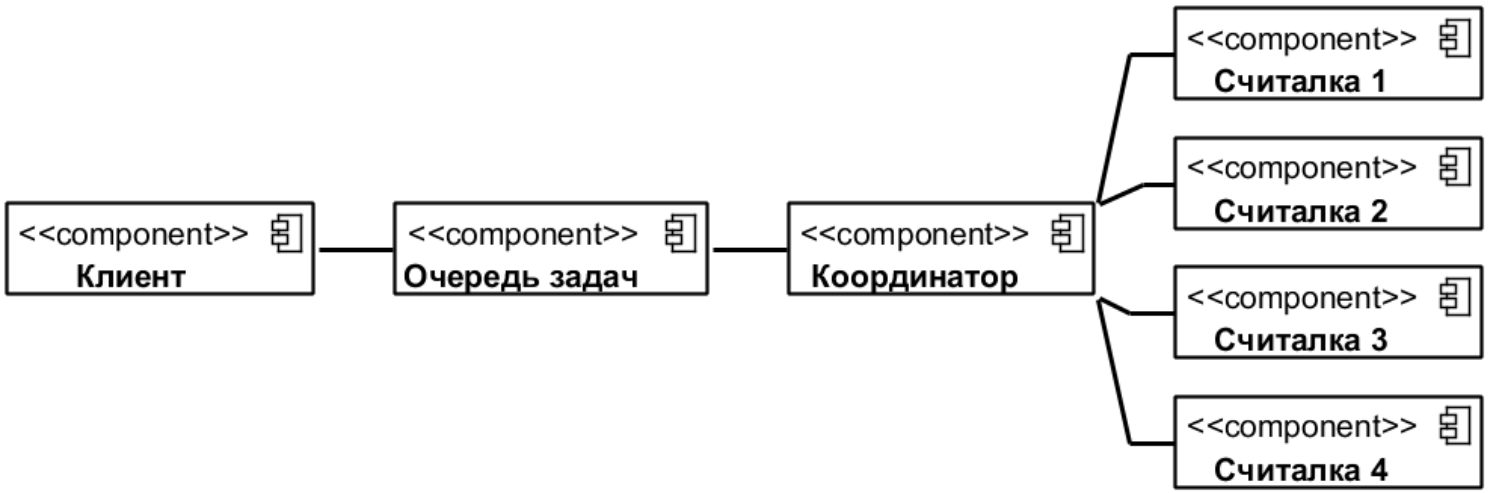
Лекция 14: Проектирование распределённых приложений

Часть вторая: стратегические вопросы

Юрий Литвинов
y.litvinov@spbu.ru

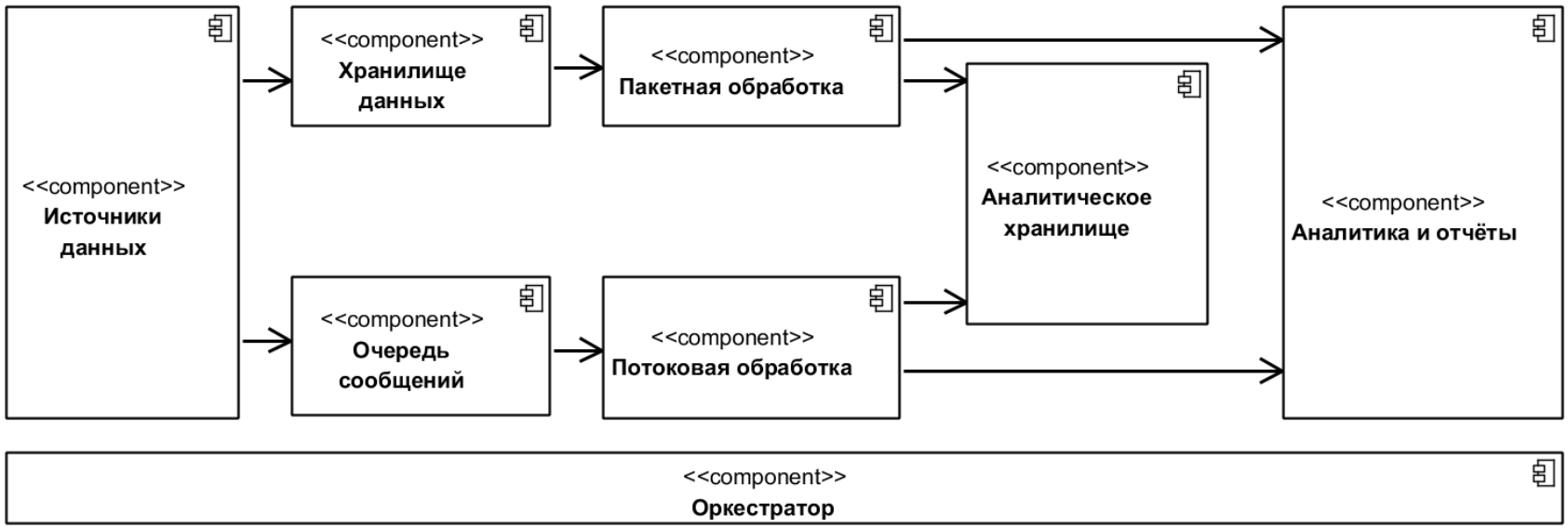
14.12.2021

Big Compute



- Для сверхсложных задач, предполагающих тысячи вычислительных узлов
- Требует «embarrassingly parallel» задачу
- Предполагает использование весьма продвинутых (и дорогих) облачных ресурсов

