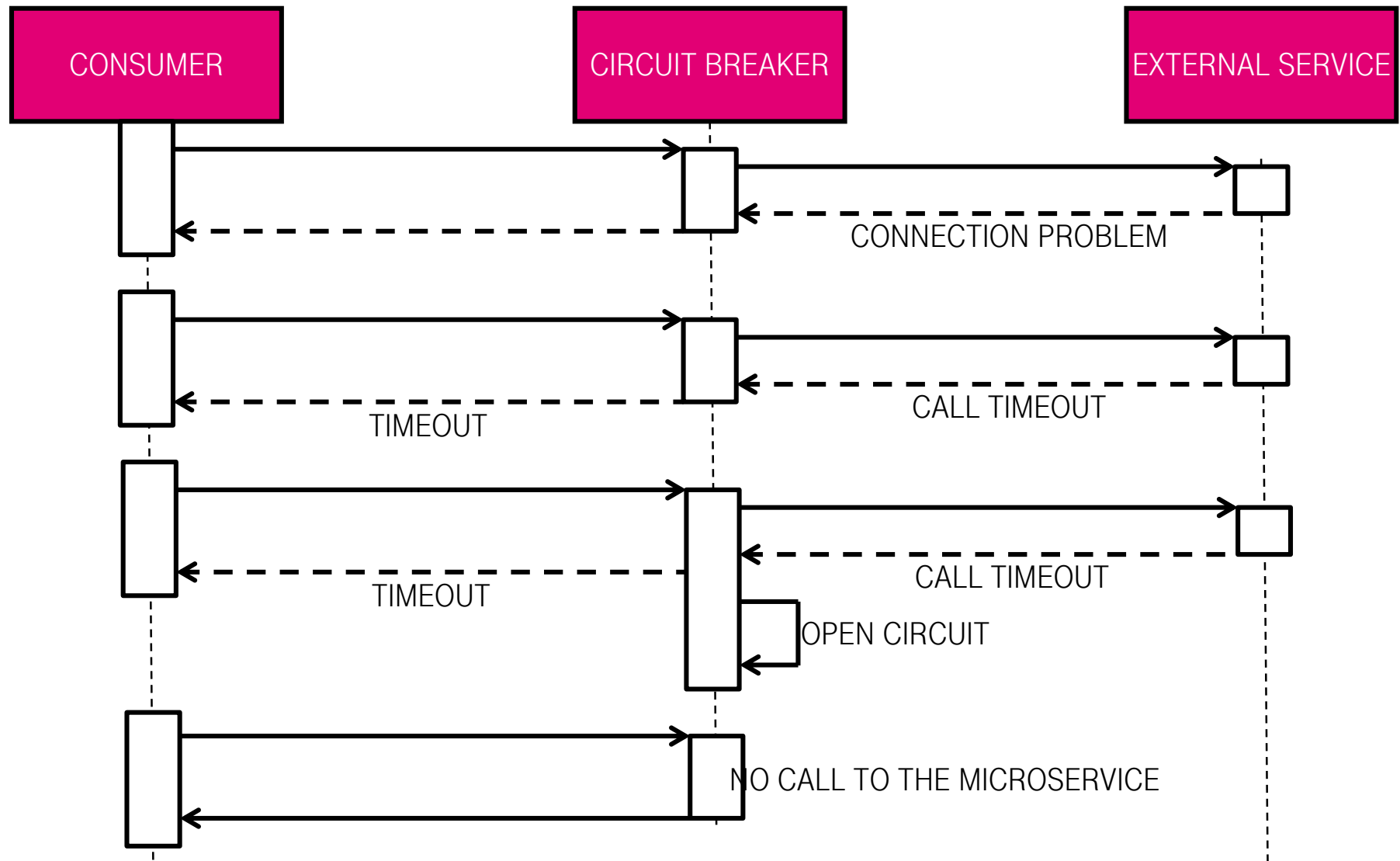


## Характеристики:

- Обработывает вызовы к внешним системам в offline-режиме, если несколько предыдущих вызовов потерпели неудачу.

.. T .. Systems ..

# Patterns of Resilience: Circuit Breaker – Example



...T...Systems.....

# Patterns of Resilience: Circuit Breaker - Hystrix

**Netflix Hystrix** – данный фреймворк имплементирует паттерн Circuit Breaker. Позволяет обернуть вызовы к внешним системам в специальные объекты - Hystrix Command.

## Example (Spring):

@Configuration

@EnableAspectJAutoProxy

public static class SpringConfig {

    @Bean

    public HystrixCommandAspect hystrixCommandAspect() {  
        return new HystrixCommandAspect();  
    }

}

@Service

public class DummyRemoteCallService implements RemoteCallService {

    private RemoteCallService remoteCallService;

    public DummyRemoteCallService() {  
    }

    @Override

    @HystrixCommand(fallbackMethod = "fallBackCall")

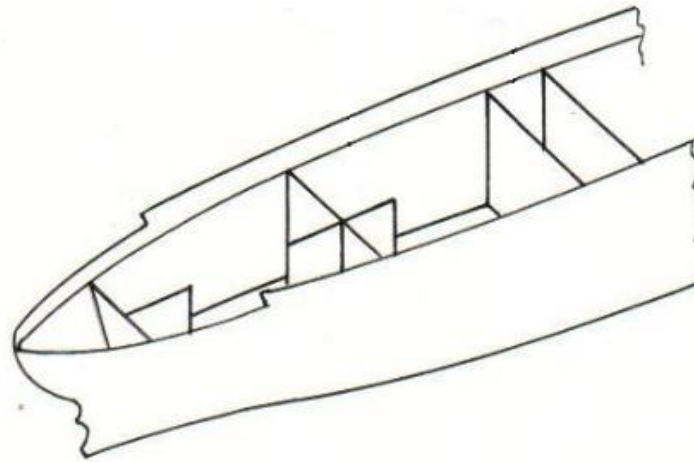
    public String call(String request) throws Exception {  
        return this.remoteCallService.call(request);  
    }

    public String fallBackCall(String request) {  
        return "FALLBACK: " + request;  
    }  
}

# Patterns of Resilience: Bulkheads

---

**Bulkheads** – данный паттерн позволяет спроектировать систему так, чтобы избежать ее полного отказа при отказе какой-либо части.

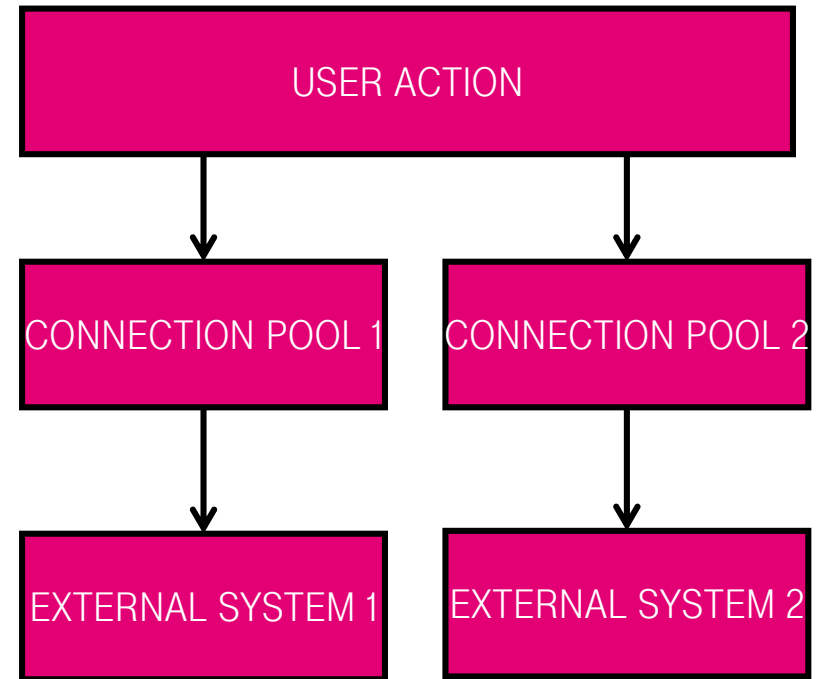
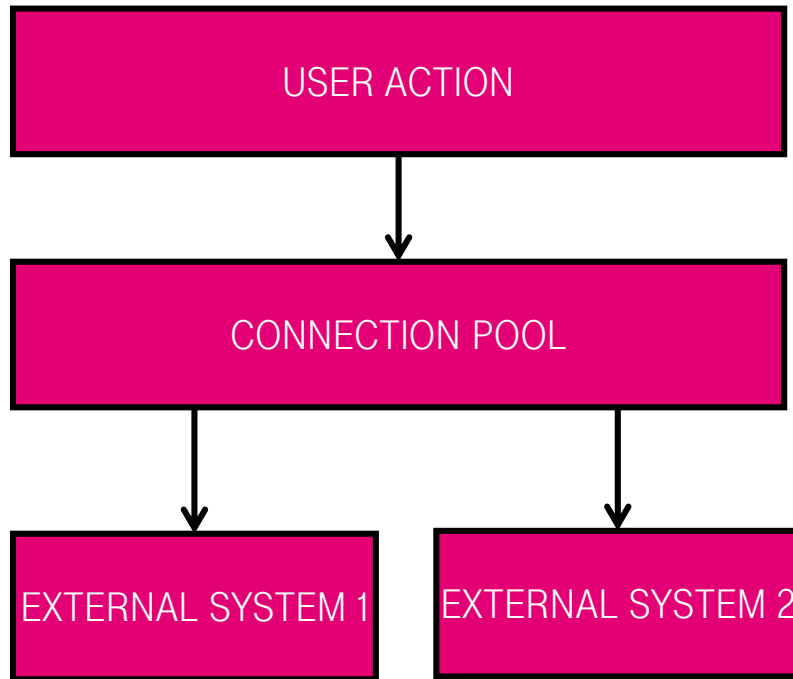


Характеристики:

- Система разбивается на независимые составляющие
- Изолирование отказавших части системы

... **T** ... **Systems** ...

# Patterns of Resilience: Bulkheads #2



... **T** ... Systems ...



# Patterns of Resilience: Monitoring

---

**Мониторинг** – это паттерн, который позволяет выявлять проблемы непосредственно во время выполнения приложения.

Характеристики:

- Отслеживает поведение и взаимодействие с внешними системами
- Сигнализирует о проблемах в компоненте системы

...T...Systems.....

