Меня хорошо слышно && видно?

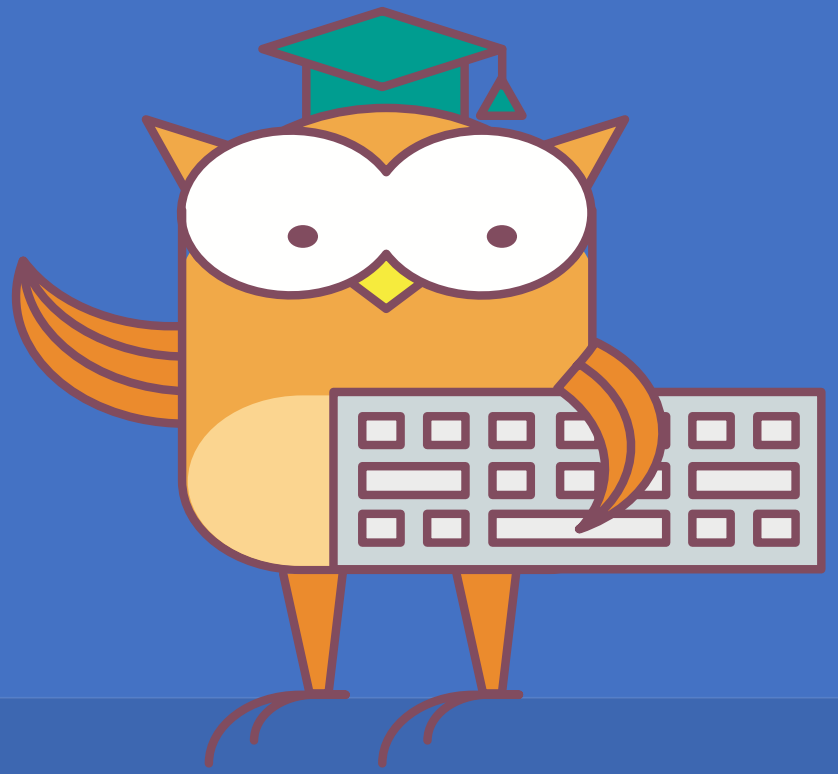
Напишите в чат, если есть проблемы!

Ставьте + если все хорошо

OTUS

Инфраструктурные паттерны

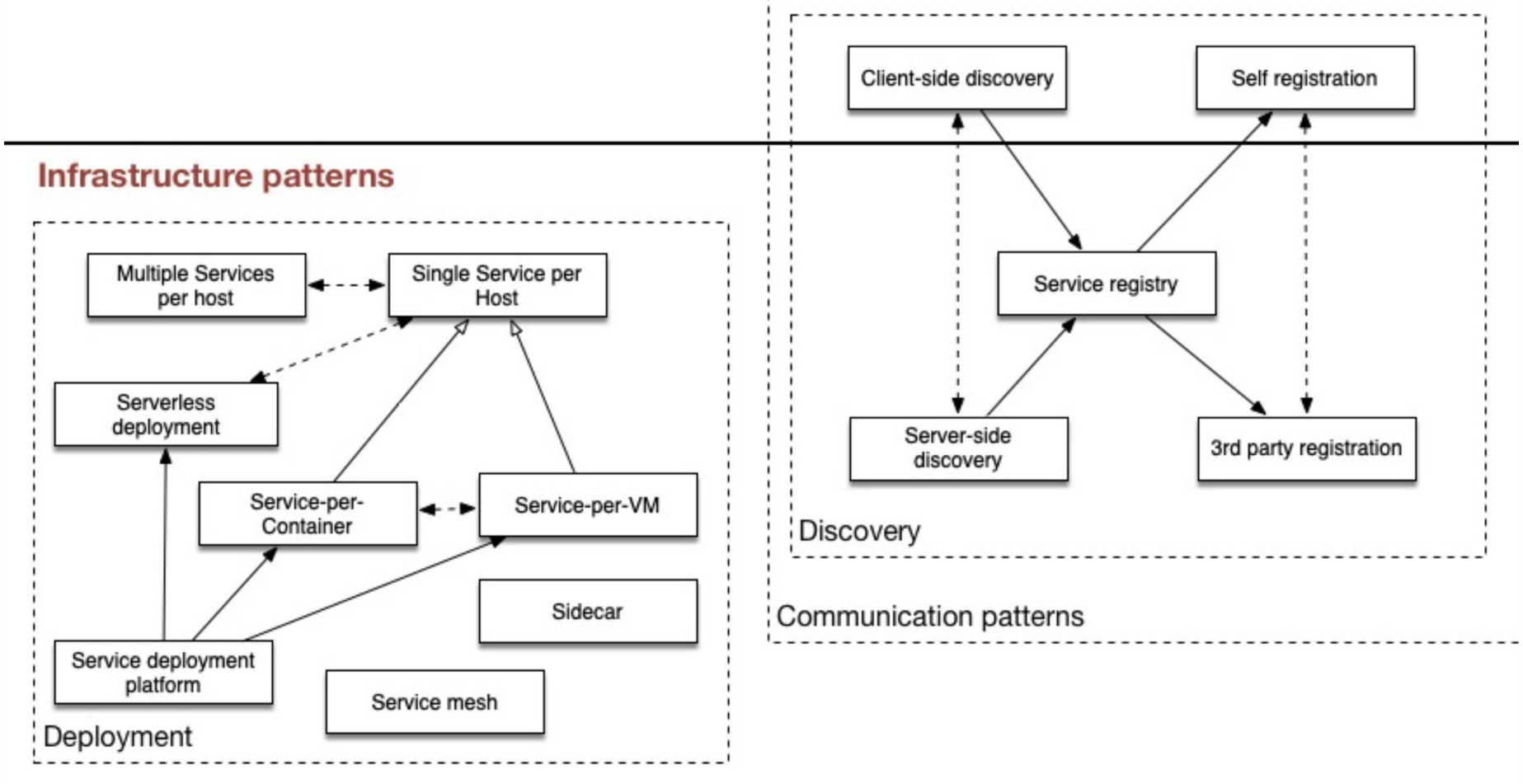
Архитектор ПО



Щетинников Стас

Карта вебинара

Рассмотрим основные инфраструктурные паттерны



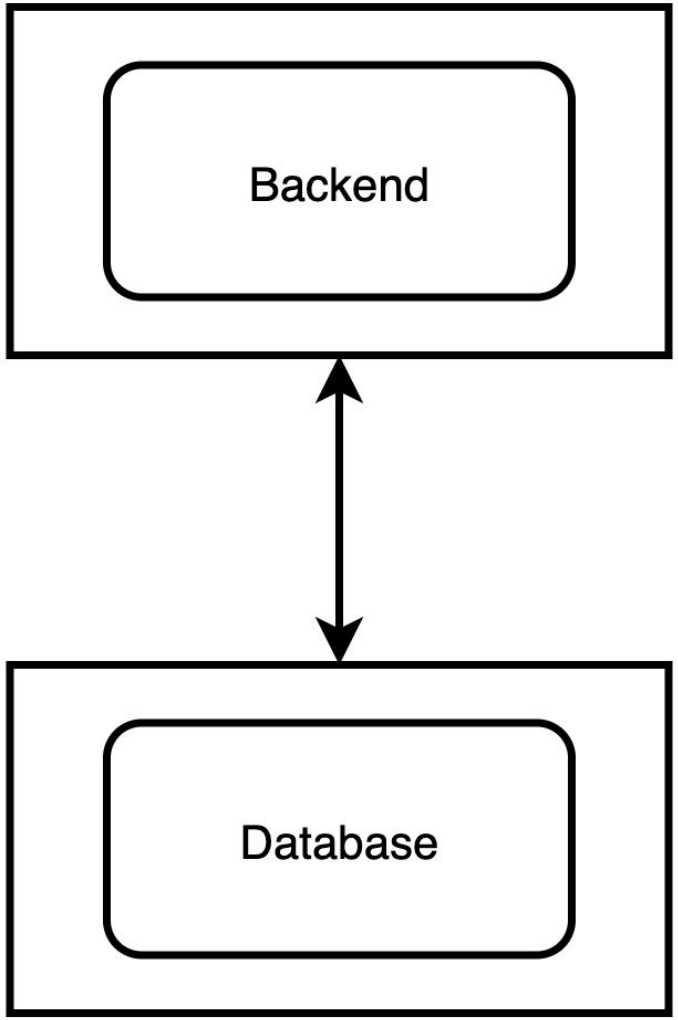
00

Инфраструктурные паттерны

Инфраструктурные паттерны

В самом начале приложение у нас представляет собой связку – приложение – БД. Приложение крутится на одном сервере, БД – на другом.

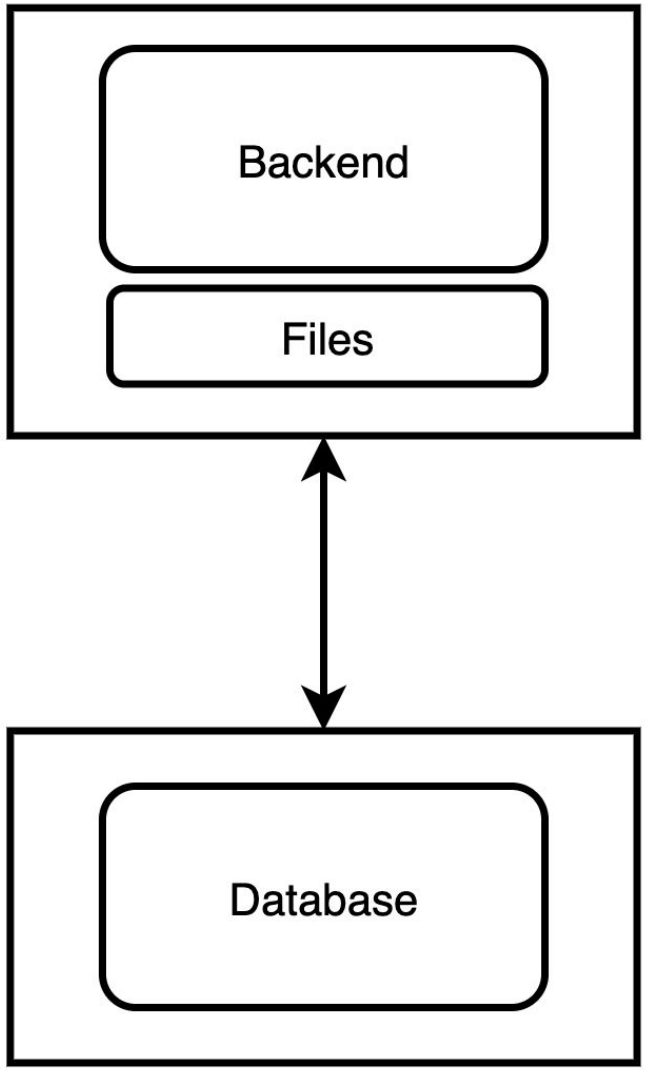
Инфраструктурные паттерны



Инфраструктурные паттерны

Приложение для работы сохраняет какие-то файлы у себя
(например, пользователь загружает какие-то файлы)

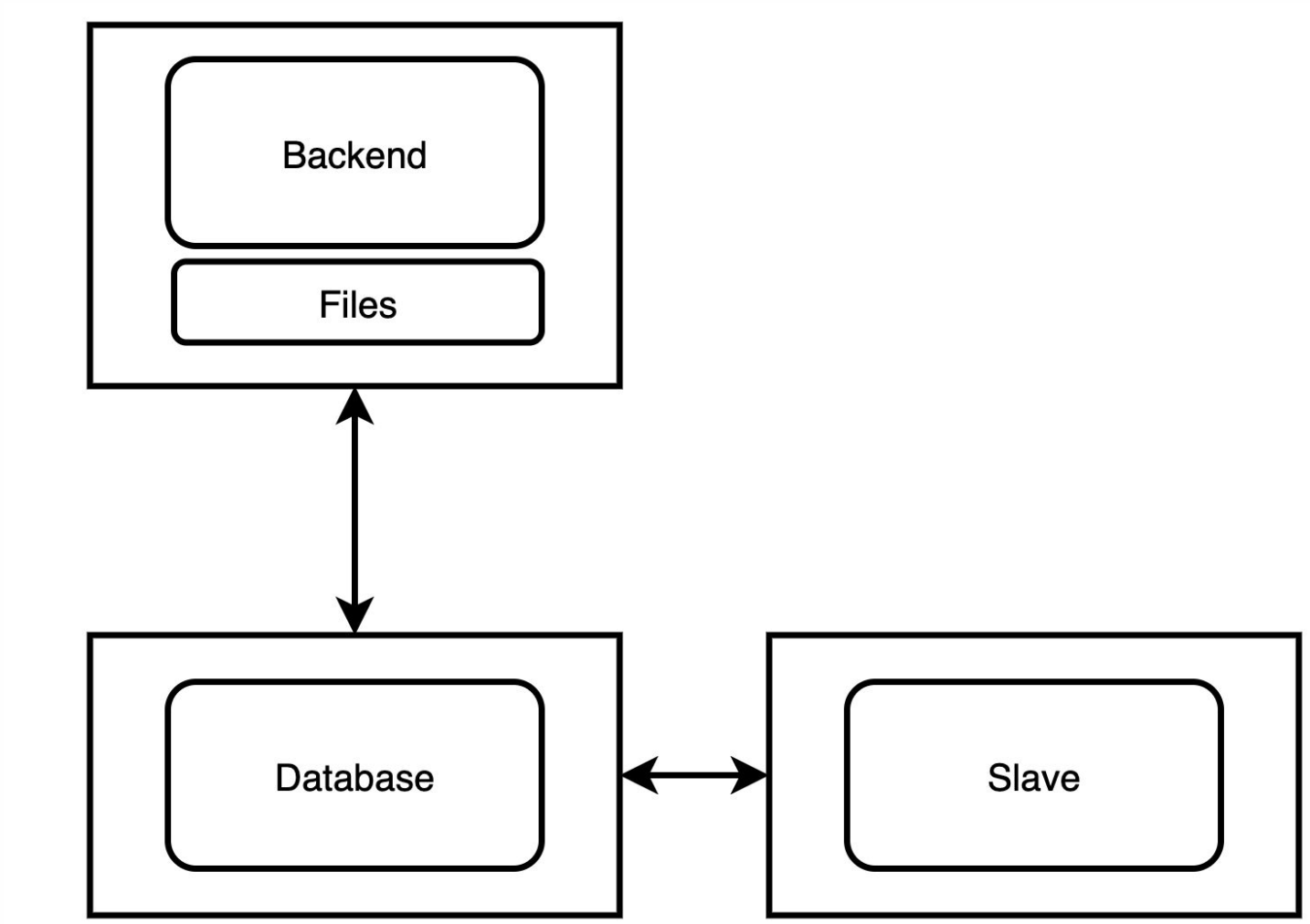
Инфраструктурные паттерны



Инфраструктурные паттерны

Проходит время, нагрузка растет и приходится приложение масштабировать. Для этого добавили slave для БД.

