

# «Микросервисы и Контейнеры»

*26 октября 16:20 – 17:50 МСК*



**Альберт Халиулов**

Технический эксперт  
IBM Cloud

**Мы начинаем в 16:20 МСК**

Материалы для практических занятий:

<https://github.com/albert-haliulov/cloud-native-workshop-2020>

Программа для стартапов:

<http://ibm.biz/startwith>

Промо-код для IBM Cloud:

<https://ibm.biz/ibmcloudcoupon>

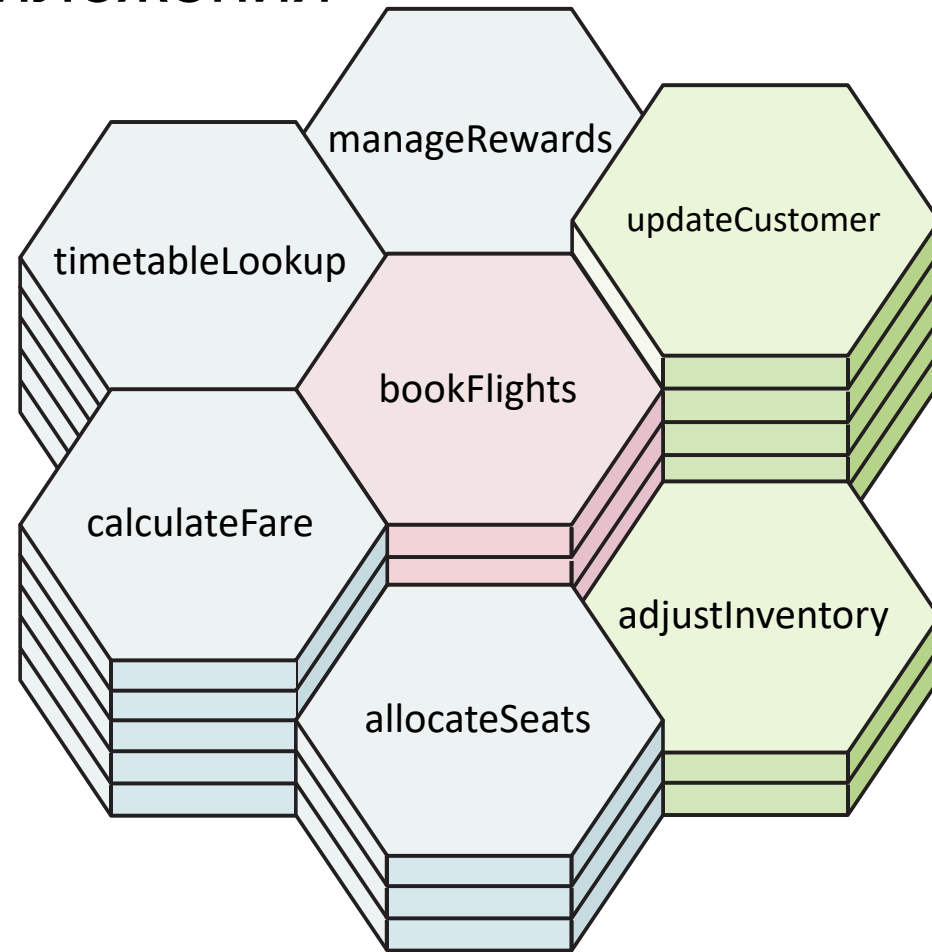
# Что такое микросервисы?

## Микросервисы это архитектурный стиль приложения

- Приложение состоит из микросервисных компонент

## Каждый микросервис:

- Миниатюрное приложение
- Сфокусированное на одной задаче
- Разворачивается и сопровождается независимо
- Не зависит от работы других микросервисов
- С четко определенным интерфейсом взаимодействия

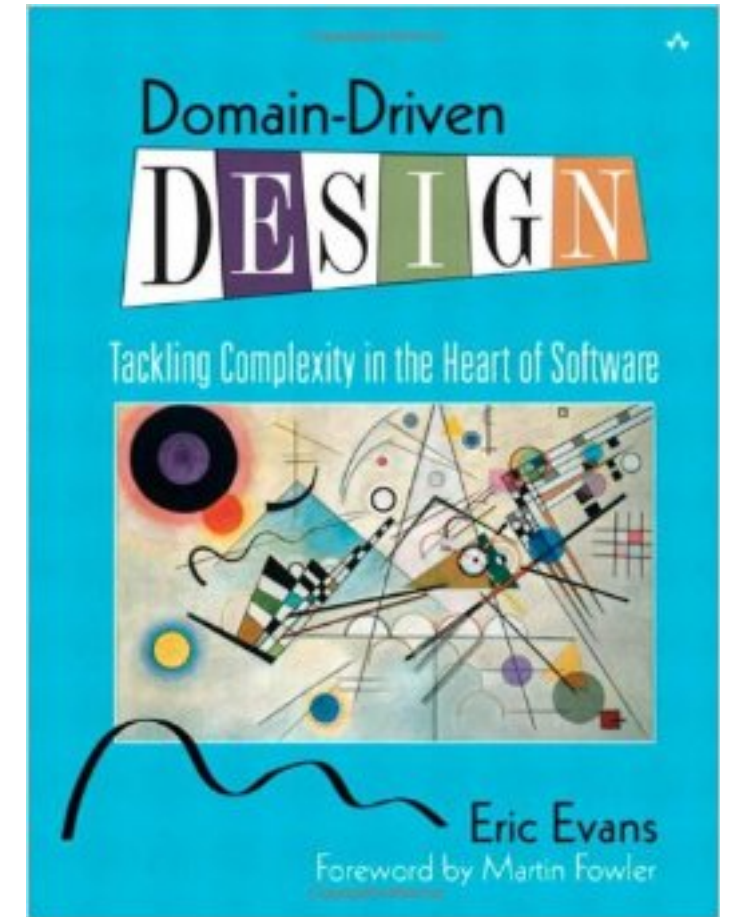


# Монолит и Микросервисы

	Монолит	Микросервис
Архитектура	Построен как логически единая исполняемая сущность	Набор отдельно запускаемых сервисов
Модульность	Основана на особенностях технологии	Основана на бизнес потребностях
Обновление	Пересборка всего приложения	Обновление того сервиса который необходимо
Масштабирование	Все приложение	Индивидуальное
Разработка	Обычно один язык программирования	Каждый сервис на подходящем языке программирования
Сопровождение	Большая кодовая база	Большое количество мелких сервисов
Развертывание	Комплексное развертывание и с запланированными простоями	Простое развертывание каждого сервиса и с минимальными простоями

# Преимущества Микросервисного подхода

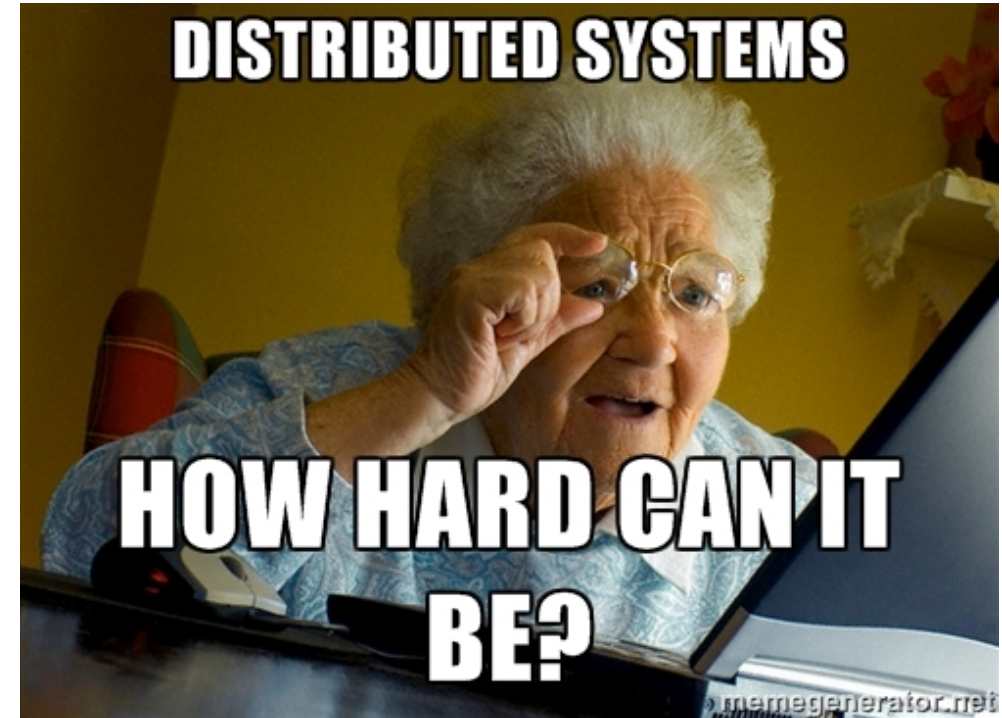
- Разрабатываются независимо
  - Ограниченная или полная независимость от других сервисов
- Разрабатывается одной командой
  - Команда небольшая
  - Все члены команды могут понимать всю кодовую базу
- Разработка идет по своему собственному расписанию
  - Новые версии поставляются независимо от других сервисов
- Каждый сервис может быть разработан на другом языке
  - Выбор наилучшего языка для реализации сервиса
- Управляет собственными данными
  - Выбор наилучшей технологии и схемы хранения данных
- Масштабируется и выходит из строя независимо от других
  - Изолирует проблемы





# Трудности при Микросервисном подходе

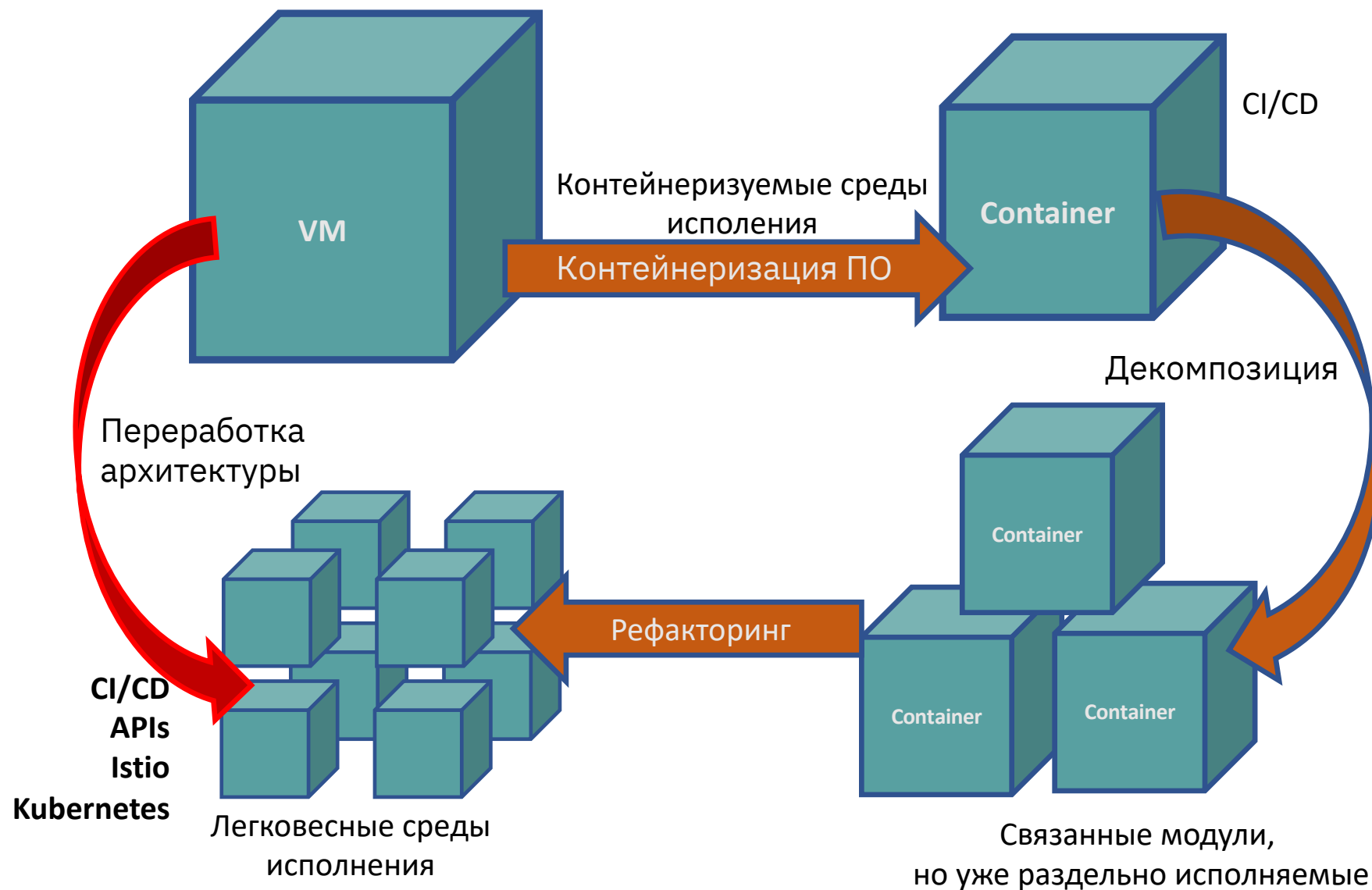
- Трудоемкость сопровождения приложения
  - Больше частей, которые нужно контролировать и управлять
- Разработчики должны обладать значительными навыками (DevOps)
  - Автоматизация всего – сборка, развертывание, тестирование
- Версионность интерфейсов взаимодействия
- Дублирование усилий при реализации сервисов
- Дополнительная сложность создания распределенной системы:
  - Сетевая задержка
  - Отказоустойчивость
  - Сериализация
- Разработка распределенных, не-транзакционных систем - сложная задача
- Обнаружение и диспетчеризация трафика к сервисам
- Безопасность взаимодействия сервисов
- Поддержание доступности и согласованности в распределенных данных
- Сквозное тестирование