Big Data

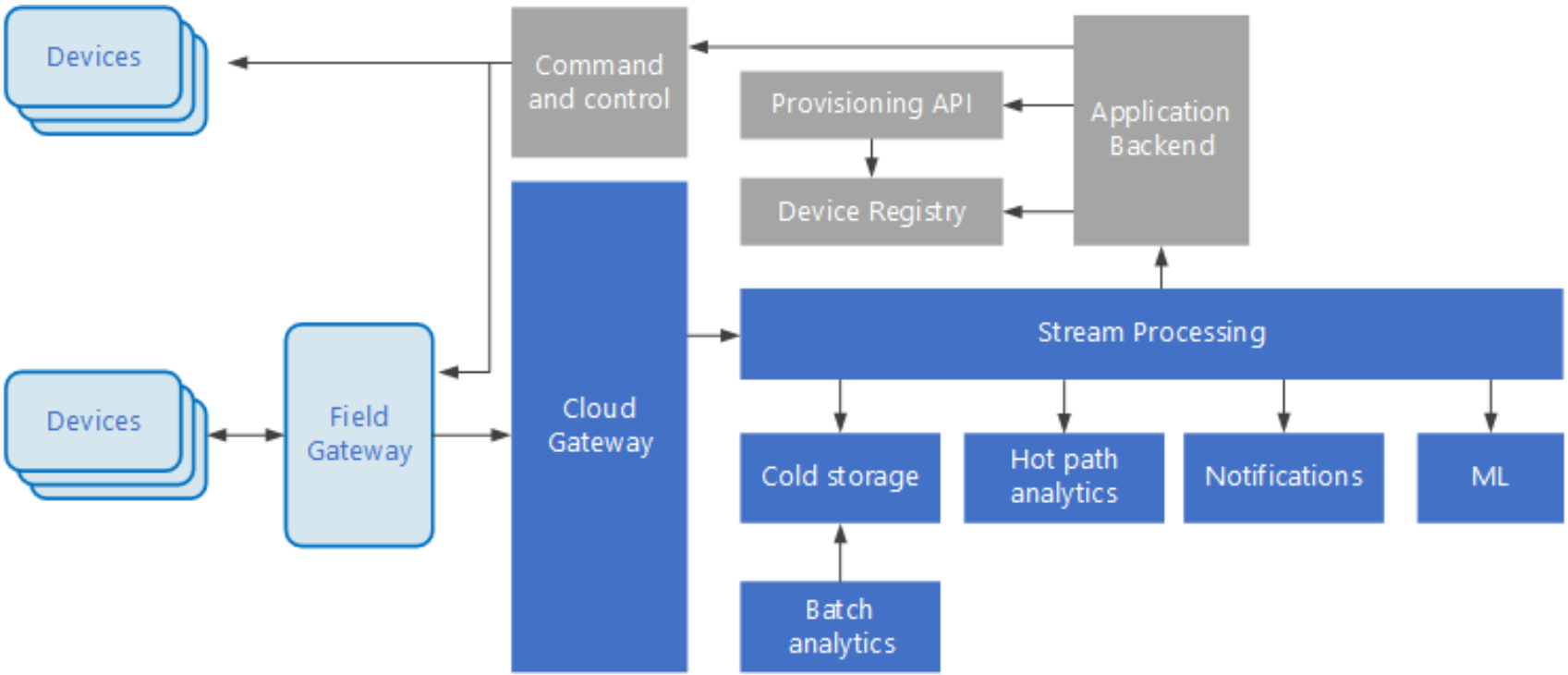


- Для аналитики над большими данными
  - Либо данных много и их можно обрабатывать неторопливо
  - Либо данных много и их надо обрабатывать в реальном времени
- Данные не лезут в обычную СУБД

Big Data, хорошие практики

- Распределённые хранение и обработка
  - Например, Apache Hadoop, Apache Spark
- Schema-on-read
  - Data lake — распределённое хранилище слабоструктурированных данных
- Обработка на месте
- Разделение данных по интервалам обработки
- Раннее удаление приватных данных

Пример: IoT

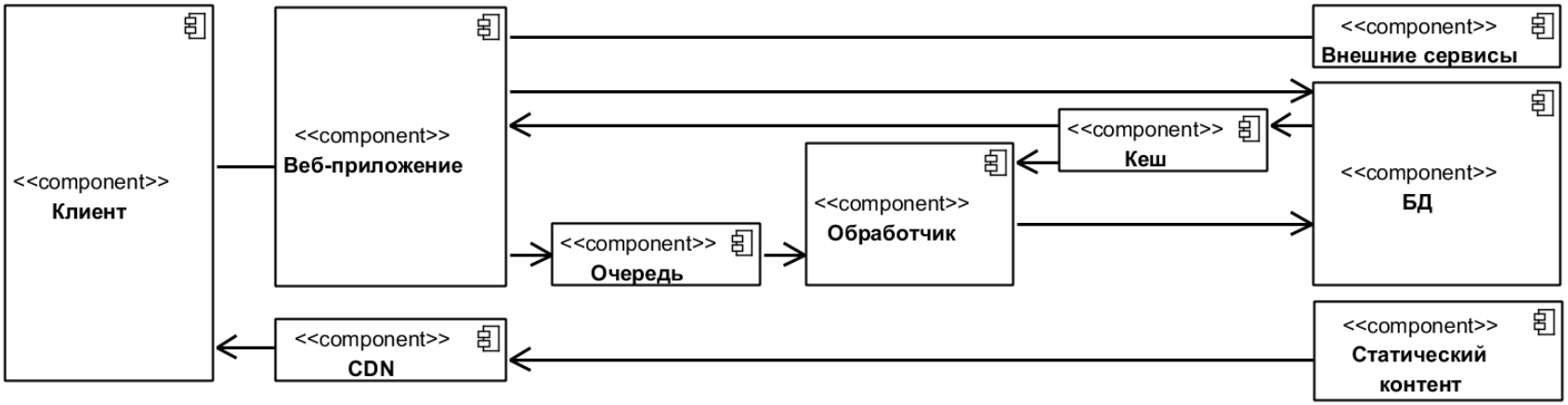


© https://github.com/MicrosoftDocs/architecture-center/blob/main/docs/guide/architecture-styles/big-data.md

