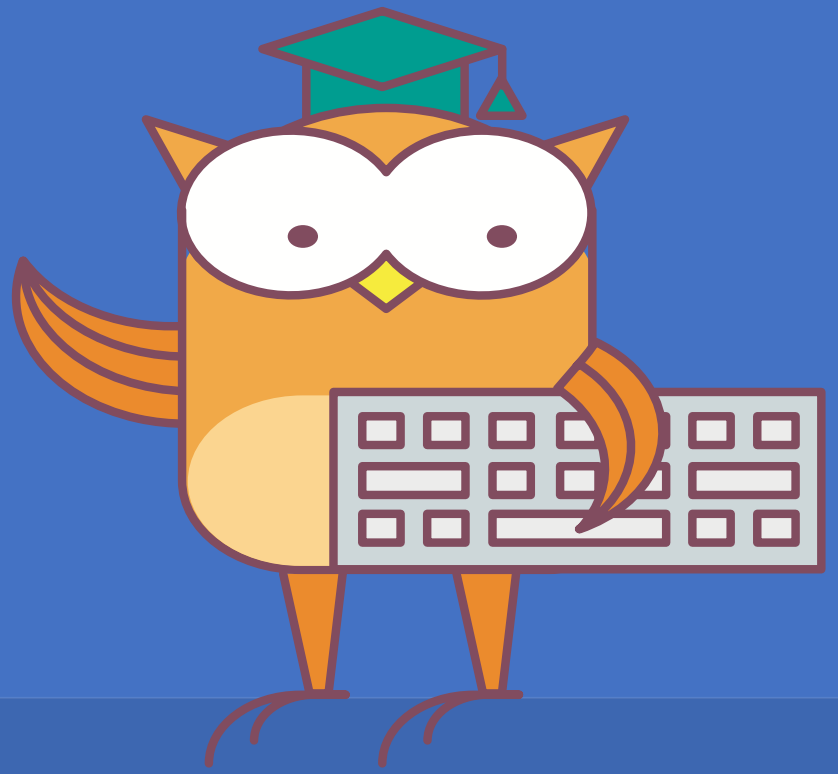
Инфраструктура микросервисов

Архитектор высоких нагрузок



Щетинников Стас

Меня хорошо слышно && видно?

Напишите в чат, если есть проблемы!

Ставьте + если все хорошо

Карта вебинара

- Зачем для микросервисов отдельная инфраструктура?
- Системы оркестрации
- App server vs virtual machine vs container
- Service discovery
- Распределенная трассировка
- Service mesh

Какие проблемы возникают в микросервисной архитектуре?

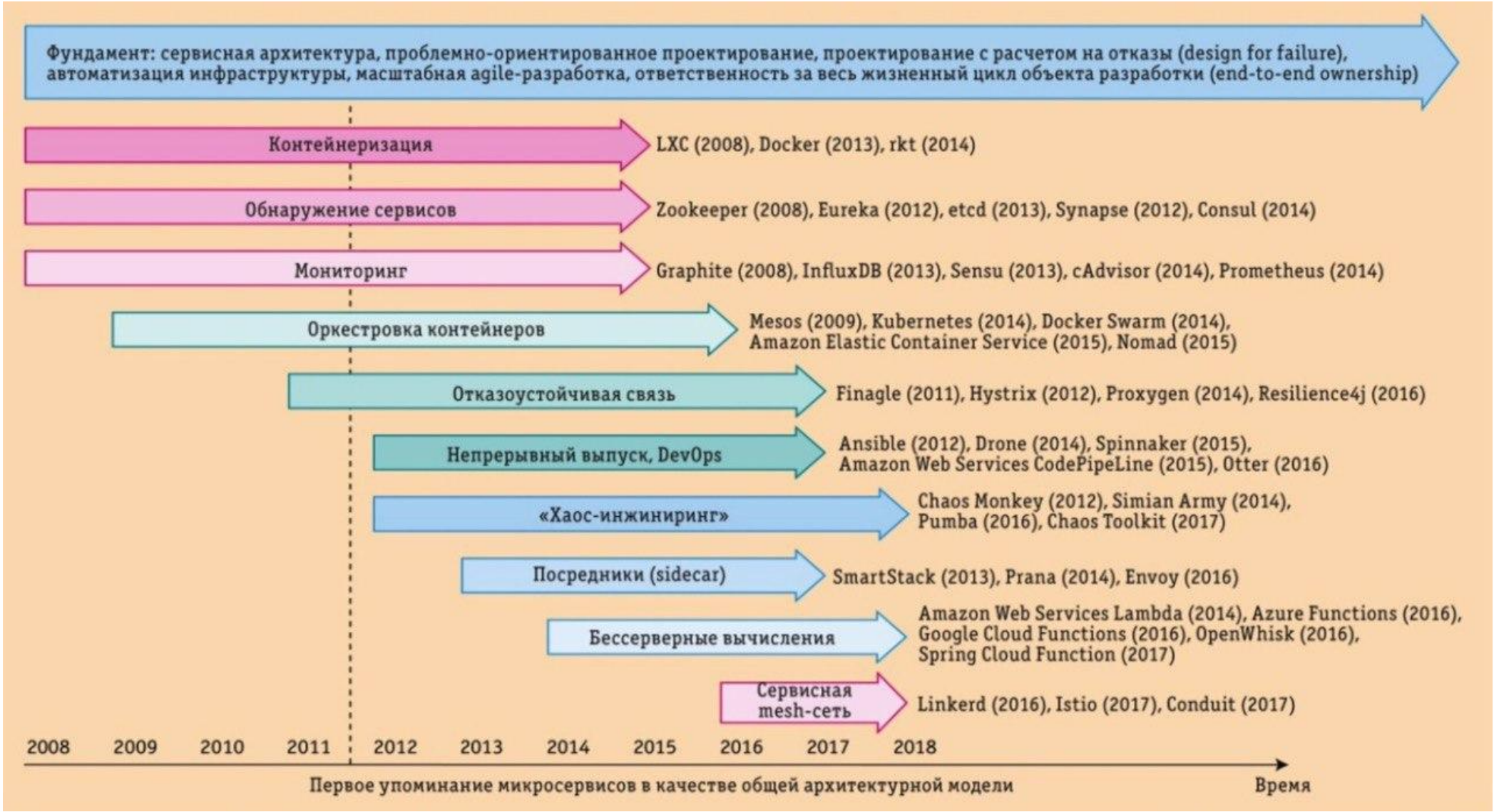
Когда сервисов много, управление ими становится сложнее. Издержки на управление растут экспоненциально с количеством сервисов.

Ручное управление перестает работать, нужны инструменты для решения проблем:

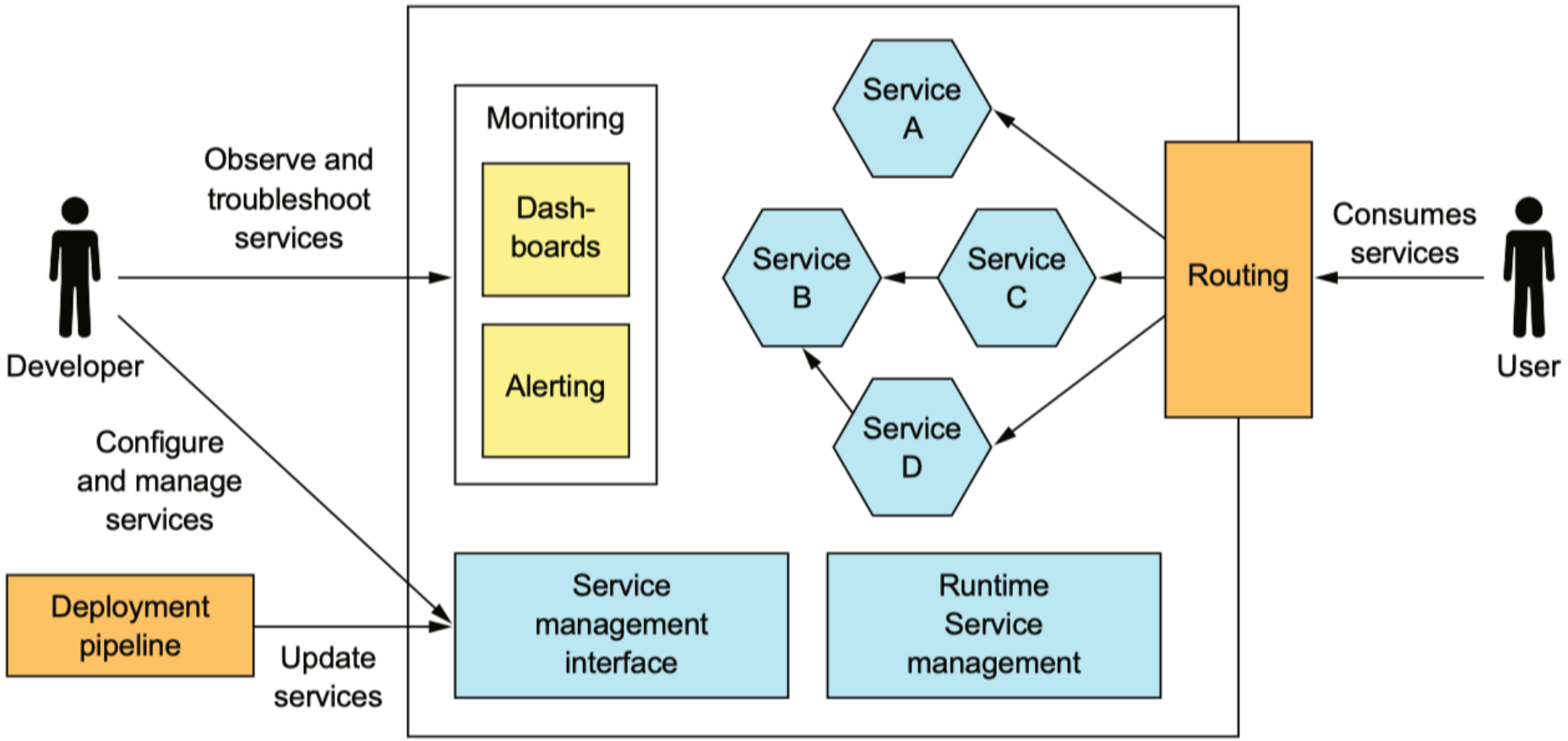
- Как на одной машине запустить процессы с разным окружением?
- Как изолировать сервисы, которые работают на одной машине?
- Как можно автоматически перекидывать сервисы в случае аварии на машине?
- Как обеспечить беспрерывную поставку?
- Как сделать обнаружение сервисов, если сервисы могут с одной ноды переезжать на другую?
- Как выбирать куда и сколько инстансов сервисов деплоить?

И т.д. и т.п.

Таймлайн инфраструктурных паттернов для микросервисной архитектуры



Основные функции инфраструктуры



Основные функции инфраструктуры

- **Автоматизированная сборка и деплой** (DevOps, CI/CD)
- **Интерфейс управления сервисами**

возможность сконфигурировать, создать или обновить сервис из командой строки, UI или API.

- **Управление runtime-ом сервисов**

изоляция сервисов, распределение по нодам, в случае падения – переподнятие, и т.д.