Инфраструктурные паттерны

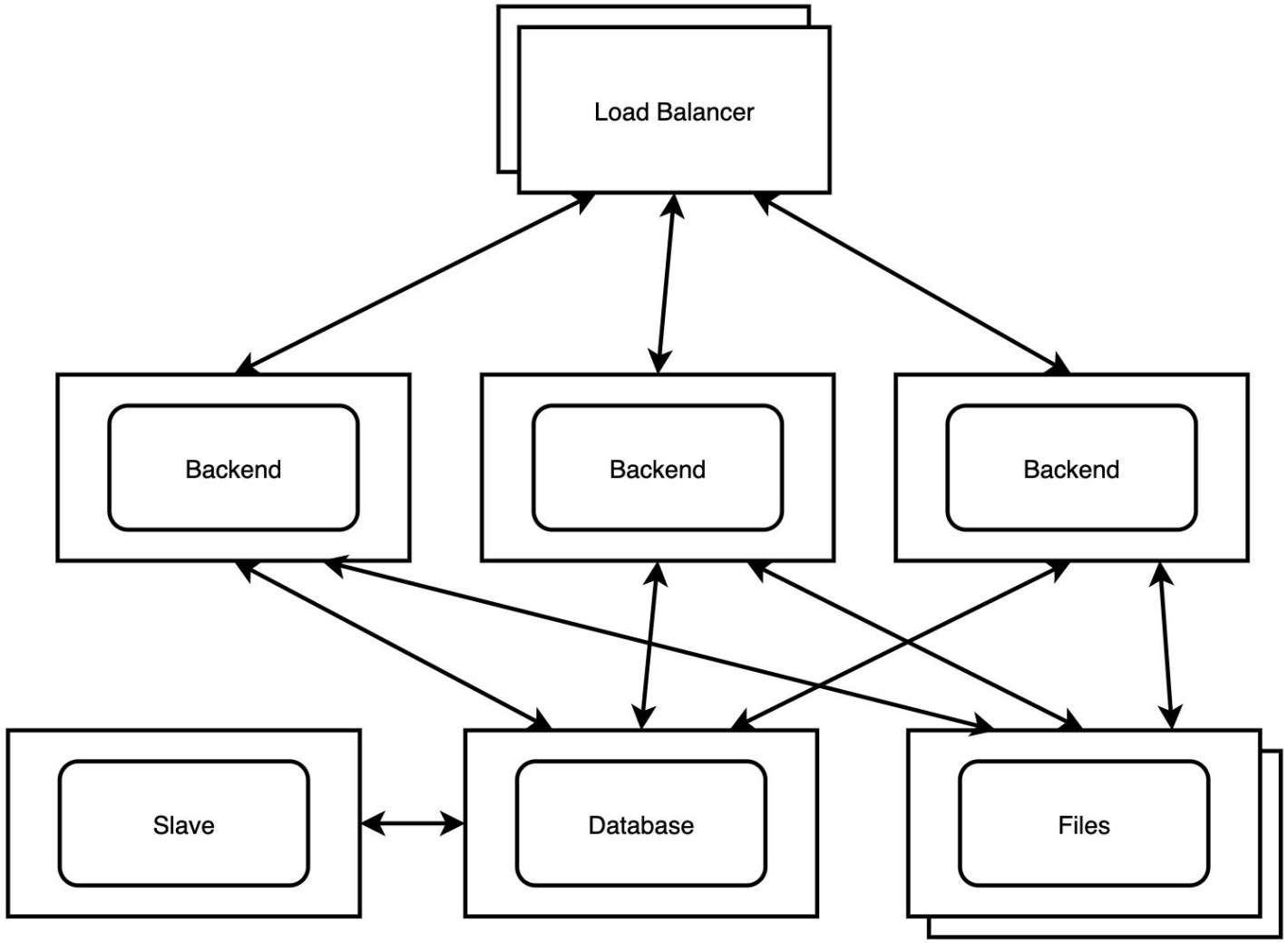


Инфраструктурные паттерны

Для отказоустойчивости и масштабирования приложения, решили добавить несколько серверов с инстансами приложения.

Для этого пришлось вынести файлы на отдельный сервер и монтировать его ко всем серверам с приложением по nfs (например).

Инфраструктурные паттерны



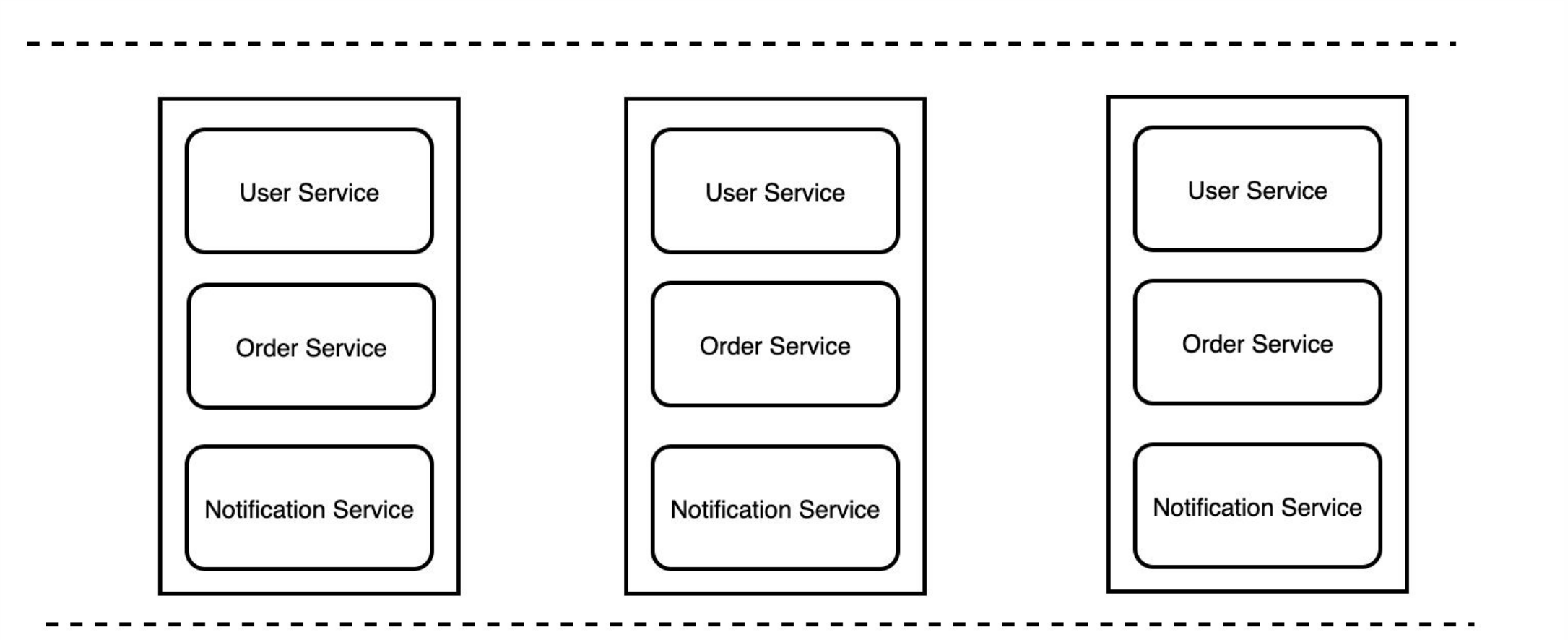
Инфраструктурные паттерны

Для удобства разработки разделили монолит на несколько микросервисов. Теперь на каждом сервере для приложений запускаем не один сервис, а несколько.

Для того, чтобы иметь возможность использовать разные технологии и фреймворки, запускаем сервисы в контейнерах.

А сервисы обращаются друг к другу по локальным портам

Инфраструктурные паттерны

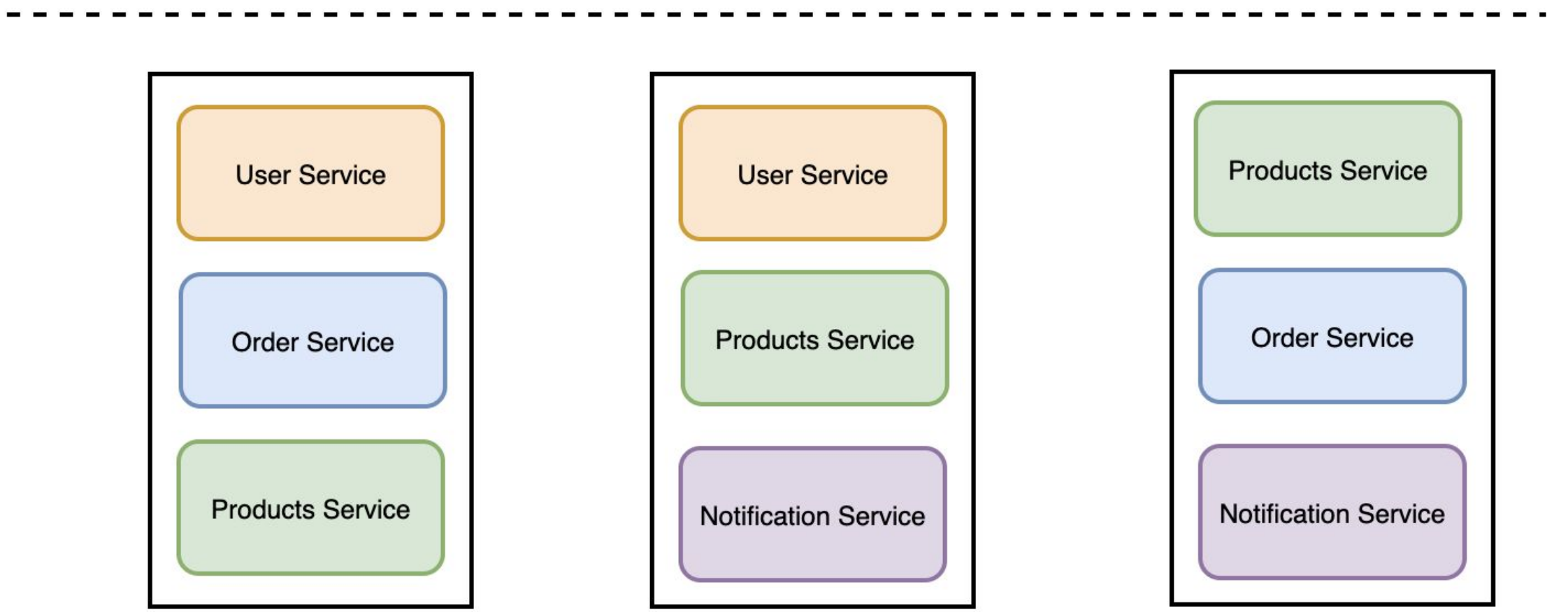


Инфраструктурные паттерны

Через какое-то все сервисы на сервер уже не помещались, потому что сервисы стали более требовательны к ресурсам. Например, сервис Products активно использовал память.

В результате, пришлось держать на каждой ноде по части сервисов.

Инфраструктурные паттерны



Инфраструктурные паттерны

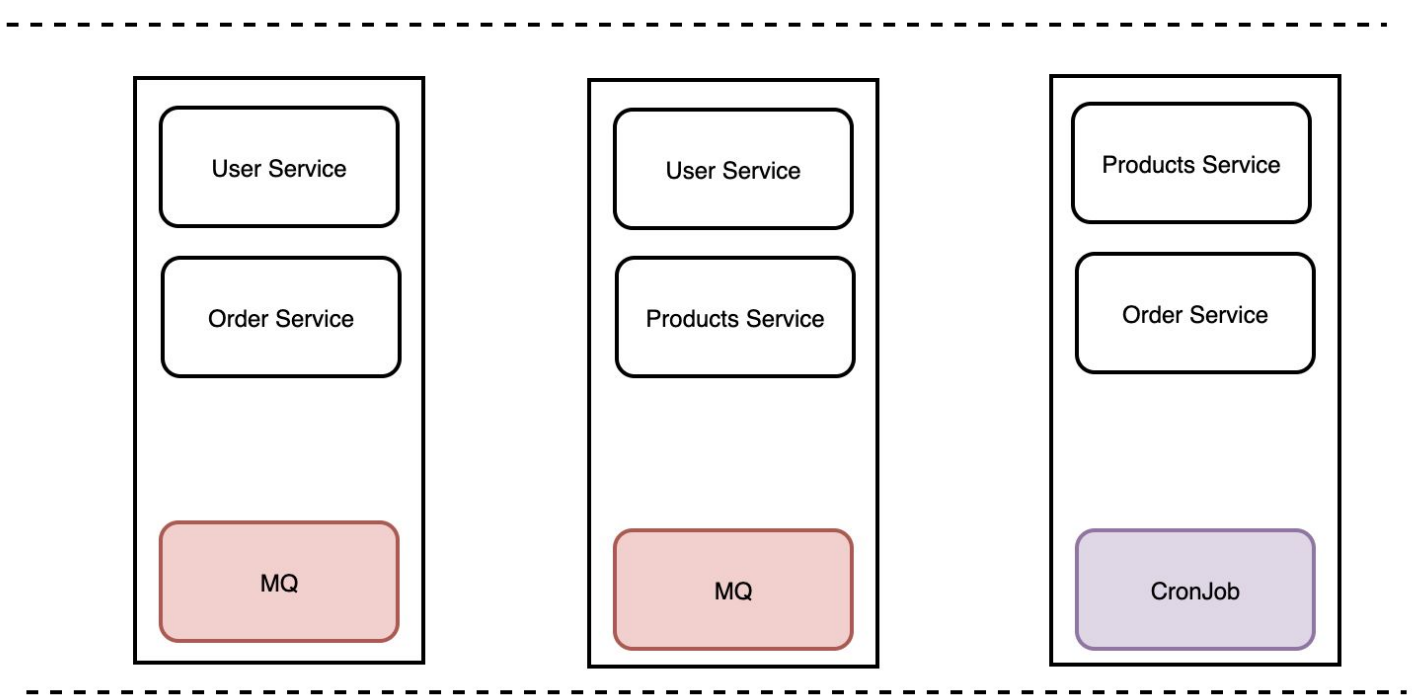
При это возник вопрос, как сервисам обращаться друг к другу.