# Когда и как строить микросервисы?

- Как начать?
  - Если нет опыта, то начать с хорошо продуманного монолита
  - Постепенно развивая решение в сторону микросервисов
- Когда переходить к микросервисам?
  - Если есть реальная бизнес необходимость
  - Если нужен быстрый и независимый вывод нового функционала
  - Если вам нужно оптимизировать часть вашего приложения

# Постепенное развитие монолита



# Как создавать микросервисы

# План на сегодня

- Пример приложения
  - Сборка, запуск в контейнере Docker
  - Подход "Build to manage"
- Развертывание в Kubernetes
  - Работа с Private Registry
  - Создание Deployment, Service, ConfigMap
  - Liveness, Readiness пробы
- Istio Service Mesh (доп. время)
  - Основы управления трафиком
  - Мониторинг, Service Graph, Tracing
  - Примеры других сценариев (A/B тестирование, канареечные развертывания)
  - Отказоустойчивость приложения
    - Timeout, Fault injections



# Приложение Кофейня

Микросервис coffee-shop

- Прием заказов
- Статус заказов
- Выдача кофе



Микросервис barista

- Приготовление кофе
- Статус заказа



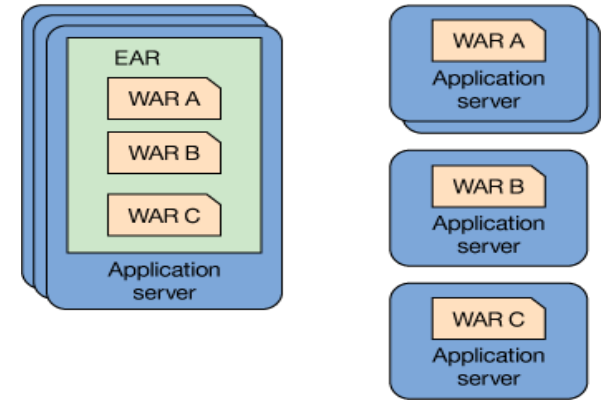
# Cloud native

- An **application architecture** designed to use the ***strengths*** and accommodate the ***challenges*** of a standardized **cloud** environment, including:
  - Elastic scaling
  - Immutable development
  - Disposable instances
  - Less predictable infrastructure

Microprofile

# What do we need to build cloud-native apps?

- We want to build cloud native apps – what do we need?
  - Lightweight app server ✓
  - Easy containerization ✓
  - Management platform for all the cattle ✓
  - Microservice and cloud-native ready Java API specification ✗

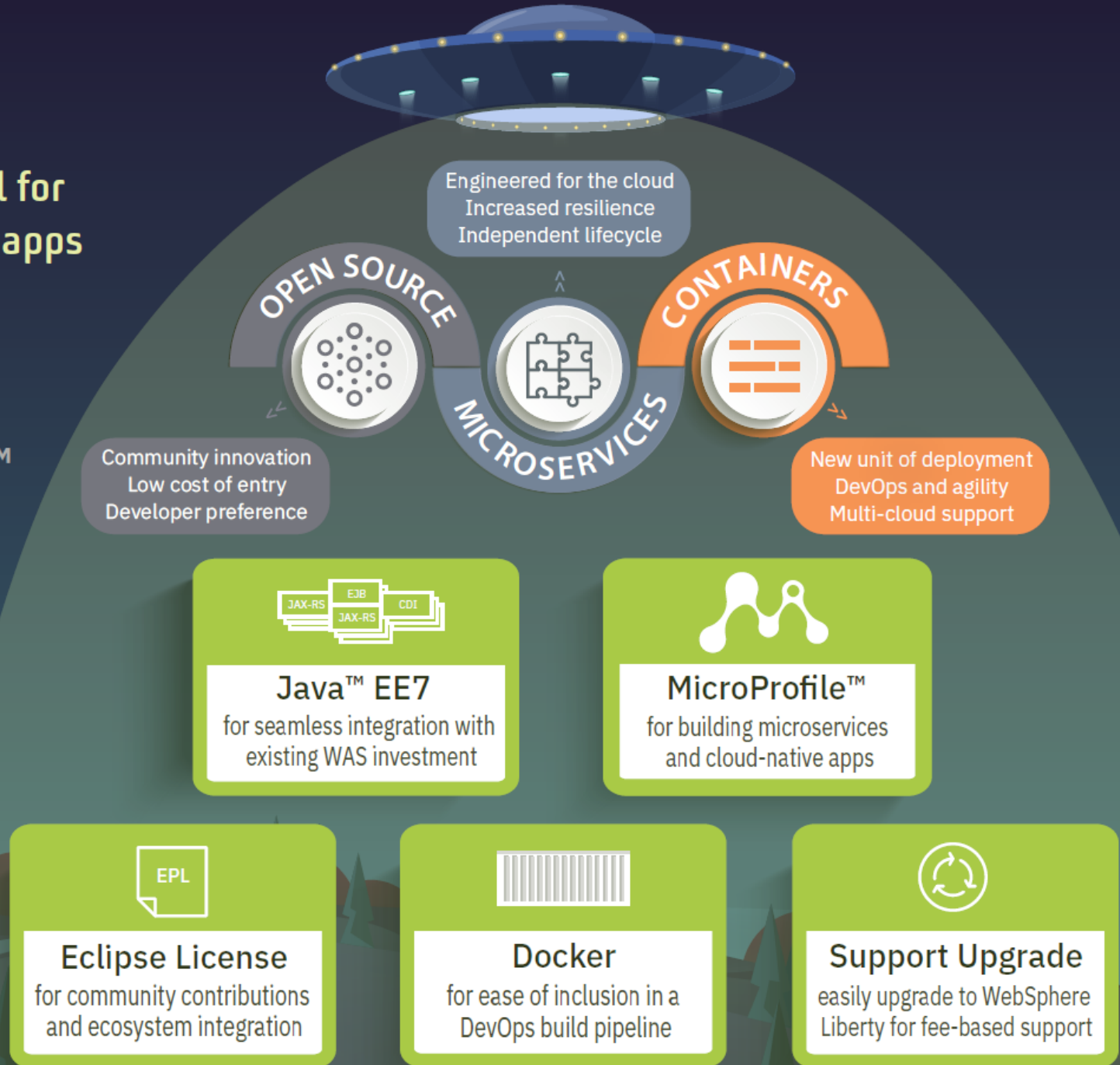


# Open Liberty

A lightweight open source server runtime ideal for building Java™ microservices and cloud-native apps

- Easy to consume
- Deploy on any cloud for Java™
- Seamlessly transition to WebSphere

<https://openliberty.io/>

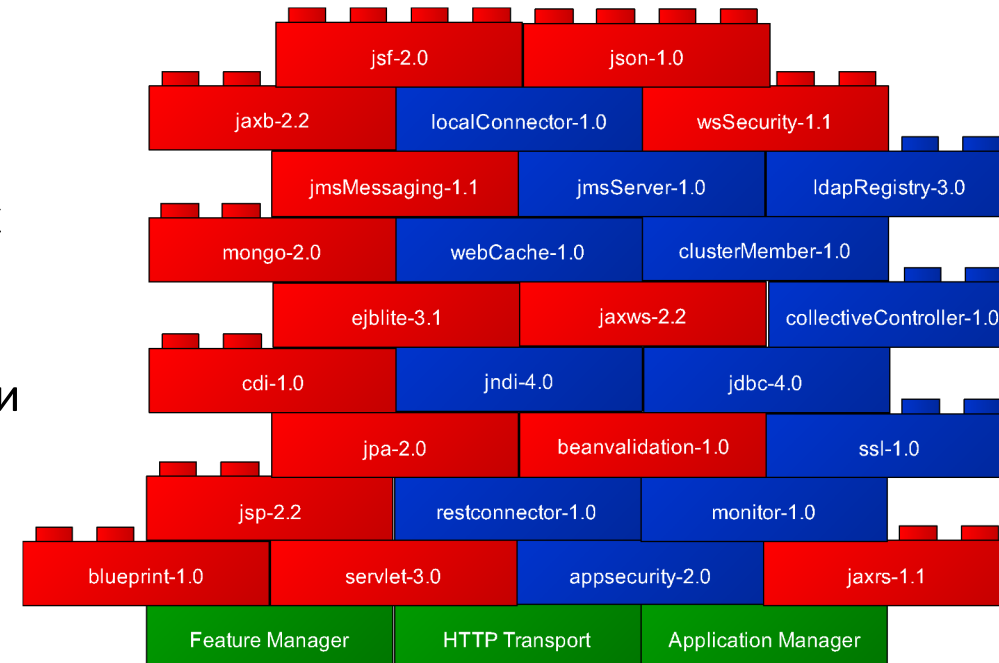




# Just enough runtime OpenLiberty

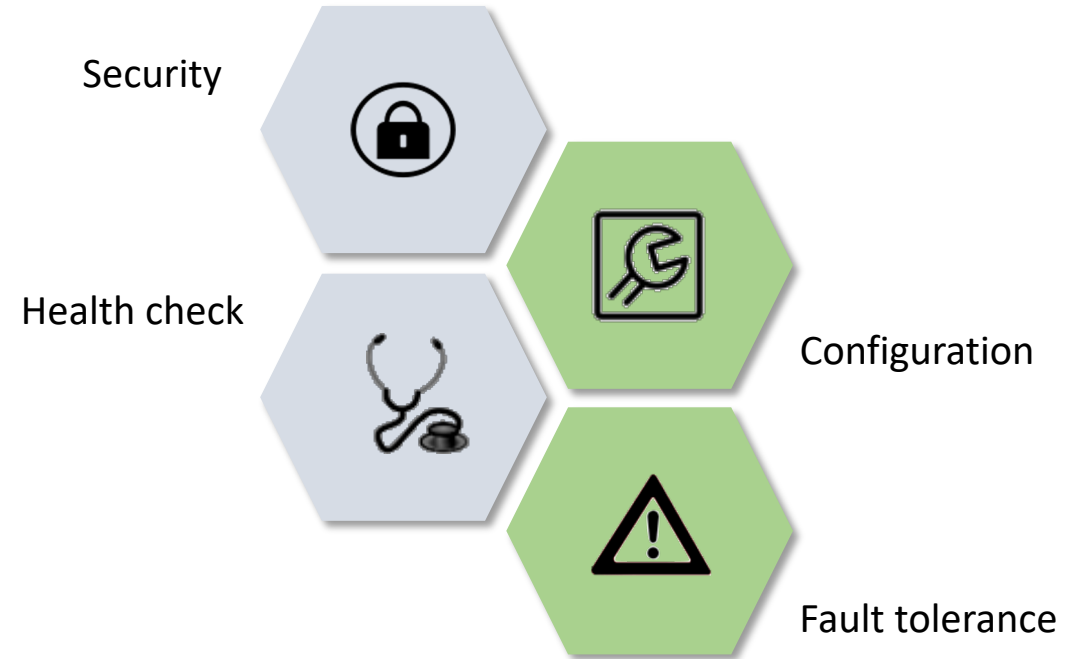
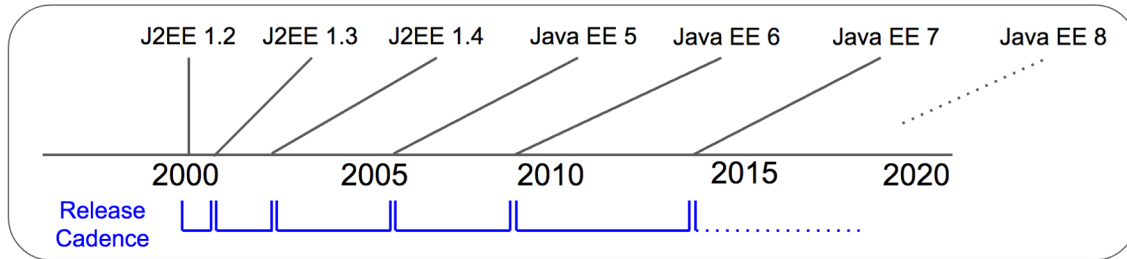
- Полная поддержка Java EE, MicroProfile, SpringBoot
- Конфигурация в виде файла для простоты DevOps процессов
- Промышленная производительность, надежность, безопасность и поддержка
- Zero migration при обновлении: EE 6, EE 7, EE 8 на одной инсталляции
- Работает одинаково как в традиционных так и контейнеризованных окружениях
- Проверен в бою: Более 50,000 Liberty в продуктивной эксплуатации

<http://openliberty.io>



# Java programming model for Microservices

Java EE does not address all the needs of microservice applications and progress is slow