- **Мониторинг и алертинг**

отслеживание живости сервисов, алертинг в случае проблем, метрики

- **Балансировка и роутинг** запросов извне

Основные функции инфраструктуры

Чаще всего функции делятся по инструментам:

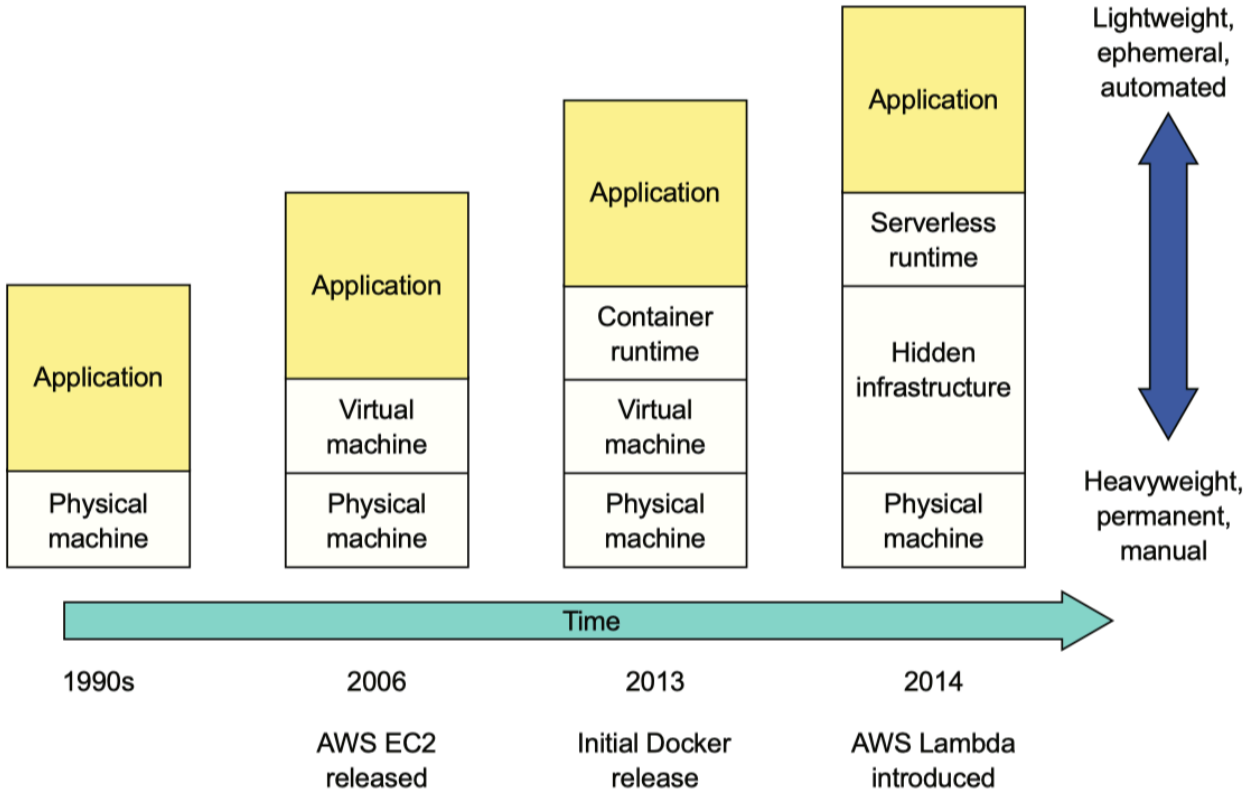
- Автоматизированная сборка и деплой – это инструмент CI/CD: Jenkins/TeamCity/GitlabCI

- Интерфейс управления сервисами, Управление runtime-ом сервисов, Мониторинг и алертинг, Балансировка и роутинг - это оркестратор+.

Примеры оркестраторов:
Kubernetes, Nomad+Consul, Mesos, Azure и т.д.

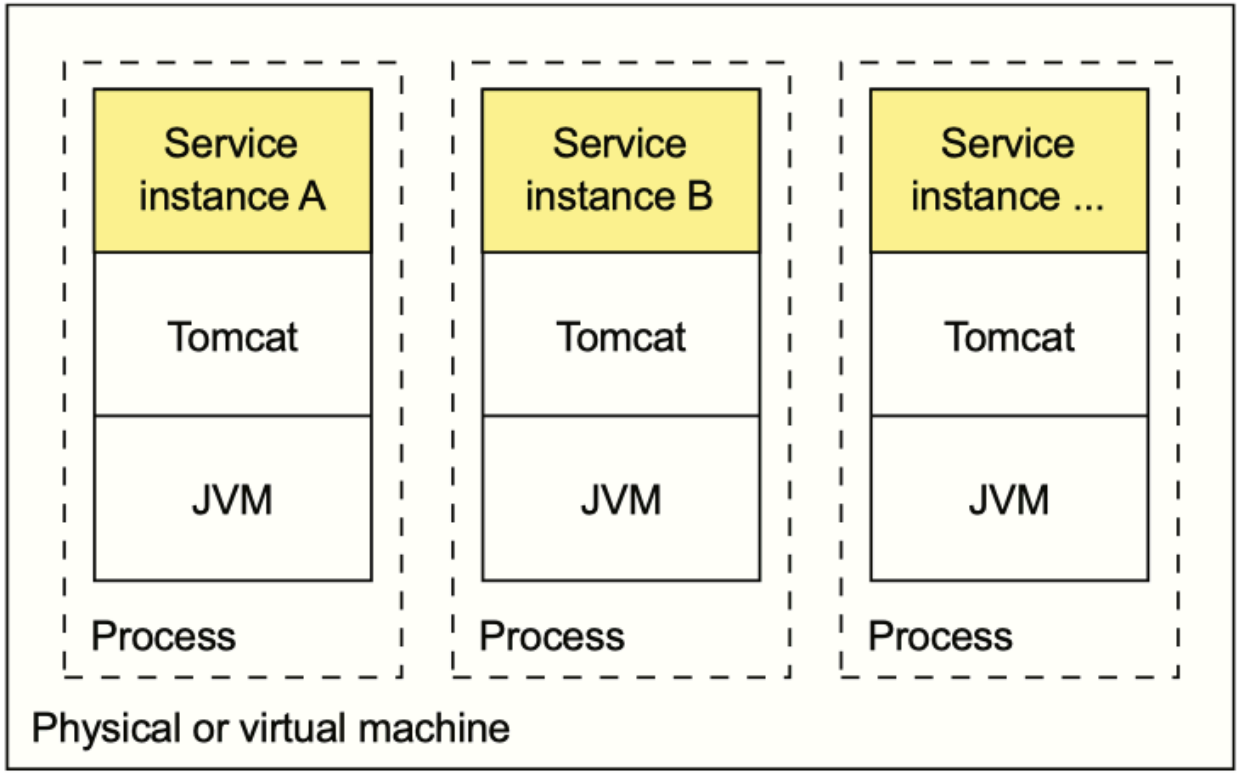
Как разместить несколько сервисов на одной машине?

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



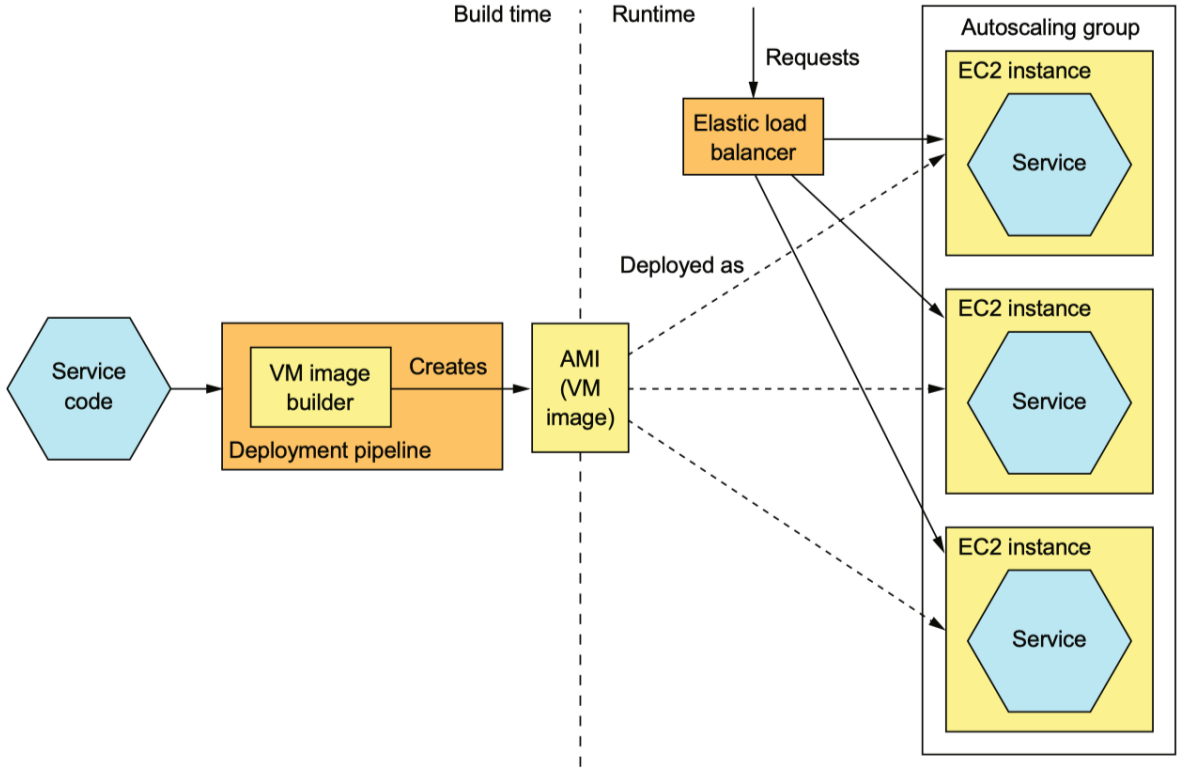
Сервер приложений

- Быстрый деплой
- Хорошая утилизация ресурсов
- Отсутствие изоляции по ресурсам между разными сервисами – CPU, memory
- Фиксированный язык программирования или фреймворк



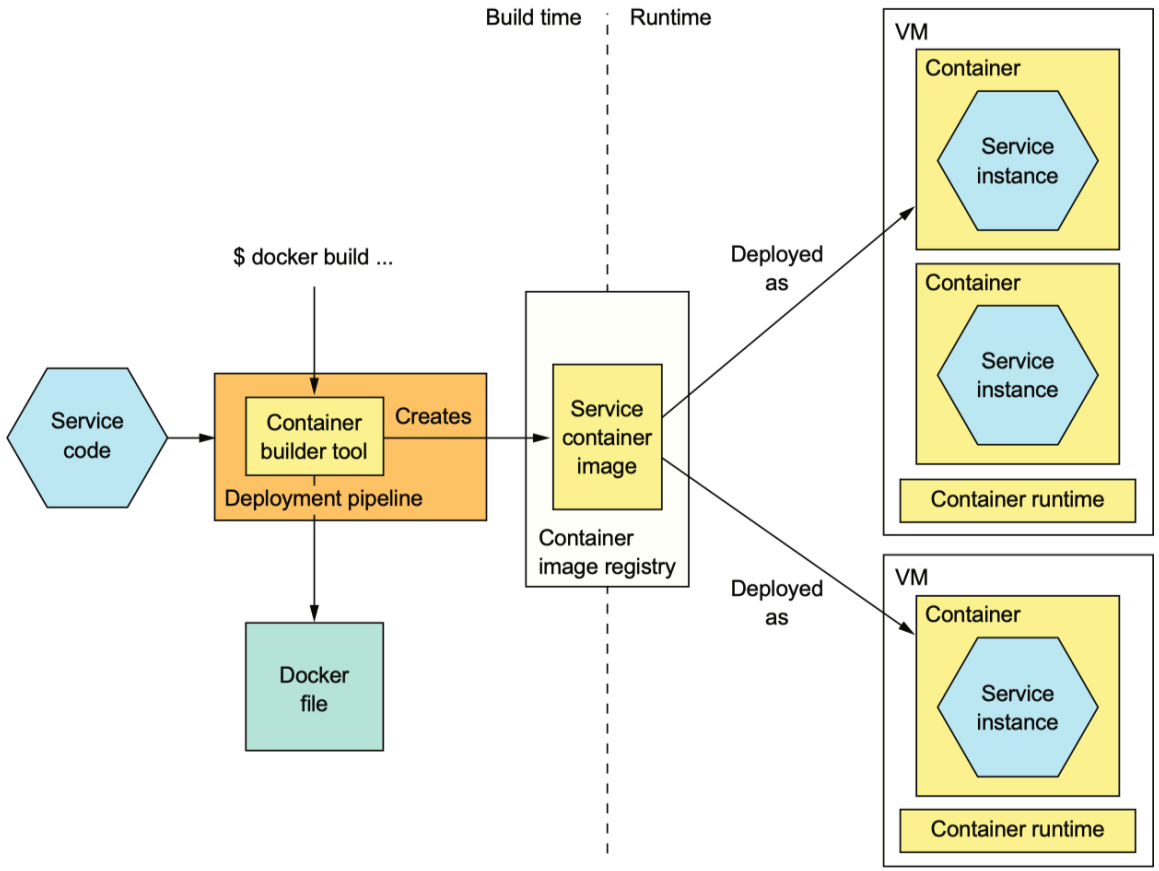
Виртуальная машина

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



Контейнеры

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



Overhead контейнеров

https://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf – исследование сравнения производительности Docker контейнеров.

- Оверхед в основном по сети (NAT)
- По диску и CPU docker container практически идентичен нативному исполнению

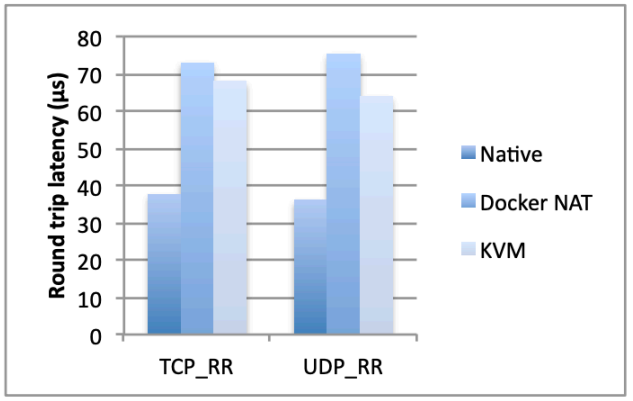


Fig. 3. Network round-trip latency (μs).