# Patterns of Resilience: Monitoring - Metrics

---

**Метрики** измеряют поведение и делятся на 2 типа:

- Общие метрики (доступная оперативная память, количество созданных потоков, версия JVM и т.д.)
- Метрики приложения (количество уникальных запросов, количество загруженных файлов)

```
{  
  "gauge.uptime":1494045951,  
  "counter.unique.uploads":423,  
  "counter.unique.downloads":104,  
  "gauge.mem":412450,  
  "gauge.mem.free":186152,  
  "gauge.classes":15509,  
  "gauge.classes.loaded":15539,  
  "gauge.classes.unloaded":30,  
  ...  
}
```

# Patterns of Resilience: Monitoring – Health-checks

**Health-check** измеряет отклонение в работе

```
public class DatabaseHealthCheck extends HealthCheck {
    private final Database database;

    public DatabaseHealthCheck(Database database) {
        this.database = database;
    }

    @Override
    protected Result check() throws Exception {
        if (database.ping()) {
            return Result.healthy();
        }
        return Result.unhealthy("Can't ping database");
    }
}
```

```
{
    "status": "FAIL",
    "databaseHealthCheck": {
        "status": "FAIL"
    },
    "diskSpaceHealthCheck": {
        "status": "OK"
    }
}
```

# Application performance monitoring

---

**Мониторинг производительности приложения** – это раздел информационных технологий, который фокусируется на проверке того, что приложение функционирует должным образом.

Основные решаемые задачи:

- Анализ
- Профилирование
- Реакция на состояние системы

**Nagios®**

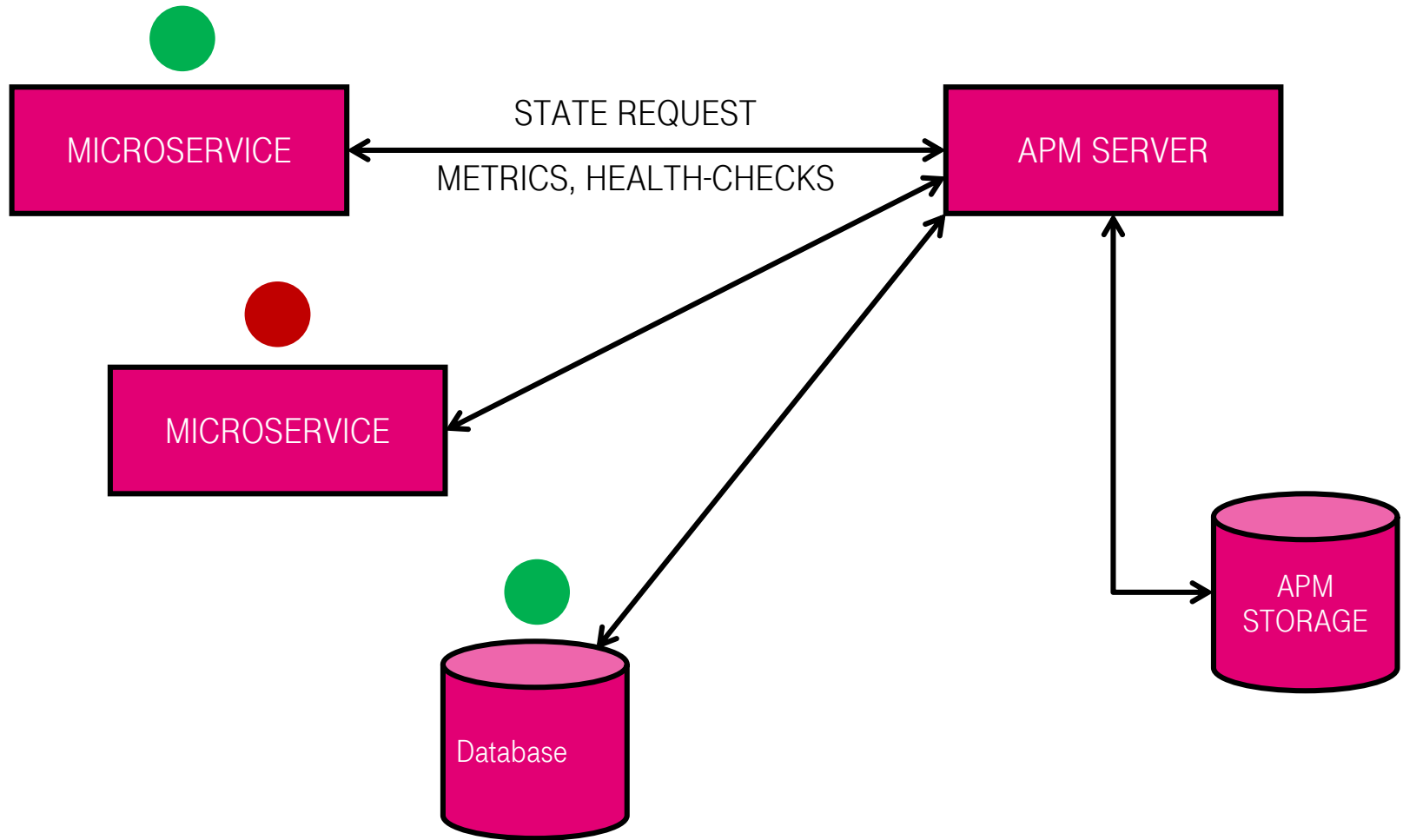
 **dynatrace**

 **New Relic®**

 **AppDynamics**

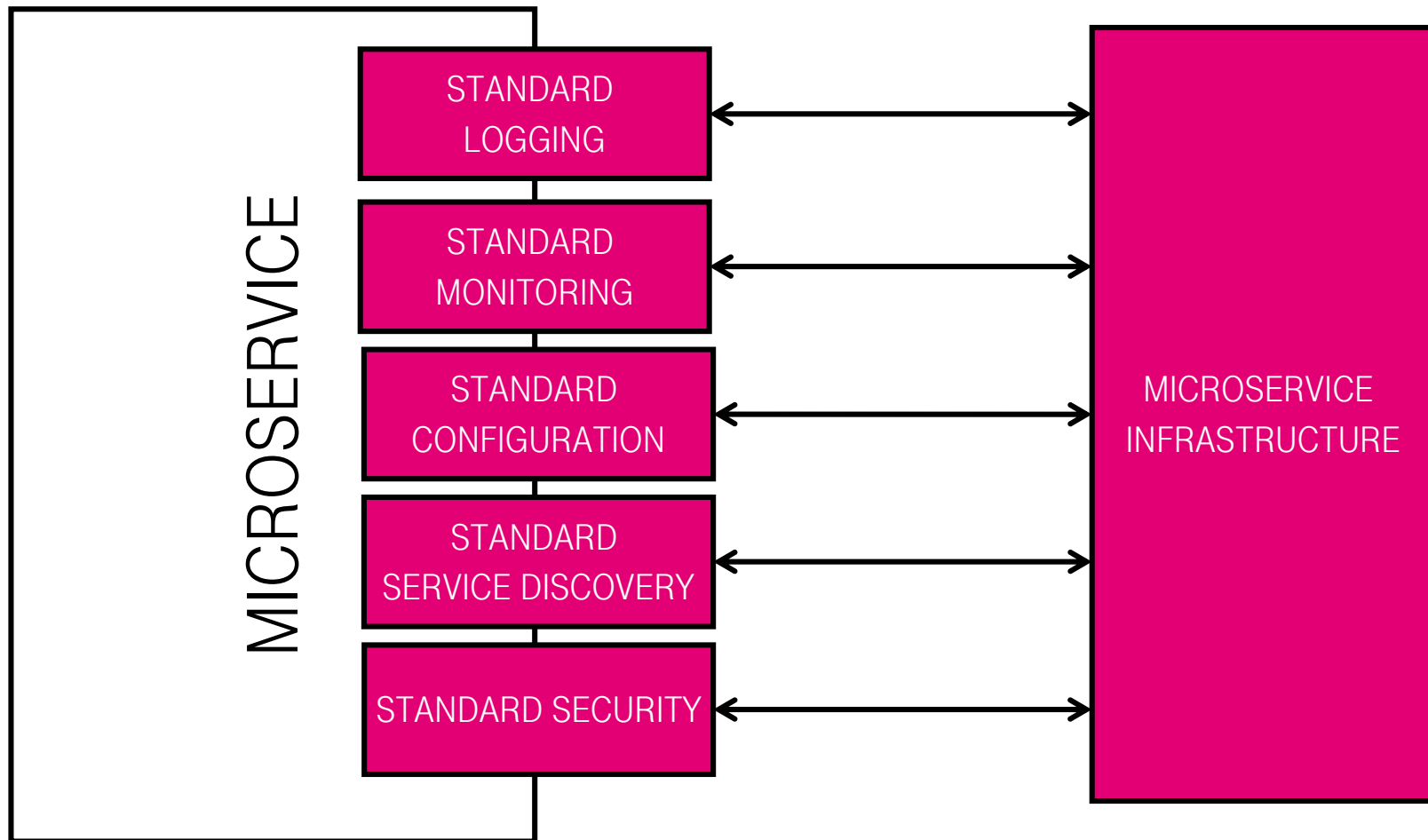
... **T** ... **Systems** ...

# Application performance monitoring #2



... T ... Systems ...

# Стандартные решения



...T...Systems...

# Шаблоны проектов

---

**Шаблон проекта** – предлагает единообразное решение общих задач

Достоинства:

- Повышает продуктивность команд
- Демонстрирует правильное использование технологий
- Содержит полностью рабочий прототип для запуска на целевой платформе

Недостатки:

- Разрабатываемые микросервисы должны адаптироваться по изменяющийся шаблон, на котором они основаны

...T...Systems.....

# Коммуникация в микросервисной инфраструктуре

---

Основные механизмы коммуникации:

- **Синхронный** (REST, SOAP, WebSockets)

Недостатки:

- Отсутствует поддержка модели publish/subscribe
- Клиент и сервер должны быть одновременно доступны

- **Асинхронный** (Message Broker)

Недостатки:

- Невозможность передать большой объем данных

...T...Systems.....



