Distributed Tracing

План:

- Observability
- Распределенный трейсинг
  - Для чего и почему?
  - Из чего состоит и как работает
  - Немного истории
- Как с этим жить?

Observability

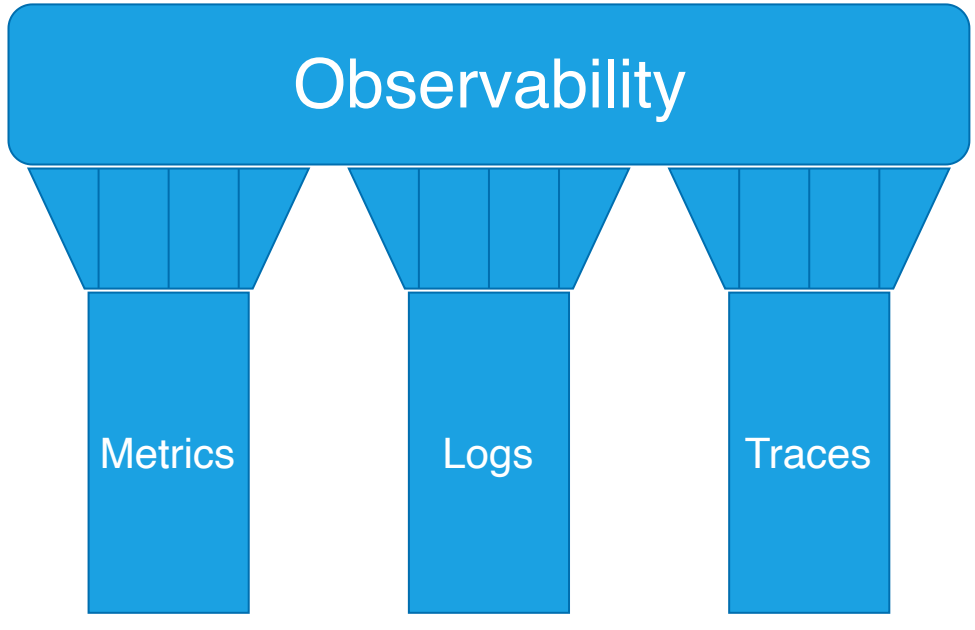
- Характеристика системы, возможность оценить ее внутреннее состояние по внешним данным (outputs)
- Чем точнее оценка - тем более "наблюдаемой" (**observable**) можно назвать систему
- В отличии от термина **Monitoring**, **Observability** подразумевает возможность получения ответов на нестандартные, неопределенные заранее вопросы

Микросервисная архитектура | Вопросы

- Через **какие сервисы** прошел запрос?
- **Что делал** каждый **сервис** при обработке запроса?
- Если запрос выполнялся дольше ожидаемого времени или выполнился неуспешно - **где была проблема?**
- Насколько ход выполнения запроса **отличался от нормального**:
  - Были ли задействованы какие-либо новые сервисы, либо не задействованы текущие?
  - Какие сервисы обрабатывали запрос дольше (или быстрее) чем обычно?
- Какой **critical path** у запроса?
- **Что будет, если**... (сломать, выключить, починить)?

Three pillars of Observability

- **Metrics** - количественные показатели по событиям
- **Logs** - подробная информация о событиях
- **Traces** - серии связанных друг с другом событий, отображающих путь прохождения запроса через систему