# Eclipse MicroProfile - Java for Microservices



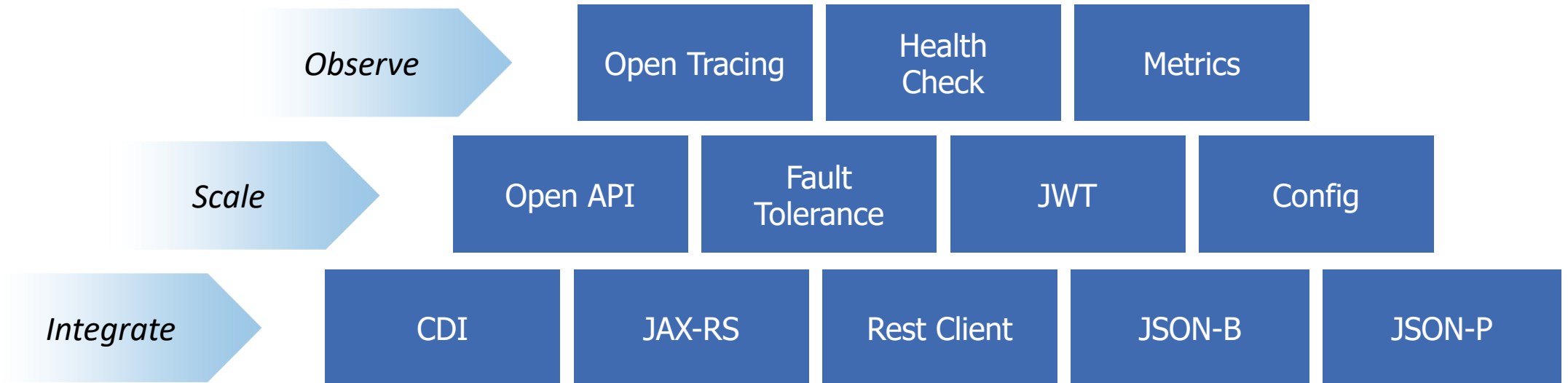
- Vendor-neutral programming model, designed in the open, for Java microservices
- Provide core capabilities for building fault tolerant, scalable, microservices
- Increasing the rate and pace of innovation beyond Java EE



## Standardizing microservices in enterprise Java via the MicroProfile community

Config	Fault Tolerance	Health Check	Health Metrics	JWT Propagation
externalize configuration to improve portability	build robust behavior to cope with unexpected failures	common format to determine service availability	common REST endpoints for monitoring service health	interoperable authentication and role-based access control

# Eclipse MicroProfile



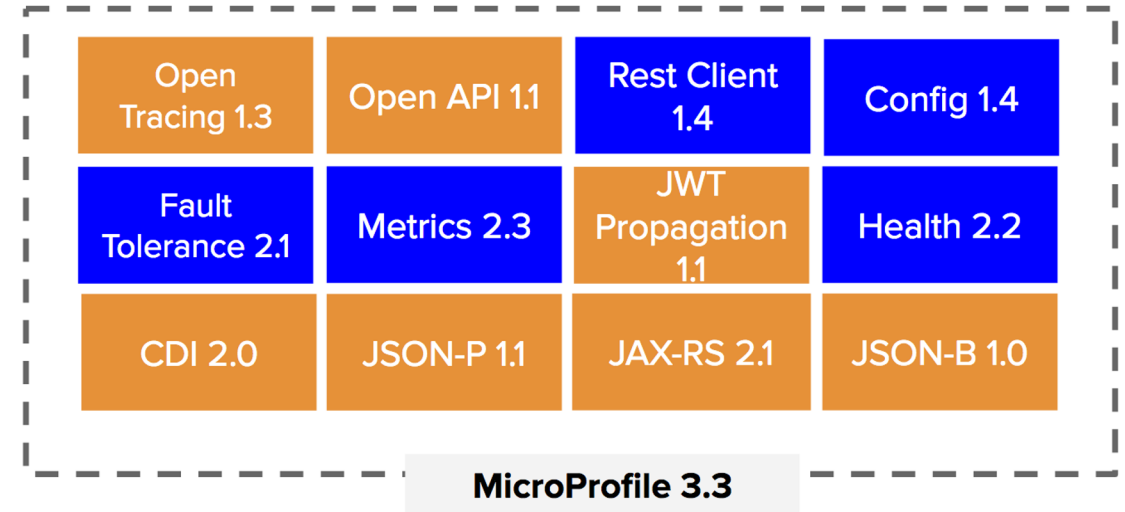
# Eclipse MicroProfile 3.3 Released!

On February 18, 2020, MicroProfile 3.3 was released. Offered in the release:

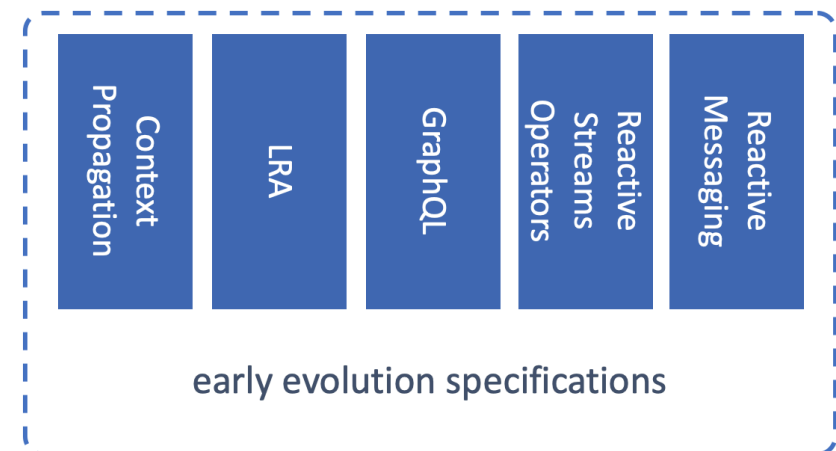
- Continued alignment with CDI, JSON-P, JSON-B, and JAX-RS
- A richer feature set for Rest Client, Config, Fault Tolerance, Metrics and Health specifications
- CDI-based and programmatic interfaces
- Test Compatibility Kit (TCK), Javadoc, HTML, PDF docs for download

## Other news:

- MicroProfile Starter 1.0 is [now generally available](https://start.microprofile.io)
- [start.microprofile.io](https://start.microprofile.io) IntelliJ extension [released](#)



- = New
- = Updated
- = No change from last release (MicroProfile 3.2)





# Current MicroProfile implementations



# Контейнеризация

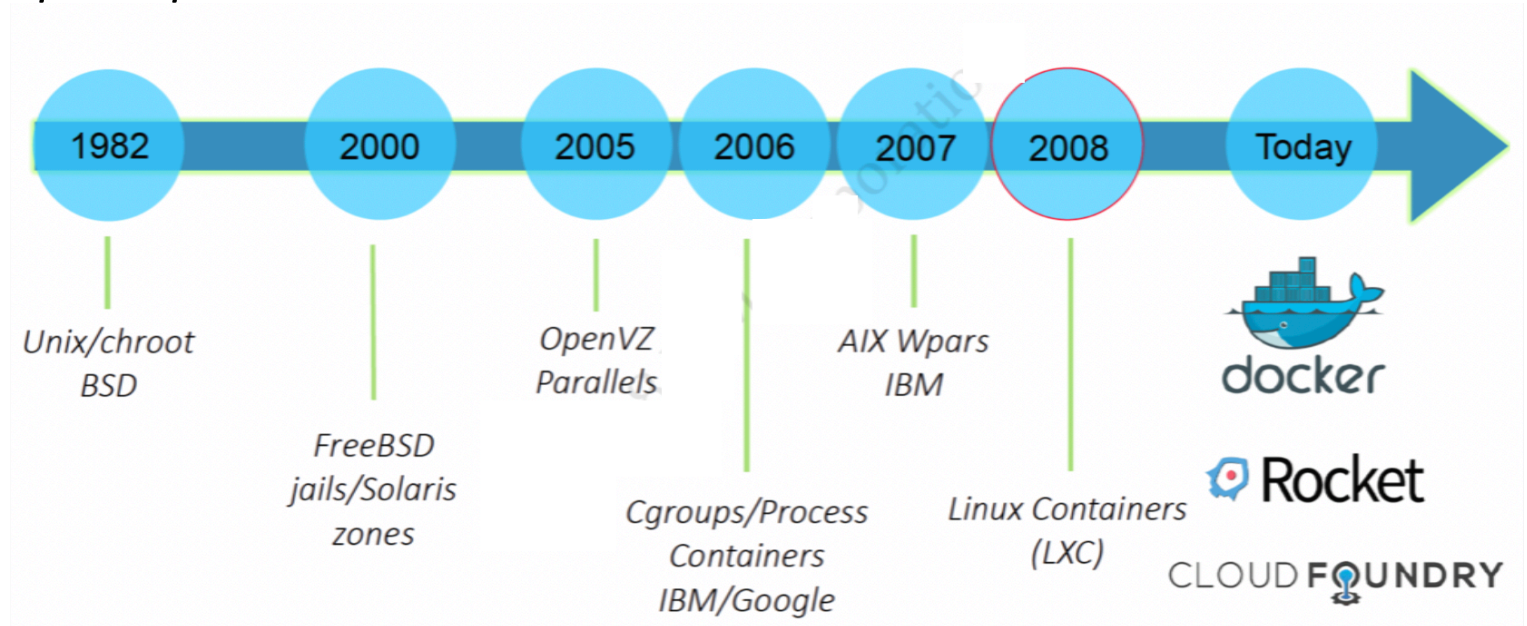
# Технологии контейнеризации

- *Контейнеризация - это способ стандартизации упаковки чего-либо.*

- *Более 10 лет опыта использования различных реализаций контейнеров в UNIX/Linux*
- *LXC – одна из популярных технологий контейнеризации в ядре Linux с 2008 года*
- *cgroups и namespaces*
- *Docker до v0.9 использовал LXC как драйвер для запуска по умолчанию*



runC (libcontainer)



Зачем нужны контейнеры?



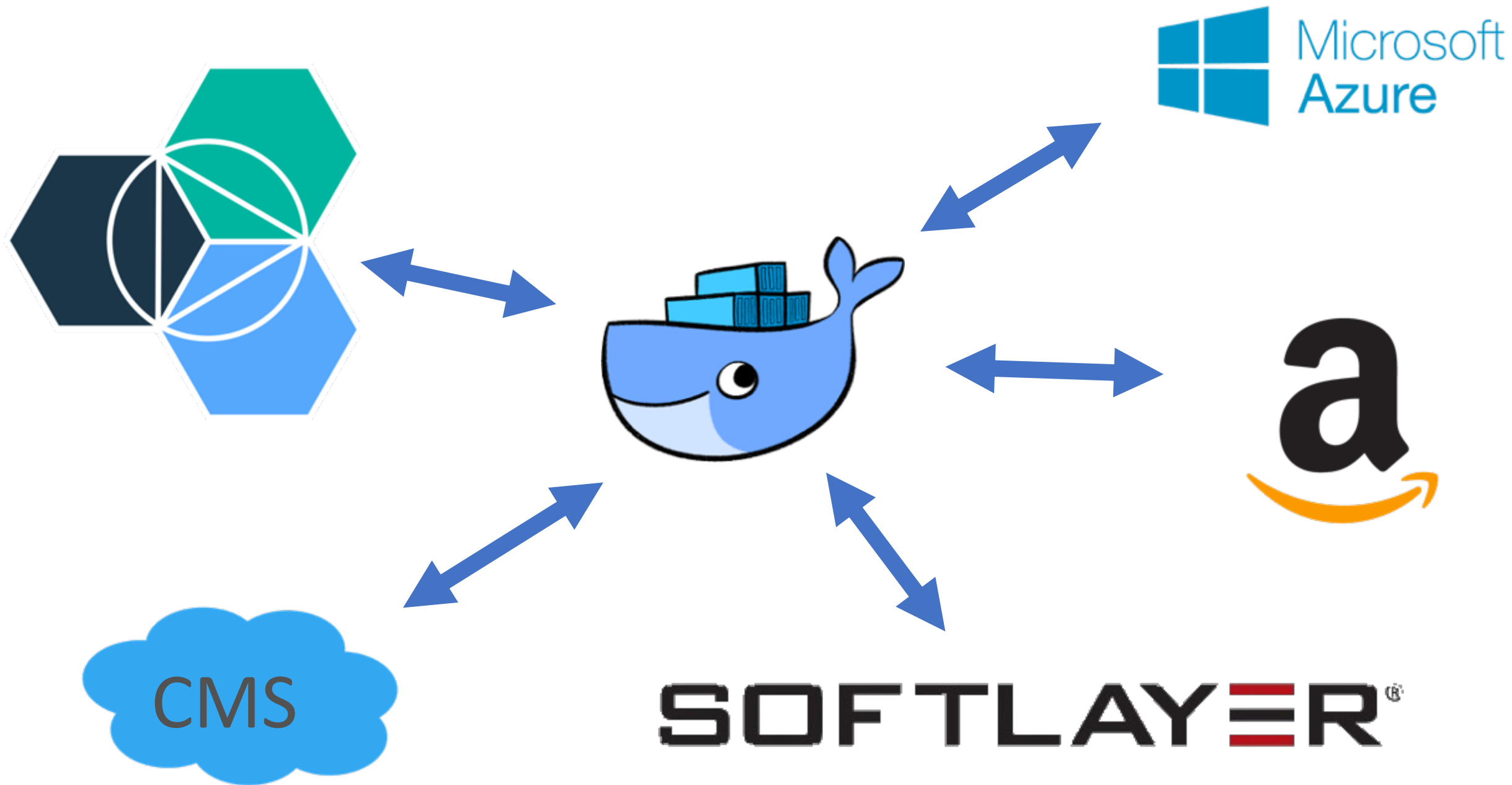
# Все любят контейнеры



Стандартный способ упаковать приложение и его зависимости и запускать его в разных средах без каких-либо доработок.