# Интерфейсы

---

**!Микросервис должен предоставлять интерфейс для того, чтобы внешние системы могли с ним взаимодействовать**

**!Интерфейсы для внешних систем фиксируются определенной версией и больше не изменяются**

**!Если изменения в интерфейсе необходимы, то разрабатывается его новая версия**

**...T...Systems.....**

Опасность



# Транзакции в микросервисной архитектуре - ACID

---

Требования предъявляемые к ACID-транзакциям :

- **Atomicity** – гарантирует, что данная транзакция либо будет выполнена полностью, либо не выполнена совсем. В случае ошибки, будет произведен откат всех изменений.
- **Consistency** – означает что данные согласованы до и после выполнения транзакции, например валидации не нарушены.
- **Isolation** – гарантирует, что операции параллельных транзакций будут выполнять раздельно и не влиять друг на друга.
- **Durability** – гарантирует что данные будут сохранены и доступны даже после физического сбоя оборудования.

.. T .. Systems ..

# Транзакции в микросервисной архитектуре - BASE

---

**Репликация** — одна из техник масштабирования баз данных. Состоит эта техника в том, что данные с одного сервера базы данных постоянно копируются (реплицируются) на один или несколько других серверов (называемые репликами).

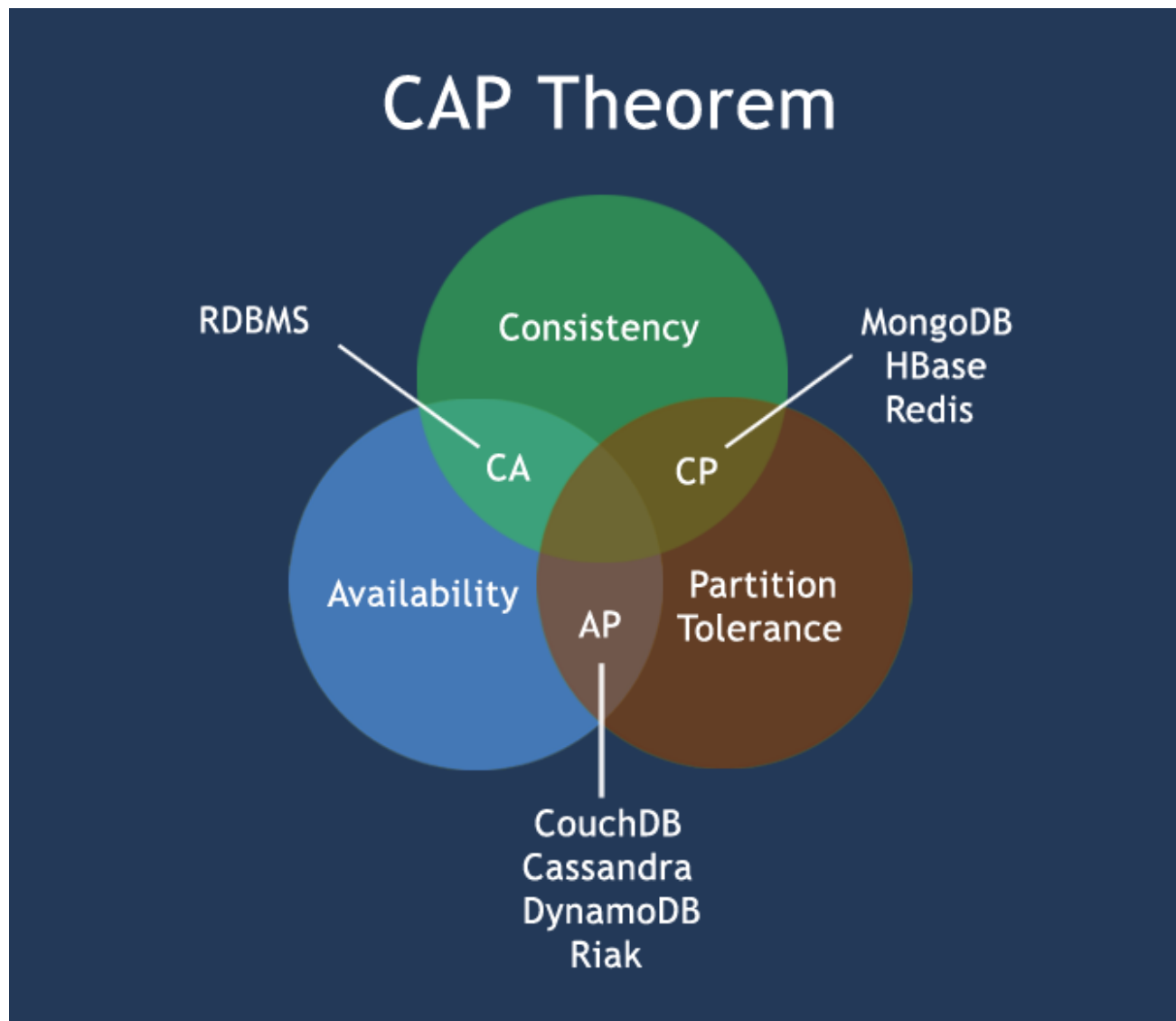
Требования предъявляемые к BASE-транзакциям:

- **Basic Availability** – гарантирует, что данные будут доступны для запросов, но они могут быть в несогласованном виде.
- **Soft State** – сохраненные значения могут меняться со временем из-за eventual consistency, даже если на вход не подавать никаких данных. Такое состояние называется soft.
- **Eventual Consistency** – как только данные добавляются в систему, состояние системы реплицируется на все ноды. При отсутствии потока входных данных, вся система рано или поздно станет согласованной.

.. T .. Systems ..



**Теорема CAP** — эвристическое утверждение о том, что в любой реализации распределённых вычислений возможно обеспечить не более двух из трёх следующих свойств:

- *согласованность данных* (англ. *consistency*) — во всех вычислительных узлах в один момент времени данные не противоречат друг другу;
- *доступность* (англ. *availability*) — любой запрос к распределённой системе завершается корректным откликом, однако без гарантии, что ответы всех узлов системы совпадают;
- *устойчивость к разделению* (англ. *partition tolerance*) — расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

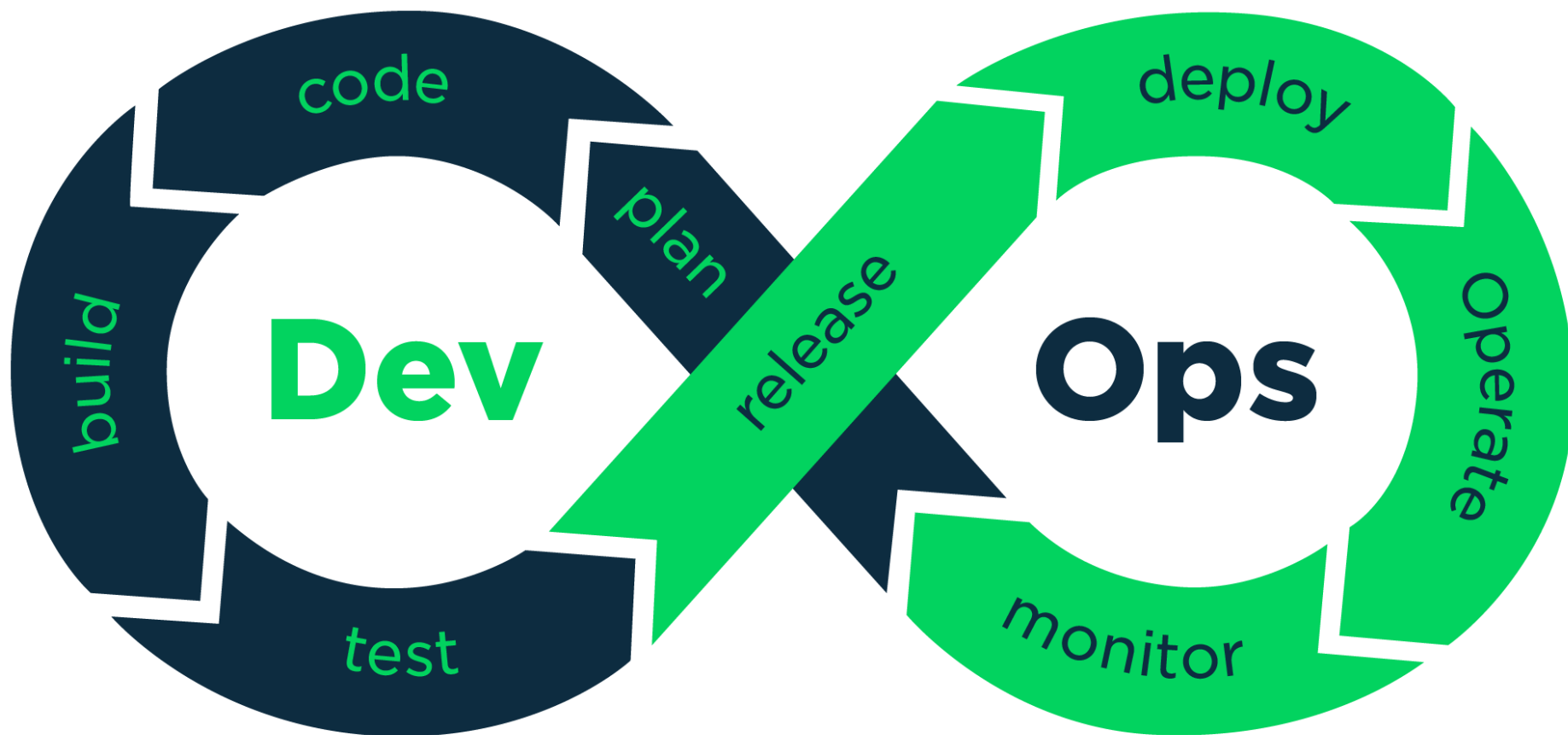


... **T** ... Systems ...

# Microservices

	
Независимость, легкие и частые релизы	Траты на мониторинг и поддержку
Независимый технологический стек	Усложнение обслуживания
Устойчивость к сбоям частей системы	«Опасные» распределенные связи
Независимая и параллельная разработка	Накладные расходы на сеть
Быстрые циклы разработки	Безопасность транзакций
Изолированные ресурсы	Обязательность компонентов (сервис дискавери и т.п.)
Масштабирование	Сложное тестирование
Заменяемость	Независимый технологический стек

.. T .. Systems ..