Событийно-ориентированная архитектура

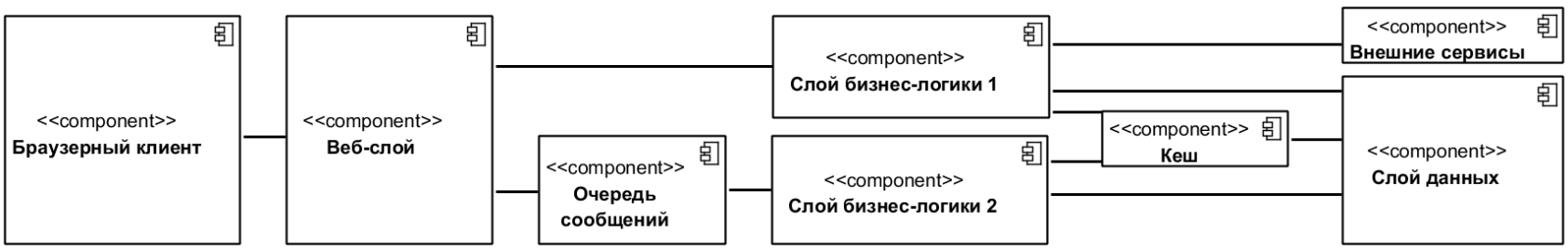
- Для обработки событий в реальном времени
- Бывает двух видов:
  - Издатель/подписчик (например, RabbitMQ)
  - Event Sourcing (например, Apache Kafka)

Web-queue-worker



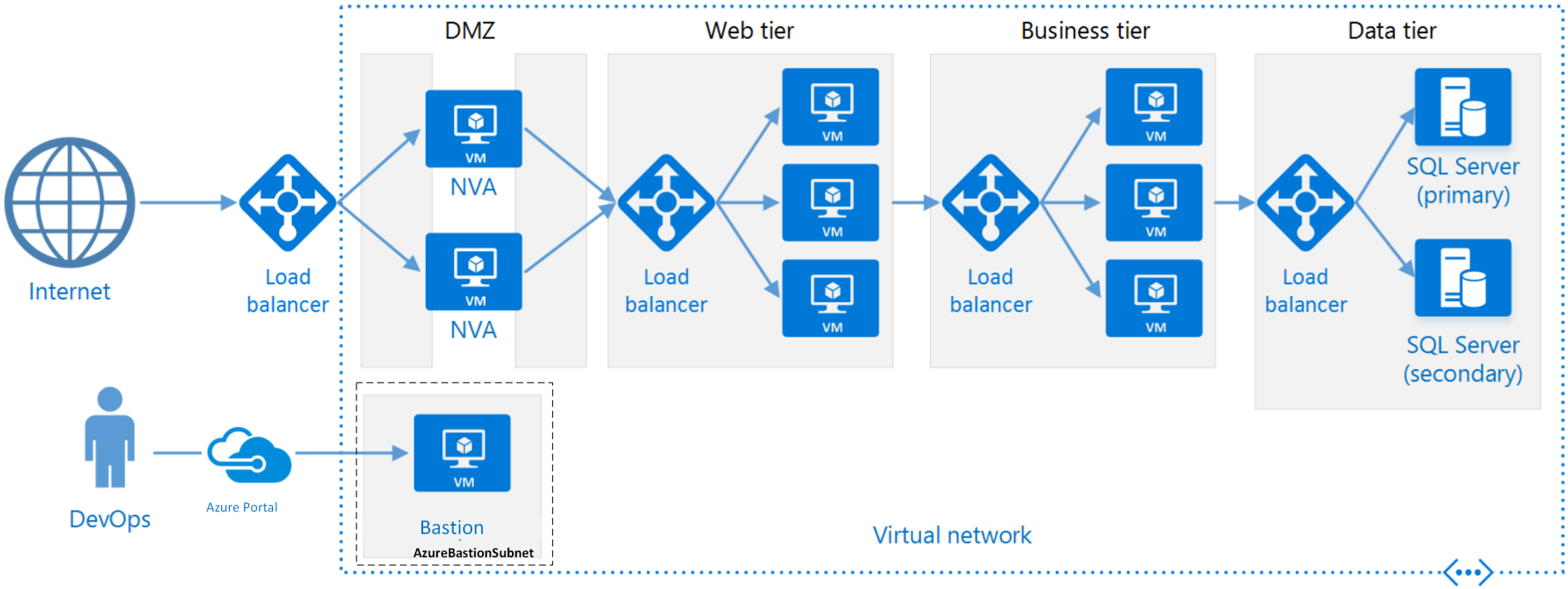
- Для вычислительно сложных задач в несложной предметной области
- Позволяет эффективно использовать готовые сервисы
- Независимое масштабирование фронтенда и обработчика
- Может превратиться в Big Ball of Mud