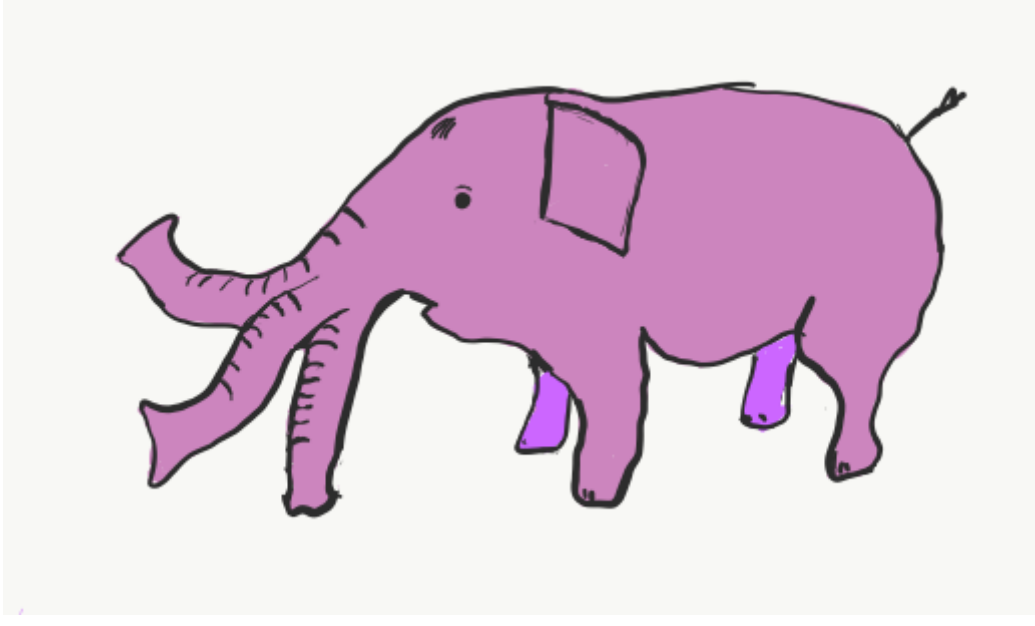
Three pillars of Observability

Итак, у нас есть:

- 3 терабайта логов
- 300 гигабайт трассировок
- 30 гигабайт метрик

Ура!!! Observability!!!

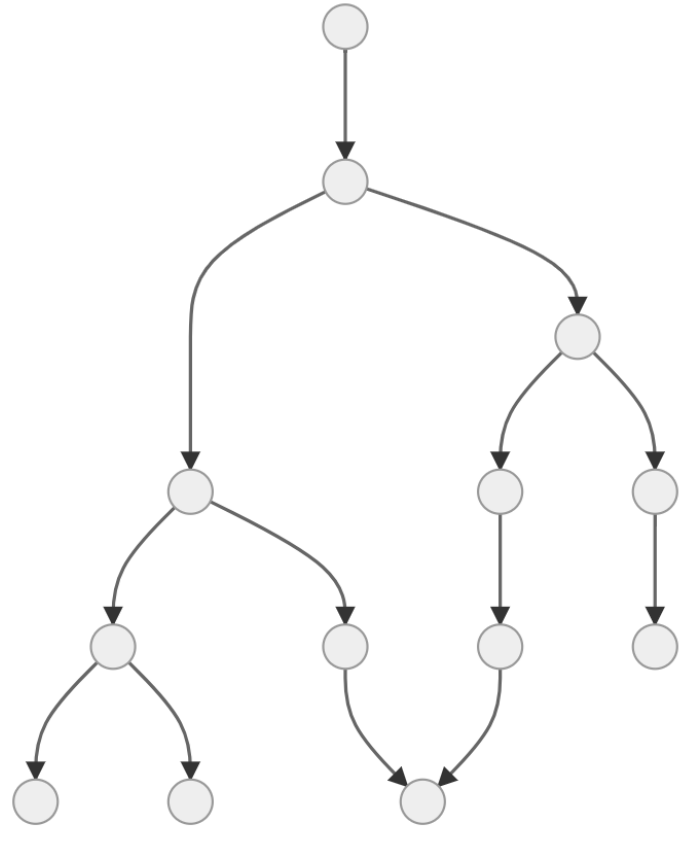
Three pillars of Observability



Трейсы, логи и метрики - это **не фундамент**, а просто **источники данных**. "Наблюдаемость" возникает из возможности получить данные из всех источников, чтобы ответить на заданный вопрос.