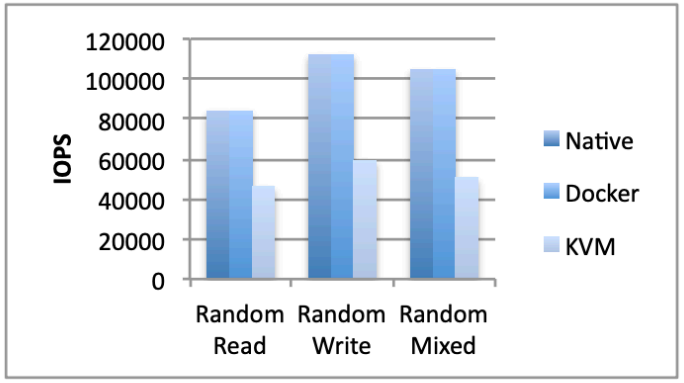


Fig. 6. Random I/O throughput (IOPS).

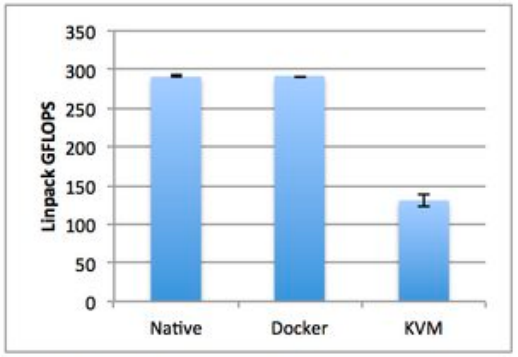
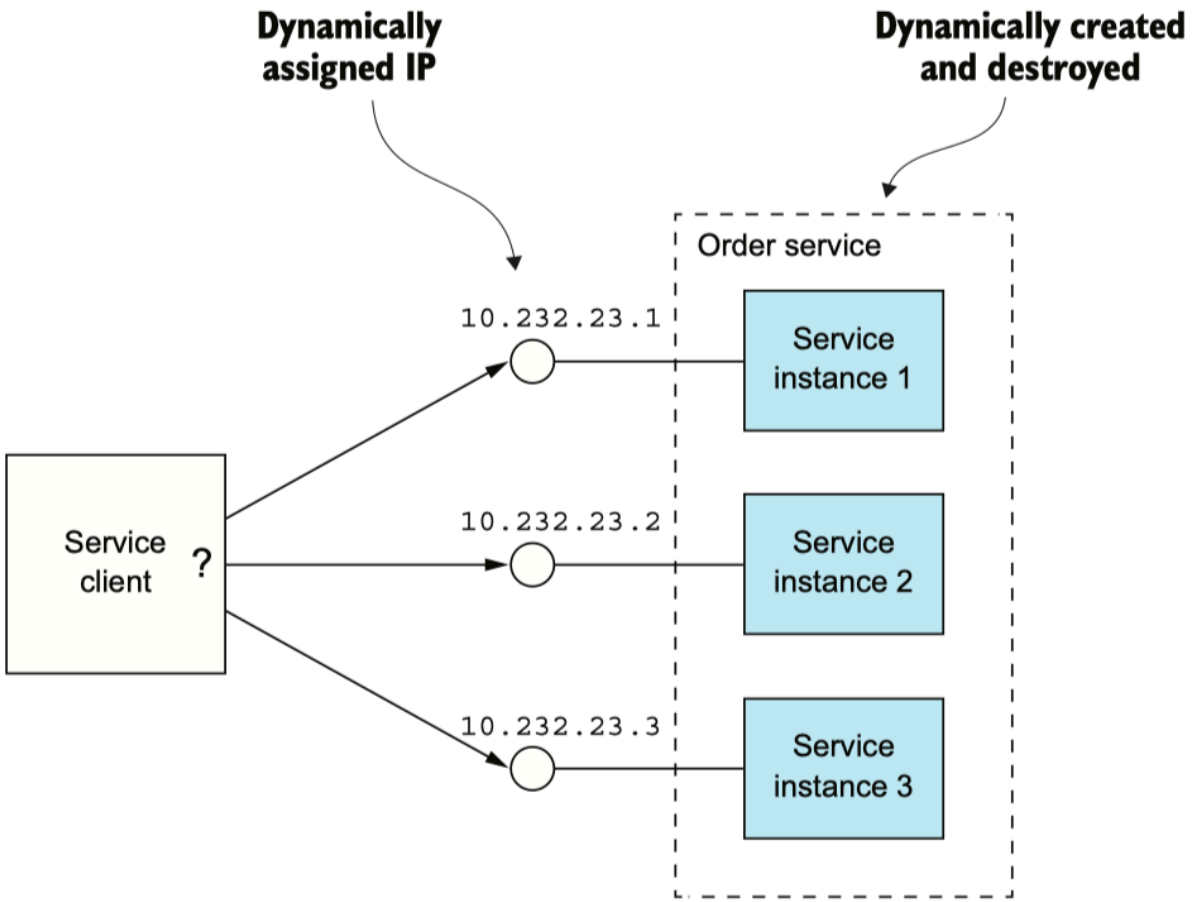


Figure 1. Linpack performance on two sockets (16 cores). Each data point is the arithmetic mean obtained from ten runs. Error bars indicate the standard deviation obtained over all runs.