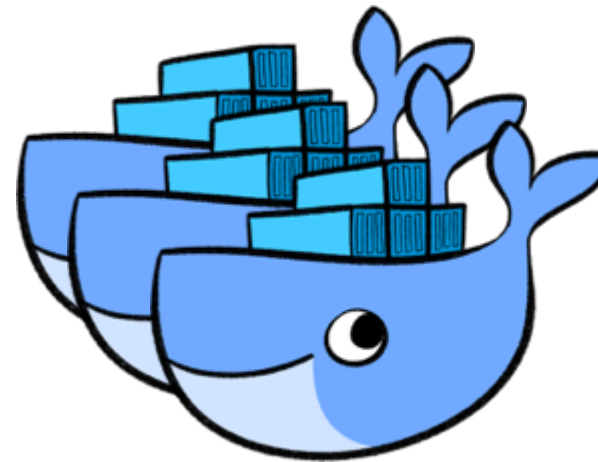
# Мобильность



# Управление версиями и зависимостями



Продуктив



Среда тестирования

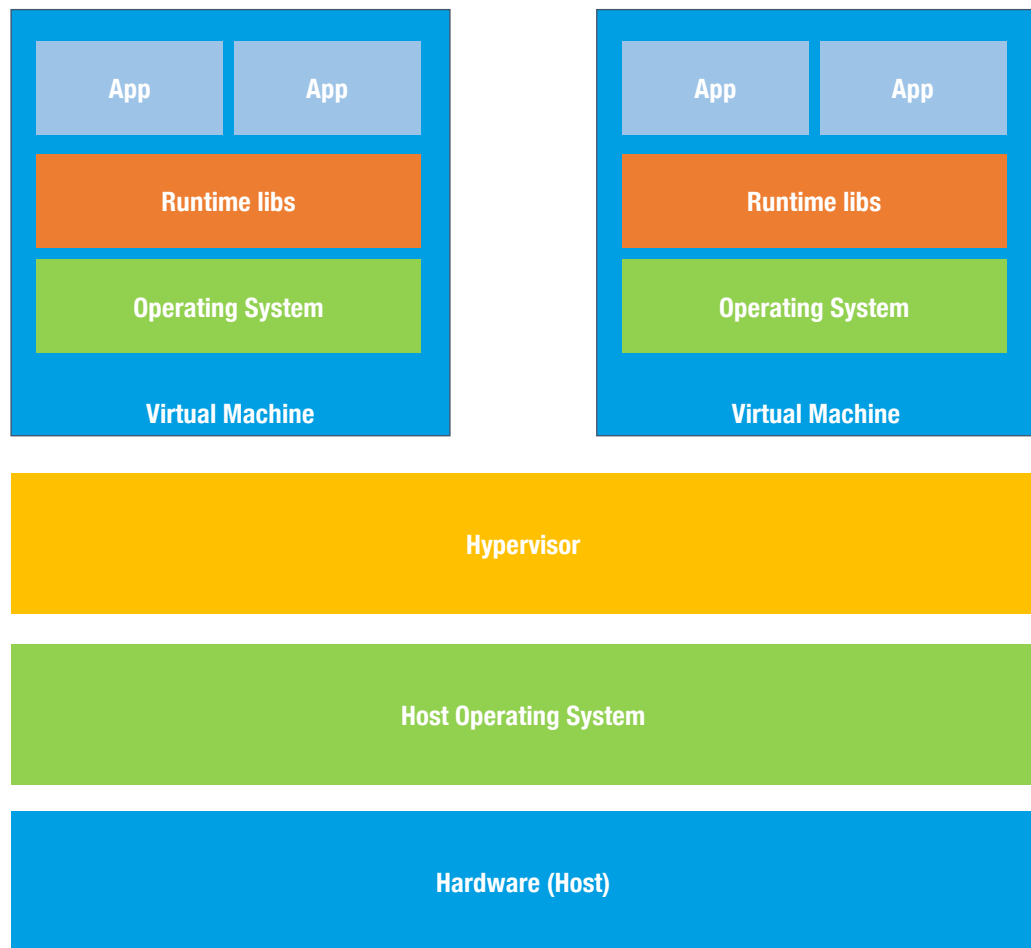


Пред-прод

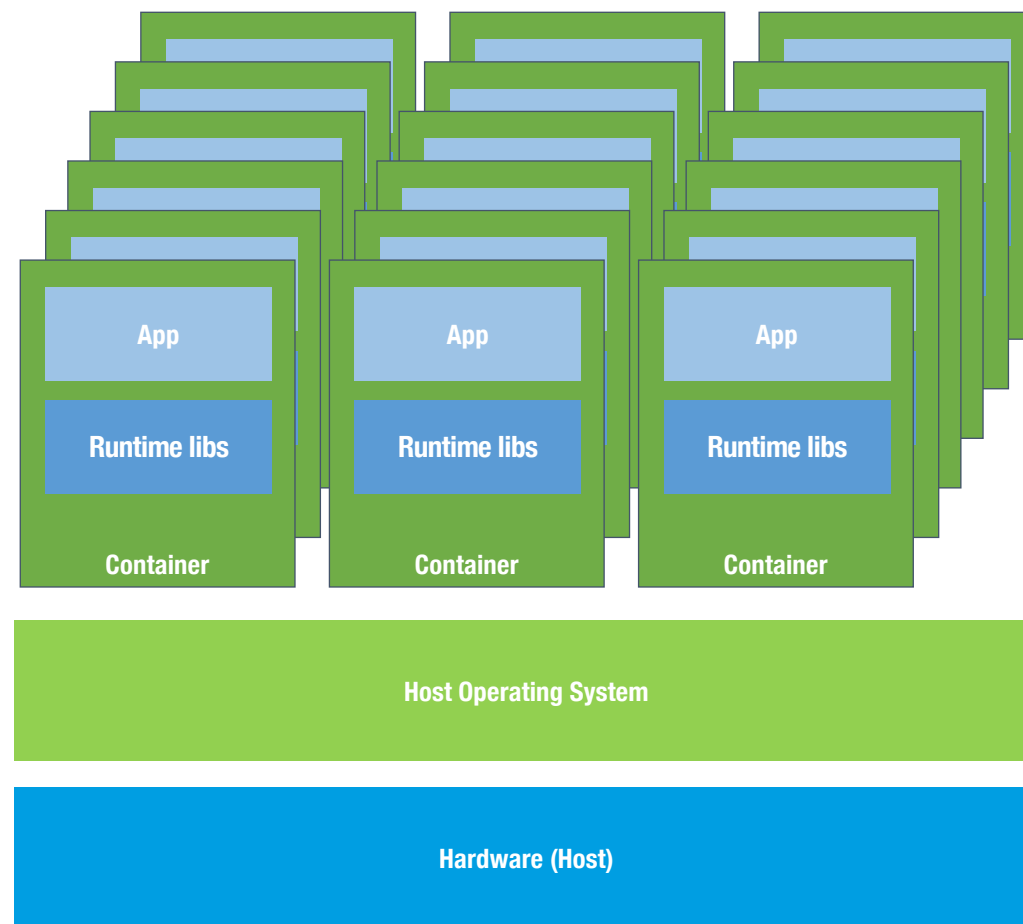


Среды разработчиков

# Уменьшение накладных расходов



**Virtual Machines**



**Containers**

# Масштабируемость и отказоустойчивость



Production environment

Kubernetes

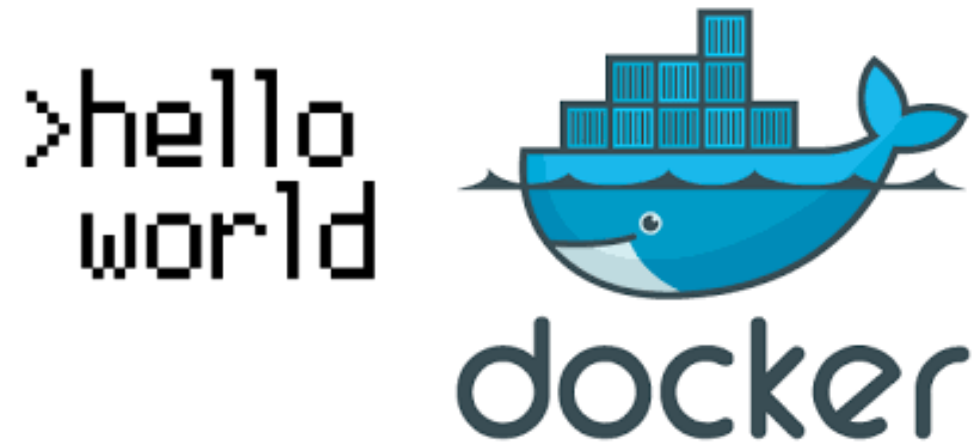


MESOS



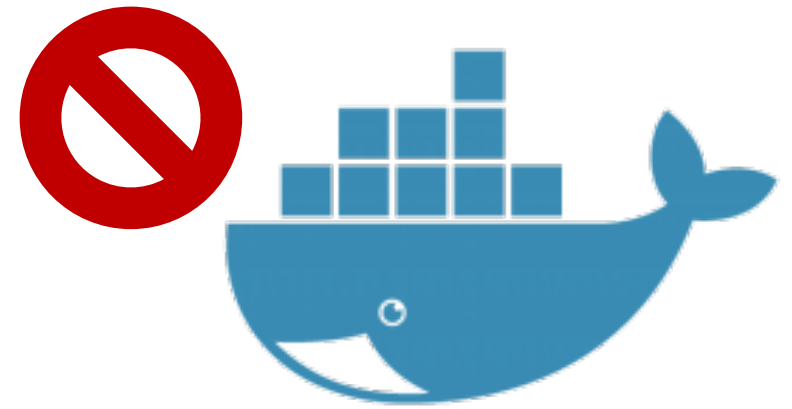
MARATHON

# Контейнеры Docker и с чего начать



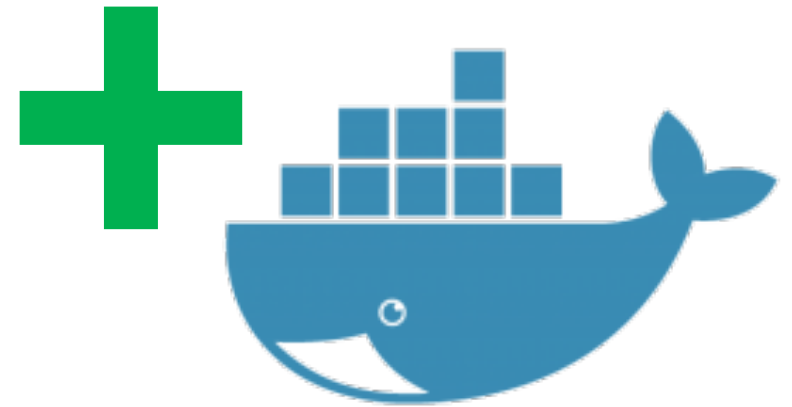
# Чем Docker не является

- Виртуальной машиной. Docker-контейнер - легковесный образ, использующий только те библиотеки и компоненты, которые необходимы самому приложению.
- Средством исполнения контейнеров. Контейнеры исполняются механизмом ядра Linux под названием cgroups. Служба docker управляет **созданием, развертыванием, запуском и конфигурацией** контейнеров.
- Средством автоматизации развертывания. Docker обслуживает только среду с контейнерами, не заботясь о состоянии виртуальных машин, операционных систем и др. инфраструктуры.



# Чем Docker является

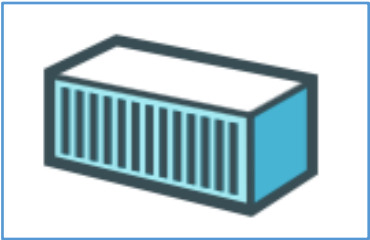
- ПО с открытым исходным кодом
- ПО верхнего уровня над средствами контейнеризации ядра ОС, предоставляющим интерфейс для упрощения процессов упаковки, доставки и развертывания сложных приложений и middleware
- Использует многоуровневую файловую систему. Каждый уровень версионруется и может быть использован множеством работающих контейнеров из одного образа
- Использует реестр для хранения образов



# Компоненты Docker

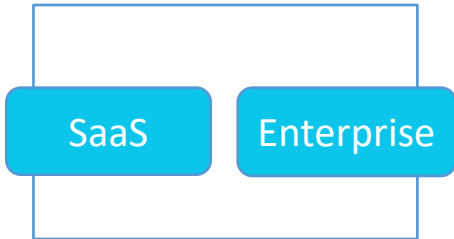


**Image** («образ») – заготовка для будущего контейнера с набором инструкций, монтируемая в виде набора read-only слоёв.

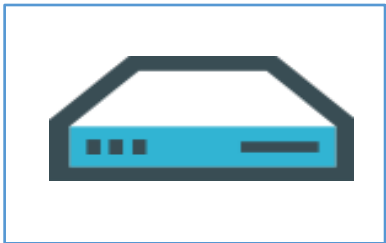


**Container** – запущенный в виде экземпляра Image.

- Относительно изолирован от других контейнеров и host-системы, степень изоляции можно контролировать через network, storage и другие подсистемы
- Конфигурация контейнера определяется через image и через параметры для запуска
- При удалении контейнера любые изменения, не сохраненные на внешнем диске, пропадают



**Docker Hub/Registry** – реестр для хранения image объектов. Может быть в виде облачного сервиса или частным репозиторием.