Проблема контроля и ответственности

- **Наш** сервис
  - Чуж**ой** сервис
    - **Не** наш сервис
    - **Дру**гой сервис
      - **API** сервис
        - **Как**ой-то сервис
  - **Бэк**енд сервис
    - **Инфра**структурный сервис
  - **WTF** сервис



Проблема контроля и ответственности

1. В **зоне контроля** продуктовой команды - их сервис.
2. Но **в зоне ответственности - все сервисы**, от которых он зависим (они все влияют на качество услуги, за которую отвечает команда).
3. В свою очередь, **те сервисы зависят от других**, другие от третьих и т.д. И эти **зависимости неочевидны**.
4. При диагностике и разработке трейсинг позволяет значительно **сузить область поиска** (и объем ненужных коммуникаций)
5. Трейсинг позволяет **сфокусироваться** на релевантных **метриках и логах**

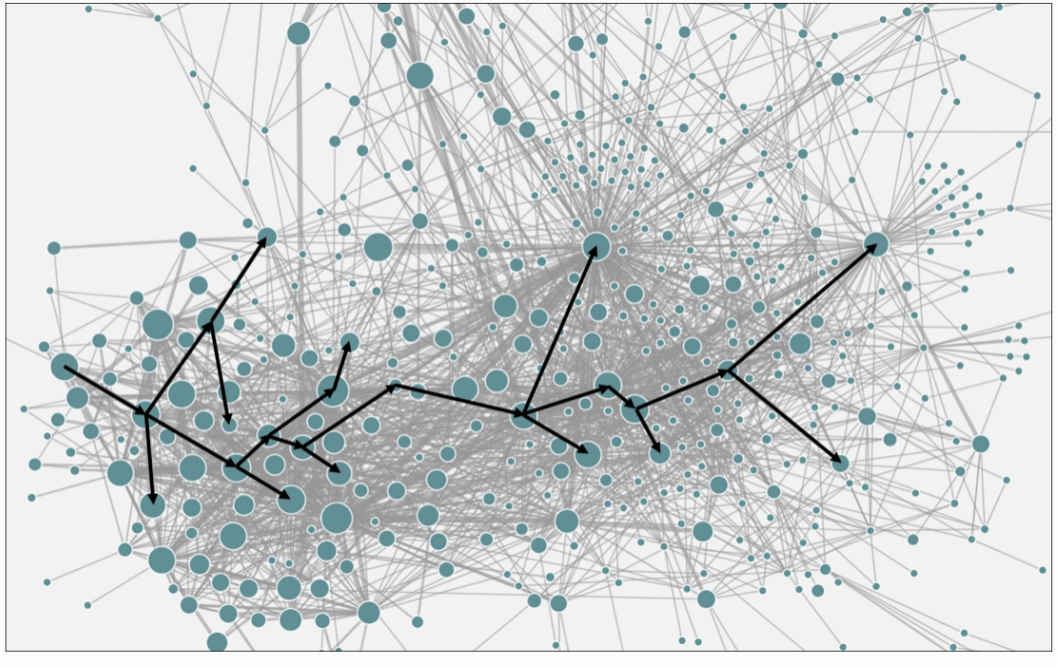
Наконец-то, про трейсинг

Есть метрики и логи, зачем трейсинг?

- **Метрики** - дешевый способ сбора аналитических данных, нет причинно-следственных связей.
- **Логи** - можно помечать каждый отдельный запрос уникальным `Request ID` (**сквозное логирование**), но могут быть проблемы с корректным порядком отображения из-за `Clock Skews` (расхождение времени между разными серверами)

Микросервисная архитектура | Uber

Назовем путь прохождения запроса распределенной транзакцией (**Distributed transaction**)



Основная идея

- С помощью внешнего или встроенного в код инструментария мы получаем данные о работе компонентов (**профилирование**)
- **Данные** профилирования:
  - **собираются** в общей базе
  - **связываются** с конкретным запросом
  - **упорядочиваются** в последовательность вызовов
  - собираются в **единый трейс**, который можно отобразить в интерфейсе (или использовать для анализа)