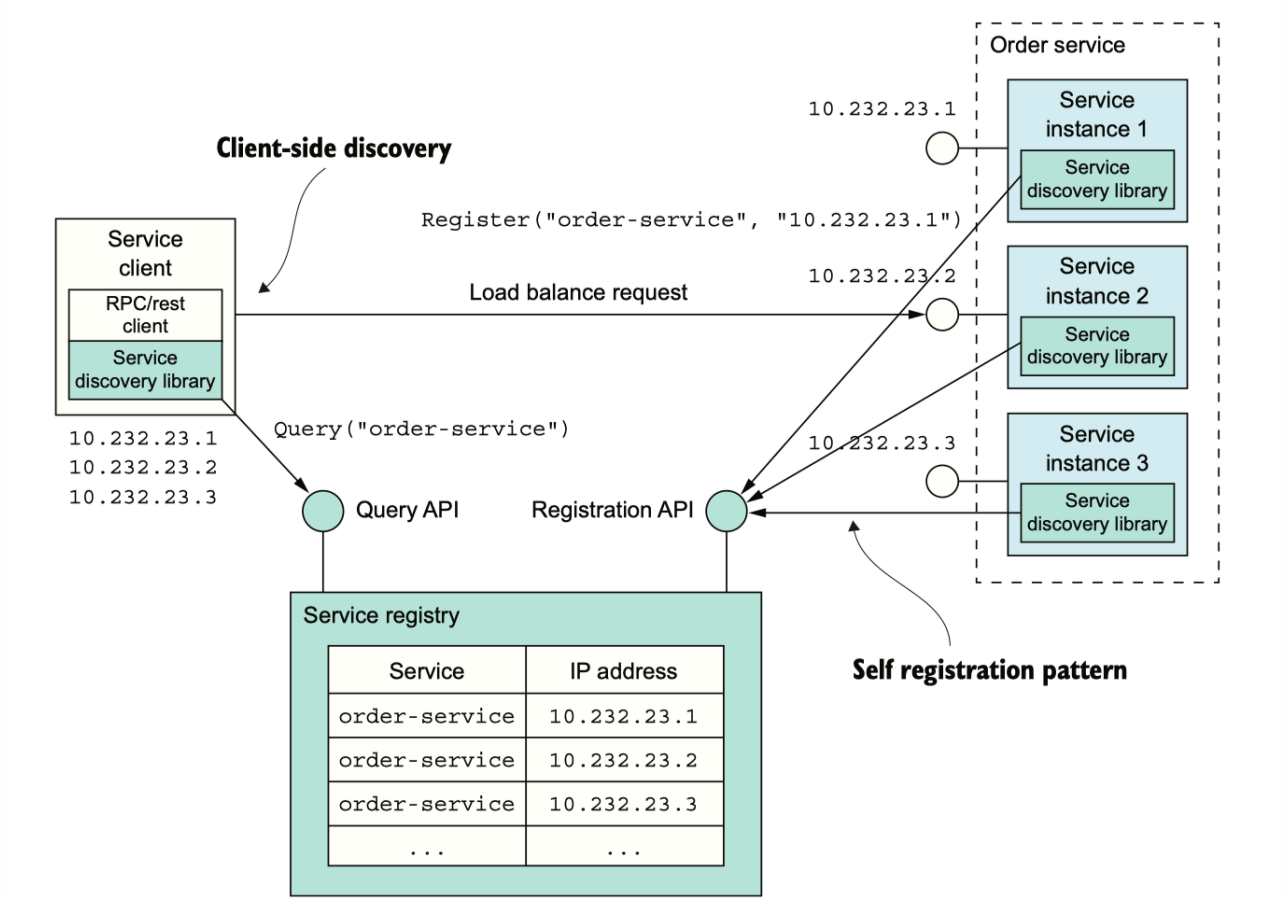
Service discovery

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

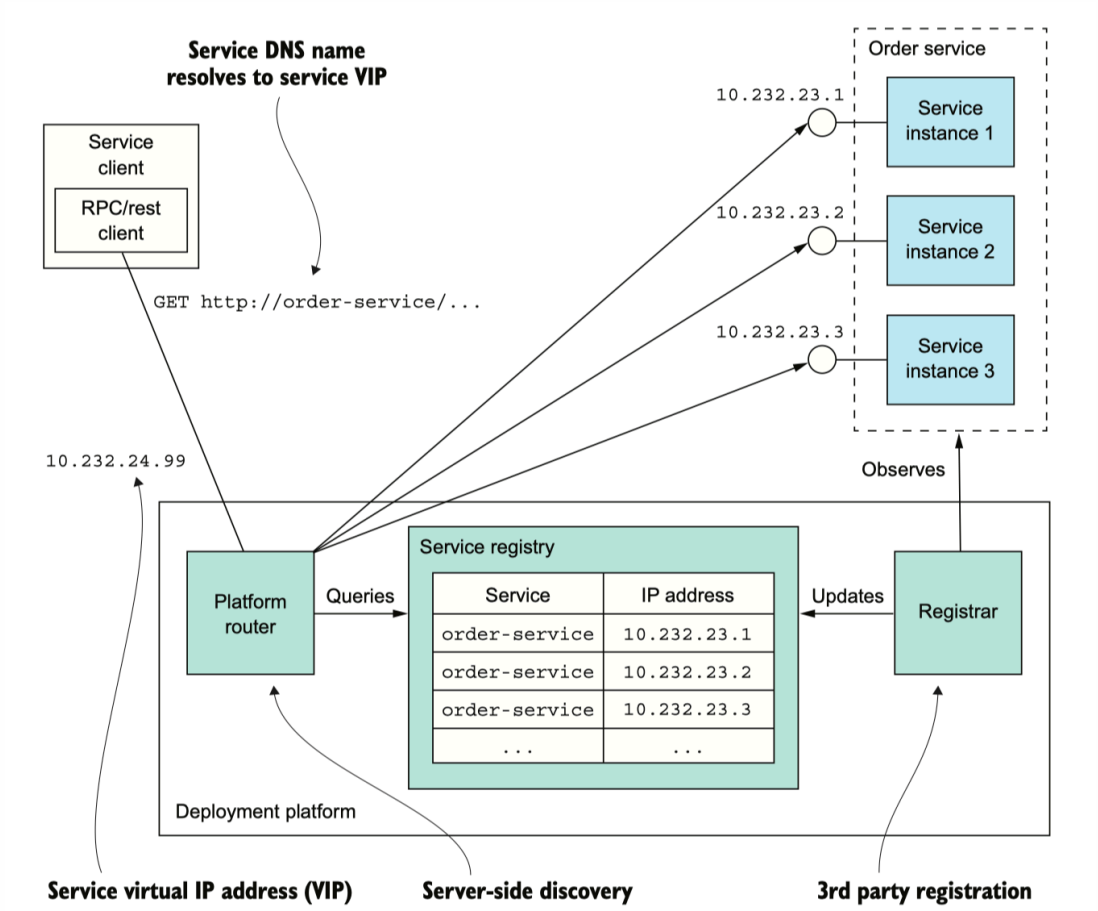
```
@Autowired
private EurekaClient eurekaClient;

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

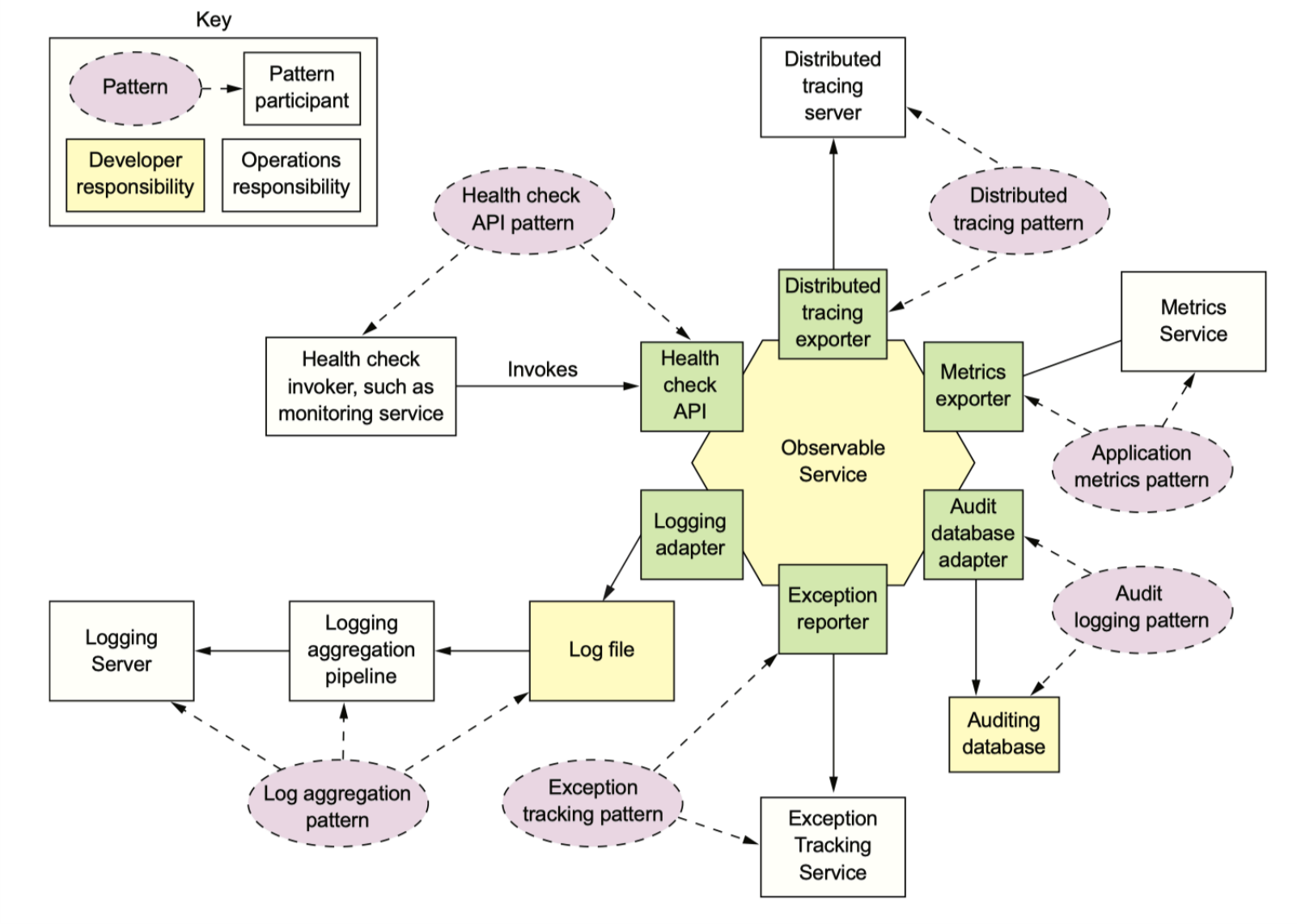
- Работает с несколькими системами оркестрации одновременно: k8s, standalone-сервисы, nomad и т.д.
- Зависит от поддержки языка программирования и фреймворка сервисов

Server-side discovery

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



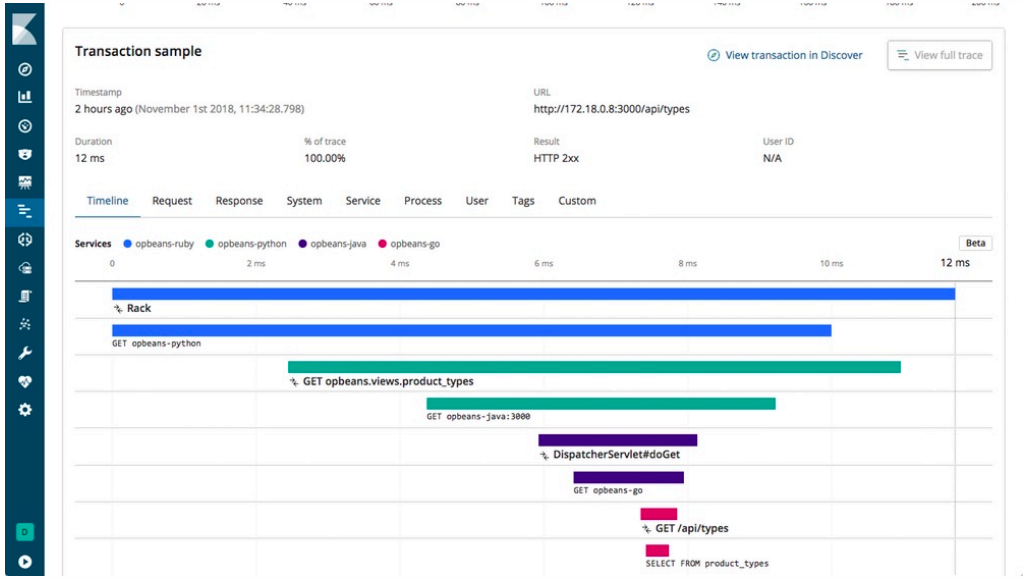
Observability



Distrubuted tracing

Распределенная транзакция – это путь прохождения запроса по разным сервисам.

- При трассировке, к каждому запросу добавляются метаданные о контексте этого запроса и эти метаданные сохраняются и передаются между компонентами, участвующими в обработке запроса
- В различных точках трассировки происходит сбор и запись событий вместе с дополнительной информацией (URL-запроса, идентификатор клиента, код запроса к БД)
- Информация о событиях сохраняется со всеми метаданными и контекстом и явным указанием причинно-следственных связей между событиями



Для чего используется tracing?

- Упрощенное взаимодействие между командами - при регрессах можно скинуть TraceID, связать систему трэкинга ошибок с трейсами
- Оценка критического пути выполнения запроса и влияния разных факторов на время выполнения (сетевые проблемы, медленные запросы к БД)
- Графы зависимостей - с кем взаимодействует мой сервис, кого затронут изменения в нем?