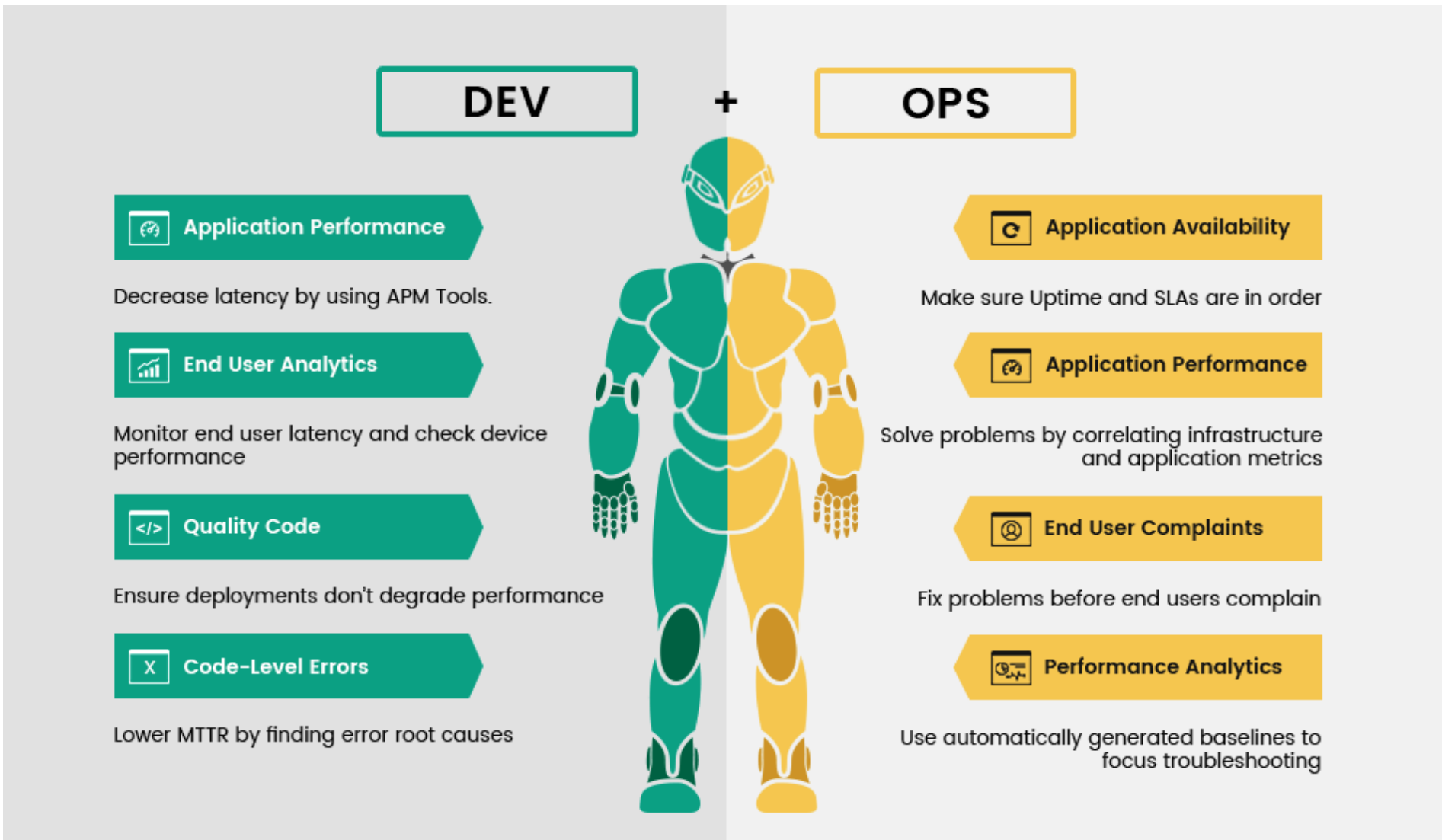
... **T** ... **Systems** ...



**DevOps** (акроним от англ. development и operations) — это методология разработки ПО, сфокусированная на предельно активном взаимодействии и интеграции в одной упряжке программистов, тестировщиков и админов, синхронизировано обслуживающих общий для них сервис/продукт



# Девы против опсов



...T...Systems...

## CONTINUOUS DELIVERY

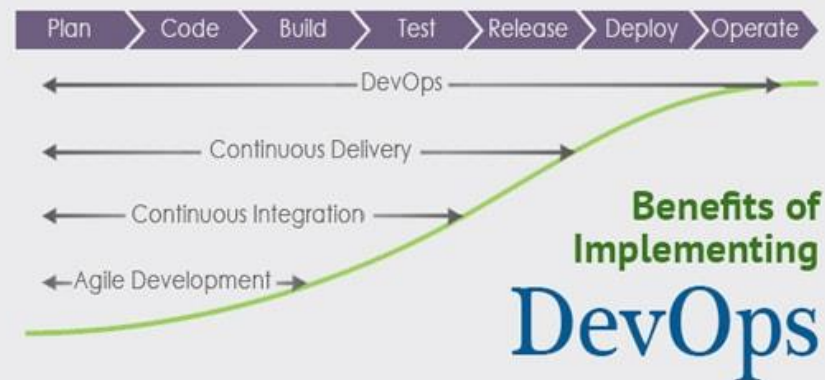
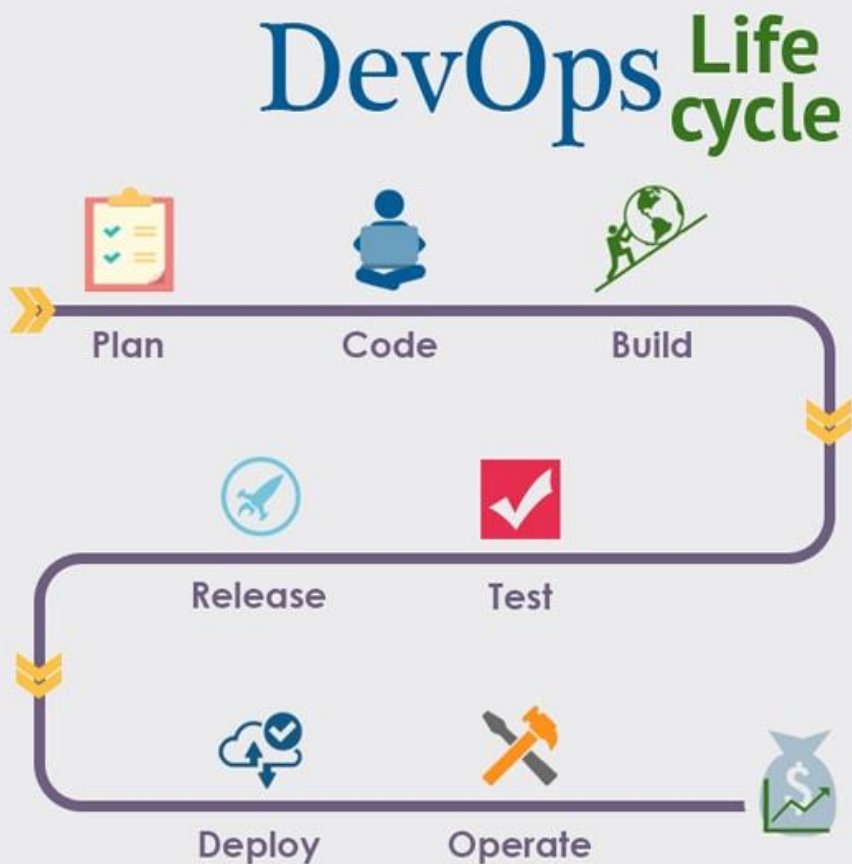


## CONTINUOUS DEPLOYMENT



... T ... Systems ...

# Цикл сопровождения



**21% Reduction**  
in time spent fixing & maintaining applications

**19% Improvement**  
in application quality and performance

**18% Increase**  
in revenue

**33% Increase**  
in time for infrastructure improvements

**15% Increase**  
in time for self-improvement

**60% Decrease**  
in application release time

... T ... Systems ...

---

# Code time

..T..Systems.....

# Task

---

## Description

<http://localhost:9090/> - config

<http://localhost:9091/> - eureka service

## Post service

<http://localhost:2142/swagger-ui.html>

<http://localhost:2142/h2>

## Document service

<http://localhost:2143/swagger-ui.html>

<http://localhost:2143/h2>

## User service

<http://localhost:2144/swagger-ui.html>

<http://localhost:2144/h2>

# Task

---

## Exercise#1

- 1) Up config service
- 2) Up eureka service
- 3) Up others service
- 4) Get parcels for email1 user

## Exercise#2

How many users in users db?