Для этого стали использовать локальные DNS имена, которые резовлились в IP балансеров, на прописывалось, куда где какой сервер искать и роутить.

Инфраструктурные паттерны

Также стало понятно, что такая схема позволяет запускать и работать не только клиентским сервисам, но и инфраструктурным:

Например, в похожей парадигме можно было запускать rabbitmq, через который общаются сервисы, redis, и т.д.



Основные инфраструктурные проблемы

- Как изолировать сервисы друг от друга? Как сделать так, чтобы сервис А не влиял на сервис Б, если они запущены на одном сервере?
- Как организовать межсервисное взаимодействие? Как сервисам находить друг друга?
- Как обеспечить отказоустойчивость и в случае проблем с сервисом перезагружать его или выключать из балансинга? Как находить проблемы у сервисов?
- Как обеспечить равномерное распределение ресурсов и избежать перегруженности конкретных нод?
- Как обеспечить конфигурирование сервисов?
- Как автоматизировать создание нового сервиса?
- Как обеспечить простое масштабирование сервисов?
- Как обеспечить автоматическое обновление конфигураций балансеров, отвечающих за входящий трафик при создании нового сервиса или старого сервиса?
- Как обеспечить обновление новой версии?

Основные инфраструктурные проблемы

Чем больше сервисов, тем больше проблем. Издержки на управление растут экспоненциально с количеством сервисов (в том числе и инфраструктурных).

01

CI/CD

CI/CD

Без автоматизации работ с инфраструктурой и сборкой, тестированием и выкладкой кода в микросервисной архитектуре ничего сделать не получится.

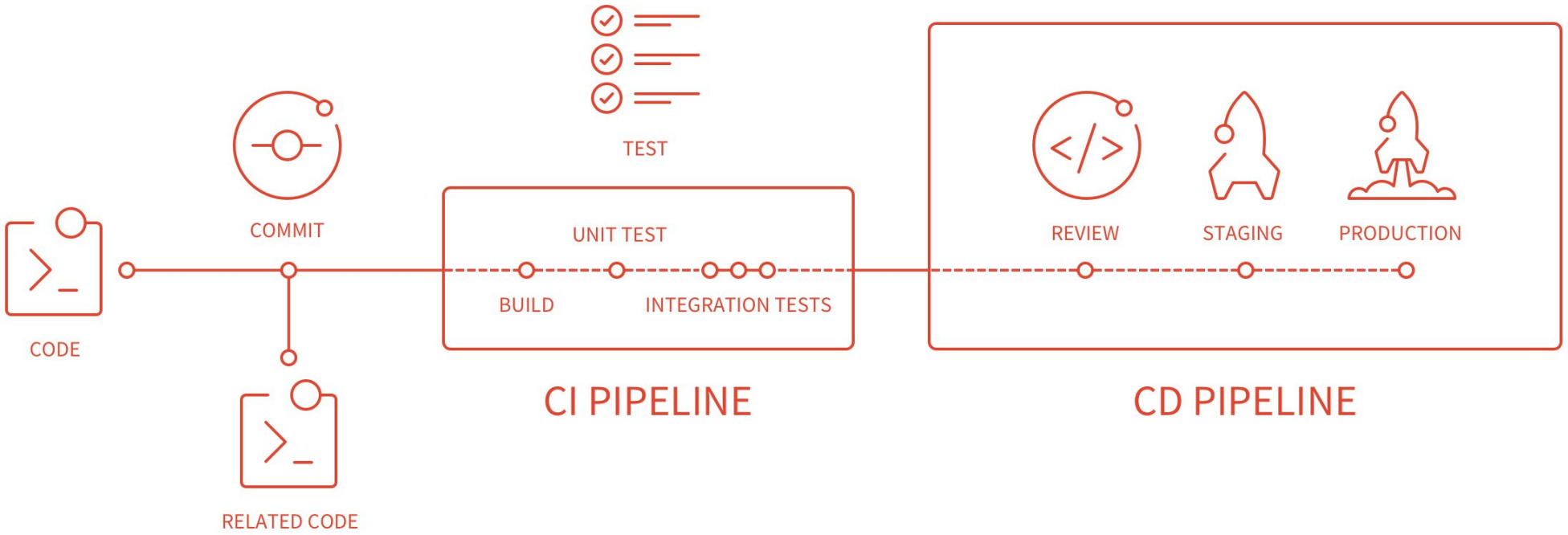
CI/CD

CI – continious integration – практики автоматизации процесса интеграции приложения – сборка, тестирование

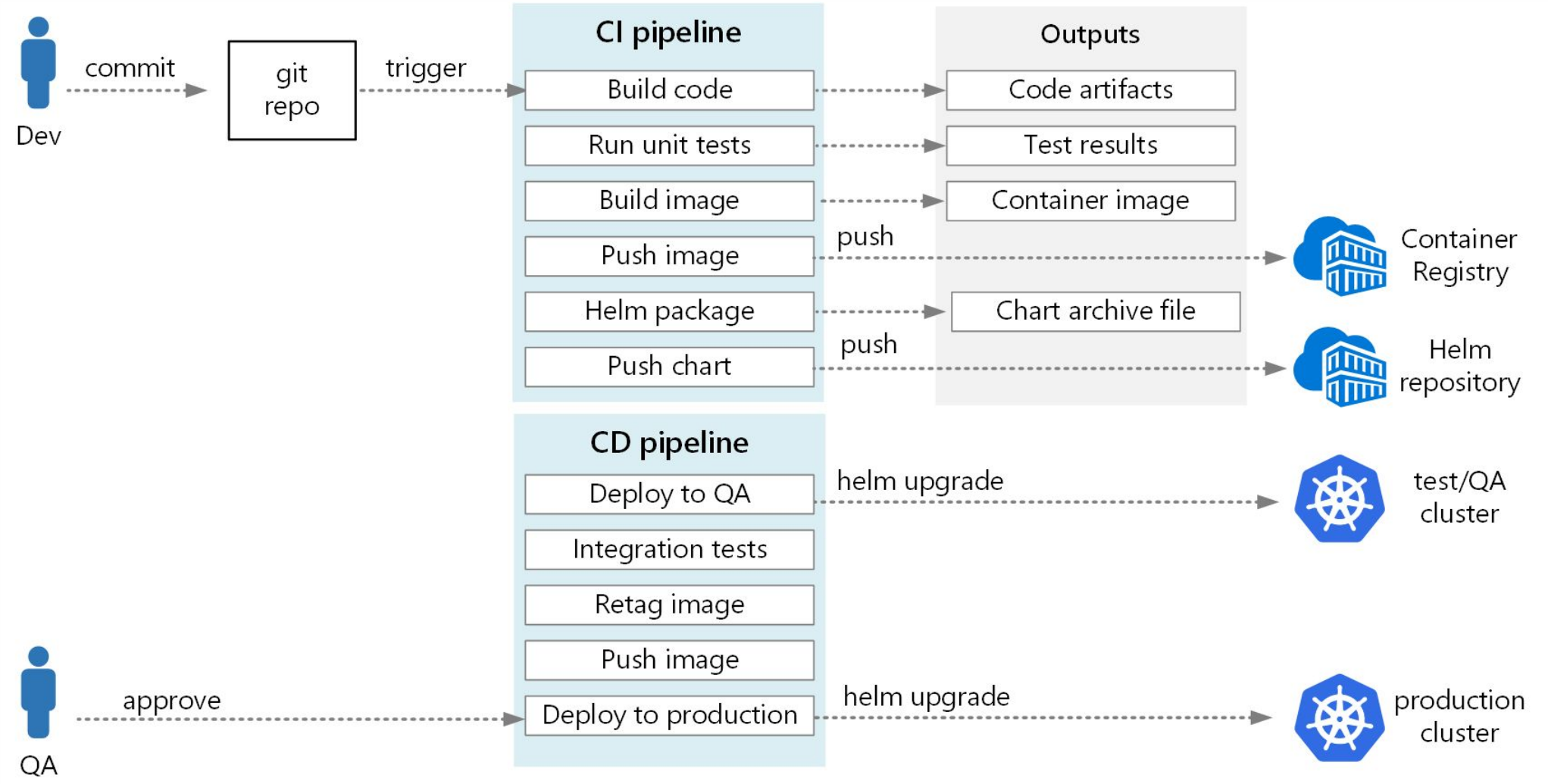
CD – continious deployment – практики автоматизации процесса выкладки приложения на разные среды

Для CI/CD используется инструменты типа Jenkins, Gitlab, Bamboo, TeamCity и т.д.

CI/CD


TEST
COMMIT
UNIT TEST
REVIEW
STAGING
PRODUCTION
CODE
BUILD
INTEGRATION TESTS
CI PIPELINE
CD PIPELINE
RELATED CODE

CI/CD с Kubernetes



CI/CD

Также крайне важна автоматизация задач, связанных с управлением и работой с инфраструктурой: масштабирование (увеличение количества реплика), запуск различного рода одноразовых задач и миграций, переконфигурирование приложения и т.д.

02

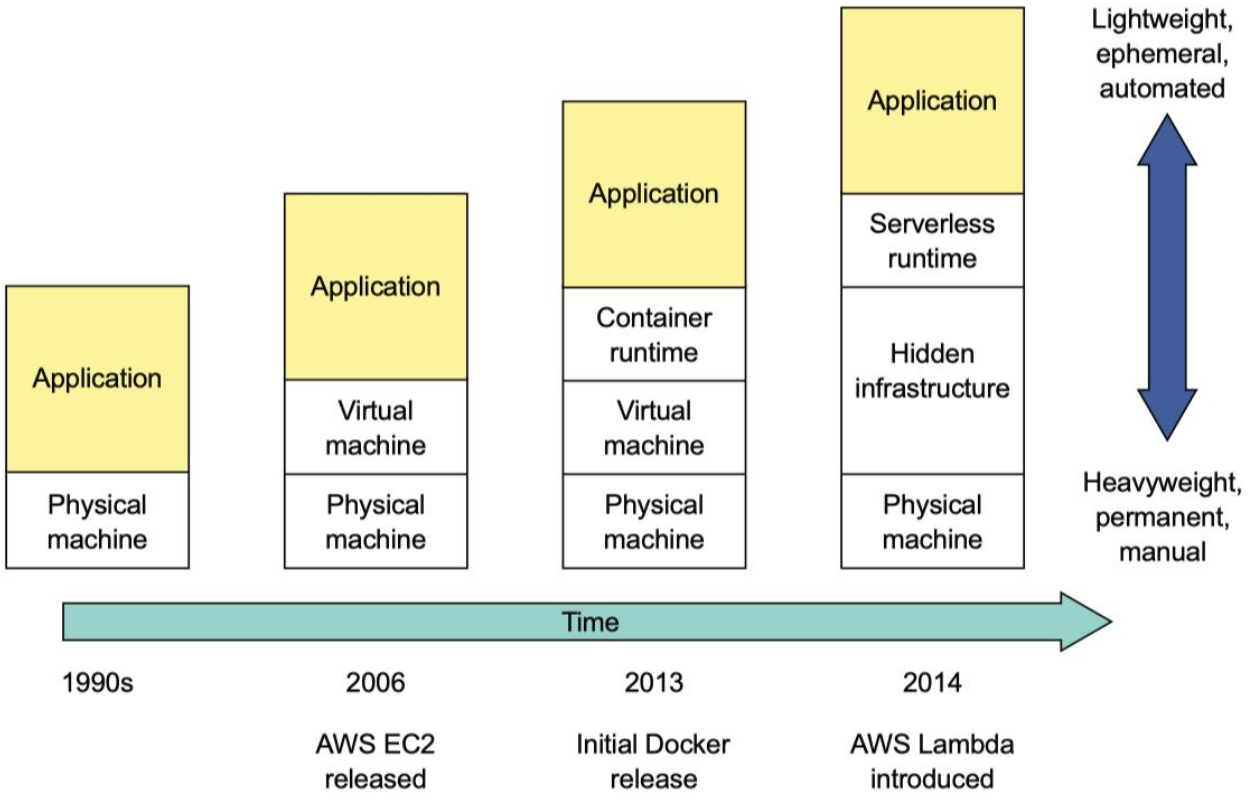
Дистрибуция артефактов

Артефакт и сервис

- Что такое артефакт и чем он отличается от сервиса?

Как разместить несколько сервисов на одной физической машине?