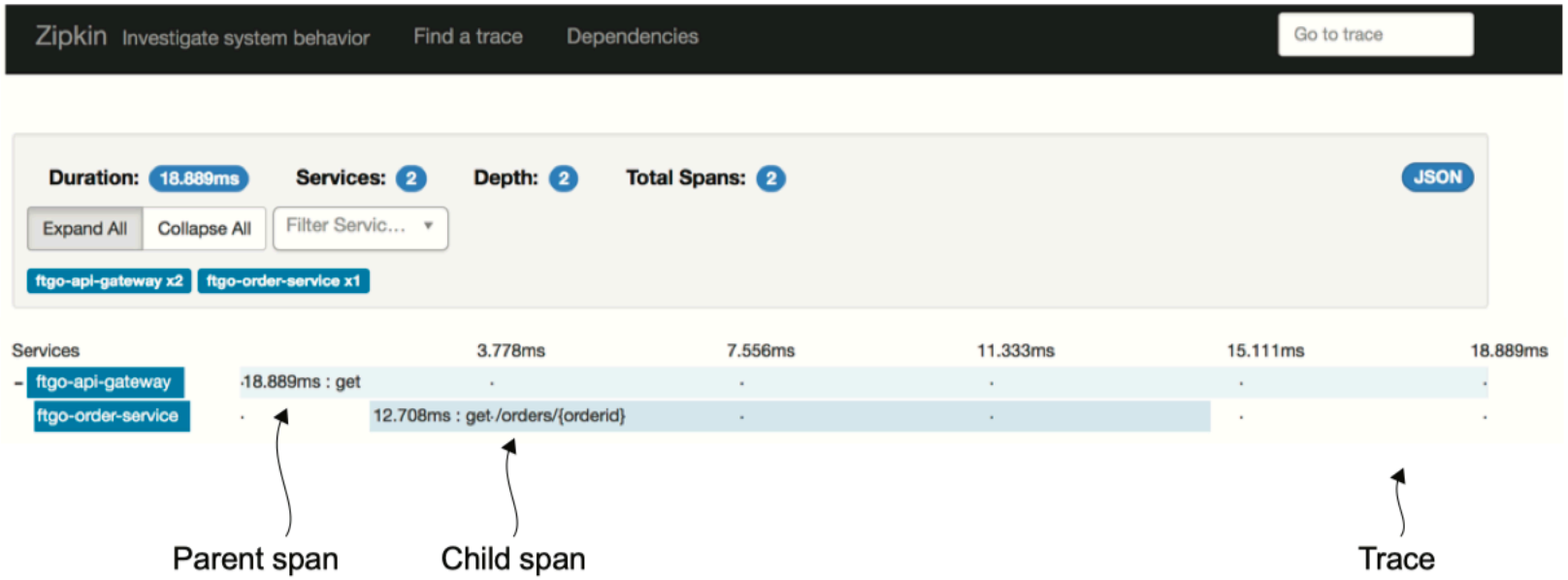
Основая терминология

Span - запись об одной логической операции по обработке запроса (тайминги и метаданные).

- Каждый спан обязательно содержит ссылку на Trace-ID
- Каждый спан содержит свой уникальный идентификатор Span-ID

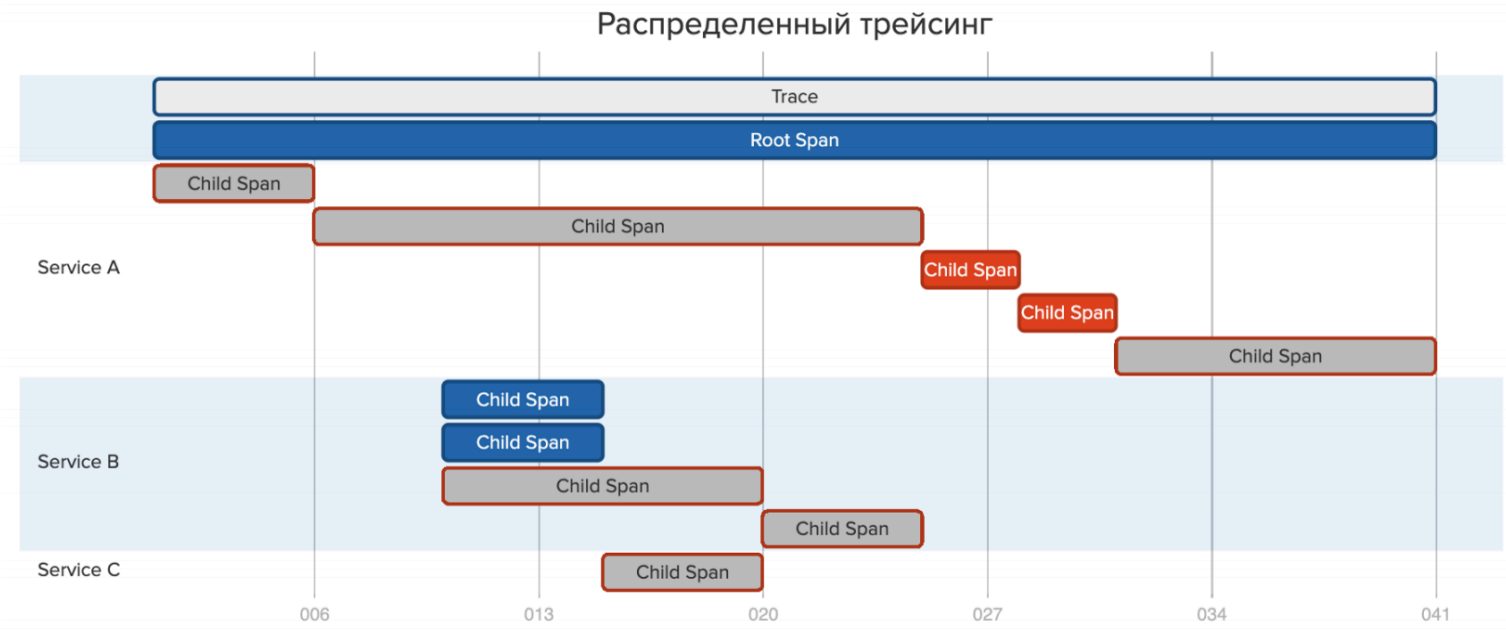
Trace - коллекция связанных записей (Spans), описывающая обработку одного запроса (end-to-end)

- каждый трейс имеет свой уникальный идентификатор - Trace ID



Основая терминология

- **Root Span** - это спан, у которого нет ссылки на родительский спан (только Trace ID), он показывает общую длительность выполнения запроса



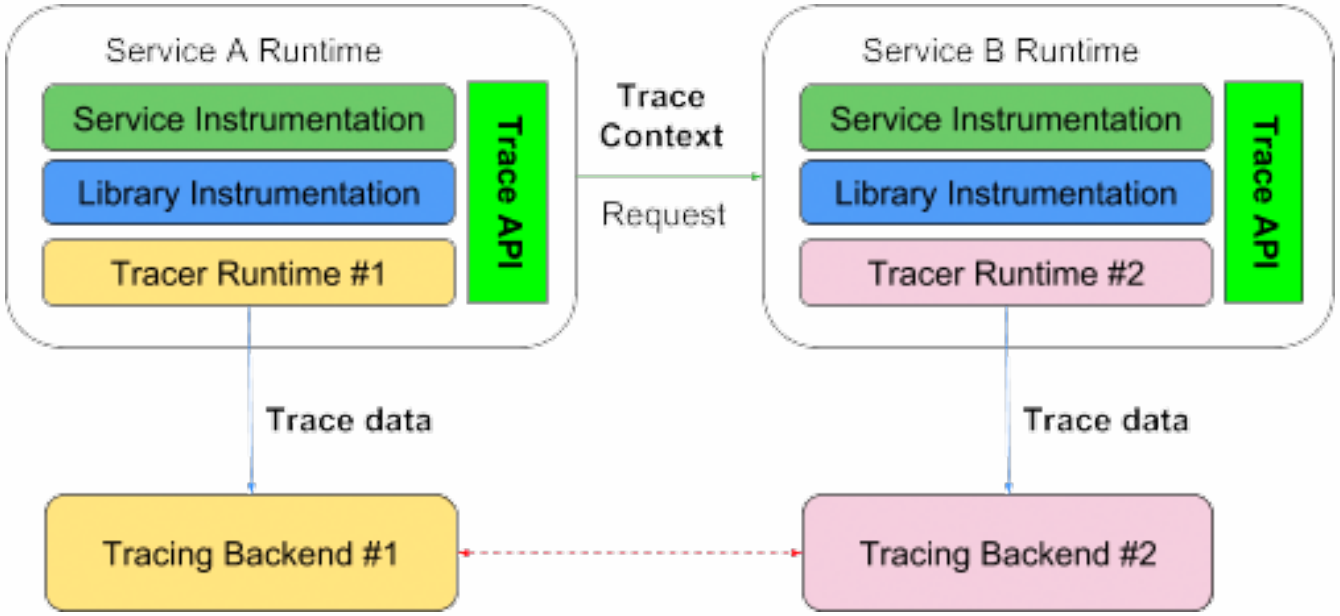
Какие есть проблемы

- Не видны проблемы общей инфраструктуры (состояние очередей, IOPS и т.п.), "серые ошибки" в облаках
- В трассировках нет "низкоуровневых" данных - состояние ОС, ядра и т.п., то что добывается strace, ss и прочим
- Для протоколов, где нет метаданных, надо писать свои обвязки и прокидывать
- Надо выбирать нагрузку и частоту сэмплирования

Инструментарий для трейсинга

Обычно разработчик сталкивается со следующими вопросами:

- Как систему для трейсинга использовать?
- А что если я захочу поменять трейсер? Мне придется переписывать весь код?
- А что будет с библиотеками, которые используют используют разные трейсеры?



Opentracing

Opentracing – это спецификация на то, как должен выглядеть класс Tracer в разных языках программирования, чтобы его можно было поменять, не меняя код сервиса. https://opentracing.io/specification/