N-звенная архитектура



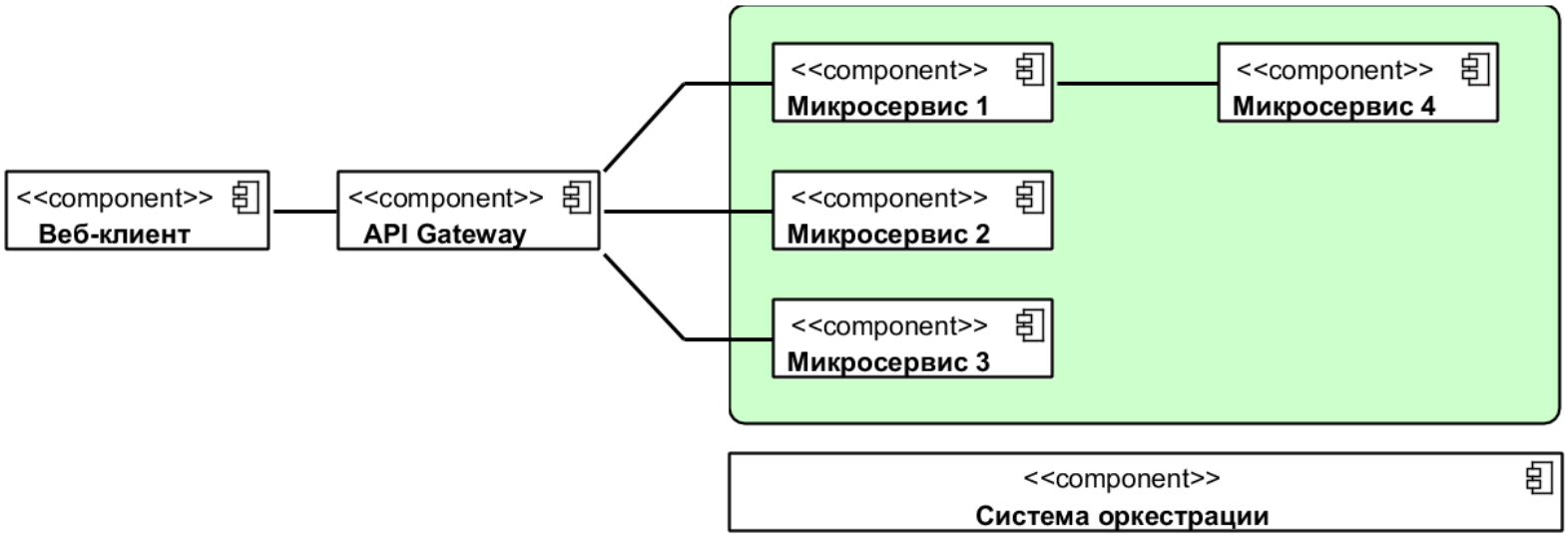
- Для быстрого переноса монолита в облако
- Для простых веб-приложений
- Проблемы с масштабированием и сопровождаемостью

Пример: N-звенное приложение на Azure



© https://github.com/MicrosoftDocs/architecture-center/blob/main/docs/guide/architecture-styles/n-tier.md

Микросервисная архитектура



- Для приложений со сложной предметной областью
- Альтернатива монолиту, со своими достоинствами и недостатками
- Микросервис пишется одним человеком за две недели
  - На самом деле, пишется и поддерживается небольшой командой
- Микросервис — ограниченный контекст в смысле DDD

Особенности

- Каждый микросервис — отдельное приложение
  - Независимость языков и технологий
  - Имеет своё хранилище данных, не имеет права шарить данные
    - В каком-то смысле, объект из ООП
    - Каждому сервису наиболее подходящая СУБД
- Мелкозернистая масштабируемость
- Независимое развёртывание
- Изоляция ошибок
- Маленькая и простая кодовая база

Проблемы

- Сложность перекладывается с реализации на оркестрацию
  - Неочевидно, неразвитые инструменты
  - В целом сложнее, чем рассмотренные выше стили
  - Сложное управление и мониторинг, требуется развитая культура DevOps
  - Сложная в плане управления зависимостями разработка
- Технологический зоопарк
- Нагрузка на сеть
- Сложно поддерживать целостность данных
  - Eventual Consistency

Representational State Transfer (REST)

- Самая популярная сейчас архитектура веб-сервисов
- Передача всего необходимого в запросе
  - Нельзя хранить состояние сессии
- Стандартизованный интерфейс, очень простые запросы
- Стандартные протоколы (в основном поверх HTTP)
- Обычно JSON как формат сериализации
- Кеширование

Интерфейс сервиса

- Коллекции
  - http://api.example.com/customers/
- Элементы
  - http://api.example.com/customers/17
- HTTP-методы (GET, POST, PUT, DELETE), стандартная семантика, стандартные коды ошибок
- Передача параметров прямо в URL
  - http://api.example.com/customers?user=me&access_token=ASFQF

Пример, Google Drive REST API

- GET https://www.googleapis.com/drive/v2/files — список всех файлов
- GET https://www.googleapis.com/drive/v2/files/fileId — метаданные файла по его Id
- POST https://www.googleapis.com/upload/drive/v2/files — загрузить новый файл
- PUT https://www.googleapis.com/upload/drive/v2/files/fileId — обновить файл
- DELETE https://www.googleapis.com/drive/v2/files/fileId — удалить файл

Дизайн REST-интерфейса

- API строится вокруг ресурсов, не действий
  - http://api.example.com/customers/ — хорошо
  - http://api.example.com/get_customer/ — плохо
- Отношения между сущностями: http://api.example.com/customers/5/orders
  - Максимум одно отношение — надо будет, сделают ещё запросы
- API — модель предметной области, не данных
- Семантика HTTP
  - Заголовки Content-Type, Accept
  - Коды возврата (200, 204, 404, 400, 409)
- Механизмы фильтрации и «пагинации»
- Поддержка Partial Content
- HATEOAS
- Версионирование — не ломать обратную совместимость

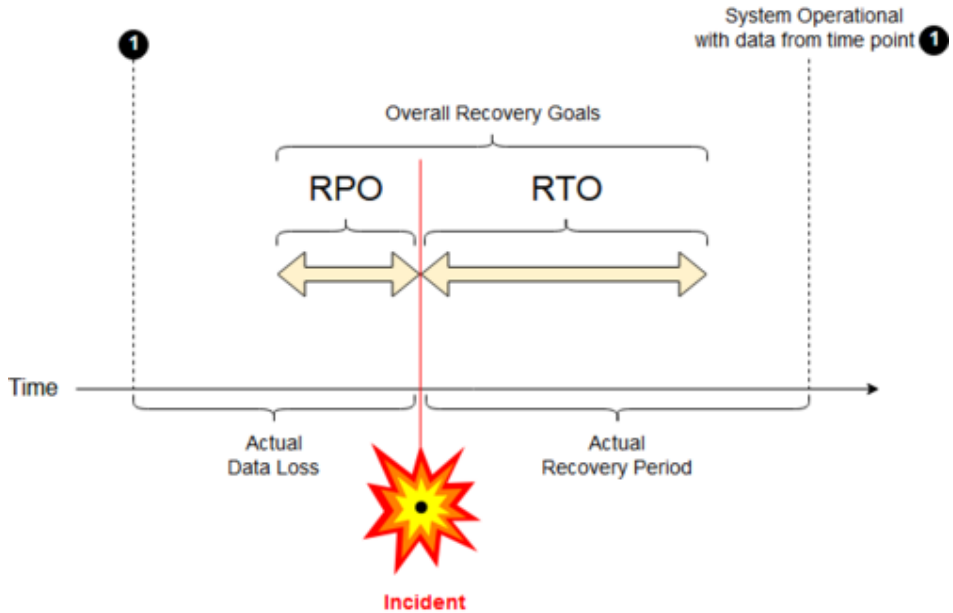
Общие принципы дизайна распределённых приложений