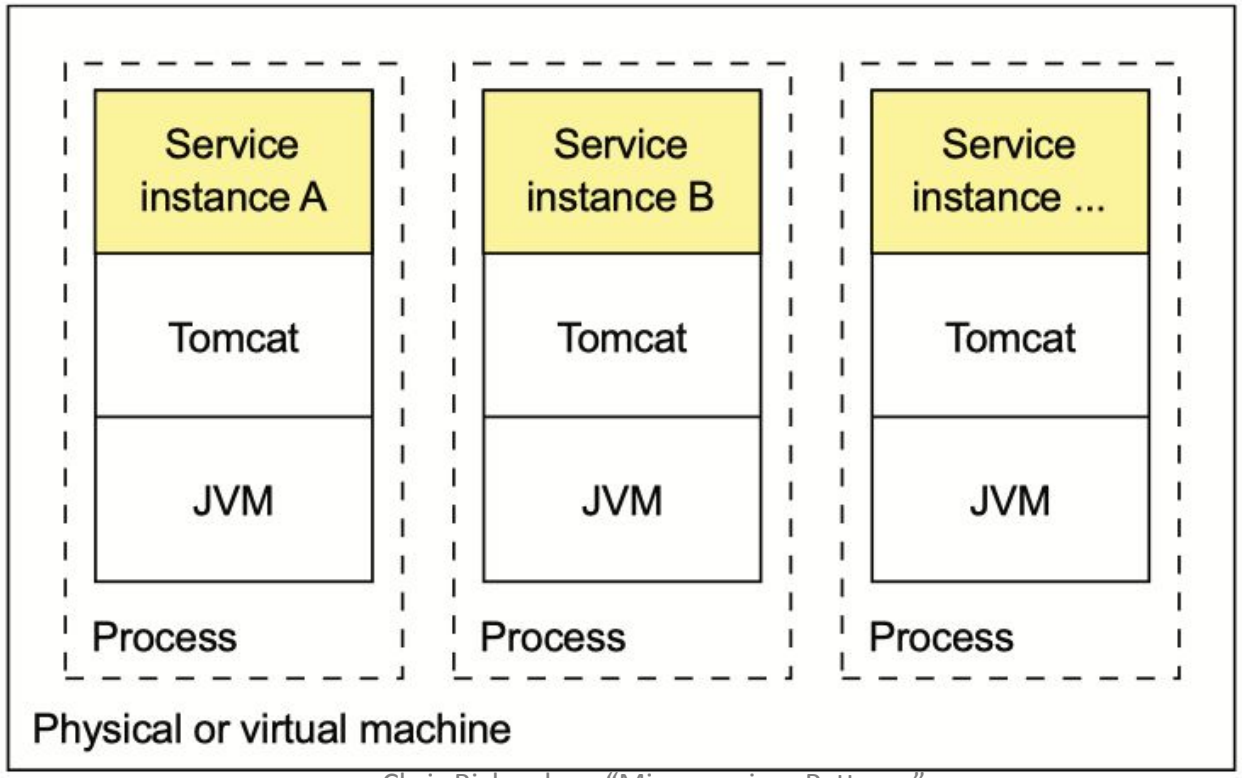
- **Сервер приложений** jvm tomcat/jboss, python uwsgi
- **Виртуальные машины** Vagrant, vmware
- **Контейнеры** Docker, rkt



Chris Richardson “Microservices Patterns”

Сервер приложений

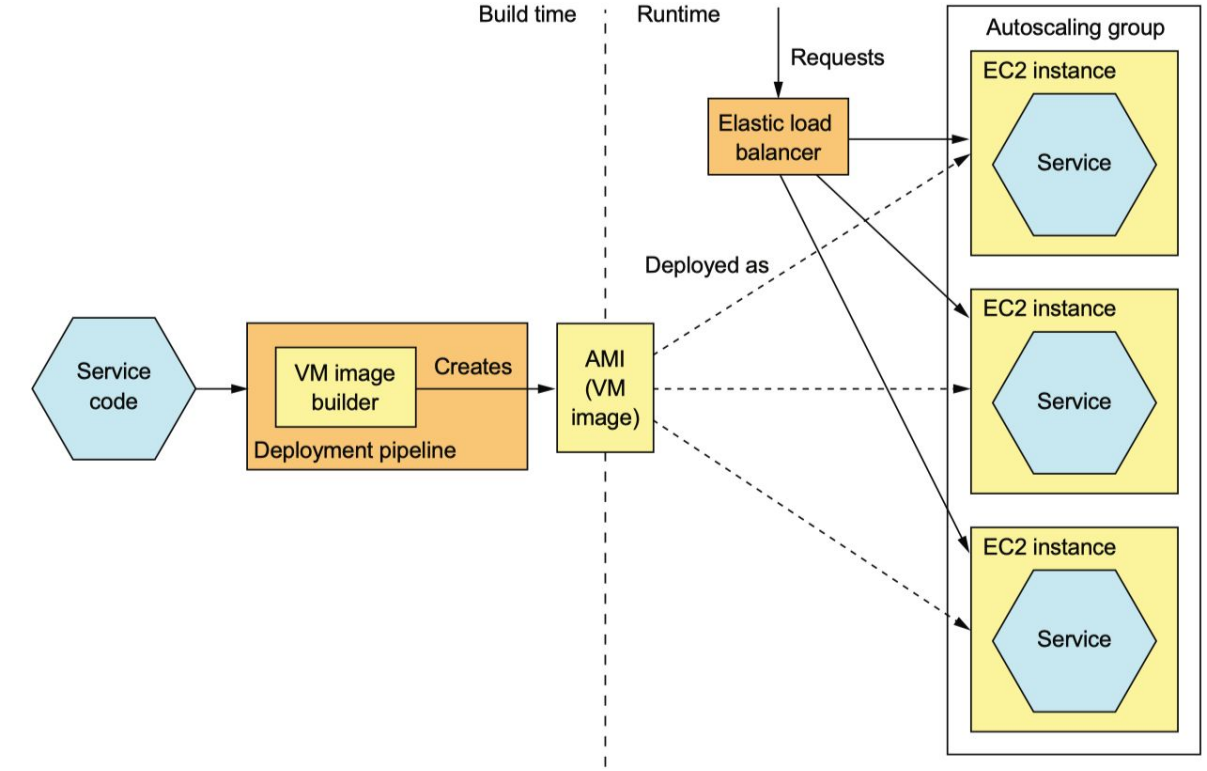
- Быстрый деплой
- Хорошая утилизация ресурсов
- Чаще всего отсутствует изоляции по ресурсам (CPU, Memory) между разными сервисами
- Фиксированный язык программирования или фреймворк



Chris Richardson “Microservices Patterns”

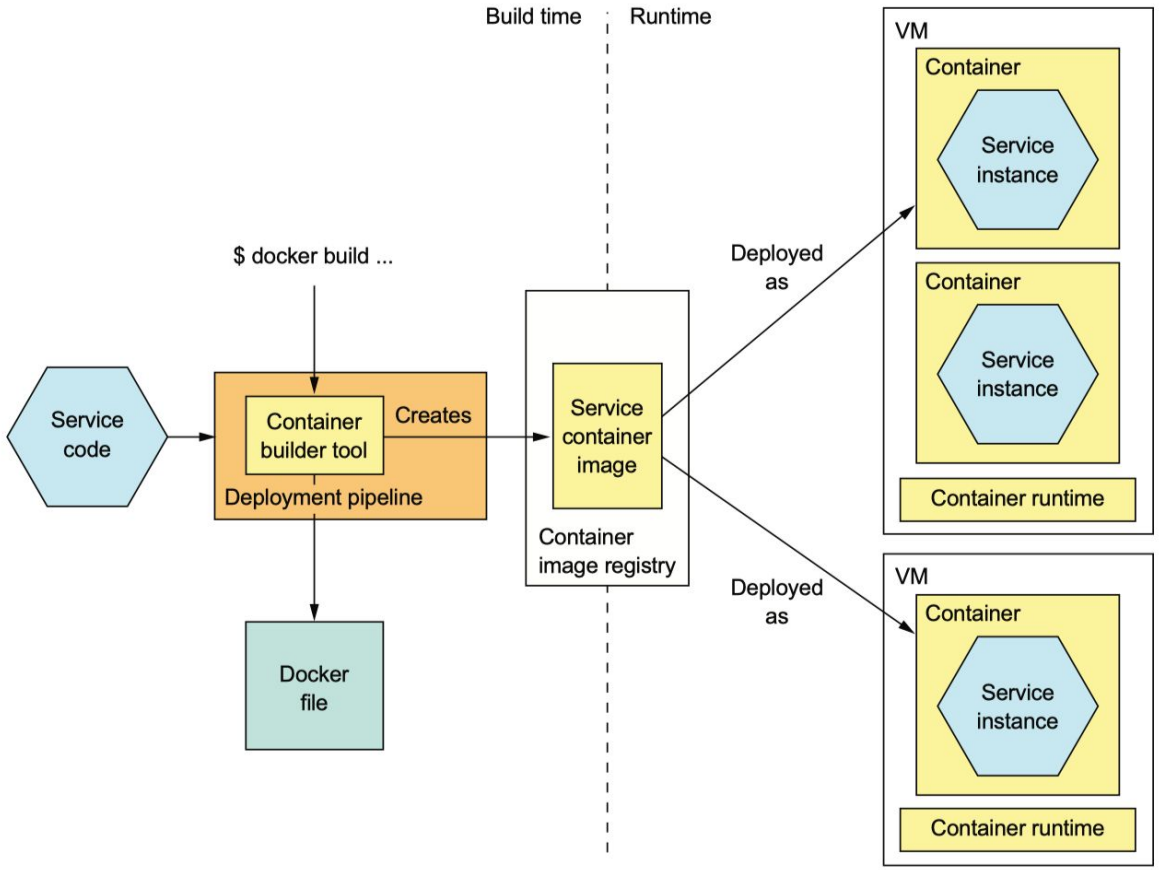
Виртуальная машина

- Technology agnostic
- Изоляция ресурсов между сервисами
- Большая утилизация ресурсов
- Долгий деплой



Контейнеры

- Technology agnostic
- Изоляция и ограничение сервисов друг от друга
- Эффективная утилизация ресурсов



Self contained приложения

Сервисы, которые не зависят от конфигурации среды исполнения, называются самоупакованными (self contained).

Т.е. сервис упакован в «контейнер», в котором все есть: все необходимые библиотеки, фреймворки нужных версий и т.д.

Что должно быть в Docker образе?

Если мы используем систему контейнеризации (docker/podman/…) в контейнер упаковывать все приложение со всеми его зависимостями.

- Системные компоненты php-fpm
- Пакетный менеджер (если нужен)
- Зависимости
- Системные библиотеки
- Собранные ассеты

И т.д.

А чего всё ещё не хватает чтобы запустить наш сервис?

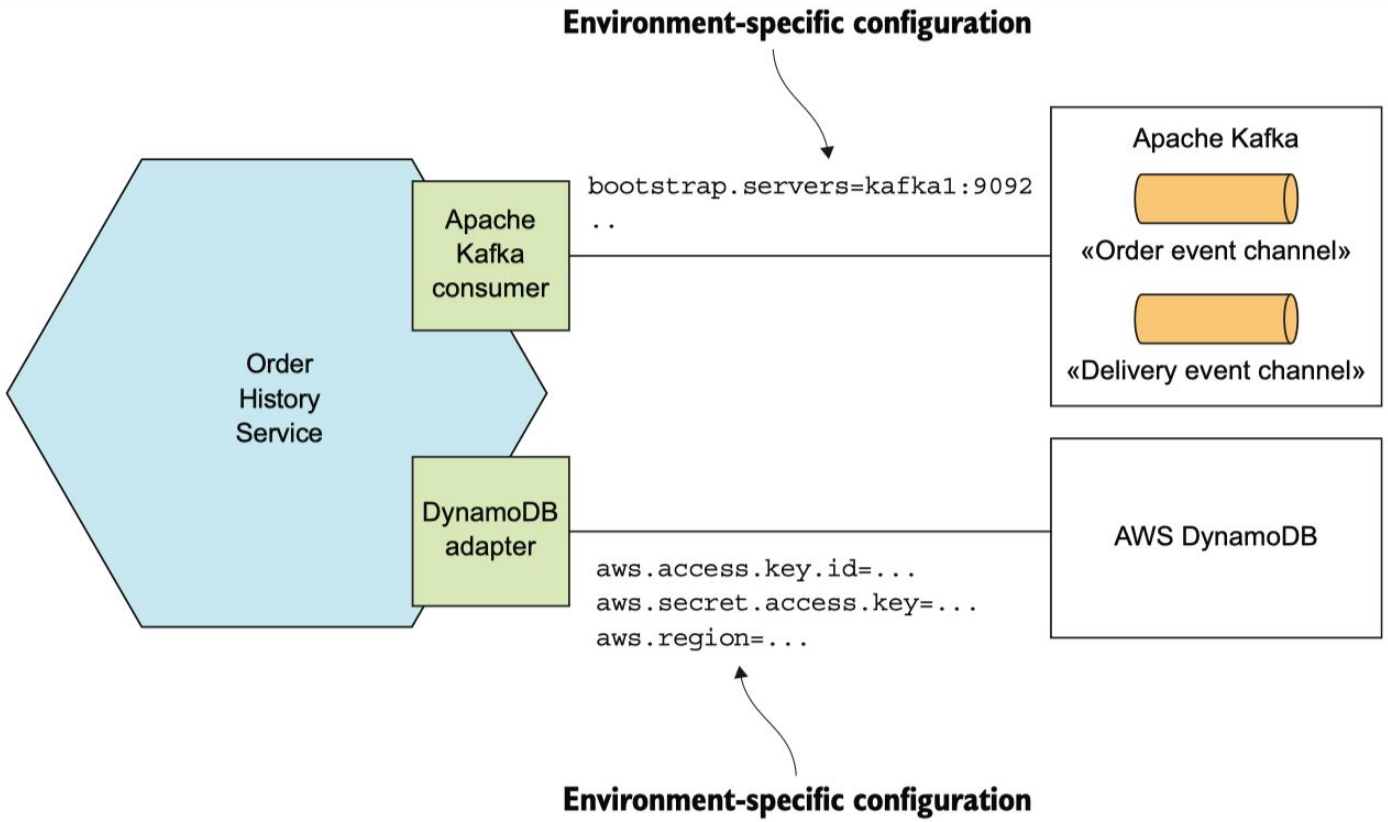
02

Конфигурирование приложений

Конфигурирование приложений

Конфигурация приложения – это все, что может меняться, между развертываниями приложений.
Например,

- Идентификаторы подключения к ресурсам типа базы данных, кэш-памяти и другим [сторонним службам]



Конфигурирование приложений

Конфигурация должна быть отделена от кода и артефактов деплоя.