# Task

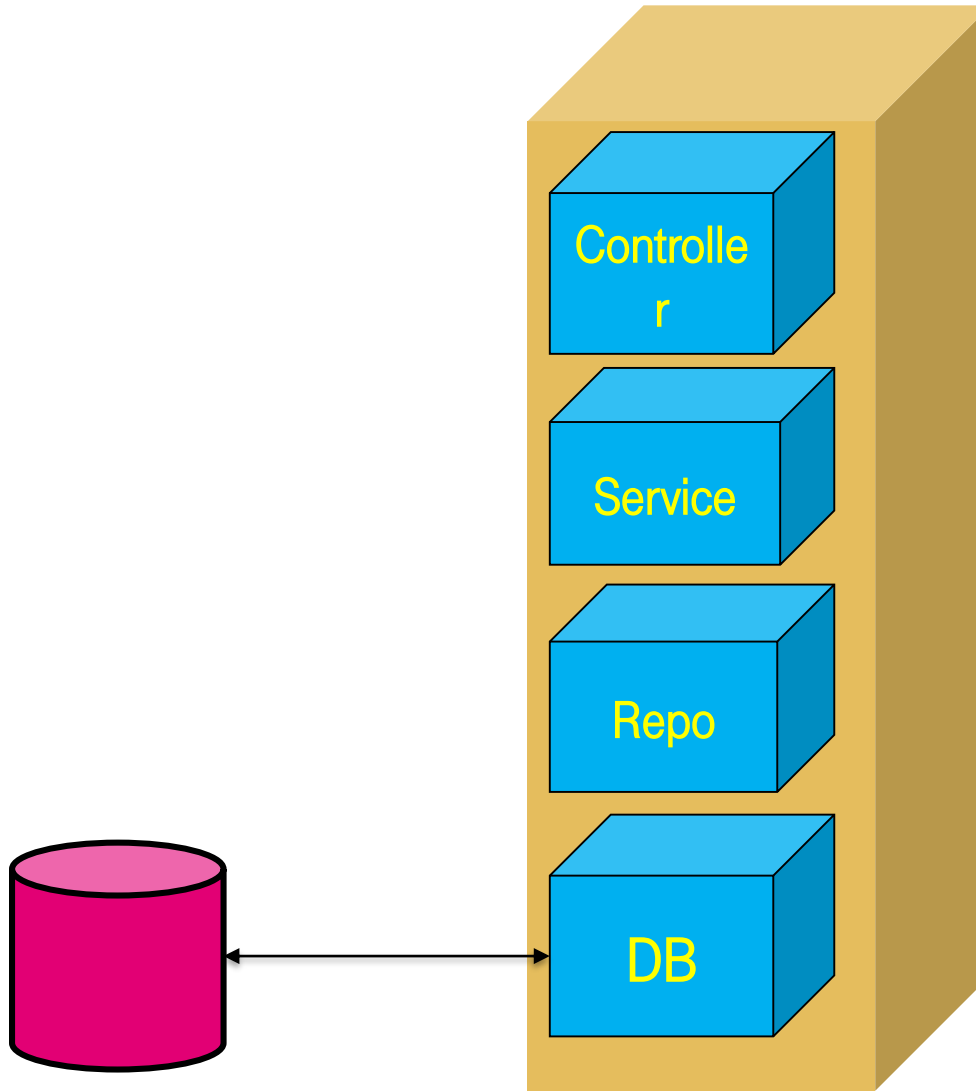
---

## **Exercise#3**

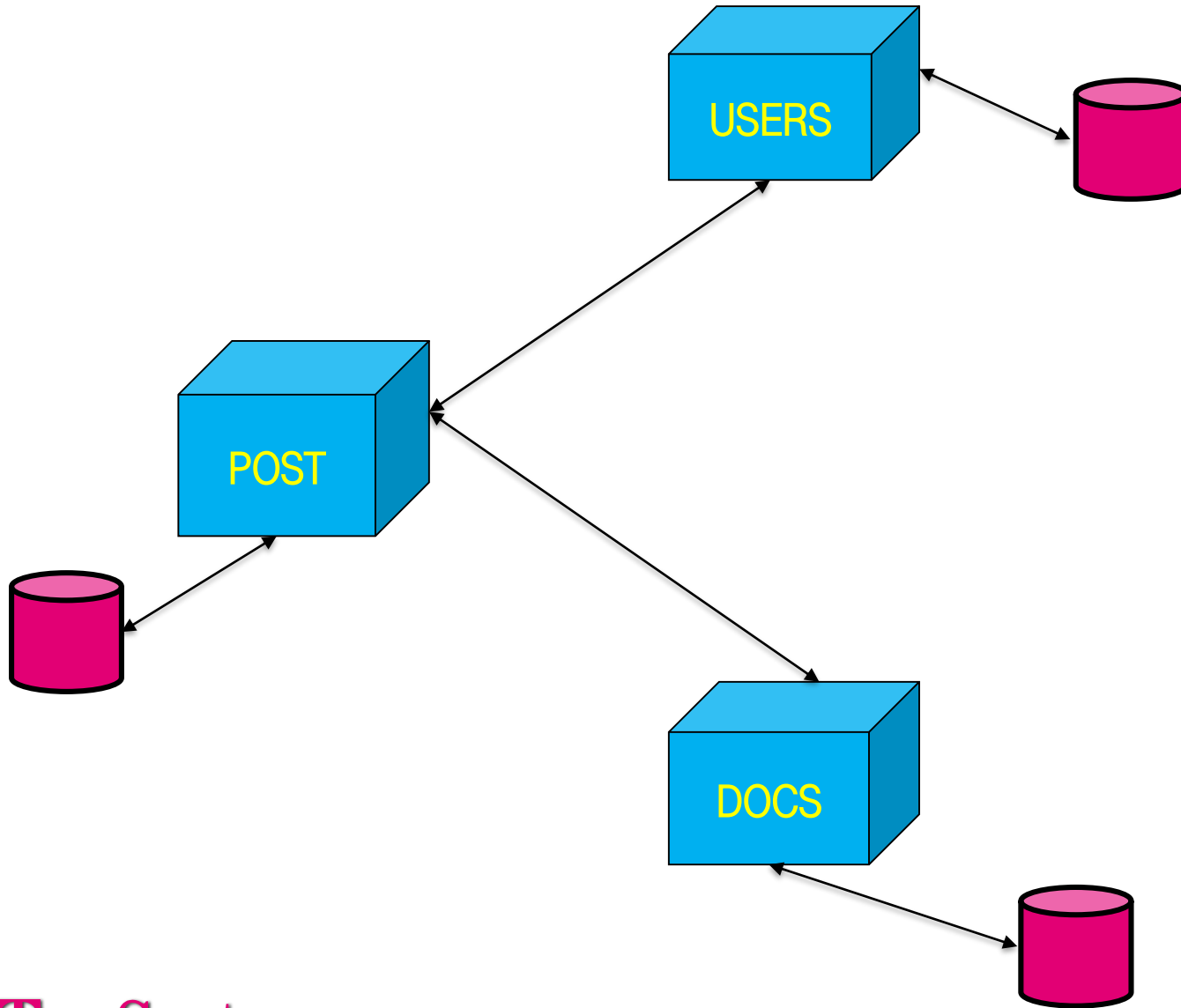
- 1) Get list of all users on user-service**
- 2) Turn off document-service**
- 3) Get list again**
- 4) What changed?**

## **Exercise#4 (Advanced)**

**Find root cause why we can't create parcel for users**



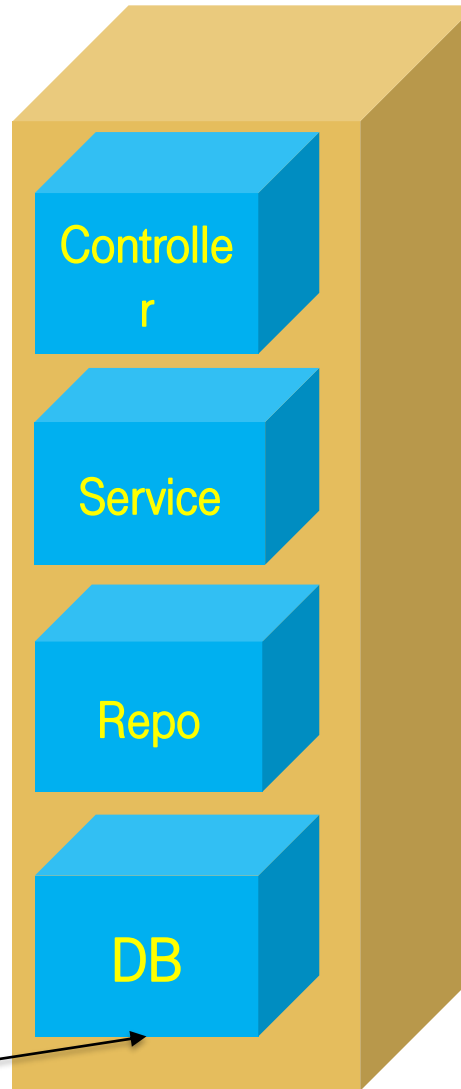
...T...Systems.....



...T...Systems...

# Task

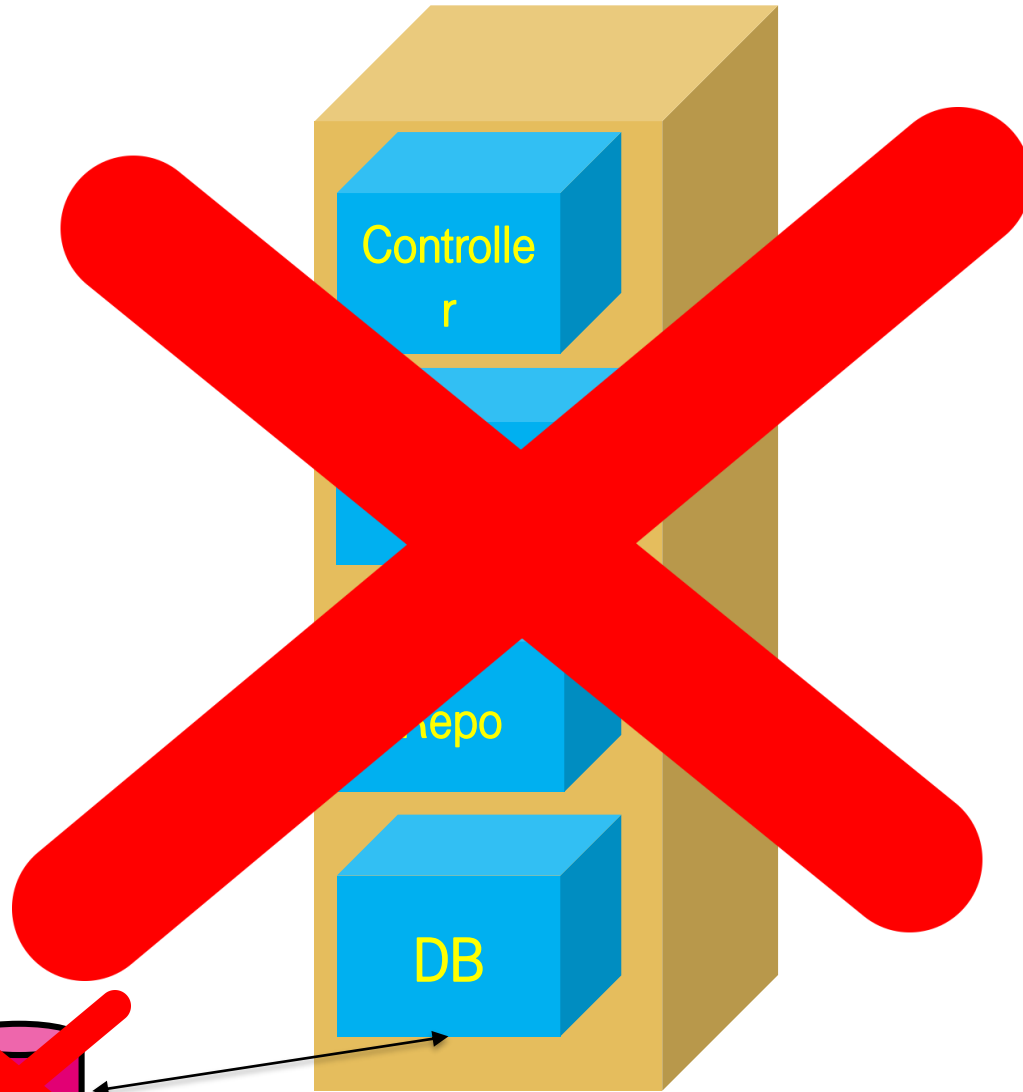
---



... T Systems ...