Самовосстановление

- Повтор при временном отказе
- Паттерн «Circuit Breaker»
- API для самодиагностики
- Разделение на изолированные группы ресурсов
- Буферизация запросов
- Автоматическое переключение на резервный экземпляр, ручное обратно
- Промежуточное сохранение
- Плавная потеря работоспособности (graceful degradation)
- Тестирование отказов, Chaos engineering

Избыточность

- Бизнес-требования к надёжности
  - Recovery Time Objective, Recovery Point Objective, Maximum Tolerable Outage
- Балансировщики нагрузки
- Репликация БД
- Разделение по регионам
- Шардирование



©

https://en.wikipedia.org/wiki/Disaster_recovery

Минимизация координации

- Eventual Consistency, компенсационные транзакции
- Доменные события (domain events)
- Паттерн «Command and Query Responsibility Segregation» (CQRS)
- Event Sourcing
- Асинхронные, идемпотентные операции
- Шардирование

CAP-теорема

В любой распределённой системе можно обеспечить не более двух из трёх свойств:

- Согласованность данных (Consistency) — во всех вычислительных узлах данные консистентны
- Доступность (Availability) — любой запрос завершается корректно, но без гарантии, что ответы всех узлов одинаковы
- Устойчивость к разделению (Partitioning Tolerance) — потеря связи между узлами не портит ответы
  - Этот пункт в распределённых системах должен быть обеспечен всегда, потому что отказы неизбежны. Остаётся выбрать один из двух

ACID vs BASE

ACID:

- Atomicity — транзакция не применится частично
- Consistency — завершённая транзакция не нарушает целостности данных
- Isolation — параллельные транзакции не мешают друг другу
- Durability — если транзакция завершилась, её данные не потеряются

BASE:

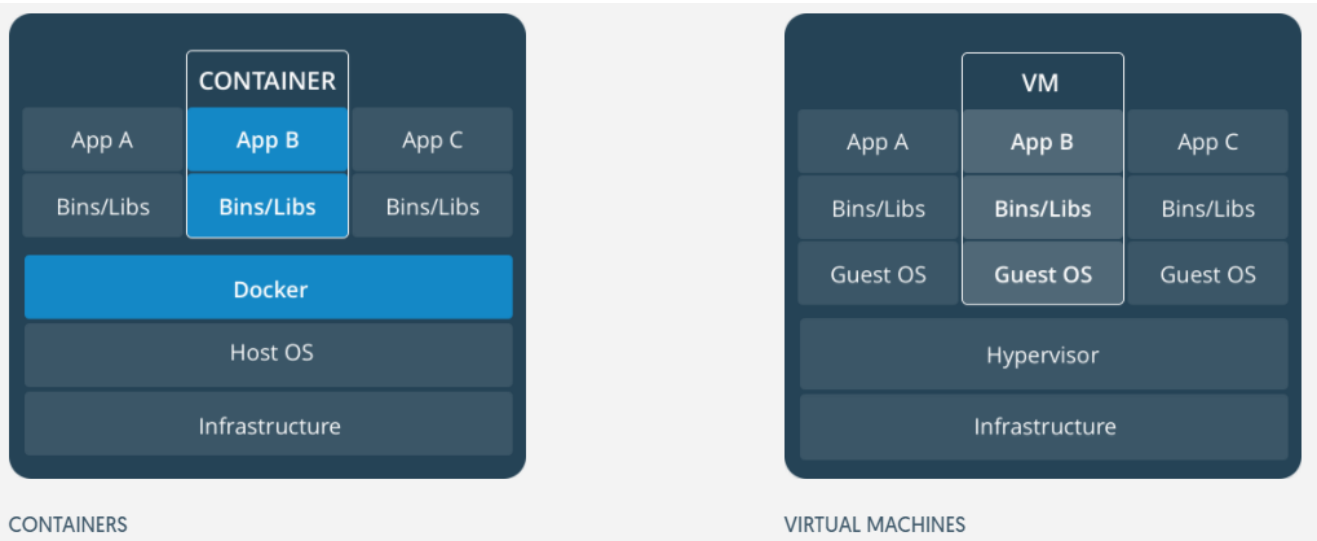
- Basically Available — отказ узла может привести к некорректному ответу, но только для клиентов, обслуживавшихся узлом
- Soft-state — состояние может меняться само собой, согласованность между узлами не гарантируется
- Eventually consistent — гарантируется целостность только в некоторый момент в будущем

Проектирование для обслуживания

- Делать всё наблюдаемым
  - Трассировка, в т.ч. распределённая
  - Логирование
- Мониторинг, метрики
- Стандартизация форматов логов и метрик
- Автоматизация задач обслуживания
- Конфигурация — это код