Подходы к формированию трейсов

- **Black-box inference** (The Mystery Machine) - подход с применением ML/Big Data. В основе лежит возможность сопоставить отдельные Event Logs и сделать вывод о состоянии конкретного запроса
- **Schema-based** (Magpie) - подход, основанный на заранее определенных и описанных схемах взаимодействия сервисов
- **Metadata propagation** - подход, основанный на добавлении к запросу неких метаданных, позволяющих однозначно идентифицировать данный запрос всем компонентам распределенной системы

Metadata propagation

- 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?

В первую очередь, трассировка инструмент для разработчиков:

- **Оптимизация производительности** - можно увидеть ненужные синхронные запросы, в том числе и в чужих системах
- **Корректность поведения** сервиса - запросы на чтение к мастерам кластера БД (вместо реплик) и т.п.
- **Понимание** работы сервиса и зависимостей - кто, куда ходит, зачем и надо ли ему это делать, внутренний "биллинг"
- **Тестирование** - сравнение трассировок развернутой версии и нового кода.

Для чего используется?

- 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?

Какие есть проблемы?

- Трейсы удобны в поиске "узкого места", но не очень эффективны в поиске *root cause* (контекст ограничен запросом)
- Можно столкнуться с буферизацией и упаковкой запросов - на уровне трейса это плохо, а для инфраструктуры - хорошо
- 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 (Kafka), надо писать свои обвязки.

Про историю и стандарты | Dapper

- На уровне идеи все началось с Google Dapper (2000 год!)
- В этой статье Google рассказал о том, как они реализовали распредленную трассировку
- Есть лишь примерное техническое описание
- Больше акцент сделан на том, как они его внедрили для всех сервисов и какой профит получили (и чего трейсинг не дает)
- Ну и еще похвастались своим BigTable...

Про историю и стандарты | Zipkin

- В 2012 году Twitter радостно сообщил, что они за неделю реализовали свой вариант Dapper под названием Zipkin
- Первое популярное open-source решение для распределенной трассировки
- Он стал достаточно популярным, несмотря на то, что в основном, был заточен на Scala/Java экосистему и твиттеровские библиотеки.
- OpenZipkin до сих пор активно используется, как и формат заголовков B3

> **FYI** Zipkin имеет 3 формата сообщений: Thrift и JSON (версий 1 и 2)

Про историю и стандарты | OpenTracing и Jaeger

- К 2015 году почти получилось стандартизовать формат сообщений и метаданных в рамках проекта OpenTracing
- Примерно в то же время Uber попробовал развернуть у себя Zipkin. И внезапно, получился Jaeger
- Он отличается от Zipkin:
  - форматом метаданных (использует OpenTracing)
  - протоколом кодирования и передачи сообщений (TChannel RPC вместо Scribe/Thrift)
  - архитектурой платформы (ближе к Dapper)
- К счастью, есть обратная совместимость с Zipkin

Про историю и стандарты | OpenCensus

- В 2018 году Google выпустил в "открытое плавание" набор библиотек OpenCensus
- Это библиотеки и вспомогательные сервисы для сбора и экспорта метрик, логов и трассировок
- Поддерживается экспорт в Prometheus, StackDriver, Zipkin, Jaeger и еще кучу всего
- Готовый middleware для трейсинга запросов и метрик HTTP, gRPC, DB

> В отличие от Jaeger, нельзя перенастроить через environment

Про историю и стандарты | OpenTelemetry, W3C

- Слияние проектов OpenCensus и OpenTracing в один, под управлением CNCF
- Проект на начальной "технической стадии"
  - пока рекомендуют использовать наработки OpenCensus
  - собственные библиотеки и сервисы - alpha-grade
- Работа W3C над Trace Context - единому формату метаданных для трассировки HTTP-сервисов и Data Interchange Format