# Task

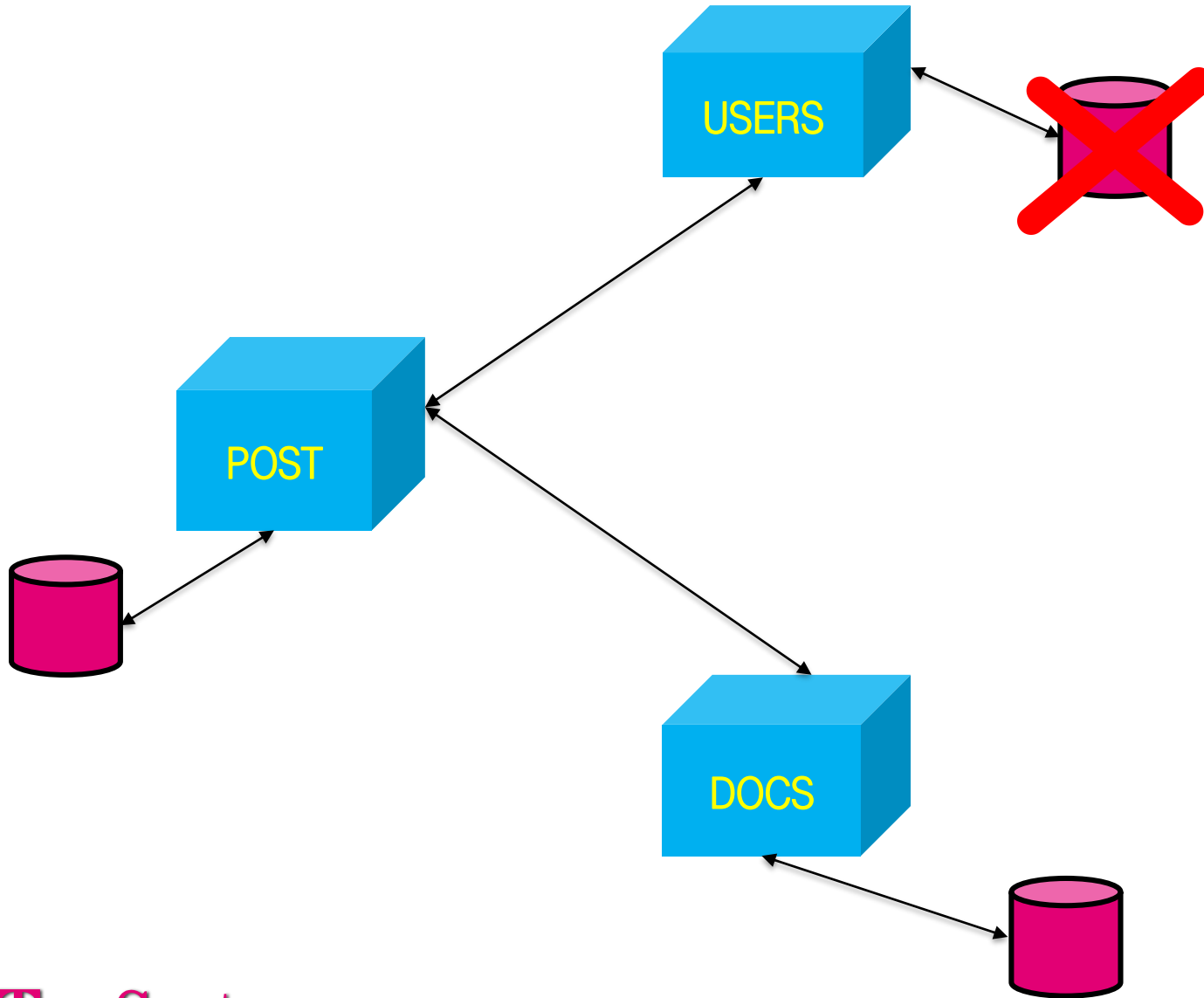
---



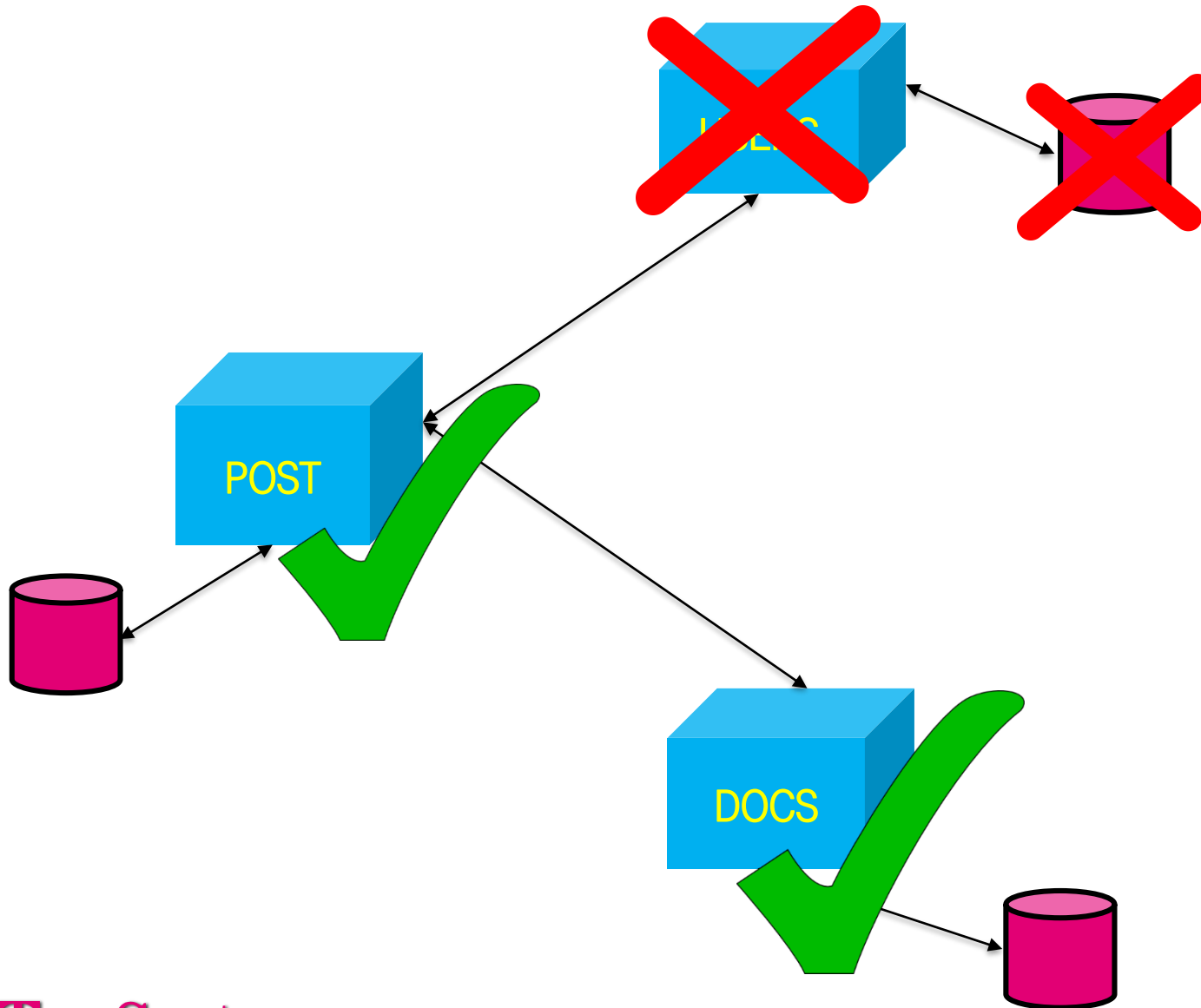
... ~~T~~ Systems ...

# Task

---



... T ... Systems ...