Docker

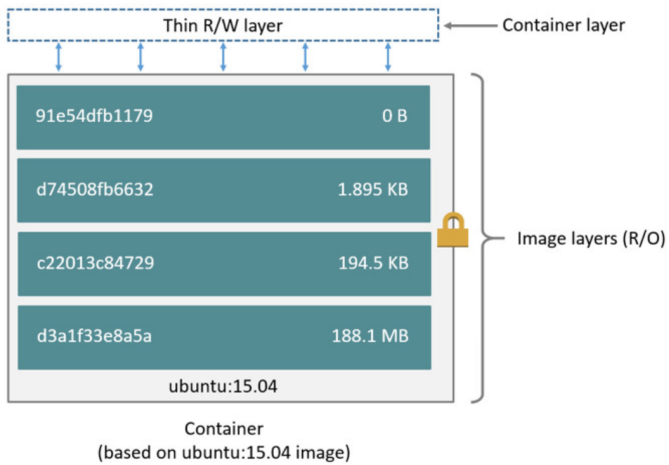
- Средство для “упаковки” приложений в изолированные контейнеры
- Что-то вроде легковесной виртуальной машины
- Широкий инструментарий: DSL для описания образов, публичный репозиторий, поддержка оркестраторами



© https://www.docker.com

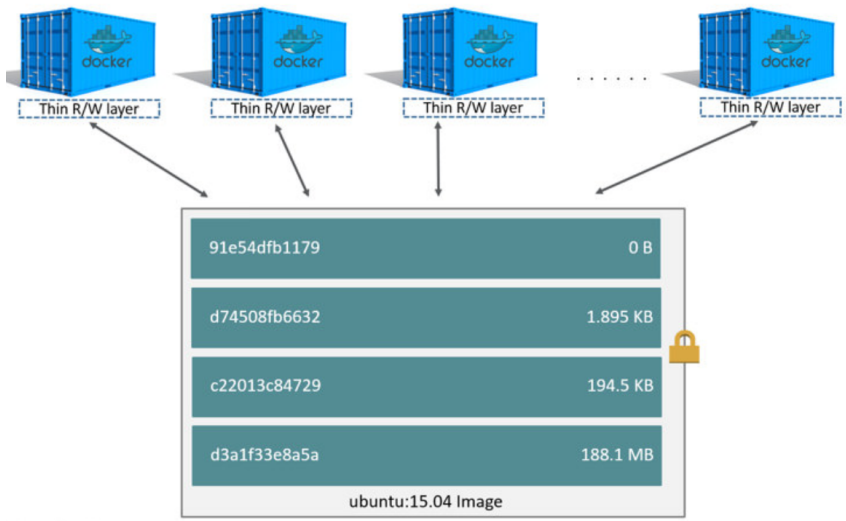
Docker Image

- Окружение и приложение
- Состоит из слоёв
  - Все слои read-only
  - Образы делят слои между собой как процессы делят динамические библиотеки
- На основе одного образа можно создать другой



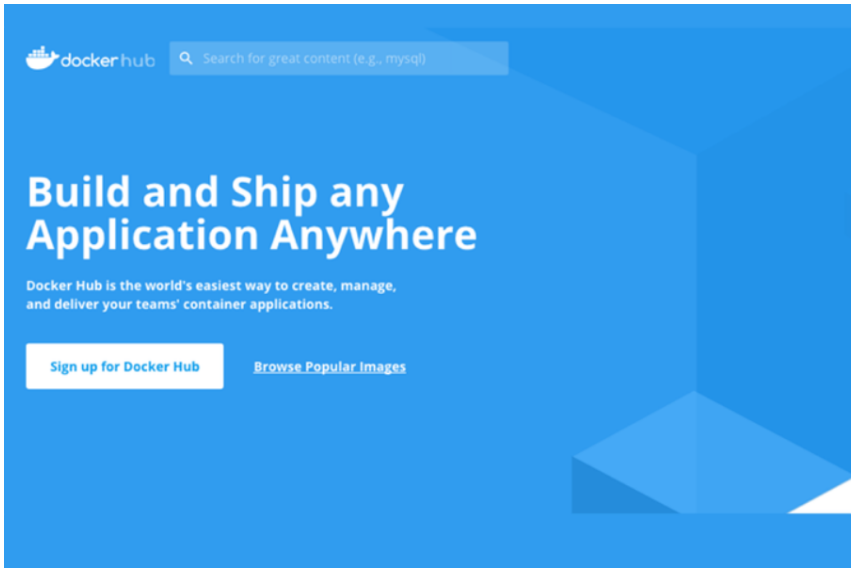
Docker Container

- Образ с дополнительным write слоем
- Содержит один запущенный процесс
- Может быть сохранен как новый образ



DockerHub

- Внешний репозиторий образов
  - Официальные образы
  - Пользовательские образы
  - Приватные репозитории
- Простой CI/CD
- Высокая доступность



Базовые команды

- docker run — запускает контейнер (при необходимости делает pull)
  - -d — запустить в фоновом режиме
  - -p host_port:container_port — прокинуть порт из контейнера на хост
  - -i -t — запустить в интерактивном режиме
  - Пример: docker run -it ubuntu /bin/bash
- docker ps — показывает запущенные контейнеры
  - Пример: docker run -d nginx; docker ps
- docker stop — останавливает контейнер (шлёт SIGTERM, затем SIGKILL)
- docker exec — запускает дополнительный процесс в контейнере