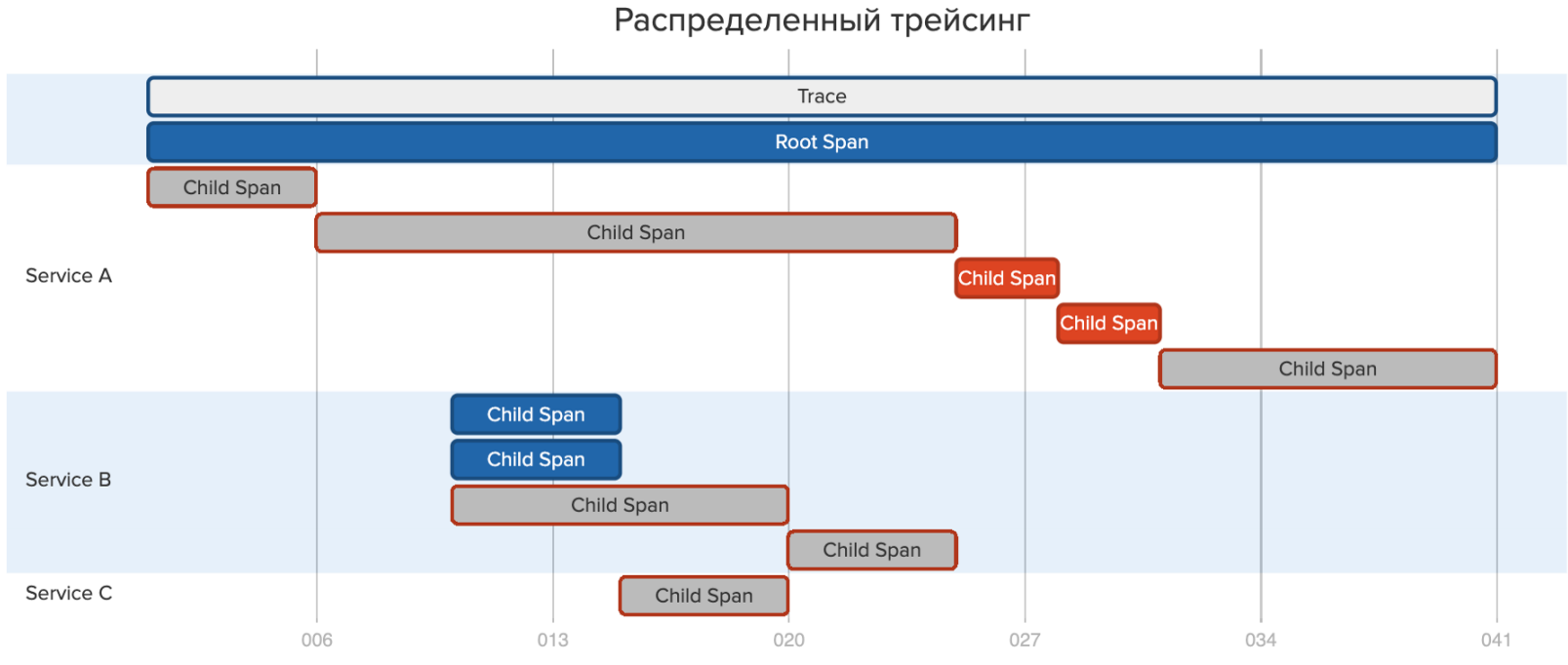
Терминология

- **Span** - запись об одной логической операции по обработке запроса (тайминги и метаданные)
  - Метаданные содержат имя операции и необходимый контекст (аргументы функций, SQL-запросы и т.п.)
  - Каждый спан обязательно содержит ссылку на `Trace-ID`
  - Каждый спан содержит свой уникальный идентификатор `Span-ID`
  - Время начала и окончания операции
  - Статус операции (успех/неуспех)
- В OpenTracing метаданные делятся на `tags` и `logs`

Терминология

- **Trace** - коллекция связанных записей (**Spans**), описывающая обработку одного запроса (end-to-end)
  - Каждый трейс имеет свой уникальный идентификатор - Trace ID
- **Root Span** - это спан, у которого нет ссылки на родительский спан (только Trace ID), он показывает общую длительность выполнения запроса

Терминология



Инфраструктура для трейсинга?