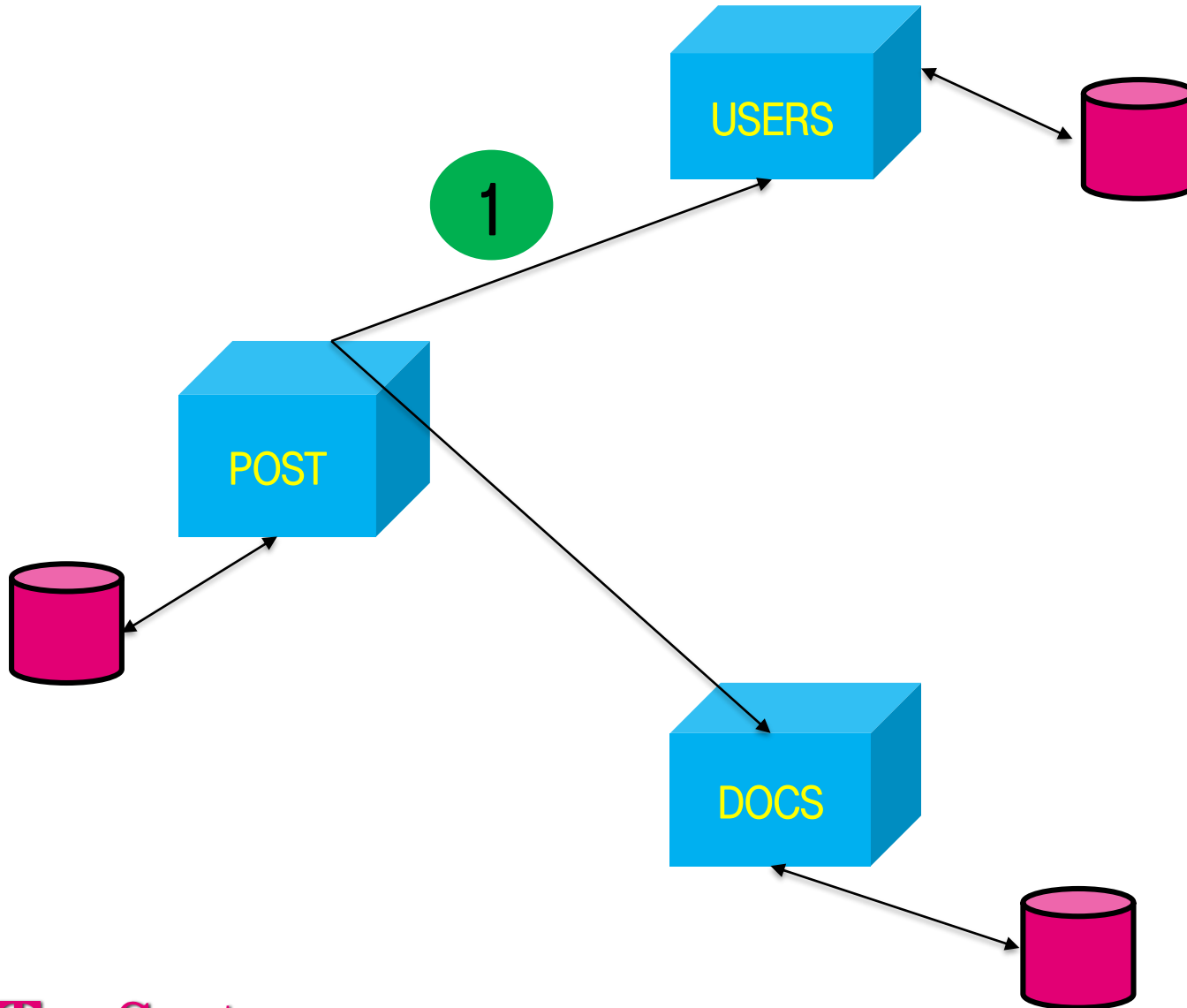
... T ... Systems ...

---

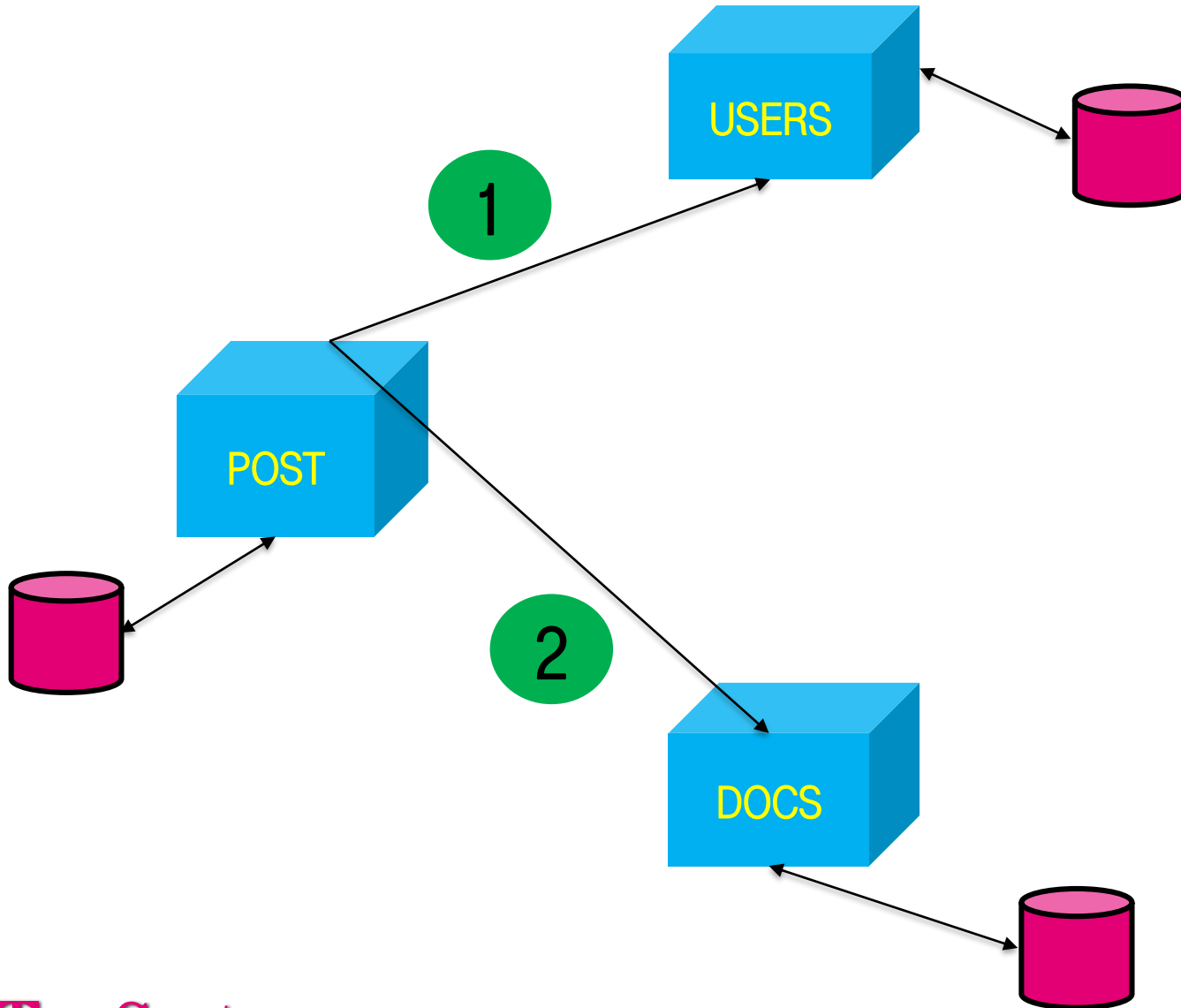
# DO CODE

...T...Systems.....

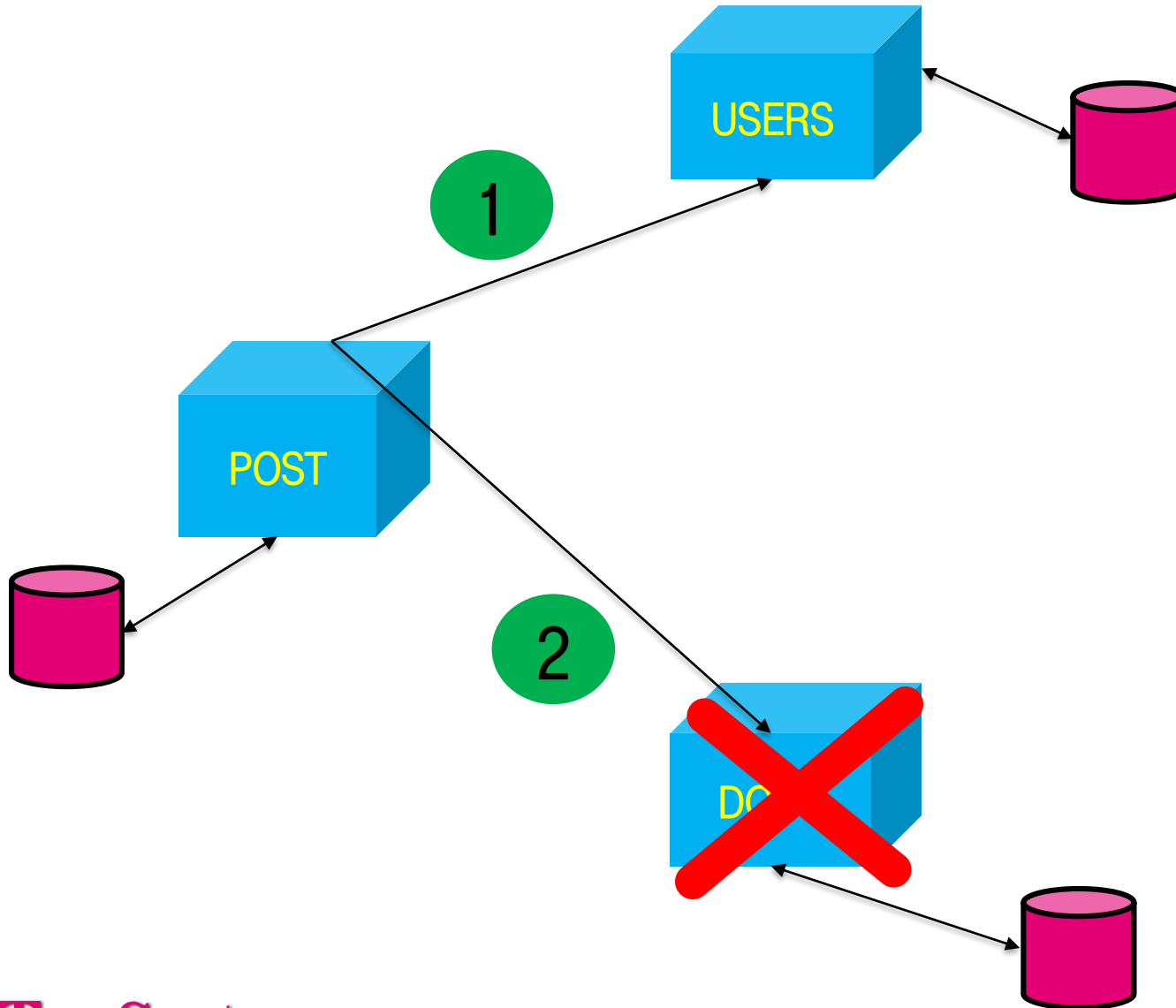




... T ... Systems ...



... T ... Systems ...