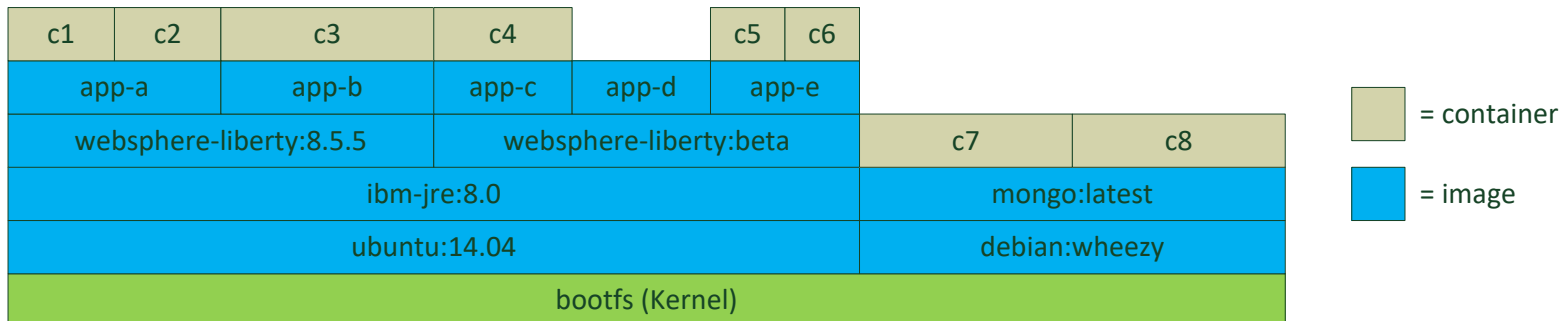
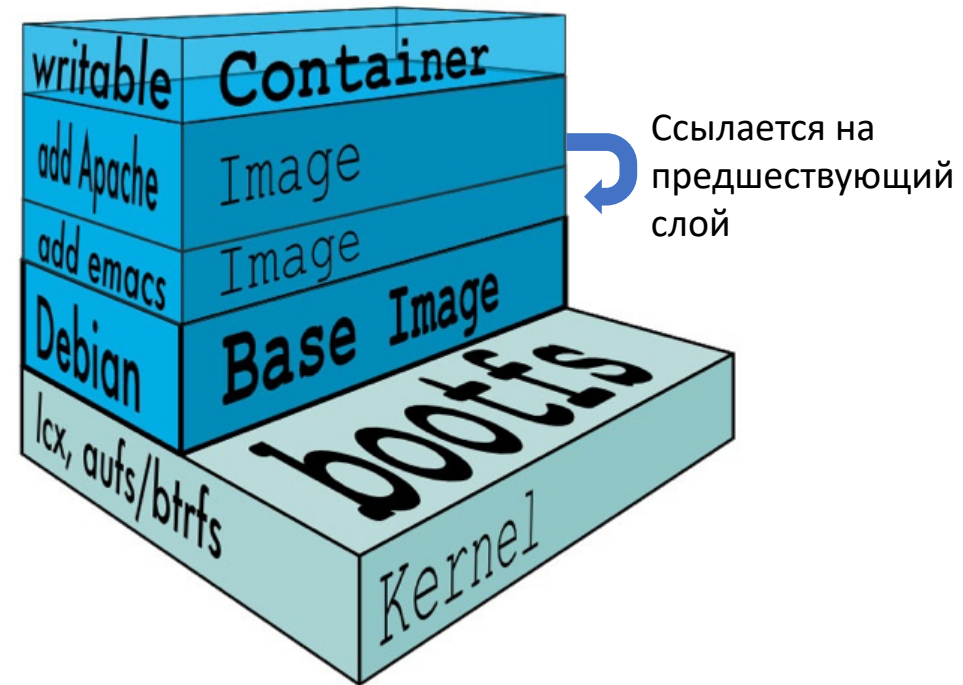
**Docker Engine** (сервер + клиент)

- Процесс (dockerd) который управляет созданием, развертыванием и конфигурацией объектов Docker
- Может общаться с другими удаленными процессами для управления Docker-сервисами
- Для работы с Docker Engine используются различные утилиты командной строки, REST API и т.п.



# Docker образ

- Образ – это набор файлов и описание (метаданные). Технически: эти файлы из корневой “/” файловой системы контейнера.
- Образ создан из слоев друг поверх друга как слоеный пирог.
- Каждый слой может добавлять, изменять и удалять файлы предшествующего слоя.
- Образы могут быть построены на общих слоях, что экономит место на диске, время его передачи по сети и память.



# Dockerfile

- Автоматизирует создание образа
- Определяет базовый образ и инструкции по созданию:
  - FROM <existing image>
  - ADD <local file> <path inside image>
  - RUN <cmd>
  - EXPOSE <port>
  - ENV <name> <value>
  - CMD <cmd>

# Dockerfile - пример

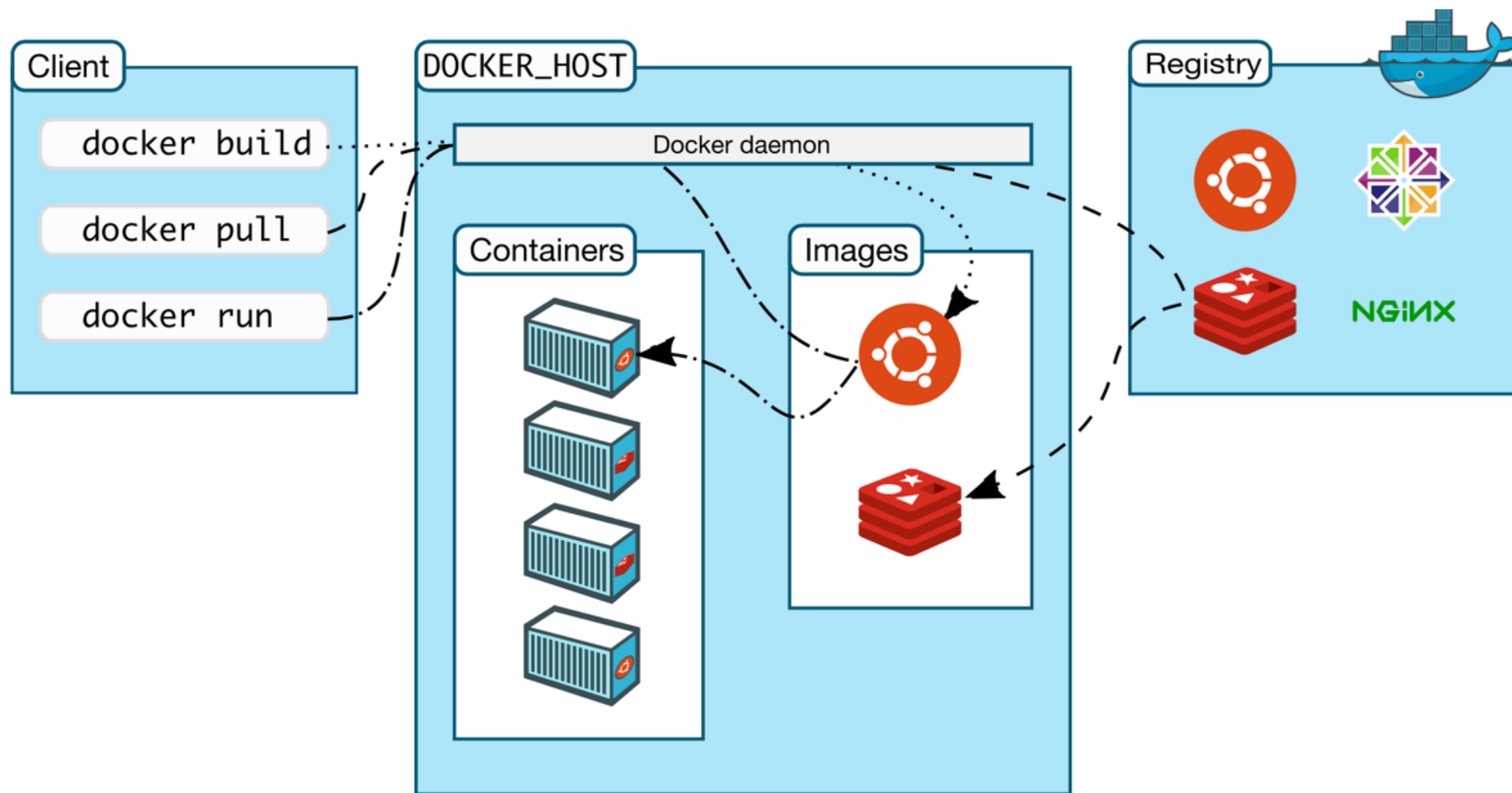
- Каждая инструкция создает слой в образе
- При изменении Dockerfile и пересборке образа, меняются только те слои, что соответствуют измененным инструкциям в Dockerfile

```
1  FROM node:8
2
3  # Create app directory
4  WORKDIR /usr/src/app
5
6  # Install app dependencies
7  # A wildcard is used to ensure both package.json
8  # AND package-lock.json are copied
9  # where available (npm@5+)
10 COPY package*.json ./
11
12 RUN npm install
13 # If you are building your code for production
14 # RUN npm install --only=production
15
16 # Bundle app source
17 COPY . .
18
19 EXPOSE 8080
20 CMD [ "npm", "start" ]
21
22
```

```
$ docker build -t node-docker-sample .
Sending build context to Docker daemon 19.46kB
Step 1/7 : FROM node:8
---> 6f62c0cdc461
Step 2/7 : WORKDIR /usr/src/app
Removing intermediate container a8b8b578327d
---> 1235d56962cf
Step 3/7 : COPY package*.json ./
---> 1ec16de8eb98
Step 4/7 : RUN npm install
---> Running in bc10ab2b19c8
added 50 packages from 47 contributors and audited
119 packages in 1.921s
found 0 vulnerabilities
Removing intermediate container bc10ab2b19c8
---> b4c395601388
Step 5/7 : COPY . .
---> 0aa7e4bdc15e
Step 6/7 : EXPOSE 8080
---> Running in 7375f160e8f8
Removing intermediate container 7375f160e8f8
---> f341b5b2c5b6
Step 7/7 : CMD [ "npm", "start" ]
---> Running in 576374c1c7dd
Removing intermediate container 576374c1c7dd
---> bf908d296742
Successfully built bf908d296742
Successfully tagged node-docker-sample:latest
```

Демо – сборка образа и запуск приложения

# Docker механизмы



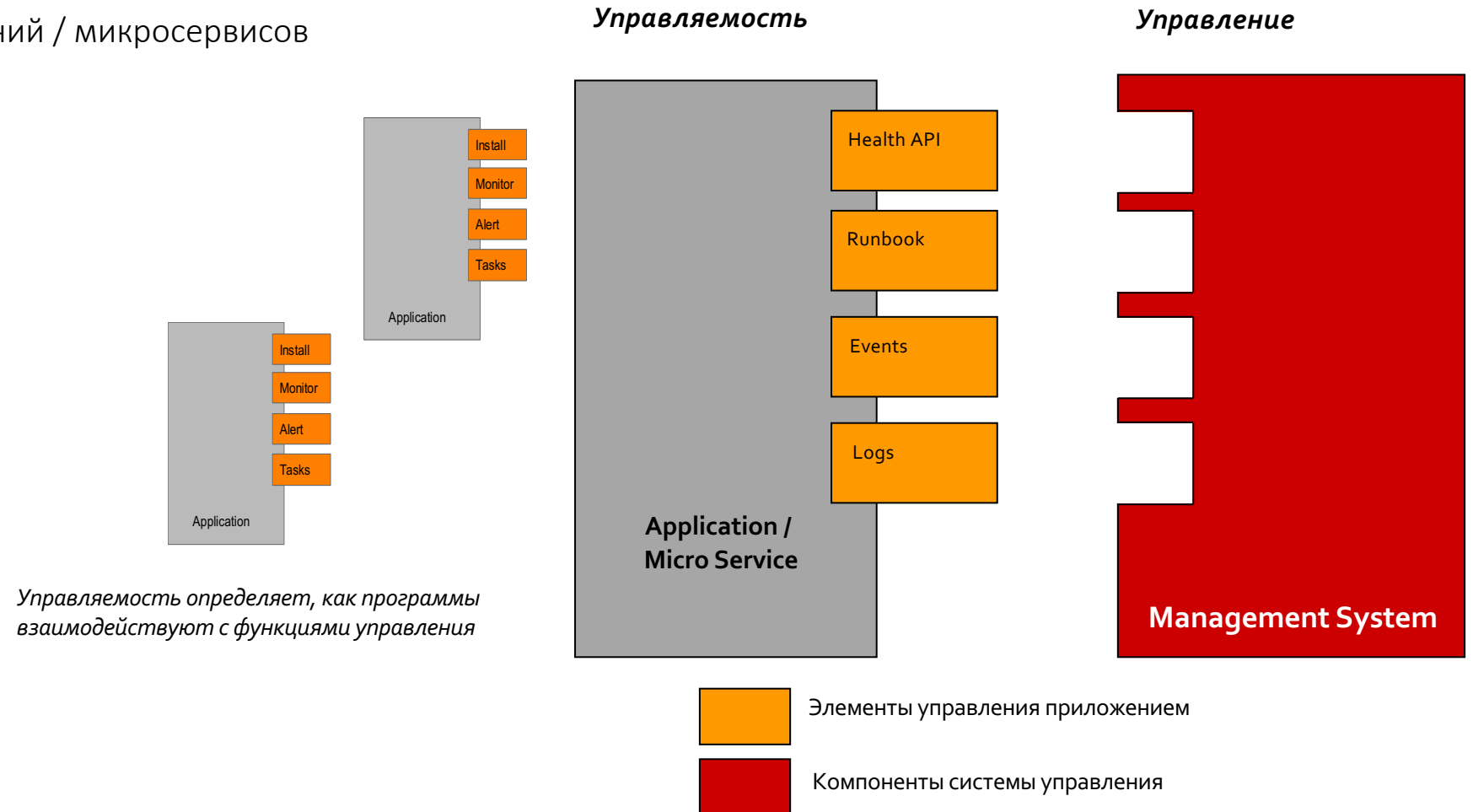
# Build To Manage – как создать управляемые приложения