Артефакт деплоя один, а конфигурации - разные.

Кодовая база приложения может быть в любой момент открыта в свободный доступ без компрометации каких-либо приватных данных

https://12factor.net/ru/config

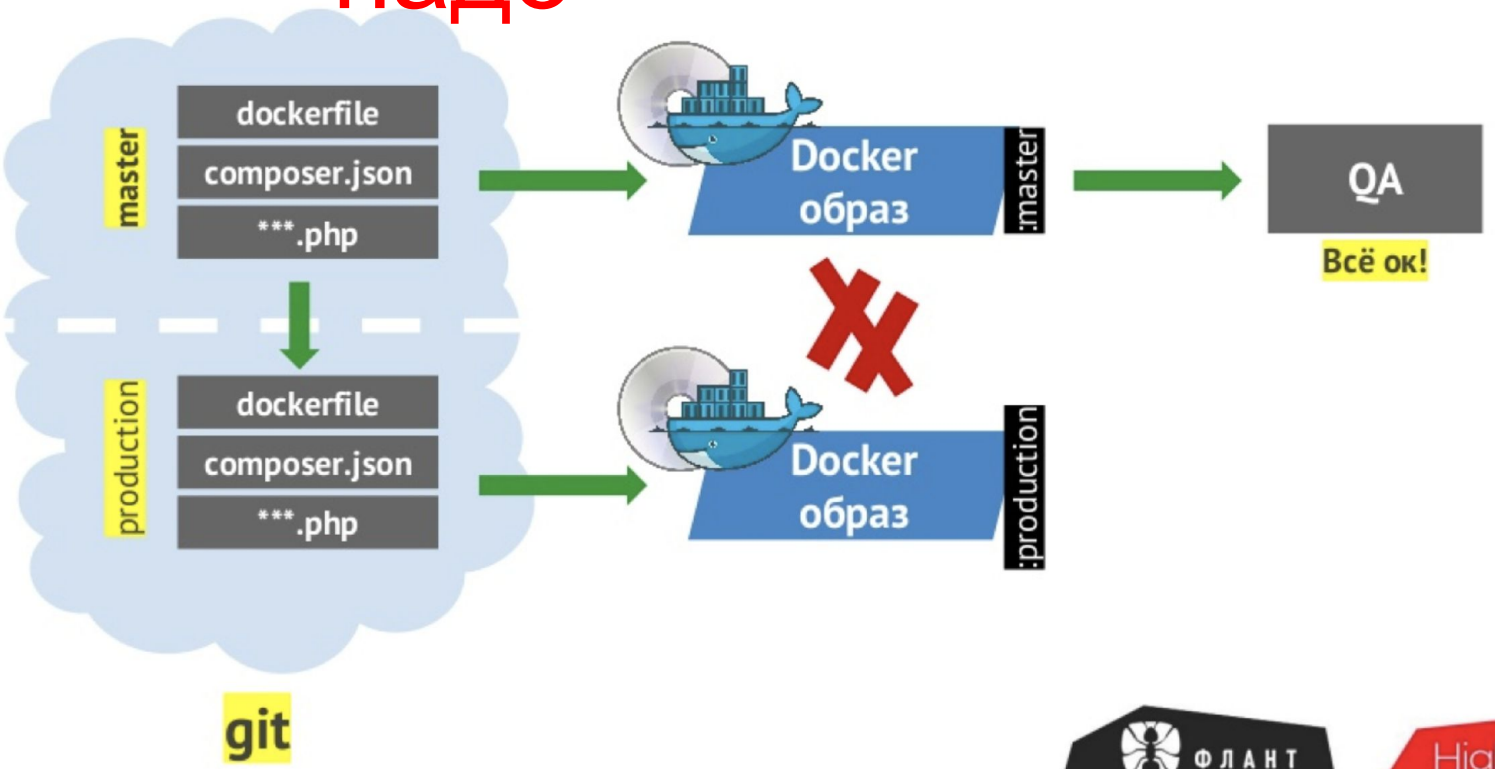
Один docker image для разных сред

Крайне НЕ рекомендуется для разных тип сред собирать РАЗНЫЕ докер image, которые отличаются друг от друга.

Почему это плохо?
Потому что тот артефакт, который мы протестировали, он самом деле будет отличаться от того, что пойдет в production.

Один docker image(артефакт) для разных сред

Так делать не надо



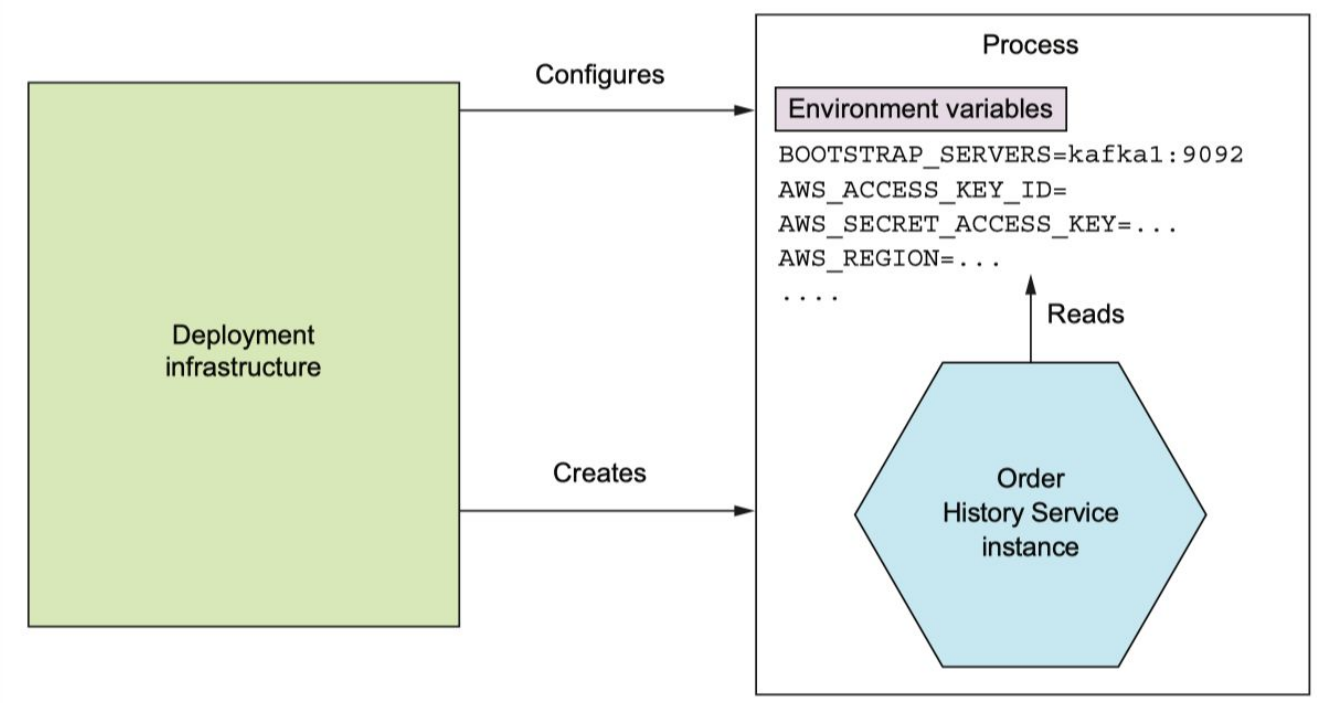
https://www.slideshare.net/profyclub_ru/cicd-kubernetes-gitlab

Push модель конфигурирования

После деплоя оркестратор передает приложению конфиг

Конфиг может передаваться через

- Переменные окружения (ENV)
- Конфигурационный файл
- Параметры командной строки

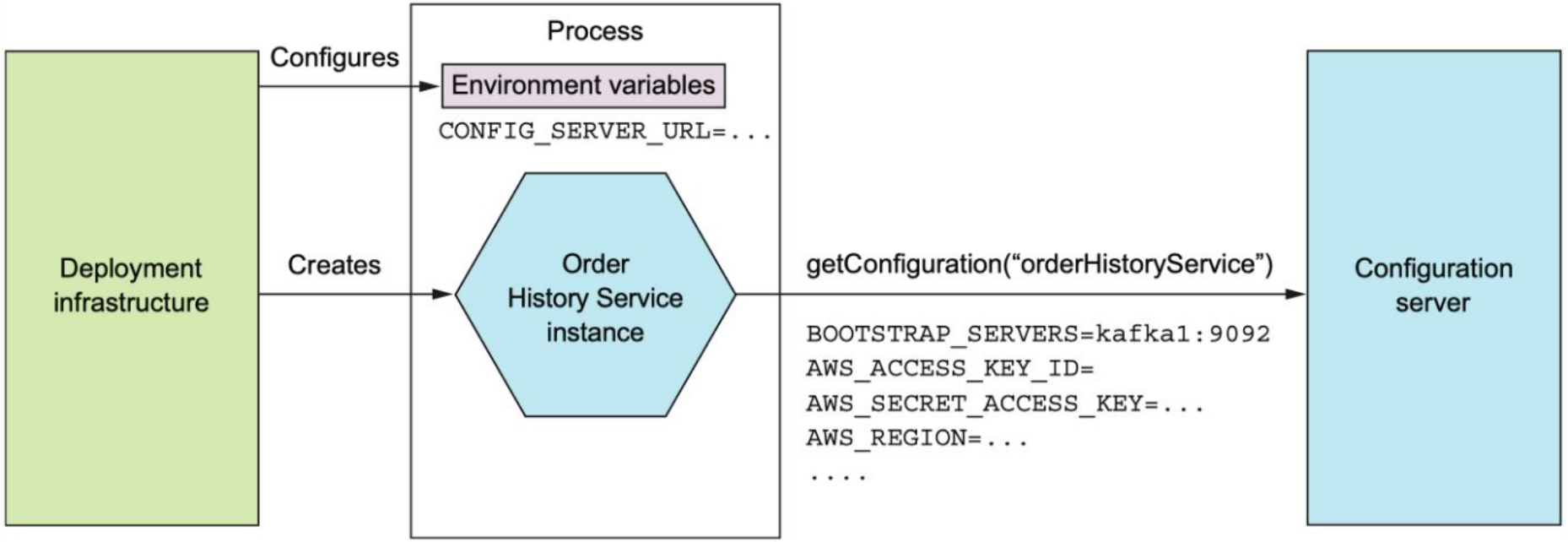


Pull модель конфигурирования

В момент старта приложение читает свой конфиг из внешнего сервиса.

Конфиг может хранится в:

- SQL
- NoSQL
- Git
- Vault
- И т.д



03

Паттерны деплоя

Стратегии деплоя

- Recreate
- Rolling update, Canary и A/B testing
- Blue/green

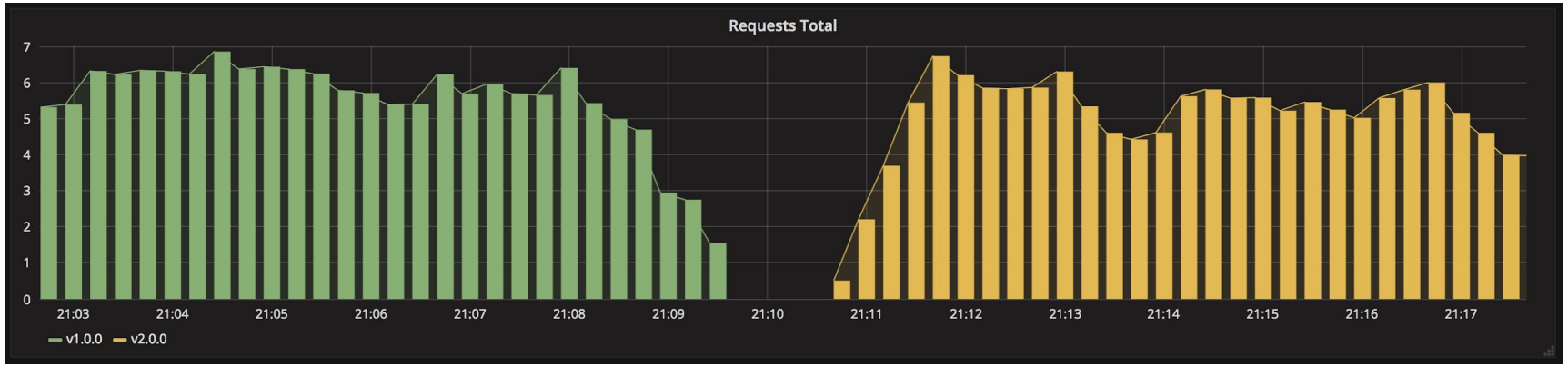
Лайфхаки

- Feature toggle
- Traffic Mirroring

Recreate

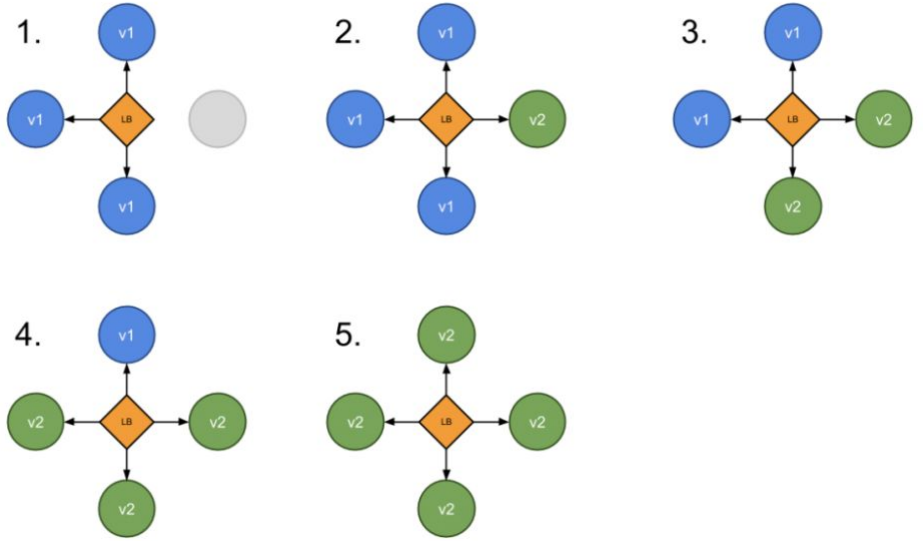
Убить существующий деплой
Поднять новый

- - Даунтайм
- + Всё очень просто



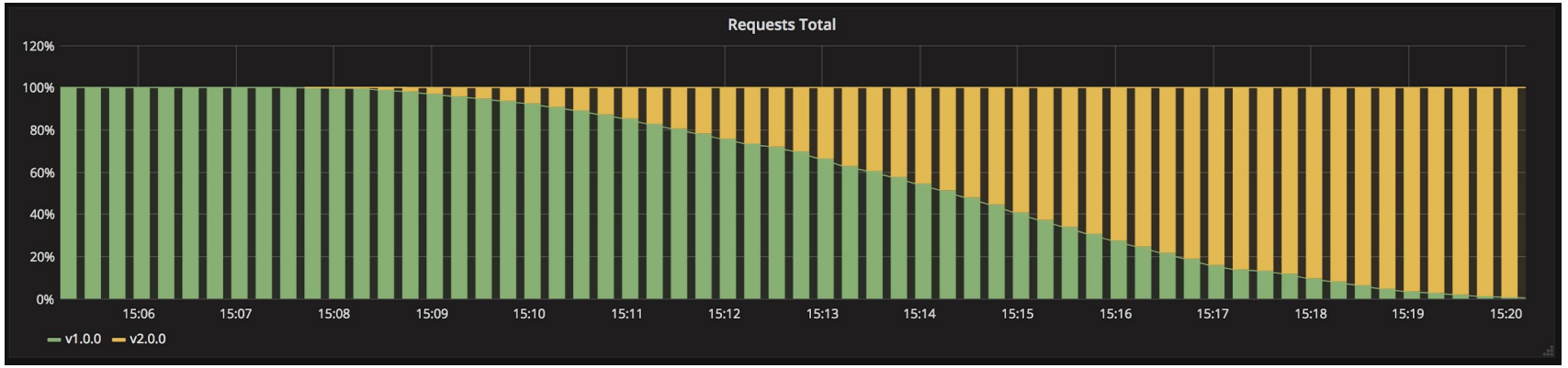
Rolling update

- Снимаем трафик с инстанса
- Поднимаем инстанс с новой версией
- Переключаем в него трафик



Плюсы и минусы:

- \+ нет даунтайма
- Время раскатки зависит от настроек
- \- API **без обратной совместимости не** раскатать

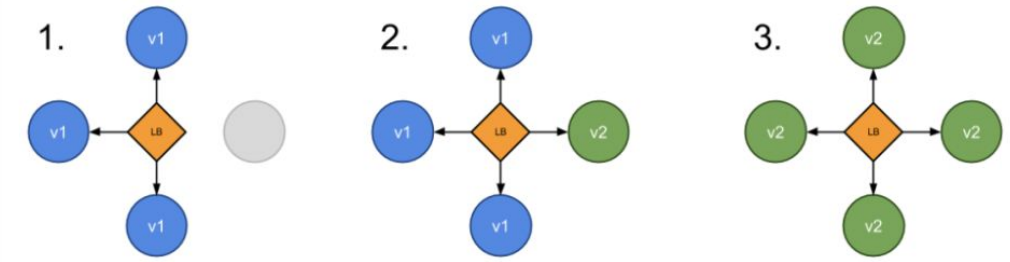


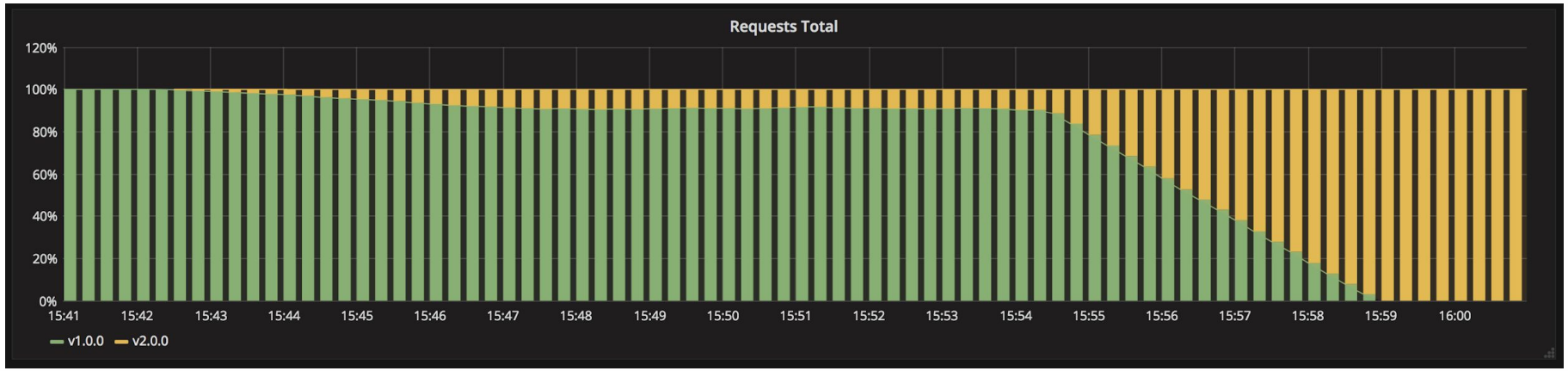
Канареечные деплои и A/B тестирование

- Поднимаем новую версию одновременно со старой
- Переключаем на нее часть трафика
- Если все хорошо, переключаем остальной трафик

Плюсы и минусы:

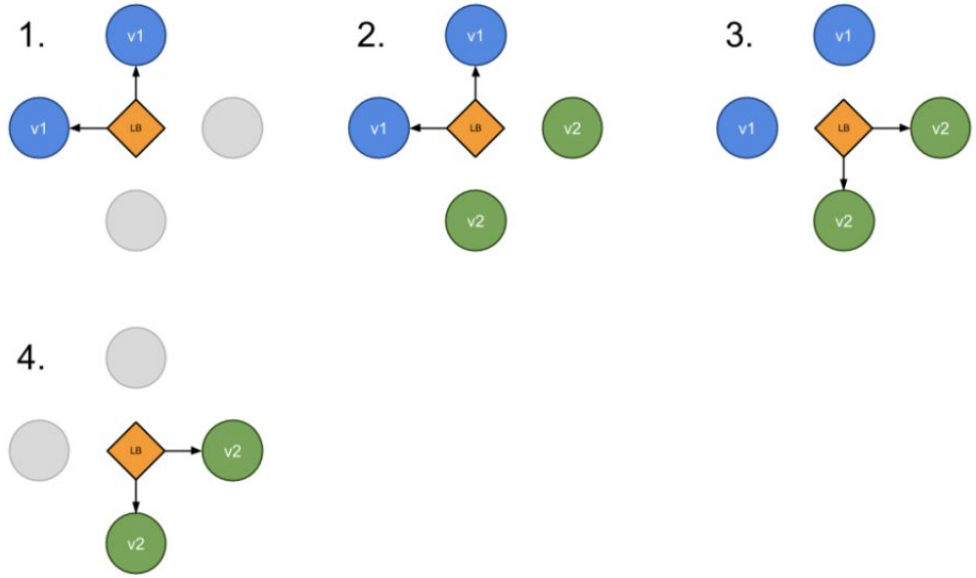
- - API **без обратной совместимости нельзя** раскатить
- \+ нет даунтайма, быстро откатить
- \+ уменьшаем риски плохого релиза
- - В случае A/B и привязки к пользователю сложнее в реализации





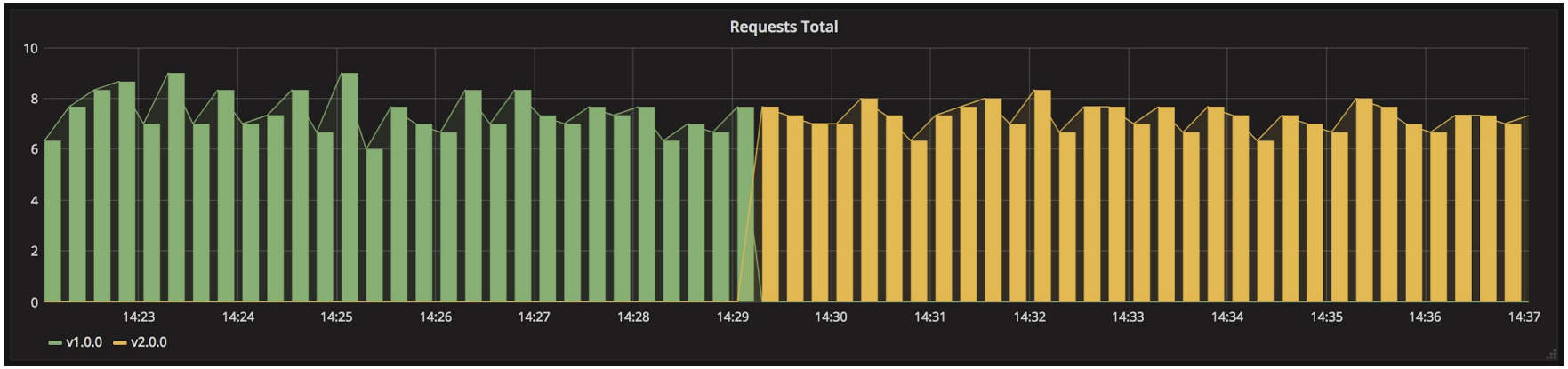
Blue/green deployment

- Поднимаем новую версию
- Проверяем ее
- Переключаем трафик в новую версию



Плюсы и минусы:

- \+ нет даунтайма
- \+ API **без обратной совместимости можно** раскатить
- \- требуется в 2 раза больше ресурсов
- **? Может ли переключение быть действительно моментальным**



Feature toggles

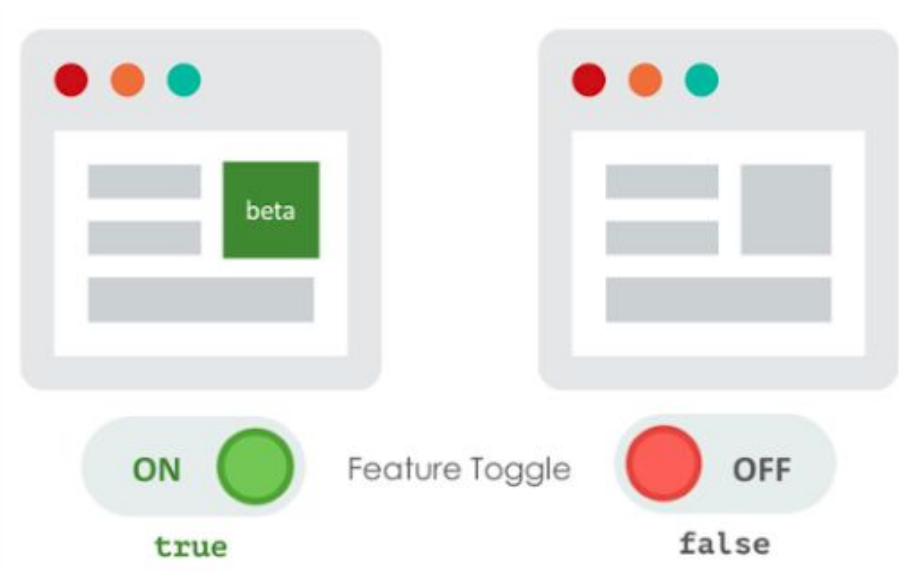
- Выкатываем фичу, но не включаем ее
- После выкладки включаем фичу на части пользователей
- После тестирования раскатываем на всех

Плюсы:

- Добавляем отдельную стадию бизнесового процесса раскатки фичи и продуктового тестирования
- Можем быстро выключить фичу, если она оказалась нерабочей

Минусы:

- Необходимо разработать данный механизм для своего приложения



Traffic mirroring

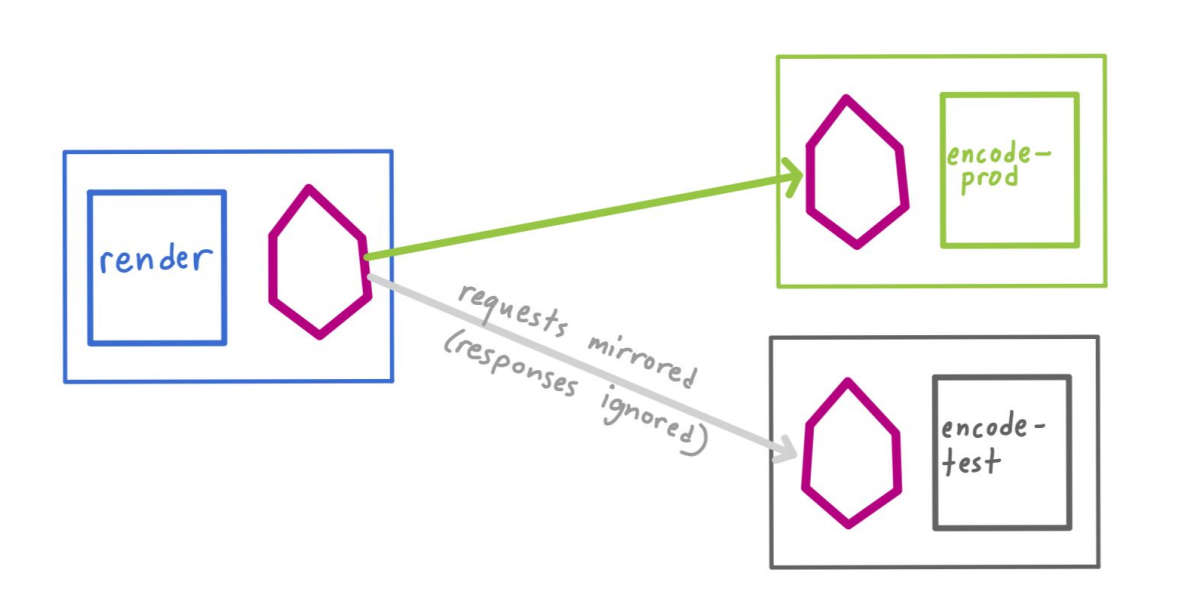
- Выкатываем новую версию, с отключенными запросами в продуктовую базу/очередь и так далее
- Пускаем любое количество трафика в % от продакшен на эту версию
- Убеждаемся что всё работает хорошо