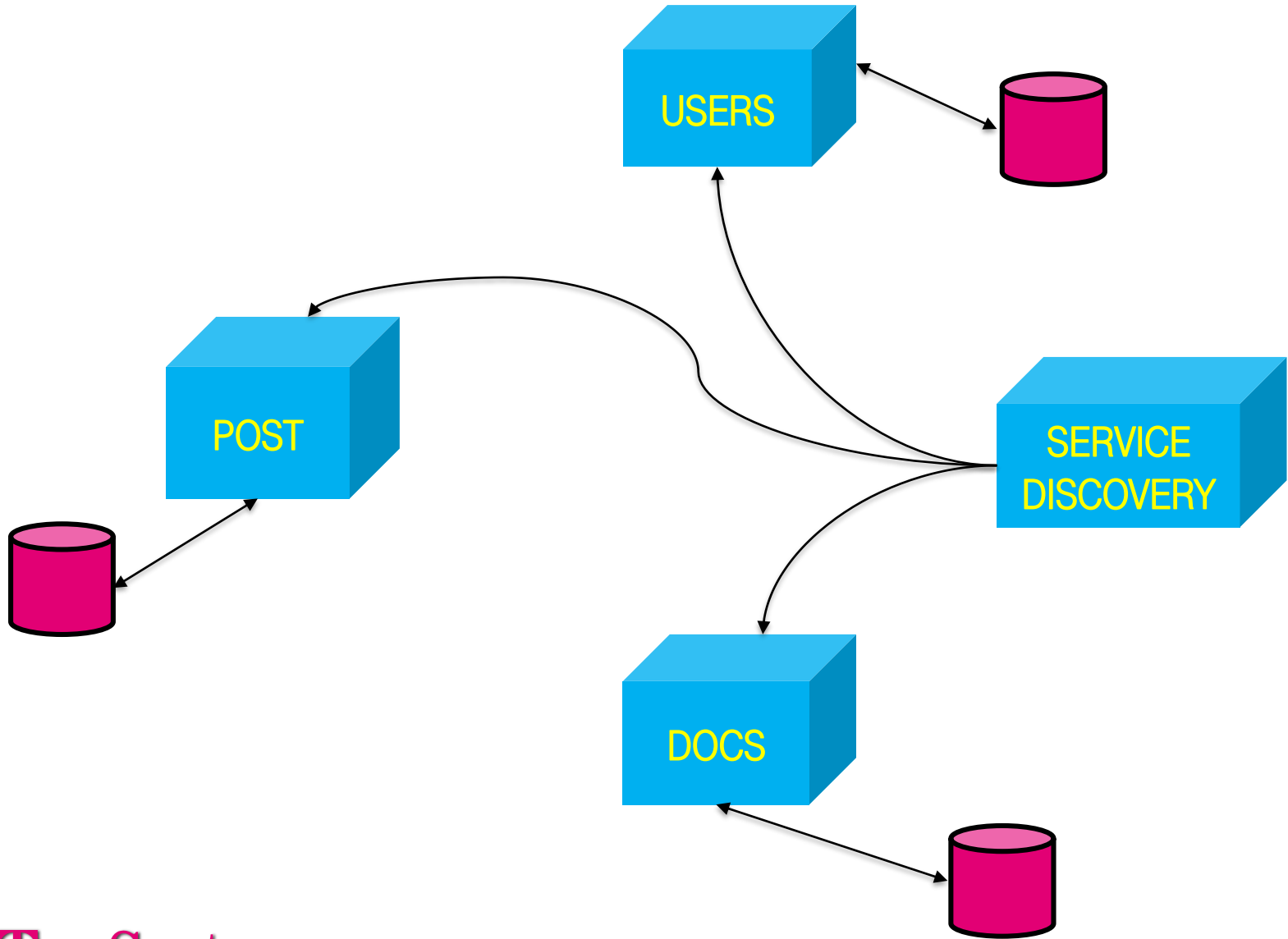


... T ... Systems ...

---

# DO CODE

...T...Systems.....



...T...Systems...

---

# DO CODE

...T...Systems.....

# Reasons to use a microservice architecture

---

- Microservices create flexibility when it comes to choosing which tool or technology one wants to use when solving problems.
- Microservices can deal with code churn. Integration can be challenging especially when it is handled by a third party. The risk of breaking integrations is lessened when microservices are used because they are very adaptable.
- Microservices are scalable, they can be used to split systems allowing more defined skill sets.
- Microservices help isolate the unknown. By using microservices your team can have the tools they need to defer issues that can make a software's architecture change and unable to respond to business requirements.

(Мартин Фаулер)

# Reasons NOT to use a microservice architecture

---

- More integration work. Microservices can cause additional work because integrating services with each other and other technologies takes time. You have to be cautious not to overload and break the microservices.
- Deployment. Deploying multiple services is more challenging than deploying a single service.
- Cognitive Complexity. It can make it nearly impossible to trace logging or issue tracking data across service boundaries. The network complexity has the potential to tarnish the integrity of the services true purpose.

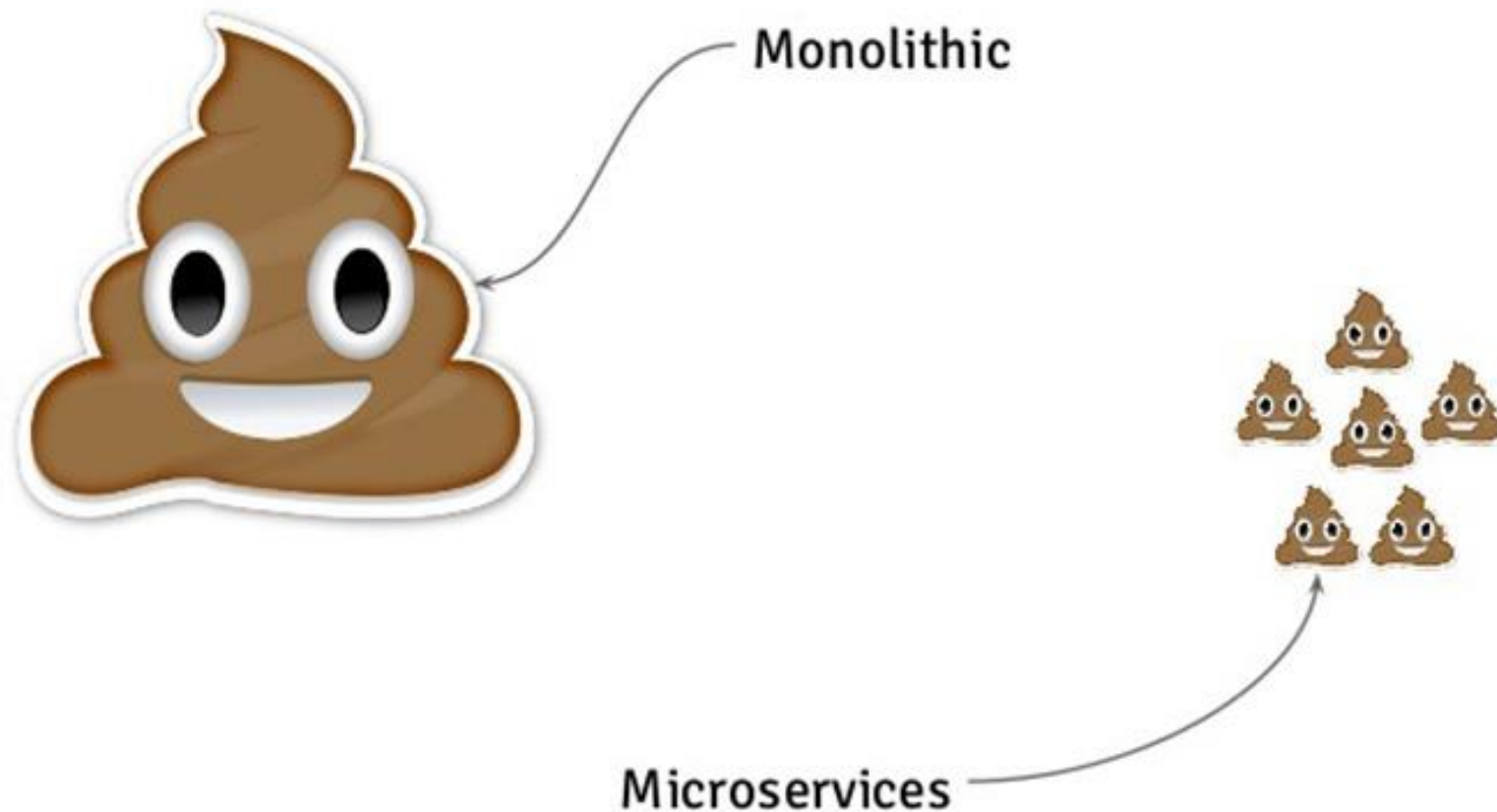
(Мартин Фаулер)



# Заключение

Категория	Монолитная архитектура	Микросервисная архитектура
Код	Единая база кода для всего приложения	Каждый микросервис имеет свою базу кода
Сложность понимания	Часто встречается запутанная логика, сложность поддержки	Чтение и восприятие намного легче, легкость поддержки
Развертывание	Комплексное развертывание с запланированными техническими поддержками и окнами профилактик	Каждый микросервис развертывается независимо с коротки либо нулевым временем простоя
Язык	Обычно разрабатывается с на одном языке	Микросервисы могут быть разработаны на различных языках
Масштабируемость	Требует масштабировать все приложение целиком, даже если известно, где находится “бутылочное горлышко”	Позволяет масштабировать необходимые сервисы без масштабирования всего приложения

## Monolithic vs Microservices



... T ... Systems ...

# Литература

---

- Шаблоны корпоративных приложений (Мартин Фаулер)
- Создание микросервисов (Сэм Ньюмен)
- <https://www.youtube.com/watch?v=wgdBVIX9ifA> (GOTO 2014 • Microservices • Martin Fowler)
- <https://www.youtube.com/watch?v=HDq1KQT51kM> (Максим Сячин — Микросервисы: первая кровь)
- [https://www.youtube.com/watch?v=7WT\\_Rl6m2DU](https://www.youtube.com/watch?v=7WT_Rl6m2DU) (Преимущества и недостатки микросервисной архитектуры в HeadHunter / Антон Иванов (HeadHunter))



**THANK YOU!**