- По мере того как Разработка и Сопровождение становятся все ближе друг к другу, возникают новые методы, позволяющие улучшить управление облачными приложениями

- Создание управляемых приложений / микросервисов

- Методы включают:

- HealthCheck API
- Deployment Correlation
- Topology Information
- Test Cases and Scripts
- Runbooks
- Log Format and Catalog
- Distributed Tracing
- Event Format and Catalog
- Monitoring Configuration
- First Failure Data Capture





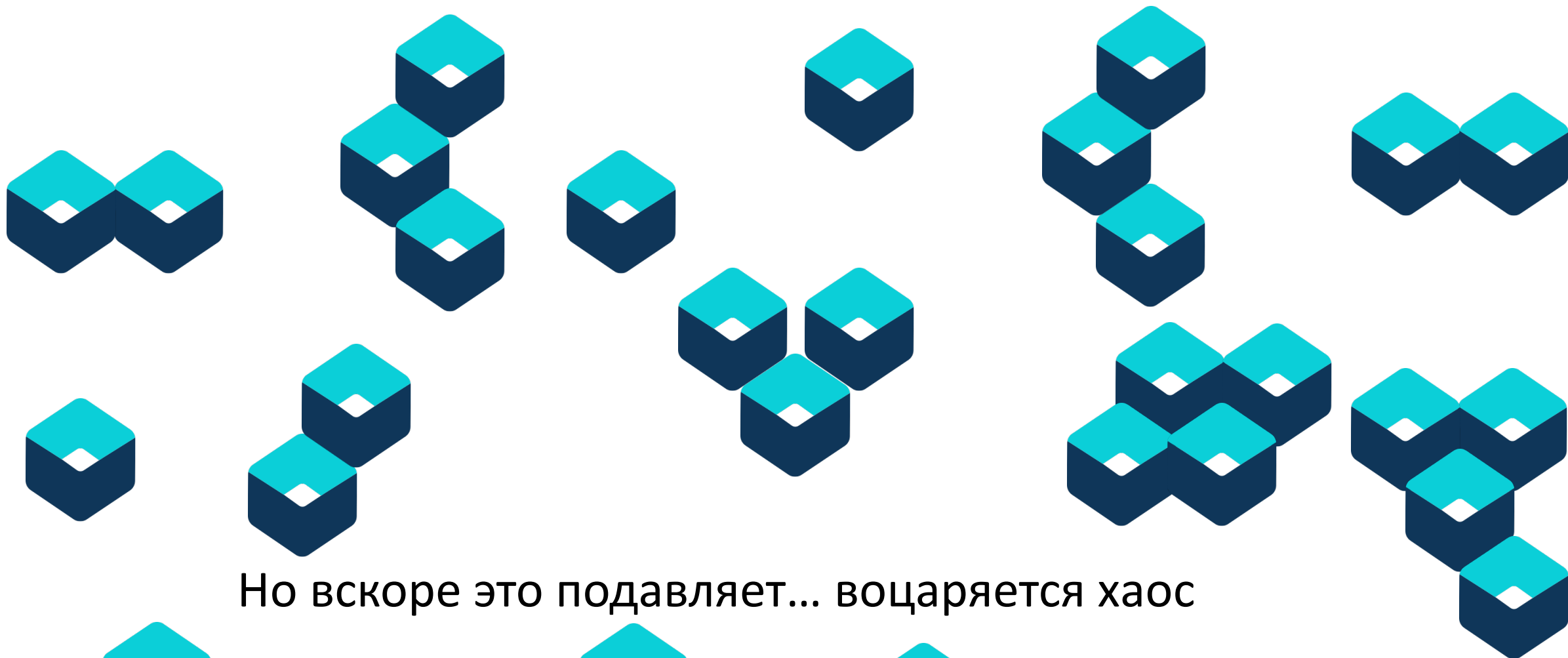
Каждый контейнерный путь начинается с одного контейнера....



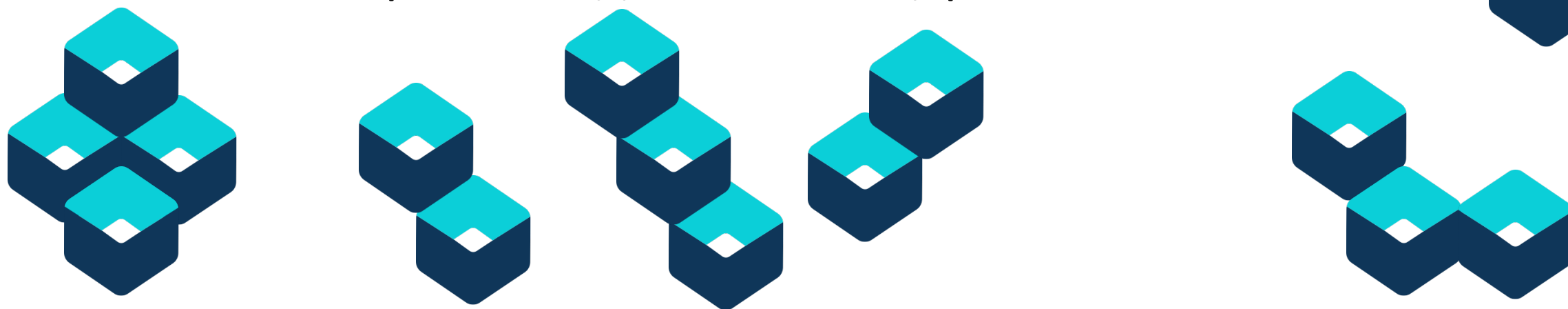
Сначала с ростом легко справиться....







Но вскоре это подавляет... воцаряется хаос



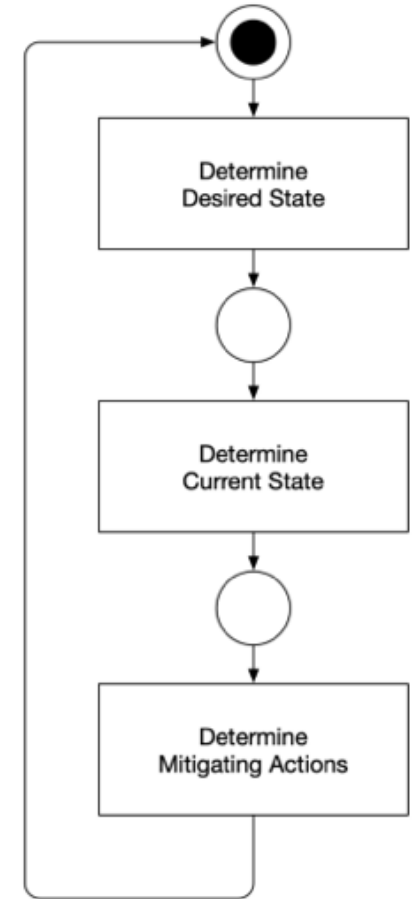
Восстановите контроль с помощью Оркестратора контейнерами



# Что такое Kubernetes

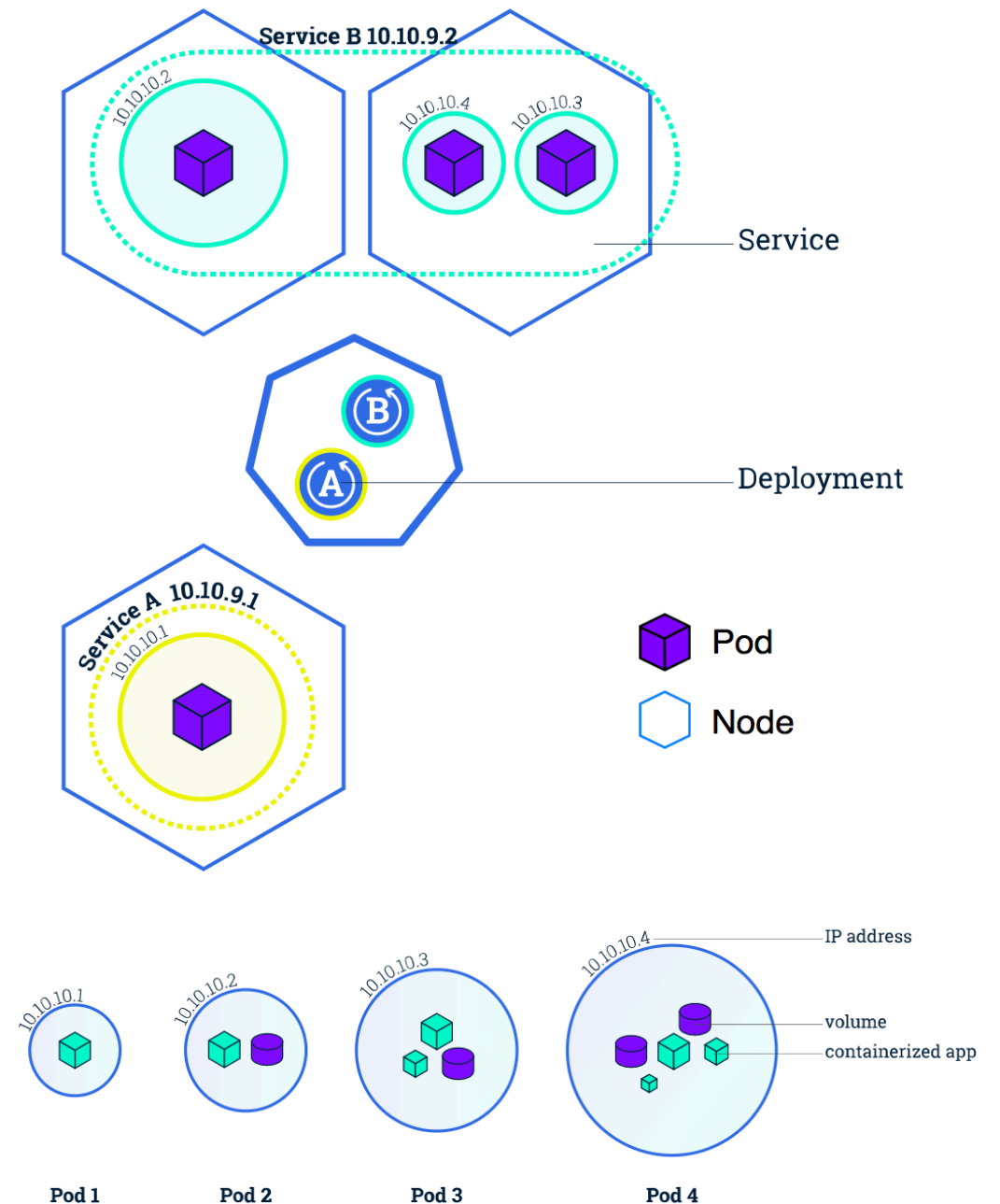


- Система оркестрации контейнерами
- В основе декларативный подход по управлению
- Инициирован компанией Google
- Создан на базе проекта Borg, но не является копией
- Один из проектов CNCF (Cloud Native Computing Foundation)
- Golang основная технология



# Kubernetes концепции

- Pod – запрос на запуск одного или нескольких контейнеров в рамках вычислительного узла.
- Deployment - определяет правила развертывания Pod и желаемое количество его копий.
- Service - предоставляет способ сетевого взаимодействия с Pod-ами.
- Volume - нужен если Pod необходимо постоянное хранилище данных.
- И т.п.