```
import io.opentracing.Scope;
import io.opentracing.Tracer;
import io.jaegertracing.Configuration;
import io.jaegertracing.internal.JaegerTracer;
...
private void sayHello(String helloTo) {
    Configuration config = ...
    Tracer tracer = config.getTracer();
    try (Scope scope = tracer.buildSpan("say-hello").startActive(true)) {
        scope.span().setTag("hello-to", helloTo);
    }
    ...
}
```

Пример, Opentracing

Например, мы можем поменять только реализацию tracer-а, а остальной код не меняем

```
import io.opentracing.Scope;
import io.opentracing.Tracer;
import io.jaegertracing.Configuration;
import io.jaegertracing.internal.JaegerTracer;
...
private void sayHello(String helloTo) {
    Configuration config = ...
    Tracer tracer = config.getTracer();
    try (Scope scope = tracer.buildSpan("say-hello").startActive(true)) {
        scope.span().setTag("hello-to", helloTo);
    }
    ...
}
```

```
import io.opentracing.Scope;
import io.opentracing.Tracer;
import co.elastic.apm.opentracing.ElasticApmTracer;
...
private void sayHello(String helloTo) {
    Tracer tracer = new ElasticApmTracer();
    try (Scope scope = tracer.buildSpan("say-hello").startActive(true)) {
        scope.span().setTag("hello-to", helloTo);
    }
    ...
}
```

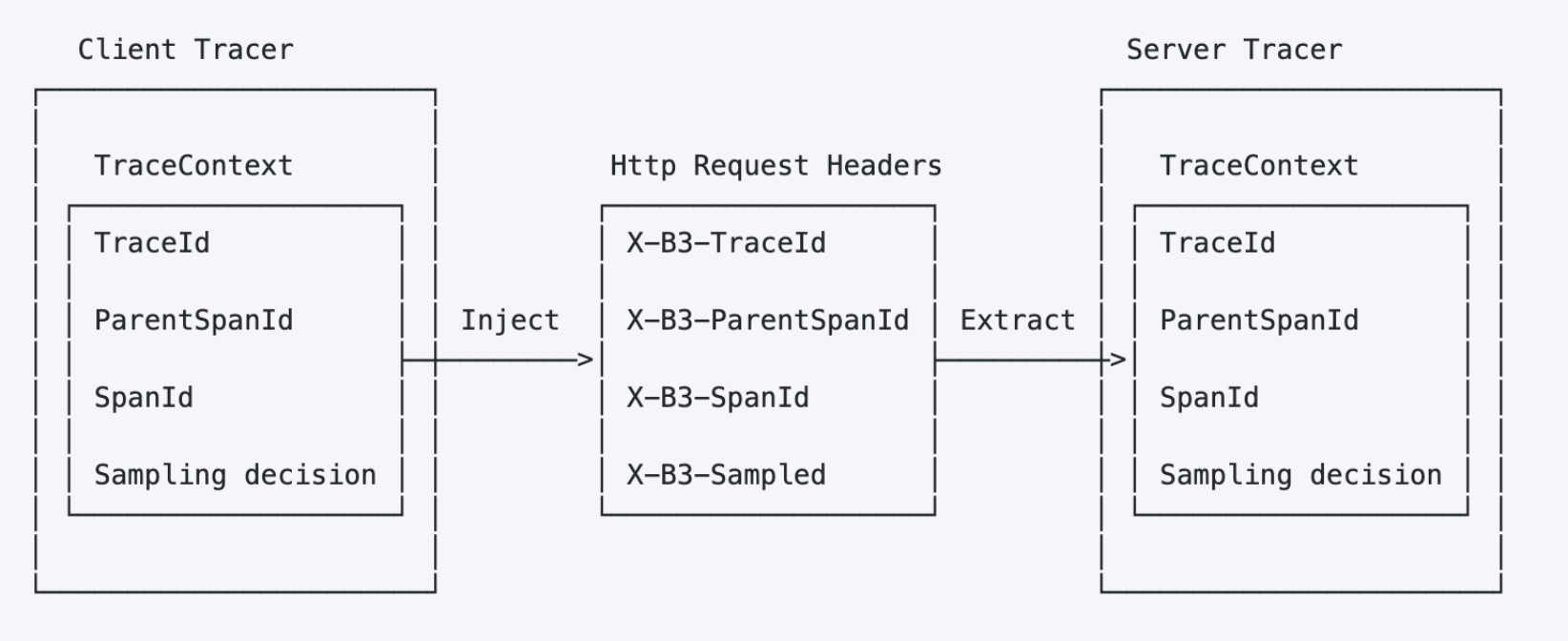
Context Trace

Context trace - каким образом мы прокидываем контекст трейса (trace-id, span-id, данные и т.д) от одного сервиса в другой.

Для HTTP есть черновик W3C Context Trace - https://www.w3.org/TR/trace-context-1/

Есть также свои реализации:
- Zipkin B3 протокол (https://github.com/openzipkin/b3-propagation)
- Uber протокол

И т.д.



Инструменты для Tracing

- **Zipkin** (https://zipkin.io/)