Плюсы:

- Нагрузочное тестирование на реальном трафике с продакшена

Минусы:

- Нужно убедиться что миррорринг безопасен с точки зрения приложения

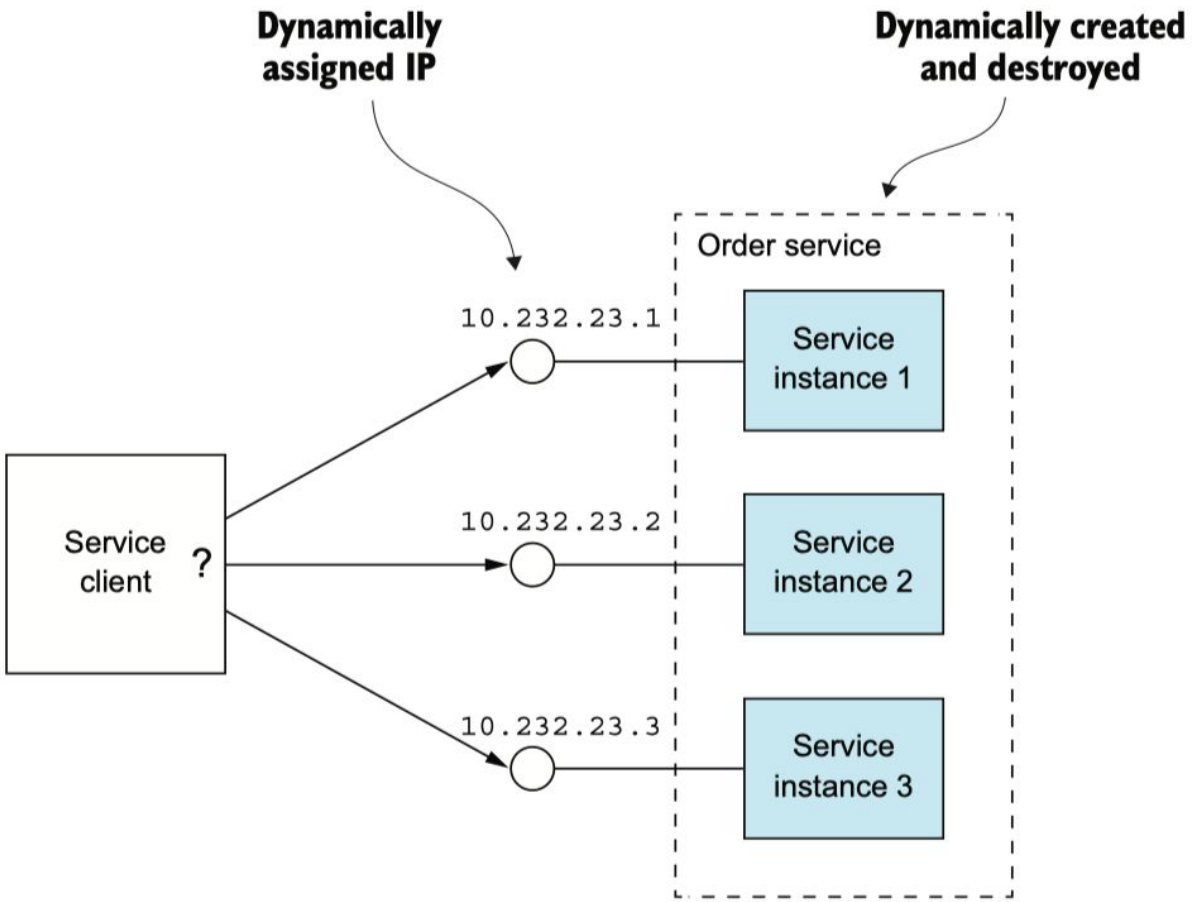


04

Service Discovery

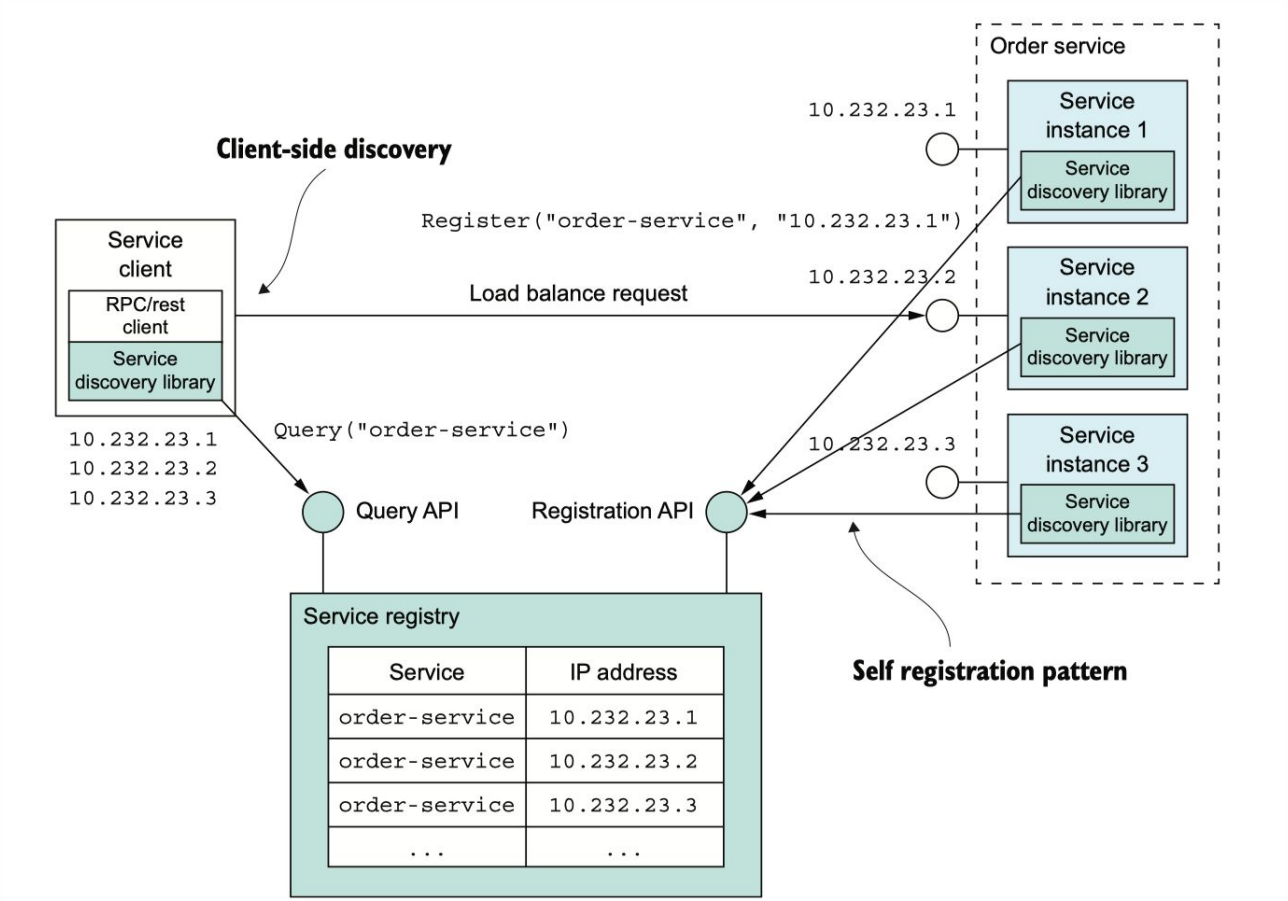
Service discovery

Как клиенту понять, где находится инстанс сервиса?



Client-side discovery

- Клиент сам ходит в реест сервисов, получает оттуда данные
- Сервисы сами себя регистрируют в этом реестре



Eureka

- https://github.com/Netflix/eureka

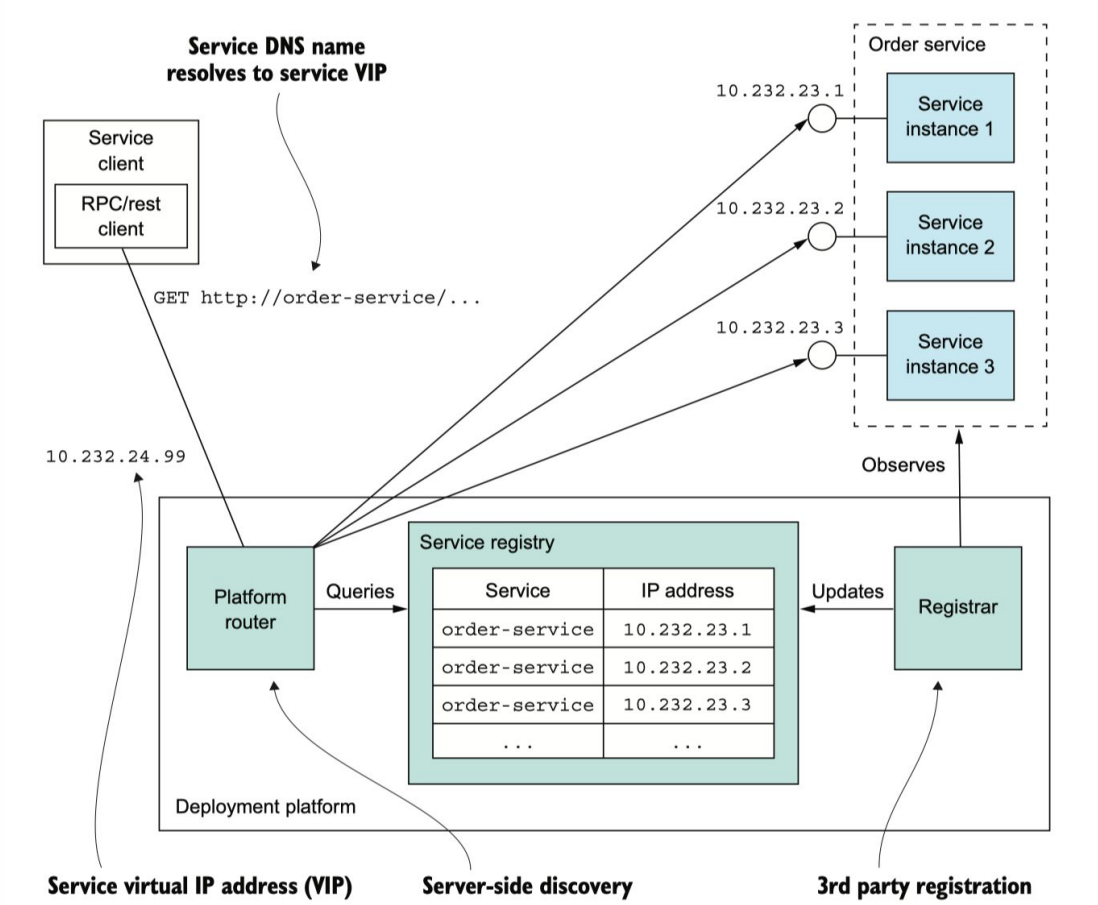
```
@Autowired
private EurekaClient eurekaClient;

public void doRequest() {
    Application application
      = eurekaClient.getApplication("spring-cloud-eureka-client");
    InstanceInfo instanceInfo = application.getInstances().get(0);
    String hostname = instanceInfo.getHostName();
    int port = instanceInfo.getPort();
   //...
}
```

Client-side discovery

- Работает с несколькими системами оркестрации одновременно: k8s, standalone-сервисы, nomad и т.д.
- Зависит от поддержки языка программирования и фреймворка сервисов

Server-side discovery

- Оркестратор регистрирует сервис в реестре
- При обращении клиент использует service name или ip
- При обращении на этот ip роутер заглядывает в таблицу маршрутизации и перенаправляет запрос (осуществляет LoadBalancing)



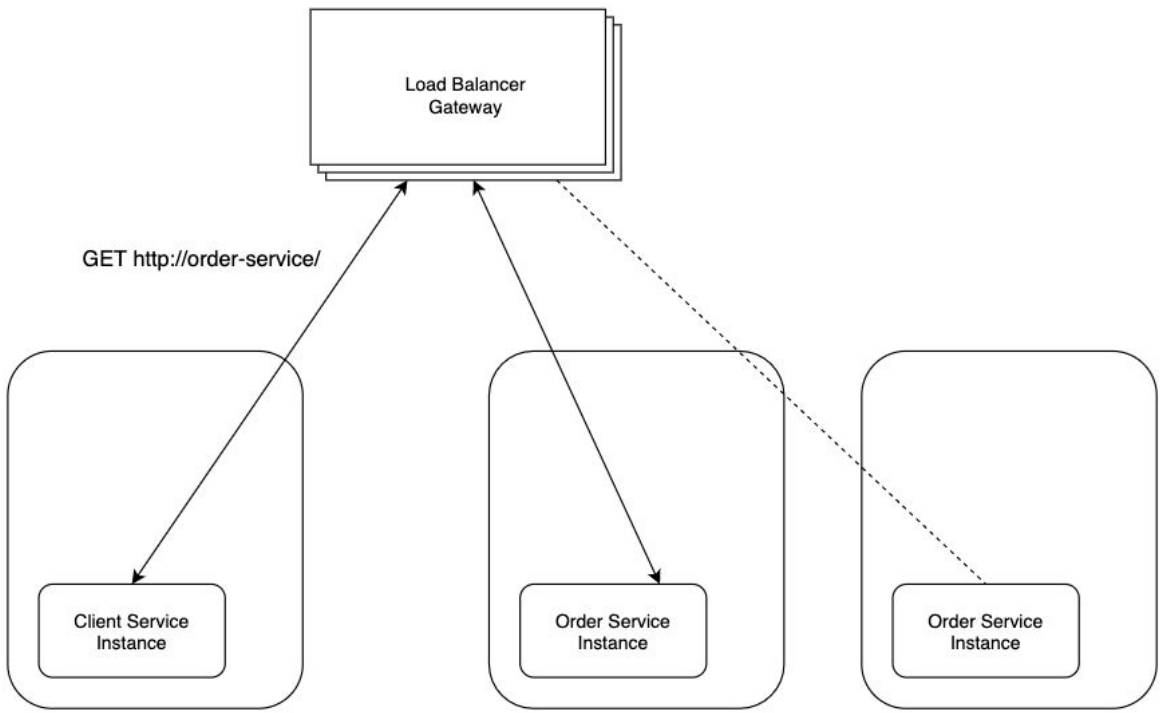
Паттерны server side service discovery

- API Gateway или Балансер
- DaemonSet
- Сетевой роутинг

API Gateway или Балансер

Сервисы общаются друг с другом не напрямую, а через GW.
Конфигурация роутинга находится в GW и обновляется с помощью средств автоматизации или руками