Чтобы запустить трейсинг потребуется:

1. Адаптированный код приложений
2. Сервер для сбора трейсов
3. База данных
4. Пользовательский интерфейс

На самом деле, все чуть сложнее...

Код приложений

1. Для нормального сквозного трейсинга **необходимо** адаптировать код приложения.
2. Варианты:
    - проброс метаданных с входящего на исходящие запросы
    - использование библиотеки "автогенератора"
    - подключение клиентской библиотеки и добавление в код генерации спанов

Код || Проброс метаданных

1. Для большинства RPC информацию можно передавать в через служебные поля в сообщениях (например, HTTP-заголовки).
2. Чтобы собрать вызовы в цепочку - достаточно просто **скопировать заголовки с ID трейса и спана и родительского Span** из входящего запрос в исходящий.
3. Этот вариант требует минимального вмешательства в код. Можно использовать как Day-0 (настроив генерацию заголовков на front-end)
4. Добытые трейсы будут не слишком информативны, с иерархией тоже будет не очень.

Код || Автогенераторы

1. Техника, пришедшая из мира APM решений
2. Для включения трассировок достаточно подключить библиотеку и вызвать функцию "патчинга".
3. Эти библиотеки переопределяют или декорируют вызовы всех или некоторых (I/O - net, file, log) функций
4. Далее, автомагически появляются трейсинг с метаданными, логирование и что-нибудь еще.
5. При бездумном применении объемы данных будут зашкаливать, а информативность падать в пол (например, подробный трейсинг livenessProbe и heartbeat).
6. Вариант таких библиотек от DataDog планируется адаптировать в OpenTelemetry

Код || Клиентские библиотеки

1. Дают достаточно простой API для генерации спанов и трейсов и разбора метаданных
2. Нужно вручную инициализировать контексты, добавлять декораторы к функциям и метаданные к спанам (SQL-запросы и прочее)
3. Требуют больше времени для изучения и внедрения, но дают наиболее информативные трассировки.

Сервер для сбора трейсов

Это сервис, который поддерживает один из стандартов API для трейсинга.

- Он принимает экспортированные спаны, сохраняет их в базе данных
- Может выполнять дополнительное сэмплирование и фильтрацию трейсов
- Может выступать как источник конфигурации для экспортеров трейсов (например, глобально задавать политики сэмплинга)
- Предоставляет API для поиска и вывода трассировок из БД

База данных

Очевидно, хранит в себе все собранные трейсы.

Обычно это ElasticSearch, но для больших объемов может быть несколько баз:

- Хранение всех трейсов - e.q. Cassandra/ScyllaDB
- Хранение и индексирование метаданных - ElasticSearch

> Для тестов или некритичных окружений удобно использовать in-memory или локальную БД (например, Badger в Jaeger)

Пользовательский интерфейс

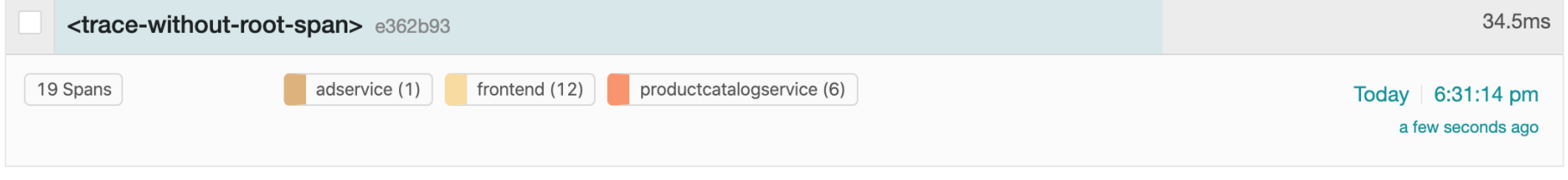
1. Обычно "из коробки" доступен простенький UI (с waterfall и несложными фильтрами). Это лучше, чем ничего, но хуже чем хотелось бы.
2. Для "того самого Observability" придется делать интеграцию между Grafana, этим UI и другими инструментами.
3. Или посмотреть на Elastic APM / Expedia Haystack / Apache Skywalking
4. Для "вдохновения" стоит SaaS (или коробочные) решения - LightStep, Honeycomb, Instana