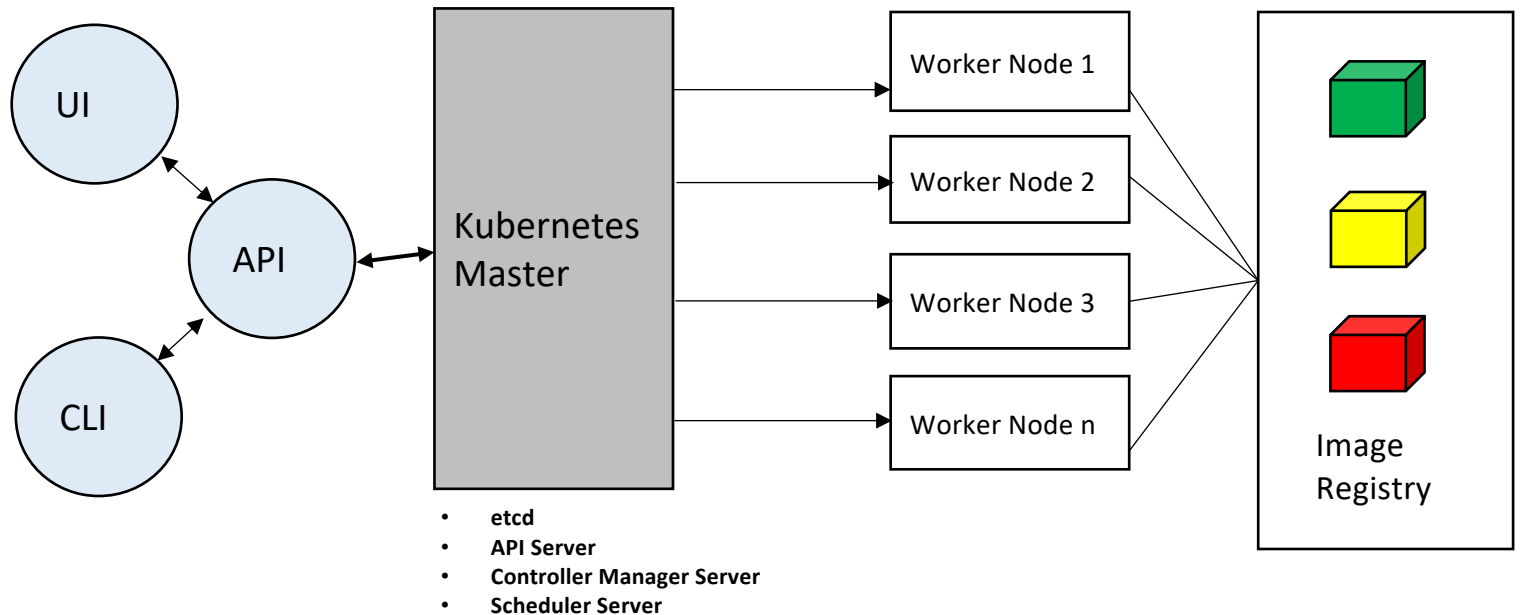
# Kubernetes архитектура

- Master узел

- etcd (распределенная key-value база)
- Kubernetes API Server
- Scheduler
- Controller Manager

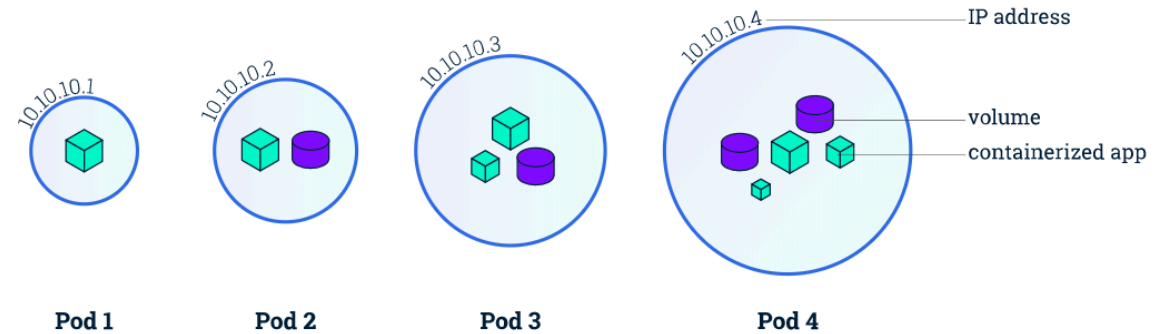
- Worker узел

- Docker
- Kubelet
- Kubernetes Proxy



# Pod

- Группа из одного или нескольких контейнеров называется *pod*.
- Контейнеры в pod разворачиваются, стартуют, останавливаются и масштабируются как единое целое.
- Контейнеры в pod делят один сетевой интерфейс на всех.



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

# Deployment

- Объект Deployment определяет шаблон по созданию Pod и желаемое количество его копий.
- Создает или удаляет Pod-ы в соответствии с требуемым количеством копий.
- Управляет обновлениями связанными с Pod-ами.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

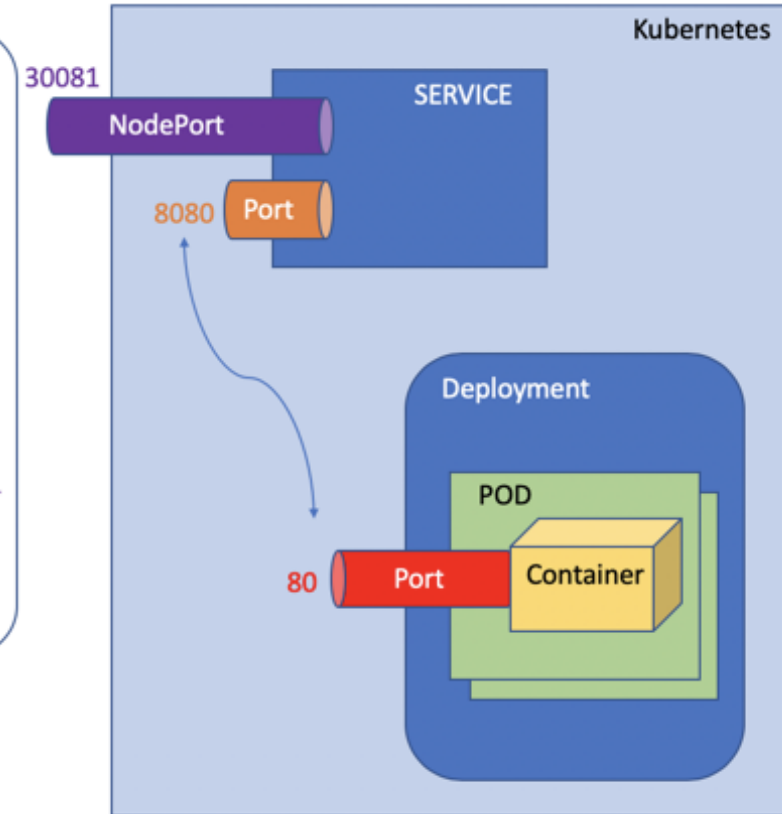
# Service

- Service предоставляет способ взаимодействия с Pod
- Так же предоставляет функции балансировки

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  ports:
    - port: 8000
      targetPort: 80
      protocol: TCP
  selector:
    app: nginx
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
labels:
  app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: auth-svc
labels:
  run: auth-svc
spec:
  type: NodePort
  ports:
    - port: 8080
      targetPort: 80
      nodePort: 30081
  protocol: TCP
  name: http
  selector:
    app: auth
```





# Volume

- Контейнерная файловая система живет и умирает вместе с контейнером.
- Volume нужен если контейнеру необходимо постоянное хранилище данных.
- Разные виды данных могут быть смонтированы как volume
  - ConfigMaps, Secrets, HostPath, ServiceAccount ....

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
    - name: redis
      image: redis
      volumeMounts:
        - name: redis-persistent-storage
          mountPath: /data/redis
      volumes:
        - name: redis-persistent-storage
          emptyDir: {}
```

# StatefulSet, DaemonSet, Job...

- StatefulSet
  - Предназначены для использования с Stateful приложениями в распределенных системах.
  - Pods создаются в определенном порядке
  - Порядковый номер и постоянные сетевые адреса
- DaemonSet
  - Стремится чтобы на всех (или определенных) узлах работала копия указанного Pod
    - работа агентов кластерной файловой системы
    - работа агентов по сборке логов
    - работа агентов системы мониторинга
- Job
  - Создает один или несколько Pod-ов и стремится чтобы определенное их количество успешно отработало
- Cron Job
  - Управляет Job-ами в привязке ко времени, периодические или по расписанию.

Демо – развертывание приложения в K8s

12 factor applications

# Что это?

- Набор лучших практик для приложений
  - Как разрабатывать, развертывать, сопровождать и управлять
- Требования к современным приложениям:
  - Облачные приложения
  - Веб-приложения
- Могут быть применимы:
  - К любому языку программирования
  - Используя любой нижележащий бэкенд

# 12 factor app – 1st 6

- I. **Codebase** – one codebase, tracked in code versioning system
- II. **Dependencies** – declare and isolate dependencies
- III. **Configuration** – store configuration in the environment
- IV. **Backing services** – treat backing services as attached resources
- V. **Build, release and run** – strictly separate build and run stages
- VI. **Processes** – run app as one or more stateless processes

# 12 factor app – 2nd 6

- VII. Port binding** – export services with port bindings
- VIII. Concurrency** – scale out using the process model
- IX. Disposability** – fast startup and efficient shutdown of apps
- X. Development and production parity** – similar as possible envs.
- XI. Logs** – treat logs as event streams
- XII. Admin processes** – run administrative tasks as one-off processes

# О чем еще стоит упомянуть

- Безопасность приложений в контейнере
- При наличии у пользователя или приложения прав супер-пользователя в контейнере – гостевая ОС может быть взломана (трояны, не квалифицированные разработчики, контейнеры из не проверенных источников)
- Усложнение инфраструктуры из-за диспетчеризации трафика
- Требуются умные балансировщики и маршрутизаторы запросов
- Управление большим числом контейнеров, конфигурациями, инфраструктурой
- Автоматизация развертывания, отслеживание живучести контейнеров и приложений в них, авто-масштабирование, мониторинг, логирование, ограничение доступа, изоляция одних от других, распределение ресурсов.