Много клиентских библиотек, свой протокол B3, который многими сторонними трейсерами поддерживается

- **Jaeger** (https://www.jaegertracing.io/)

Много клиентских библиотек. Активно развивается, бежит в сторону поддержки OpenTracing. Пока w3c context trace не поддерживают, но поддерживает B3

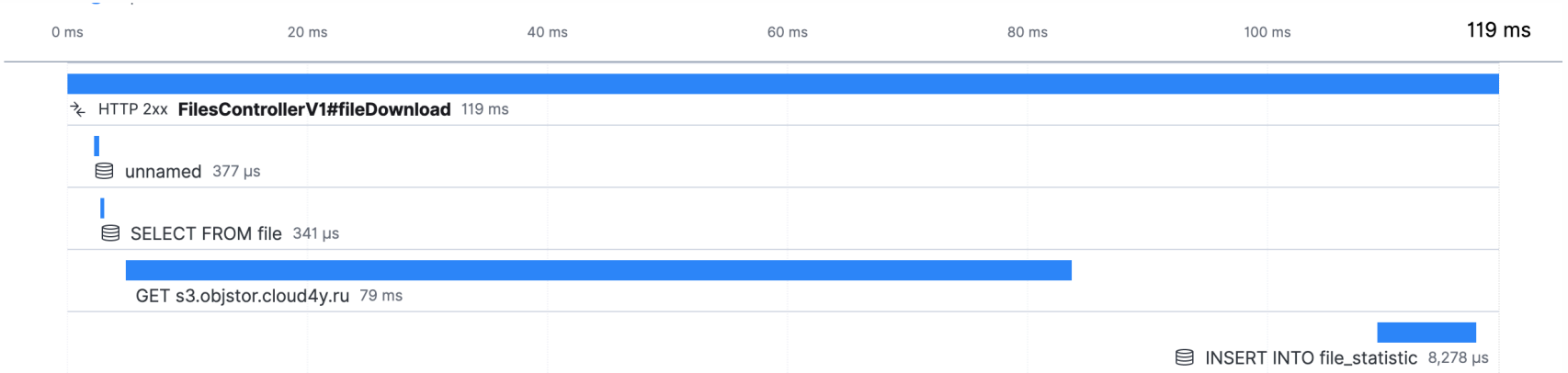
Различные APM:

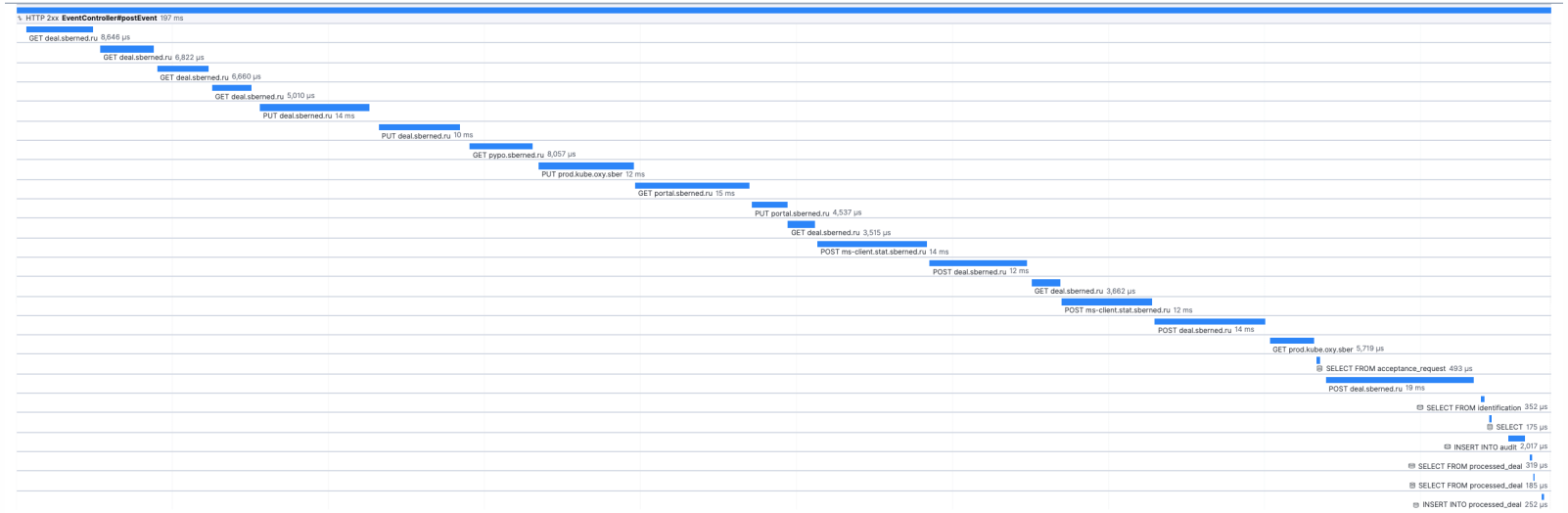
- Elastic APM, LightStep, DynaTrace, DataDog, Instana все бегут в сторону поддержки Opentracing-а и W3C Context Trace

- **OpenTelemetry** – дальнейшее развитие подхода OpenTracing с добавлением метрик Openmetrics протокол и логов. Пока в глубокой альфе.

Elastic APM

Elastic APM – средства для анализа производительности приложений с tracing-ом и метриками

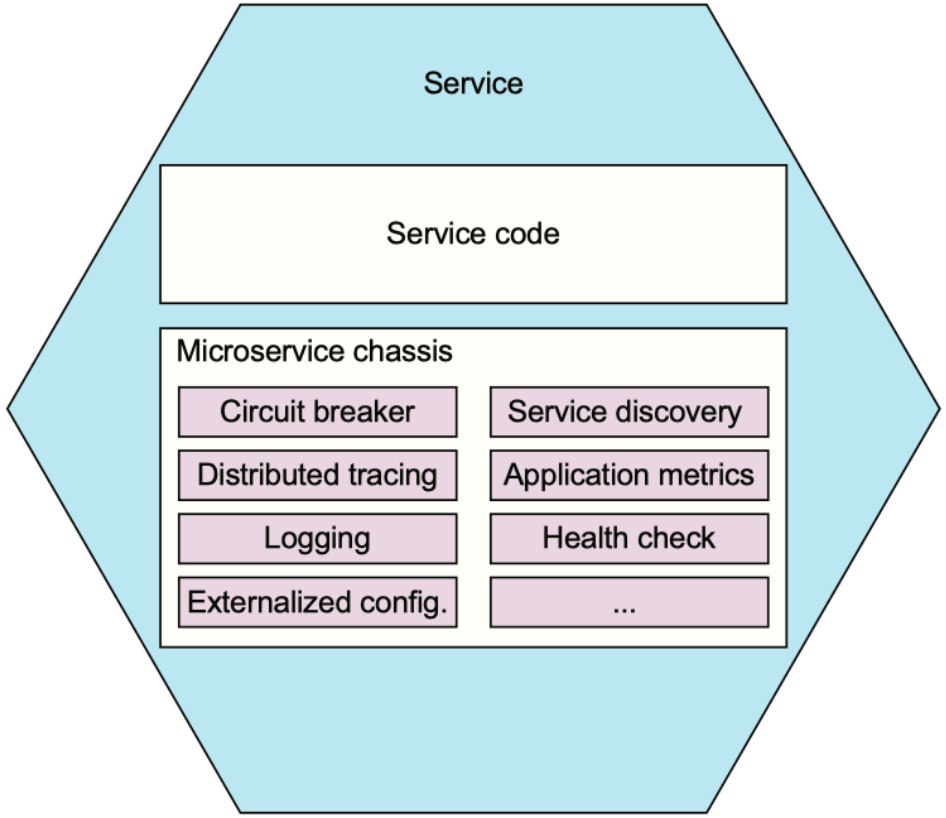




Microservice chassis

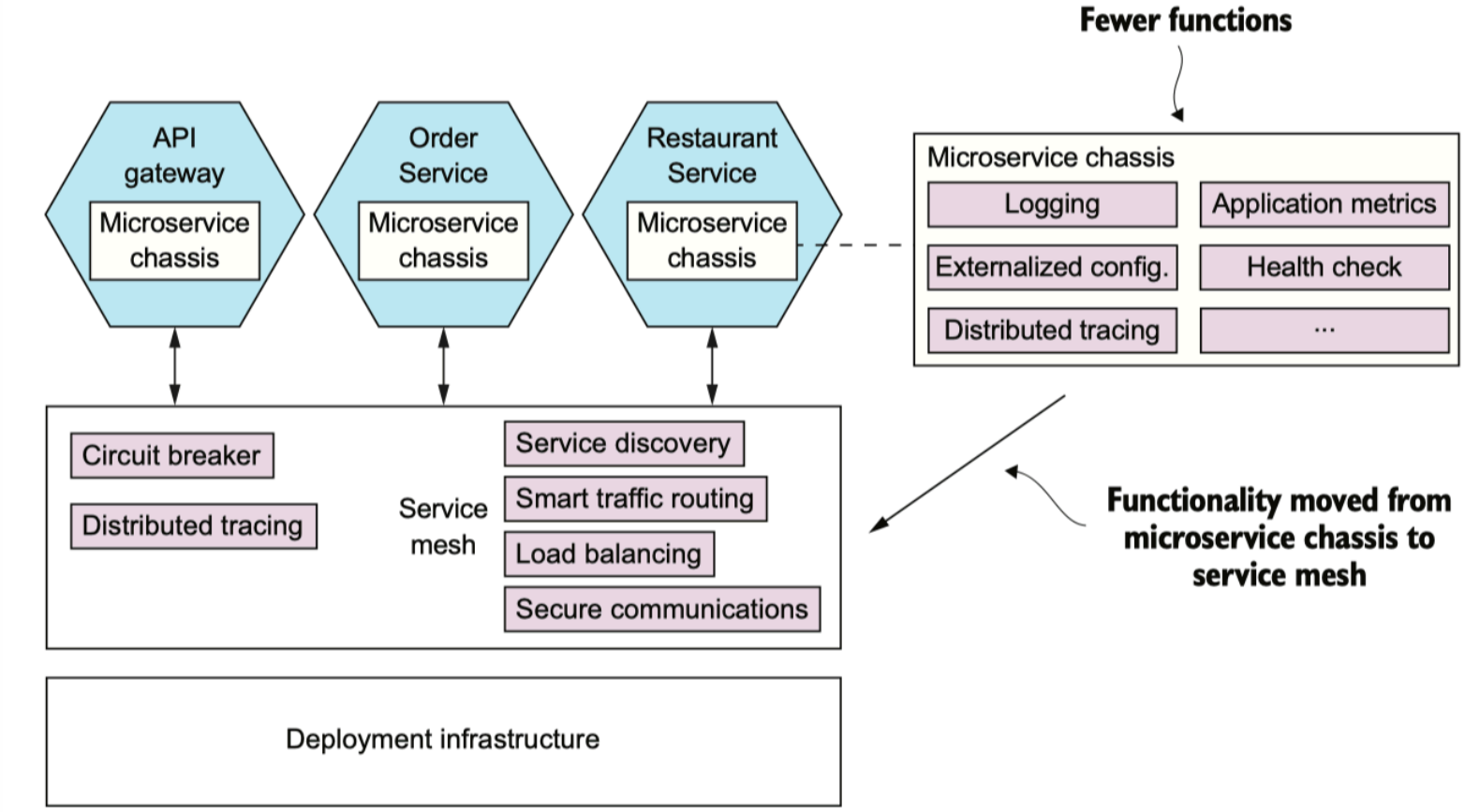
Microservice chassis – это фреймворк, который помогает встраивать сервисы в микросервисную архитектуру

Примеры: SpringBoot/SpringCloud, Go-kit



Service mesh

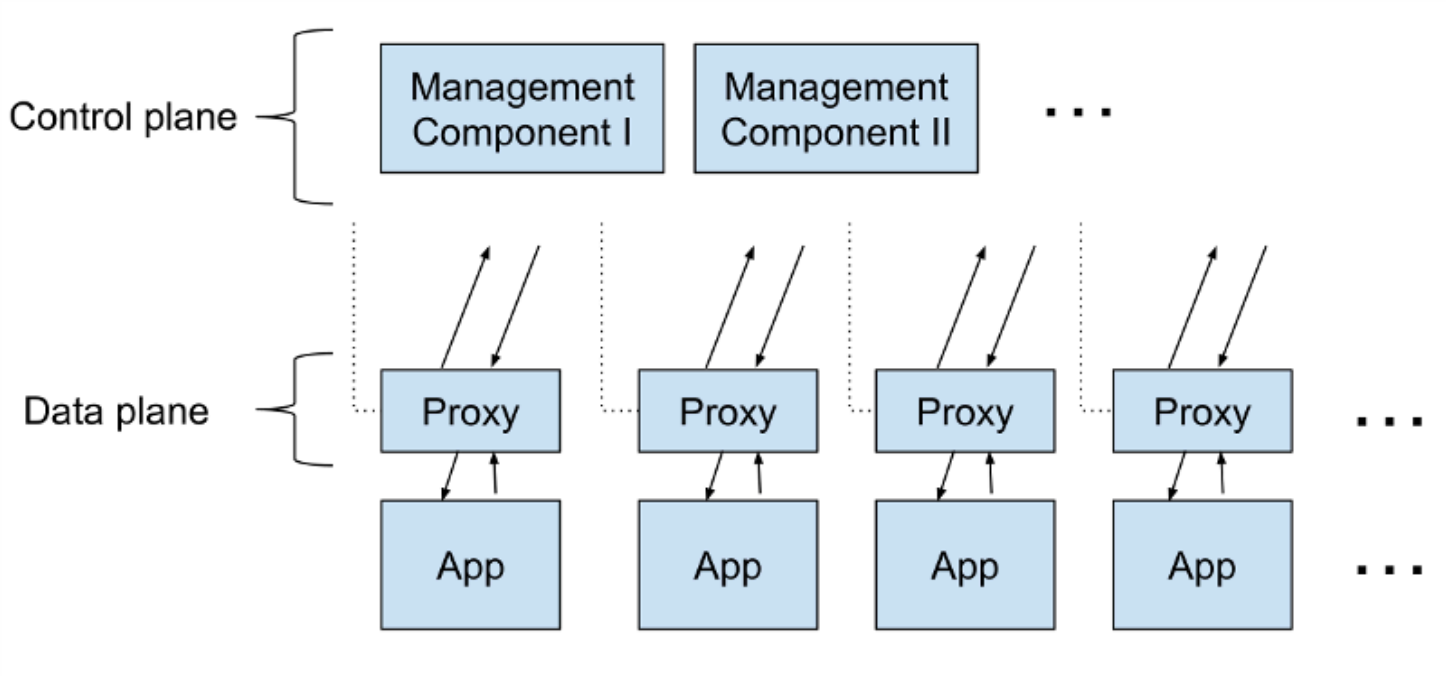
Почему бы не вынести часть логики из кода приложения в отдельный «слой» взаимодействия сервисов?



Service mesh

Давайте с каждым подом поставим side-прокси сервис, в котором будет вся логика по работе с другими сервисами:

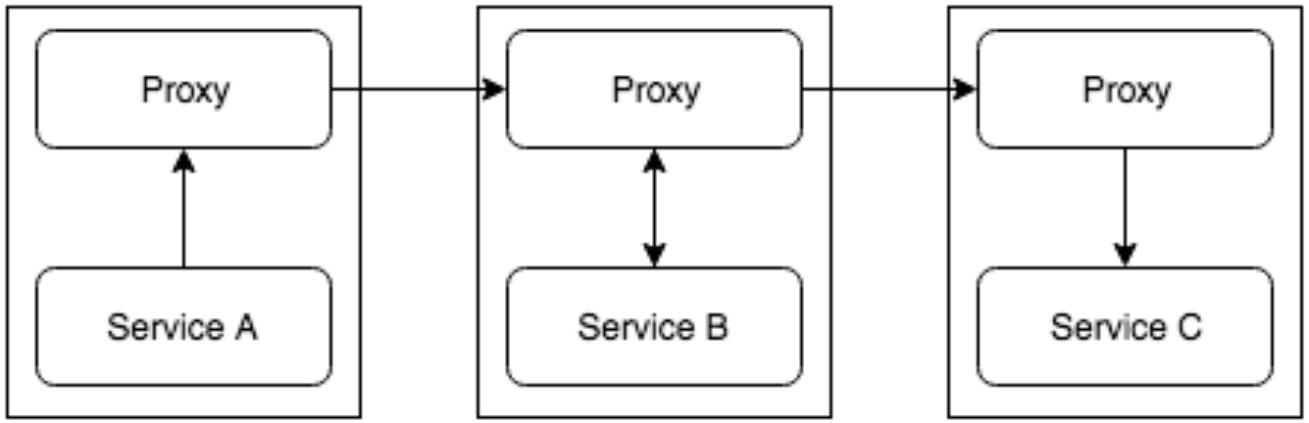
- Проверка прав доступа
- Отправка метрик и телеметрии
- Distributed tracing
- Circuit breaker
- И т.д.



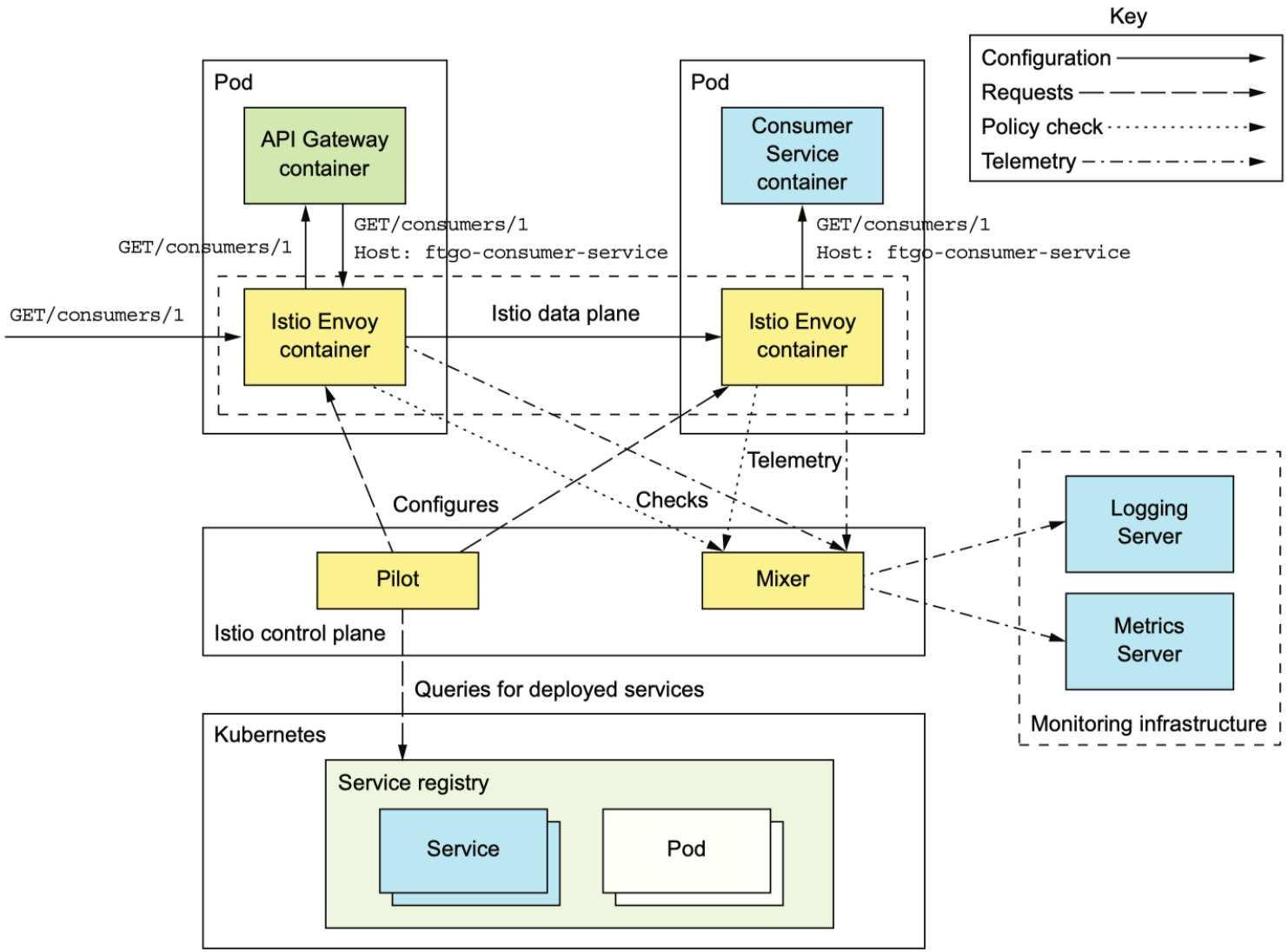
Service mesh

Неважно, что делают сервисы, но трафик между ними является идеальной точкой для добавления новой функциональности.

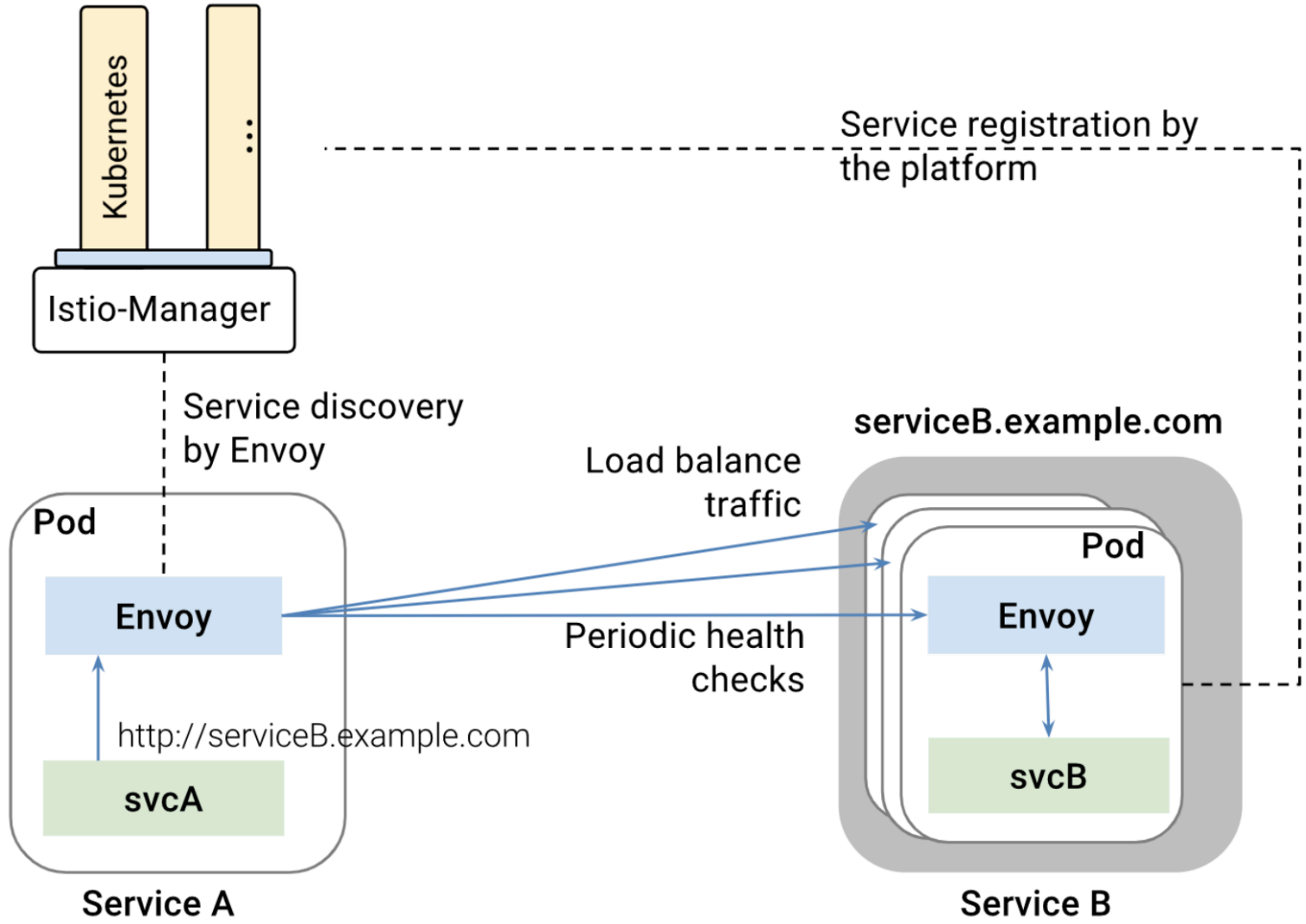
Service mesh выглядит так: вы разворачиваете кучу прокси, которые «что-то делают» с внутренним, межсервисным трафиком, и используете control plane для мониторинга и управления ими.