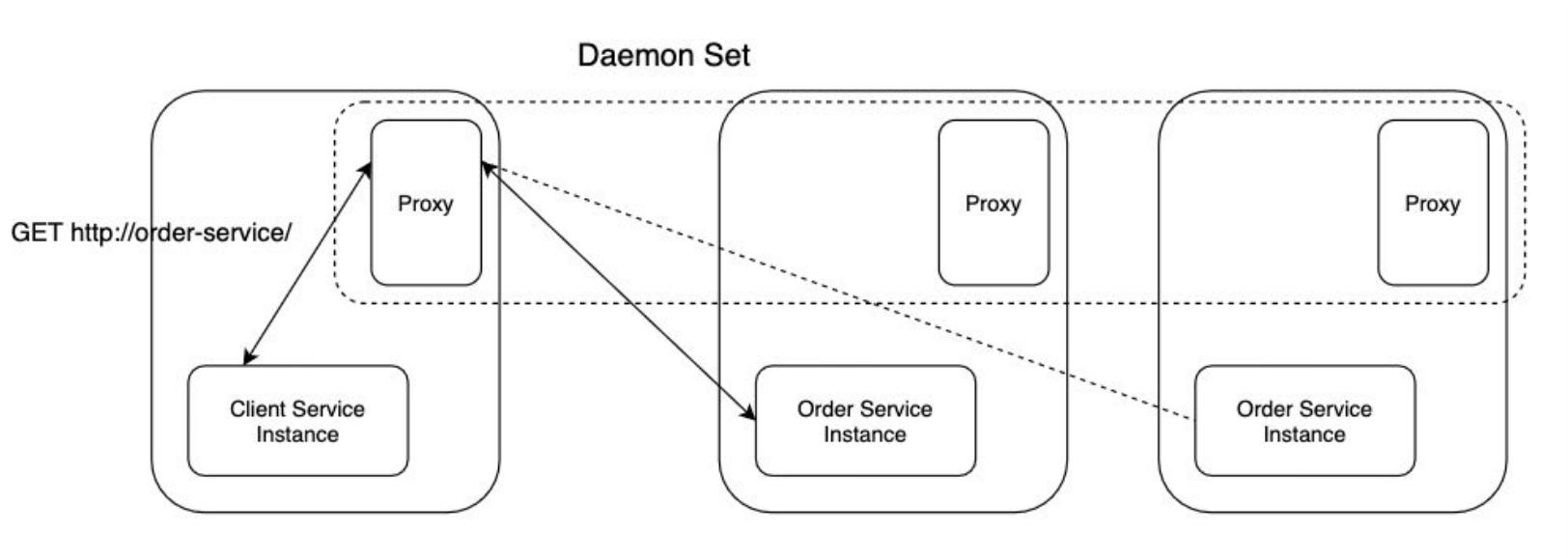
- Простая схема
- Хорошо работает, если сервисов не очень много
- Добавляет два хопа в межсервисное взаимодействие



DaemonSet

Сервисы общаются друг с другом не напрямую, а через GW или proxy, которое установлено на каждой из нод

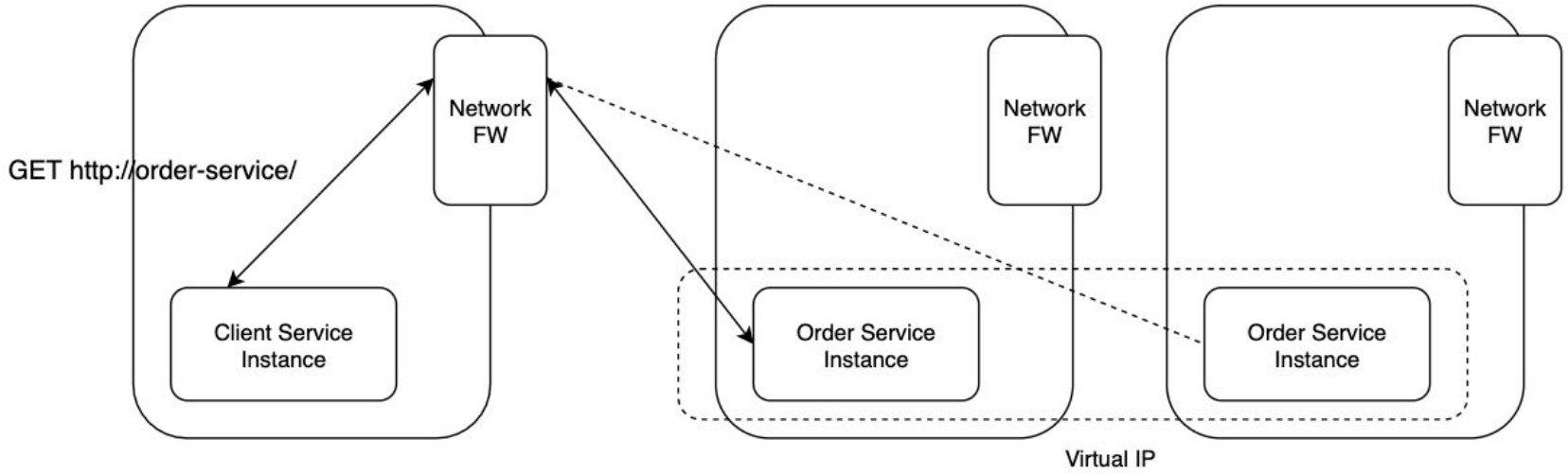
В такой схеме потенциально уменьшается latency, но усложняется поддержка конфигурации. Без автоматики синхронизировать конфиги на всем кластере довольно-таки тяжело.



Сетевой роутинг

Сервисы общаются друг с другом не напрямую, а через сетевой фильтр, который настраивается на каждой ноде.

В такой схеме потенциально увеличивается производительность за счет работы в kernel-mode. При этом сильно снижается гибкость и возможности для retry-ев бекендов инстанса, если он не отвечает.



05

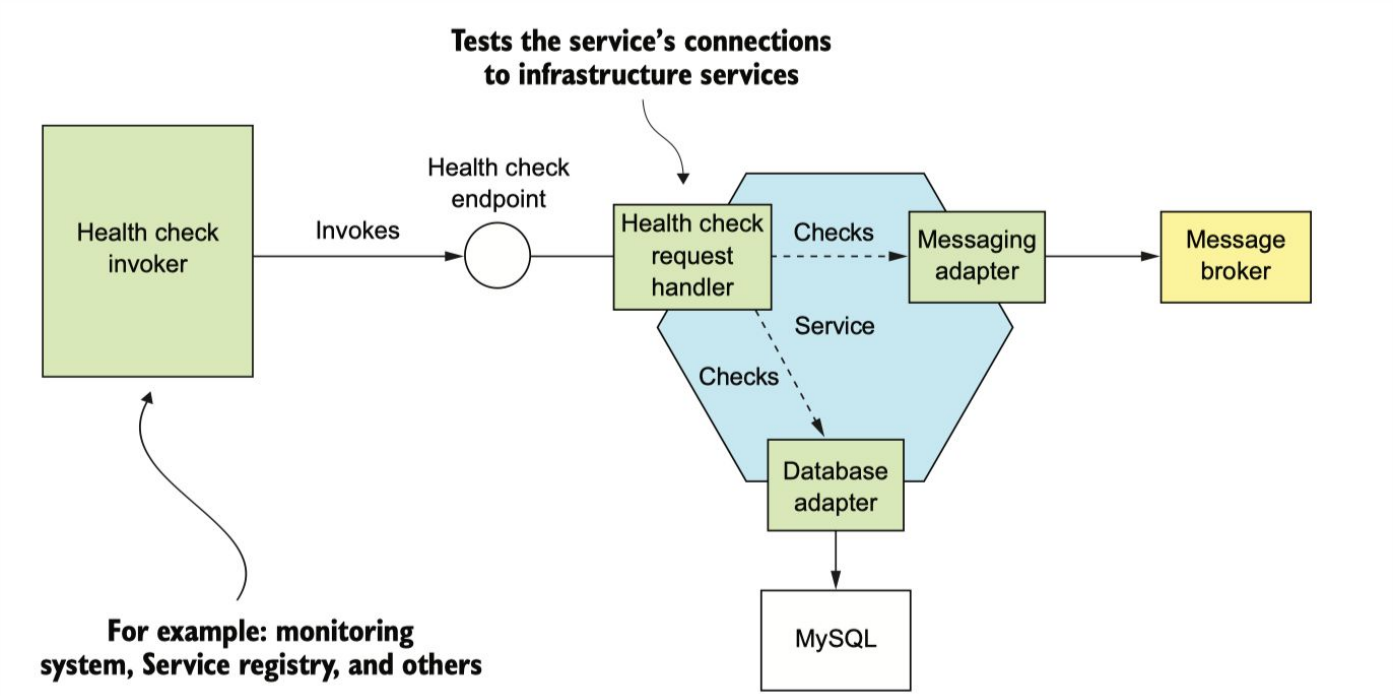
Health check

Активный health check

Чтобы проверять, умерло приложение (под) или нет заводится специальный метод (endpoint), по которому проверяется общая живость приложения.

- Liveness probe – приложение живо
- Readiness probe – приложение готов принимать трафик

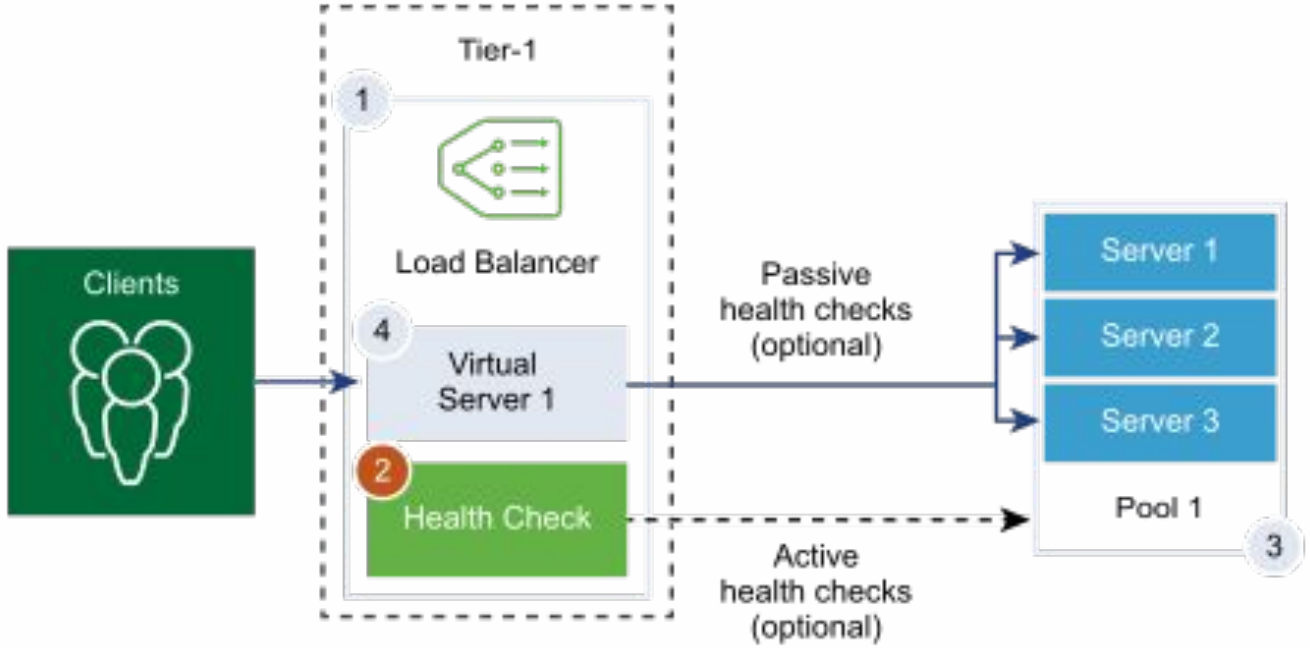
За жизнью под смотрит система мониторинга и алертинга, service registry, оркестратор, чтобы снимать трафик с больных под



Пассивный health check

Если балансер видит, что бекенд «плохо» отвечает (5XX http status) или не отвечает (timeout), он на время может выключить его из балансинга.

Пример:
https://docs.nginx.com/nginx/admin-guide/load-balancer/http-health-check/
- пример реализации health-check-ов в nginx для апстримов



https://habr.com/ru/company/flant/blog/470958/

Liveness & Readiness Probes в K8S

Очень аккуратно относится к Liveness & Readiness Probes.

- Если в пробе проверяется внешняя зависимость, то существует вероятность каскадного падения, если сеть к внешнему сервису лагнула, например
- Настраивать по необходимости
- Liveness Probe должна отличаться от Readiness Probe.

https://habr.com/ru/company/flant/blog/470958/

Спасибо
за внимание!
Пожалуйста пройдите опрос по ссылке