> Cindy Sridharan про проблемы с UI для трейсинга

Немного про production

Трассировка в Service Mesh

- В Istio трейсинг **из коробки**! На самом деле - **нет**.
- Sidecar proxy и Mixer могут генерировать спаны и отправлять их на коллектор
- По умолчанию, в Envoy используется B3-стандарт для кодирования метаданных. Поменять не так просто, и в приложениях придется с этим считаться.
- Сервер для сбора трейсов, БД и UI - ставится отдельно
- Если коллектор недоступен при старте proxy - придется перезагружать proxy.



To sample or not to sample?

- Без сэмплирования - плохо для больших объемов
- Постоянный коэффициент - плохо для малых объемов
- Постоянная частота - оптимальный вариант

Крайне желательно **сохранять в трейсе информацию о сэмплинге** (алгоритм и частота).

> В Google Dapper/Jaeger/OpenCensus - двух-уровневый сэмплинг (на уровне отправки и сохранения сообщений)

Техники сэмплирования

- Вероятностное (выбрать 1 из 100 трейсов случайным образом)
- Отложенное решение (выбрать что собирать на основе длительности или результата запроса)
- Делегированное сэмплирование (вызываемый компонент может принять решение о том, сохранять ли трейс)

P.S. - Про подбор параметров сэмплинга - в брошюре Honeycomb (в конце слайдов)

Как управлять сэмплингом?

- Агенты в Jaeger могут запрашивать параметры с сервера
- Заголовки `X-B3-*` и `X-OT-*` содержат атрибут sample-rate.
  - Если он равен 0, то спан не будет отправляться (удобно для `healthz`-эндпоинтов)
  - можно использовать для сбора 100% трейсов по определенному пользователю или группе