Dockerfile

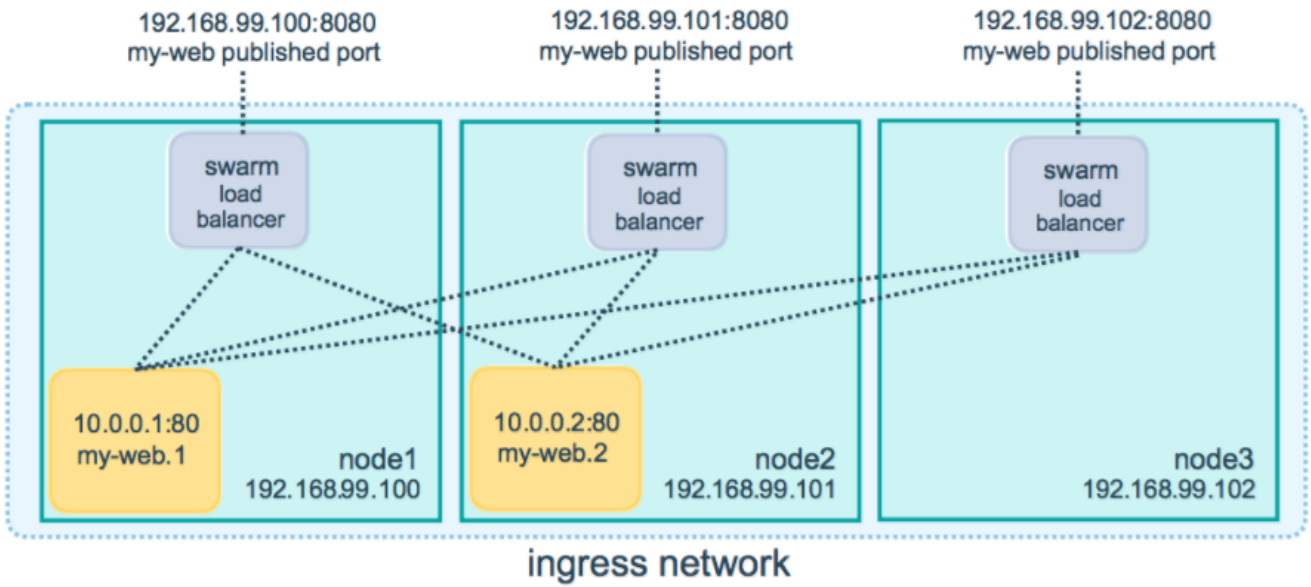
```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Docker Compose

```yaml
version: "3"
services:
  web:
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
      restart_policy:
        condition: on-failure
    ports:
      - "80:80"
    networks:
      - webnet
networks:
  webnet:
```

Docker Swarm

- Машина, на которой запускается контейнер, становится главной
- Другие машины могут присоединяться к swarm-у и получать копию контейнера
- Docker балансирует нагрузку по машинам



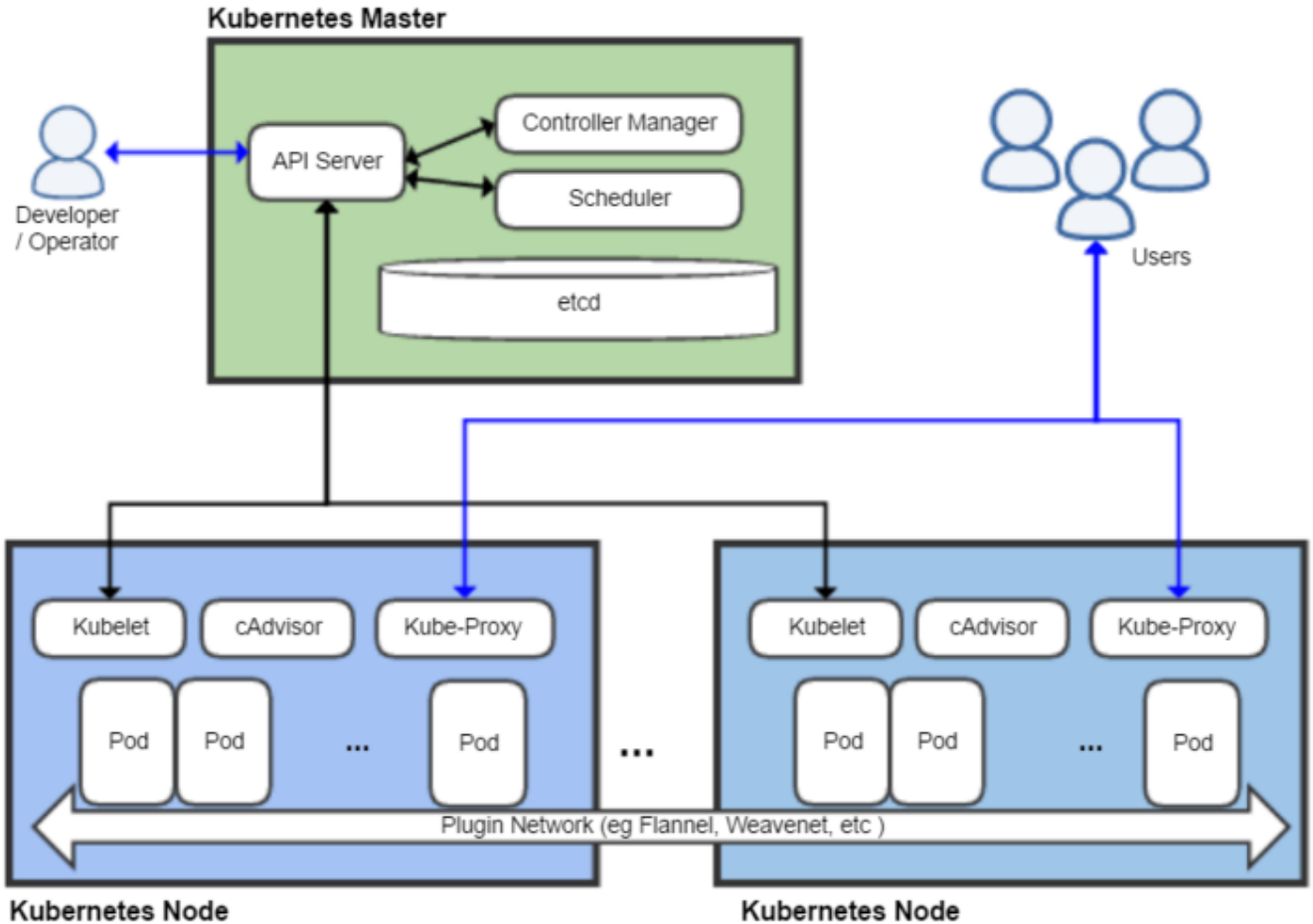
© https://www.docker.com

Kubernetes

- Оркестратор контейнеров
- Отвечает за раскидывание контейнеров по хостам, масштабирование, мониторинг и управление жизненным циклом
  - Сильно продвинутый Docker Compose
- Open-source, Google, Go

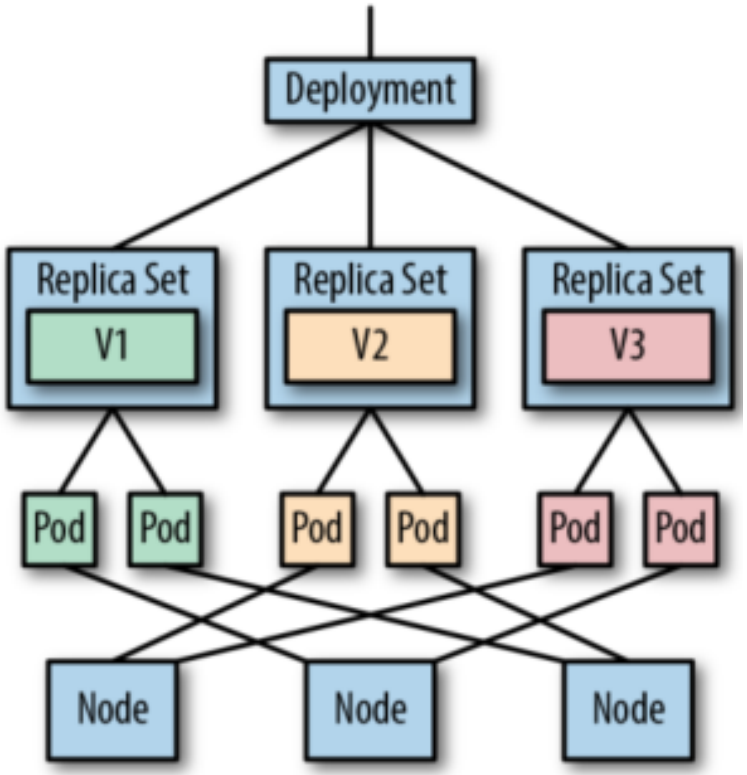
© https://kubernetes.io/

Архитектура Kubernetes



© https://ru.wikipedia.org/wiki/Kubernetes

Объекты Kubernetes



© J. Arundel, J. Domingus, Cloud Native DevOps with Kubernetes

Deployment

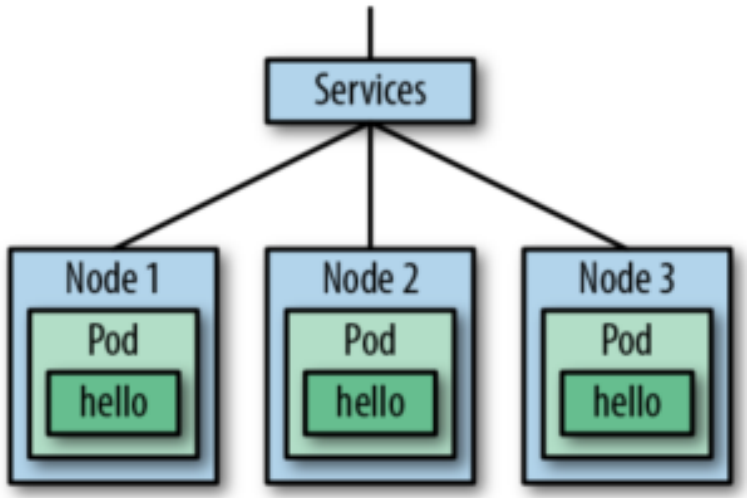
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo
  labels:
    app: demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: demo
  template:
    metadata:
      labels:
        app: demo
    spec:
      containers:
        - name: demo
          image: cloudnatived/demo:hello
          ports:
          - containerPort: 8888
```