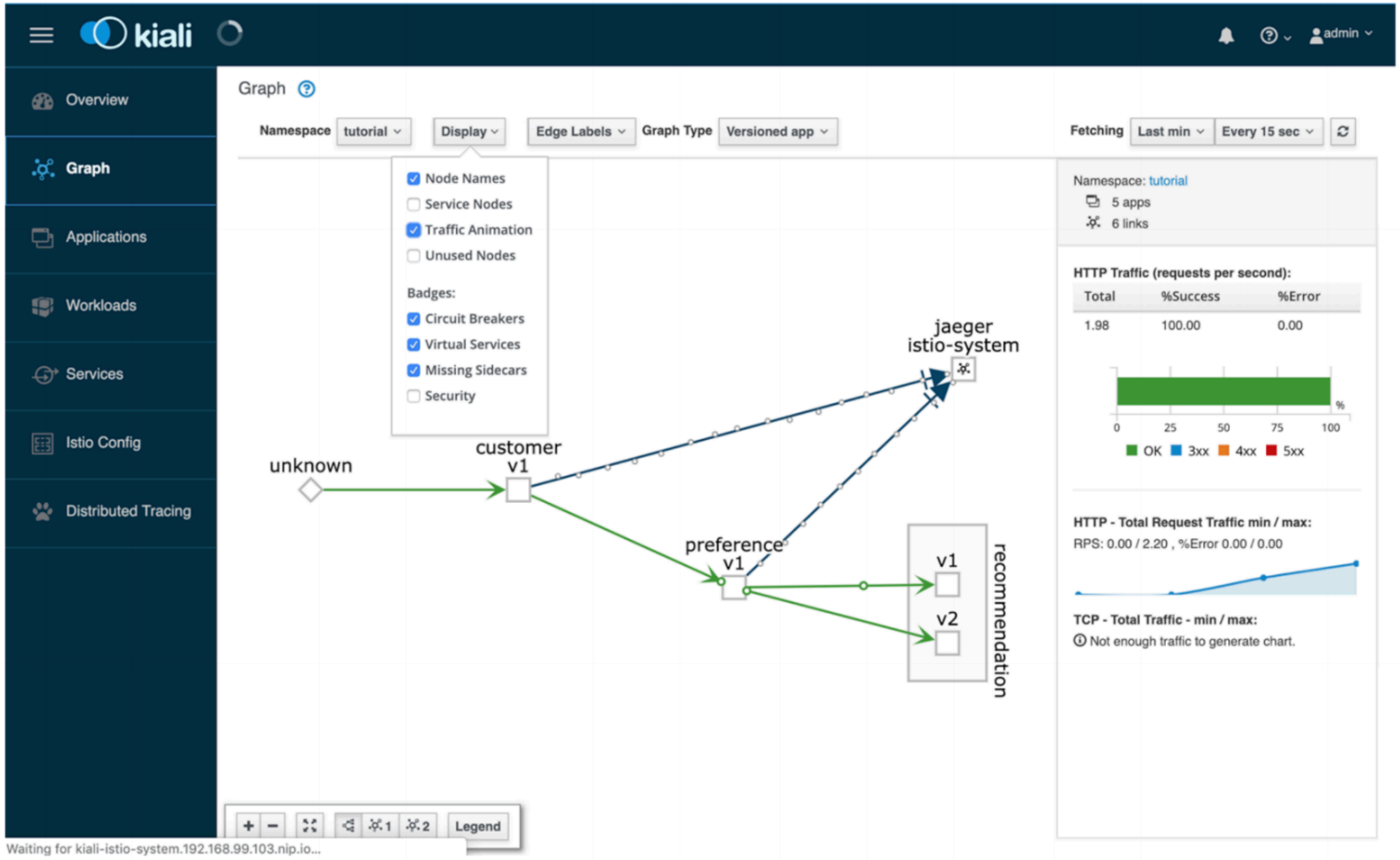
Service mesh на примере istio



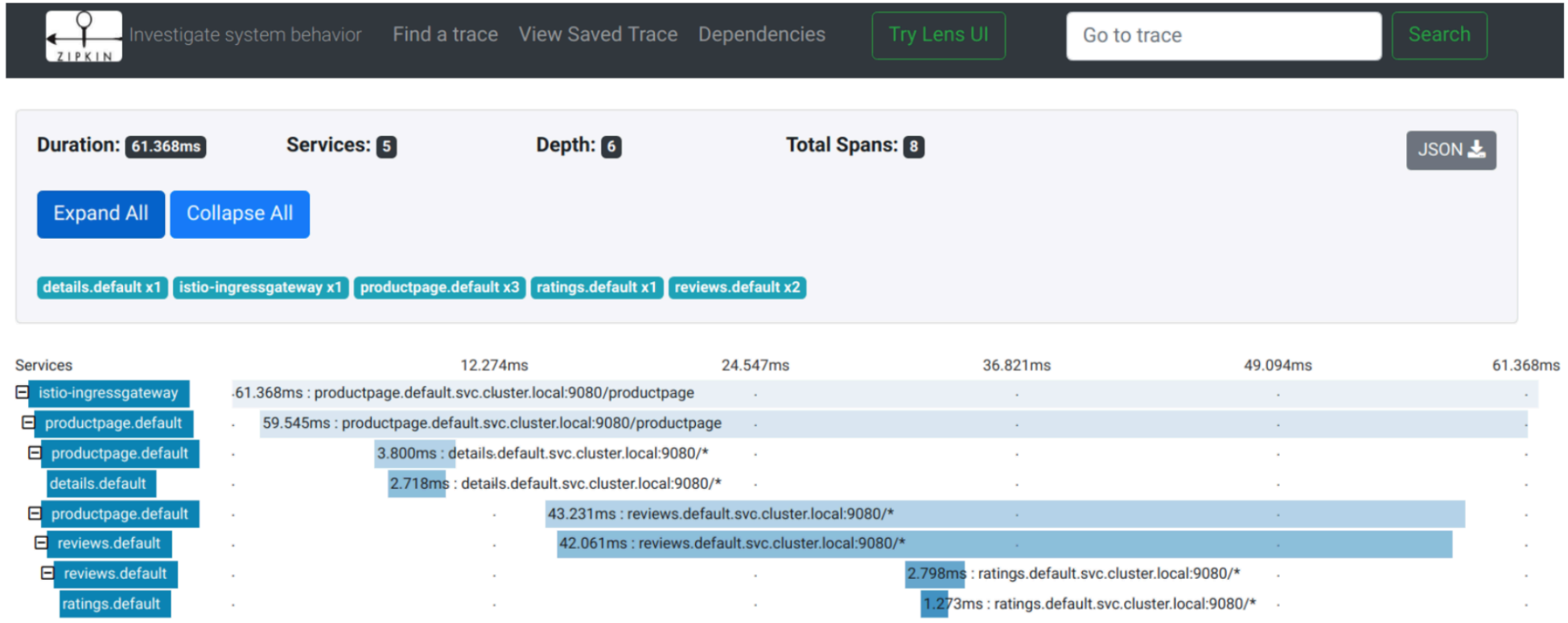
Балансировка запросов



Граф сервисов



Распределенная трассировка



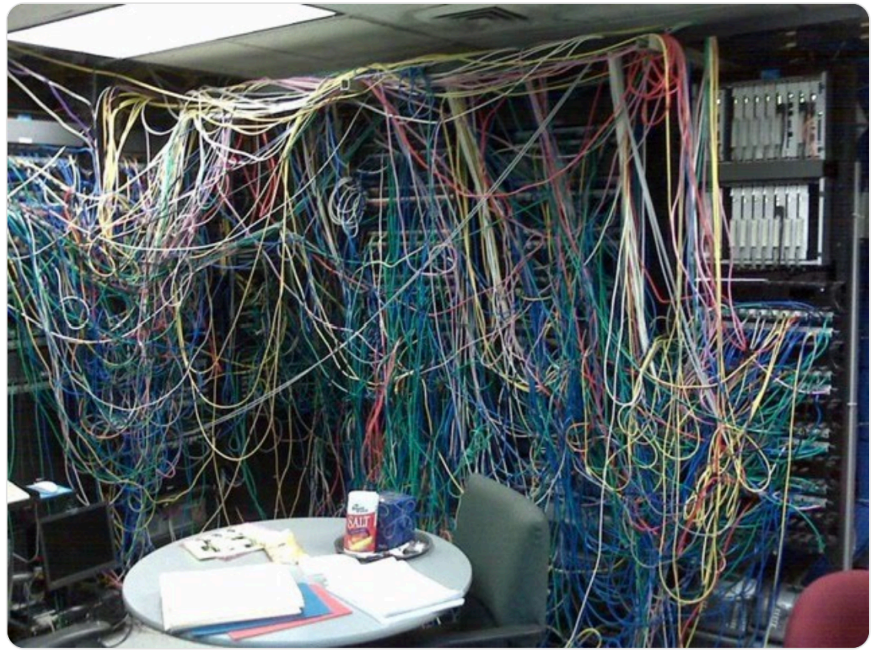
Distributed Trace for a single request

Service mesh на примере istio

Dave Strebel
@dave_strebel

Читать

“Finally got Istio into production”

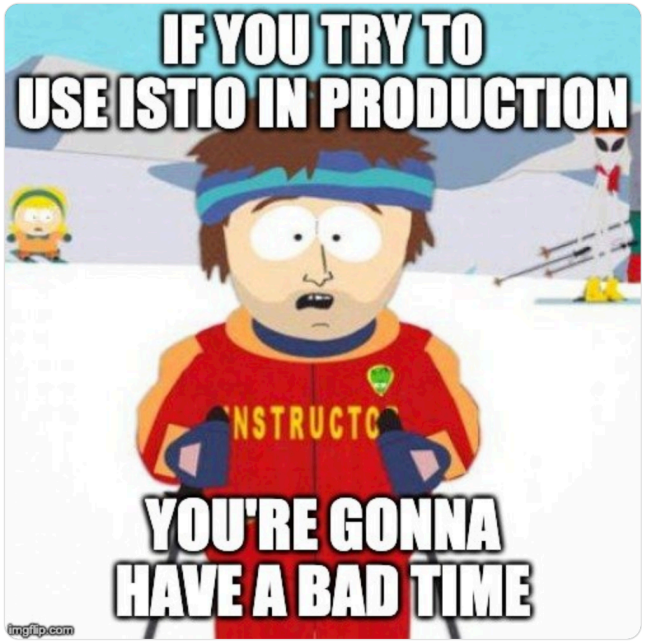


17:21 - 17 сент. 2019 г. Apple Valley, MN

Adam Phillabaum
@adamb0mb

Читать

Tech tip for the day: Don't use #istio in production. It _sounds_ awesome, but moving away from it has been one of the biggest stress-relievers I've seen in a long time.



13:23 - 26 сент. 2019 г.

Service mesh плюсы и минусы

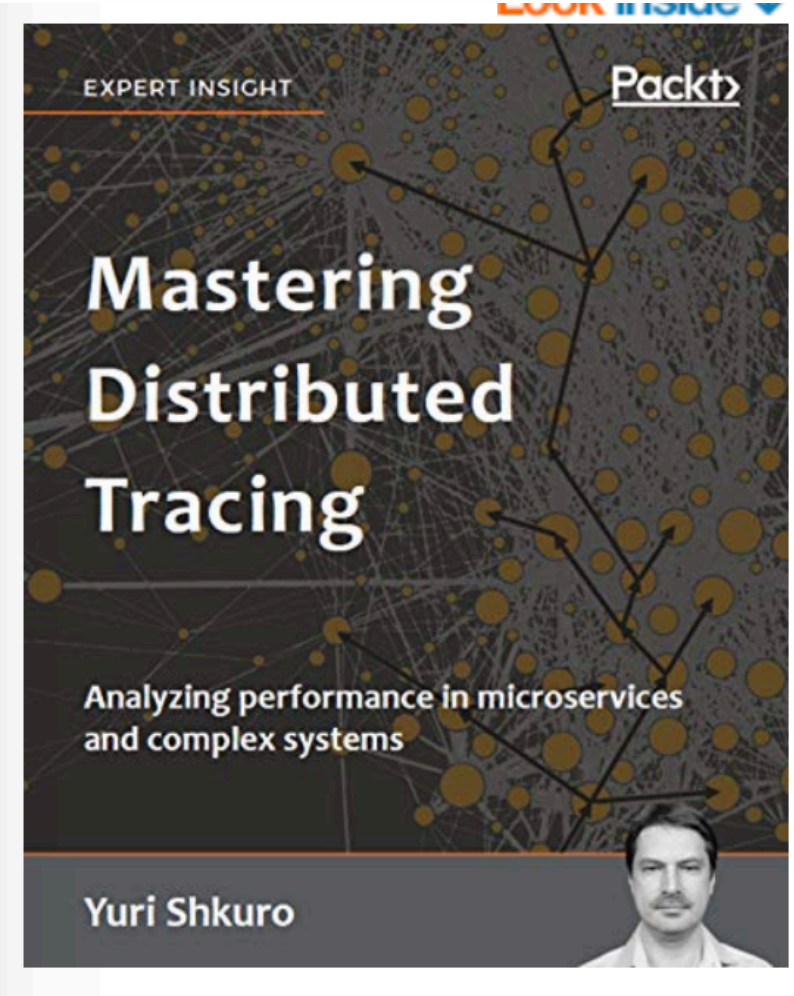
Минусы:

- Увеличение latency
- Дополнительные ресурсы (mem, cpu) на работу прокси
- Сложность поддержки

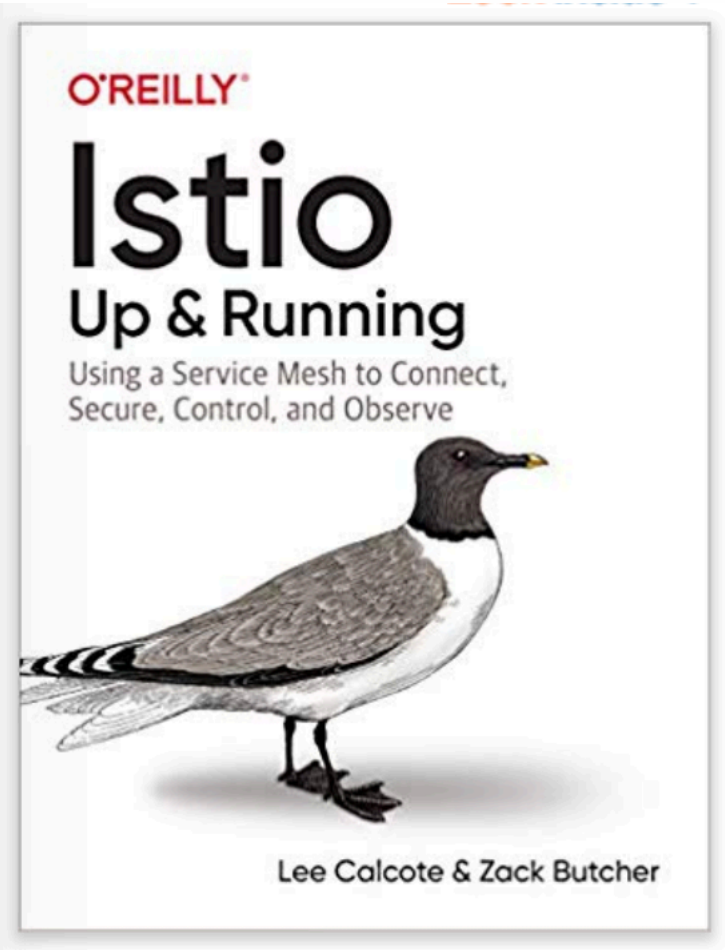
Плюсы:

- Возможность централизованно и без внесения правок в код приложений и независимо от их стека добавлять функциональность.

Материалы



https://www.amazon.com/Mastering-Distributed-Tracing-performance-microservices-ebook/dp/B07MBNGF7Q



https://www.amazon.com/Istio-Running-Service-Connect-Control/dp/1492043788

Материалы

- https://www.elastic.co/blog/distributed-tracing-opentracing-and-elastic-apm – неплохая вводная статья про опентрейсинг от Elastic APM
- https://opentelemetry.io/ - open telemetry
- https://github.com/w3c/trace-context - стандарт w3 context trace
- https://github.com/w3c/trace-context/blob/master/implementations.md - кто поддерживает trace context

Опрос

https://otus.ru/polls/5548/

Спасибо
за внимание!