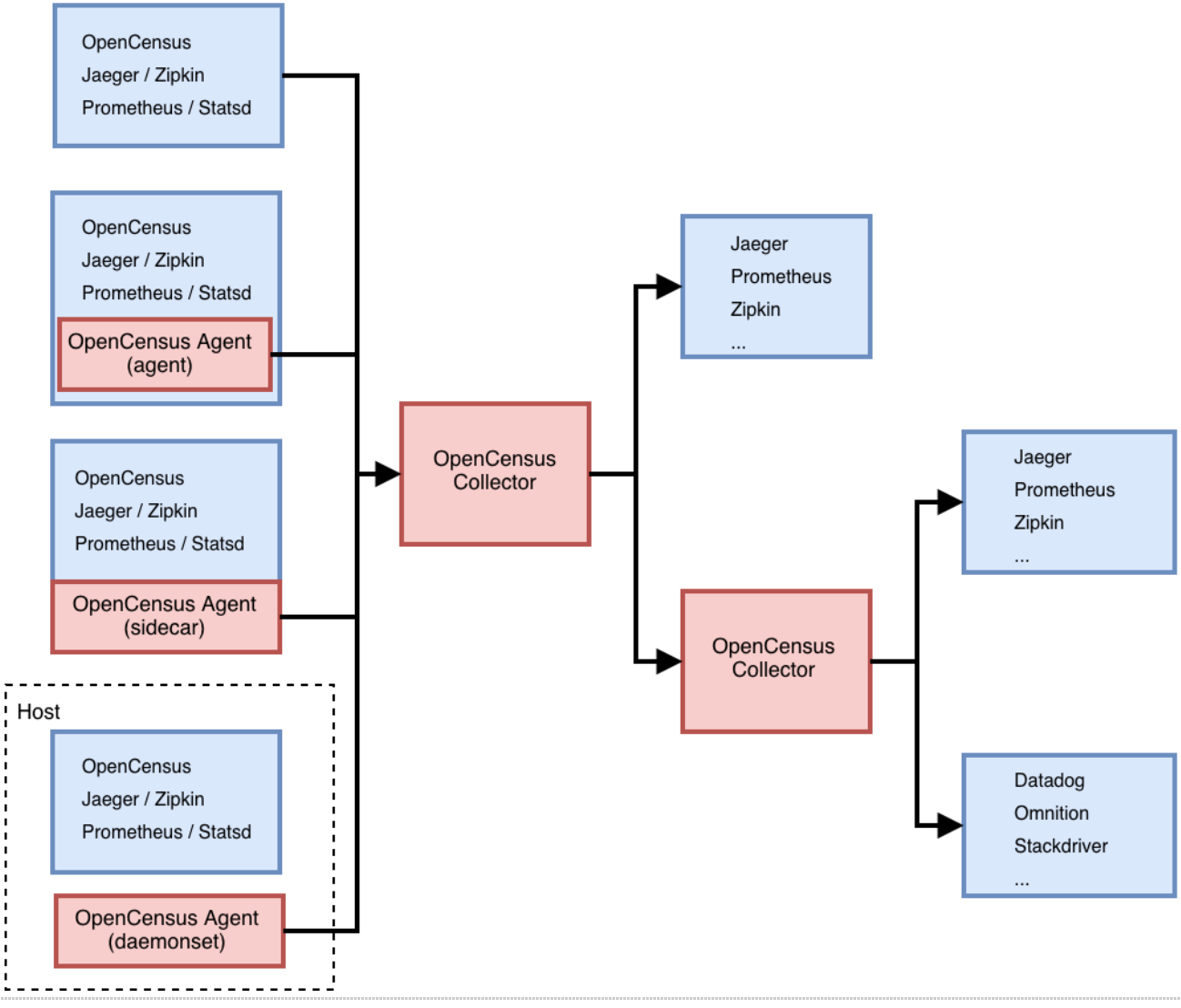
Про архитектуру

1. Сбор и **отправку трейсов** лучше **делегировать** отдельным сервисам - **агентам**
    - Они поддерживаются "из коробки" в OpenCensus и Jaeger (в Alpha в OpenTelemetry)
    - Запускаются как Sidecar (Uber-way) или DaemonSet (Dapper-way)
    - Позволяют не тащить в код лишние зависимости (service discovery, поддержку разного траснпорта и бэкендов) и блокирующие вызовы

Про архитектуру

- Дополняем агенты коллекторами OpenCensus:
  - коллекторы собирают данные с агентов
  - выполняют предобработку (в том числе, интеллектуальный сэмплинг)
  - пересылают в соответствующие бэкенды (например, Jaeger)
- По традиции, вставляем Kafka между серверами трейсинга и БД, если данных много:
  - Jaeger Collector –> Kafka –> Jaeger Ingester –> ElasticSearch/Cassandra

Про архитектуру



Полезные ссылки:

- Mastering Distributed Tracing - книга по Distributed Tracing
- Тренинги по OpenTracing на Katacoda
- Тренинги по OpenTracing
- What is the “cost” of doing instrumentation? - OpenTracing - Medium
- Доклад Егора Мыскина "Трейсинг распределённых систем"
  - Конспект Андрея Александрова по докладу "Трейсинг распределённых систем"
- Хорошая обзорная статья от Nike Engineering

Полезные ссылки:

- Еще обзорная статья по архитектуре
- Getting Started with Observability Lab: Opentracing, Prometheus, and Jaeger | USENIX
- LightStep Blog - Distributed Thoughts for a Performant World
- White papers and guides - Honeycomb
    - Брошюра от Honeycomb по Observability
    - Брошюра от Honeycomb по трейсингу