Запуск:

kubectl apply -f k8s/deployment.yaml

© J. Arundel, J. Domingus, Cloud Native DevOps with Kubernetes

Сервисы



© J. Arundel, J. Domingus, Cloud Native DevOps with Kubernetes

Service

```
apiVersion: v1
kind: Service
metadata:
  name: demo
  labels:
    app: demo
spec:
  ports:
  - port: 9999
   protocol: TCP
    targetPort: 8888
  selector:
    app: demo
  type: ClusterIP
```

Запуск:

```
kubectl apply -f k8s/service.yaml
kubectl port-forward service/demo 9999:8888
```

Рекомендации и техники

- Конфигурация — это код, не управляйте кластером вручную
- Мониторинг

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 8888
  initialDelaySeconds: 3
  periodSeconds: 3
```

- Blue/green deployment, rainbow deployment, canary deployment
- Метрики: RED, USE
- Используйте инструменты
  - Helm

Облачная инфраструктура

- Виды сервисов:
  - Infrastructure as a Service
  - Platform as a Service
  - Software as a Service
- Основные провайдеры:
  - Amazon Web Services (почти 50% рынка)
  - Microsoft Azure (порядка 10%)
  - Google Cloud
  - Всё остальное (Heroku, Yandex.Cloud, ...)

Пример: экосистема AWS

- Вычисления:
  - EC2 (Elastic Computations)
  - ECS (Elastic Container Service)
- Сеть:
  - VPC (Virtual Private Cloud)
  - ELB (Elastic Load Balancer)
- Устройства хранения:
  - EFS (Elastic File System)
  - EBS (Elastic Block Storage)
- SaaS, базы данных:
  - RDS (Relational Database Service)
  - DynamoDB
  - ElasticSearch Service

Infrastructure as Code

«The enabling idea of infrastructure as a code is that systems and devices which are used to run software can be treated as if they, themselves, are software» (Infrastructure as Code, Kief Morris)

- Платформонезависимое представление инфраструктуры
- Воспроизводимое развёртывание
- Пример: Terraform