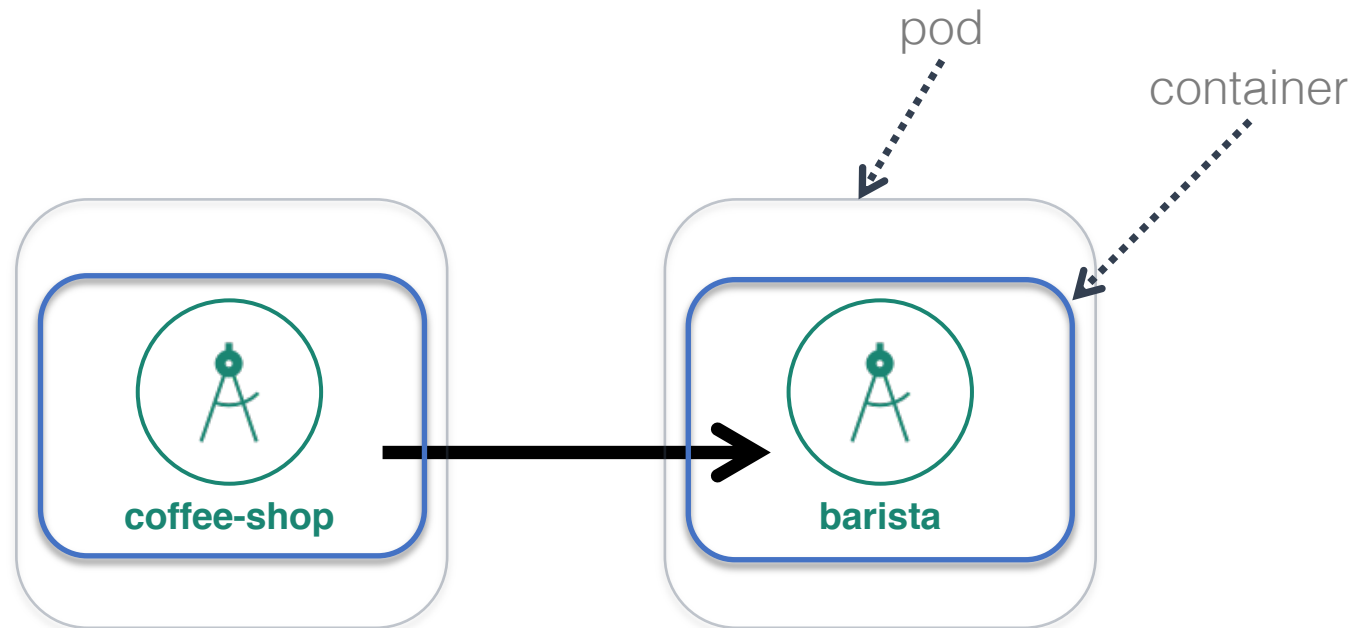


# Istio: Weaving, Observing and Securing Microservices

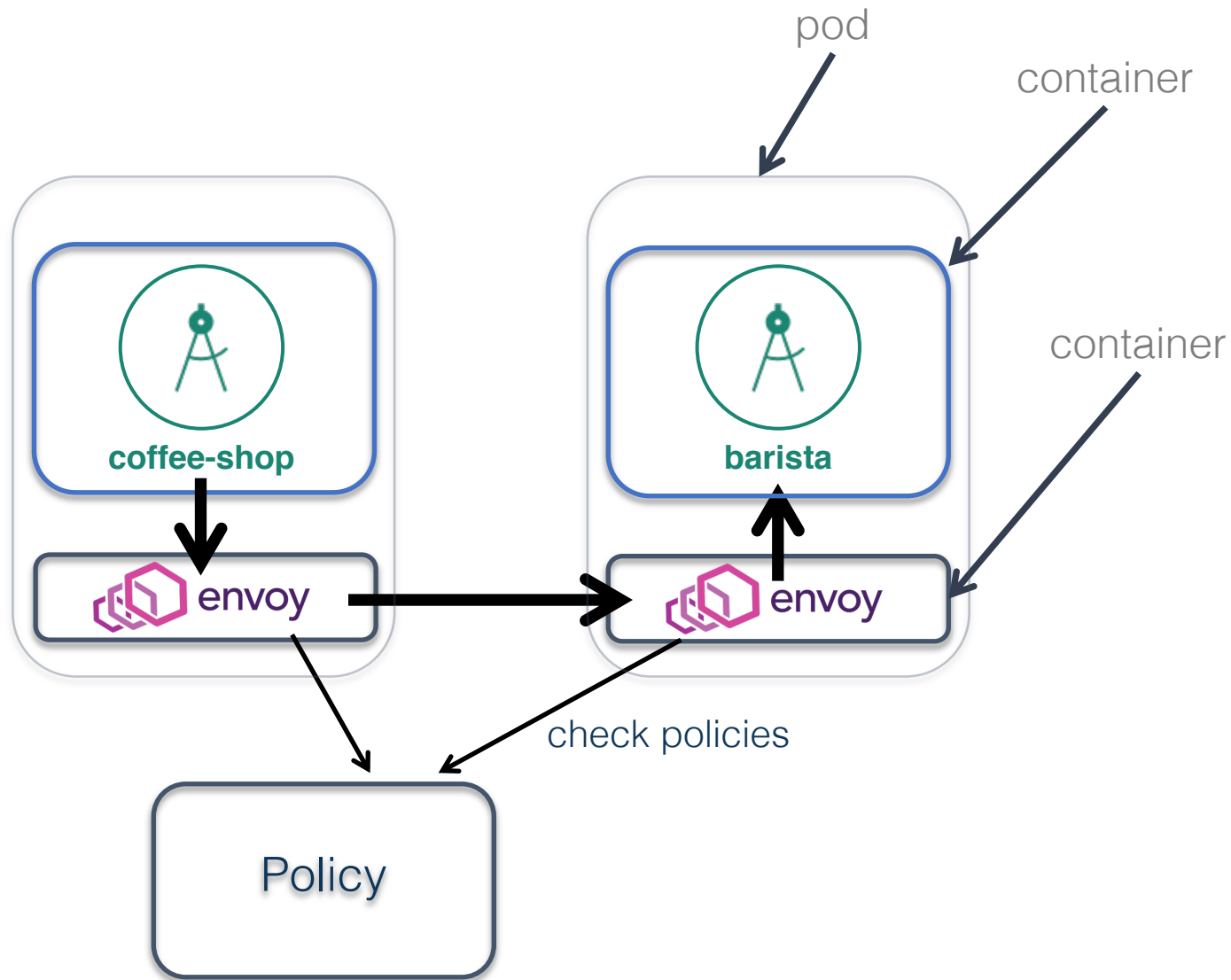




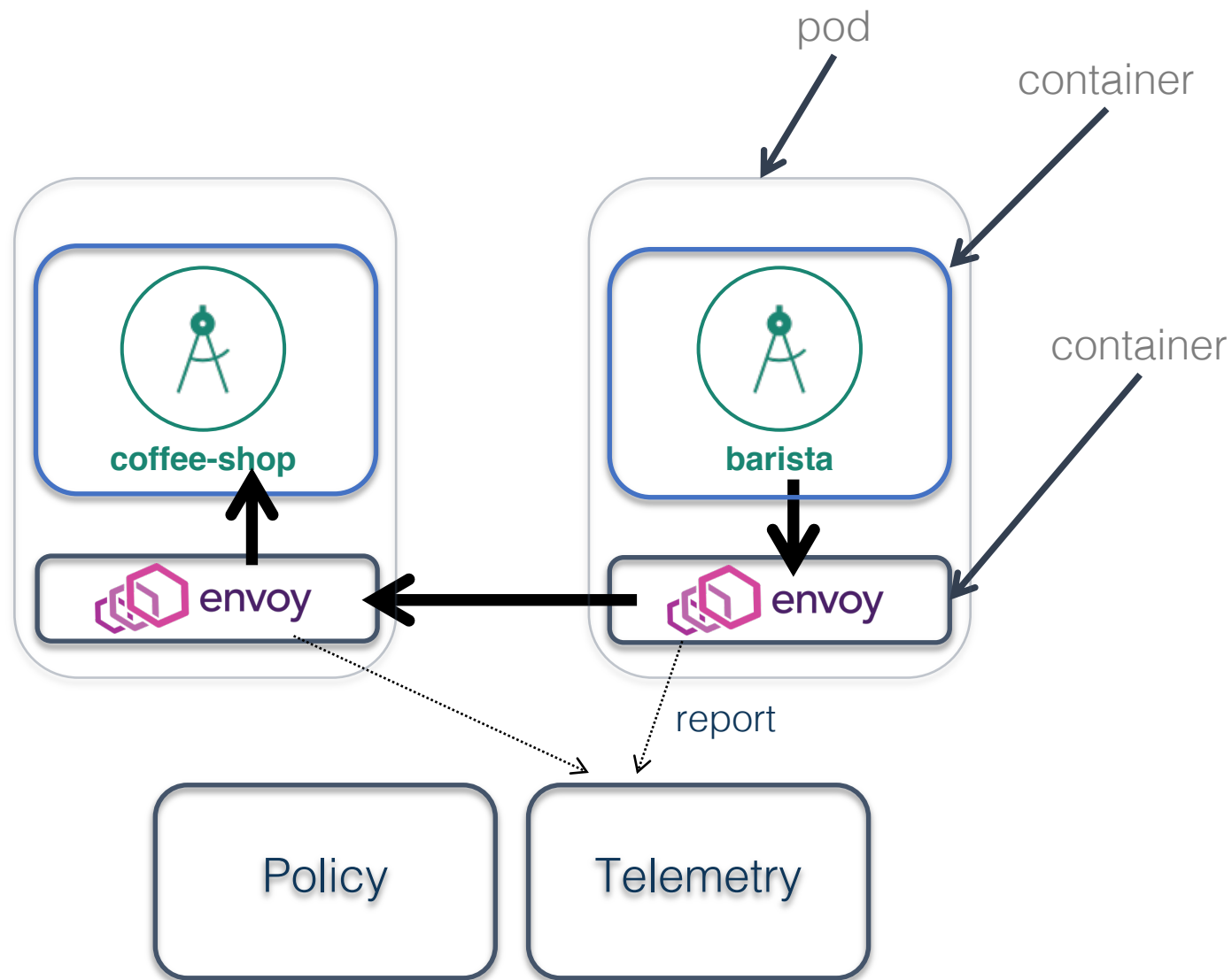
# Как это работает?



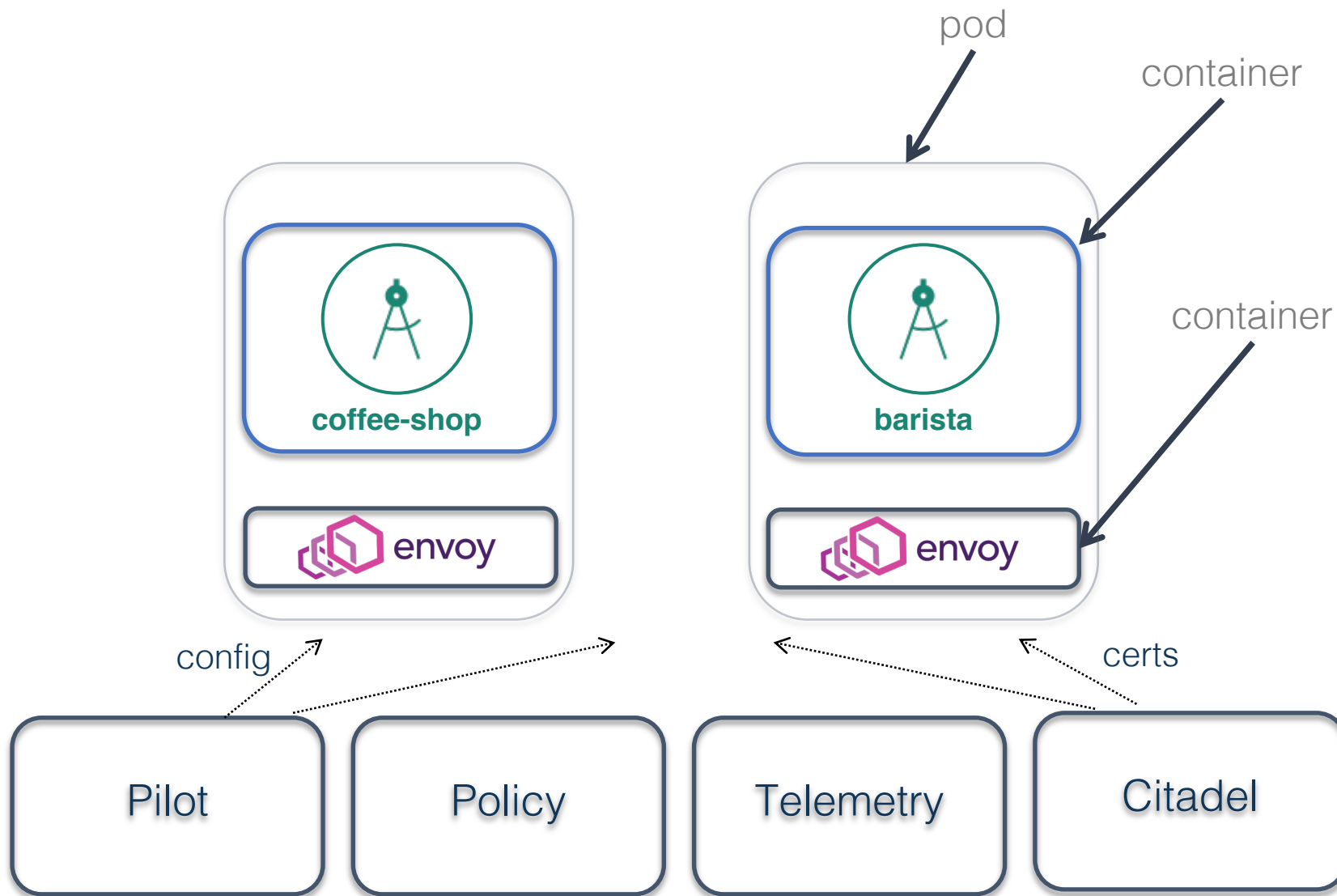
# Envoy перехватывает запросы



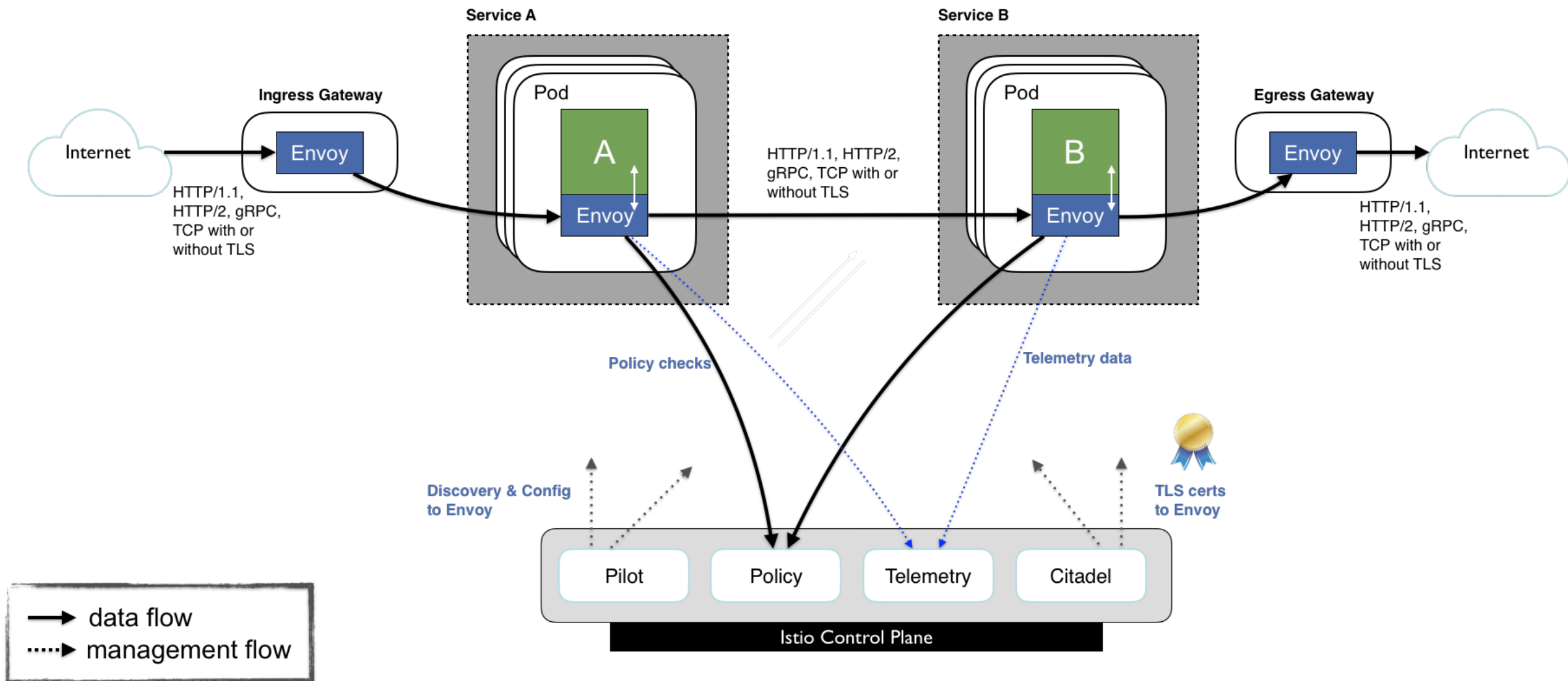
# Отбрасывает телеметрию вызова



# Pilot управляет envoy



# Istio архитектура



# Решения



**kubernetes**



CLOUD**FOUND**RY



elasticsearch



**RED HAT**  
OPENSIFT



Grafana



kibana



ZIPKIN



Prometheus



**Jenkins**