avito.tech Москва 2024

Как мы строим систему распределенного трейсинга, в которой можно терять данные

Игорь Балюк Senior Platform Engineer

Привет!

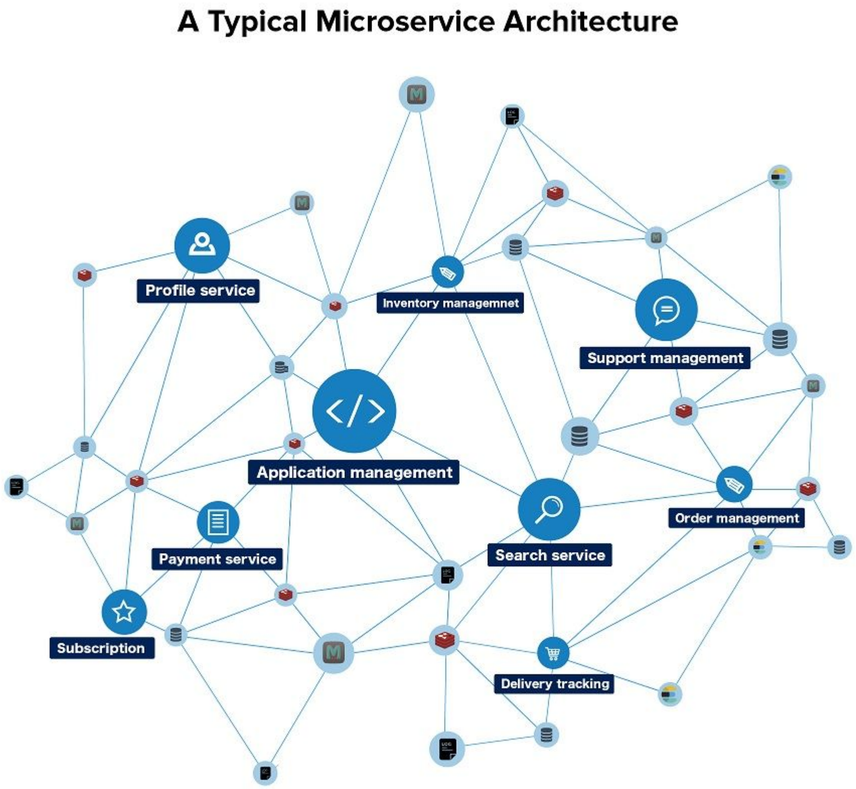
- Платформенный инженер в Авито
- Изучал распределенные системы
- Разрабатывает платформу по сбору отладочных данных

Игорь Балюк @ Авито

Контекст

Как отладить работу распределенной системы?

Дано: много сервисов, общающихся по сети
Задача: научиться быстро находить первопричину деградации



https://www.apmdigest.com/the-role-of-distributed-tracing-in-quick-problem-solving

На помощь приходят…

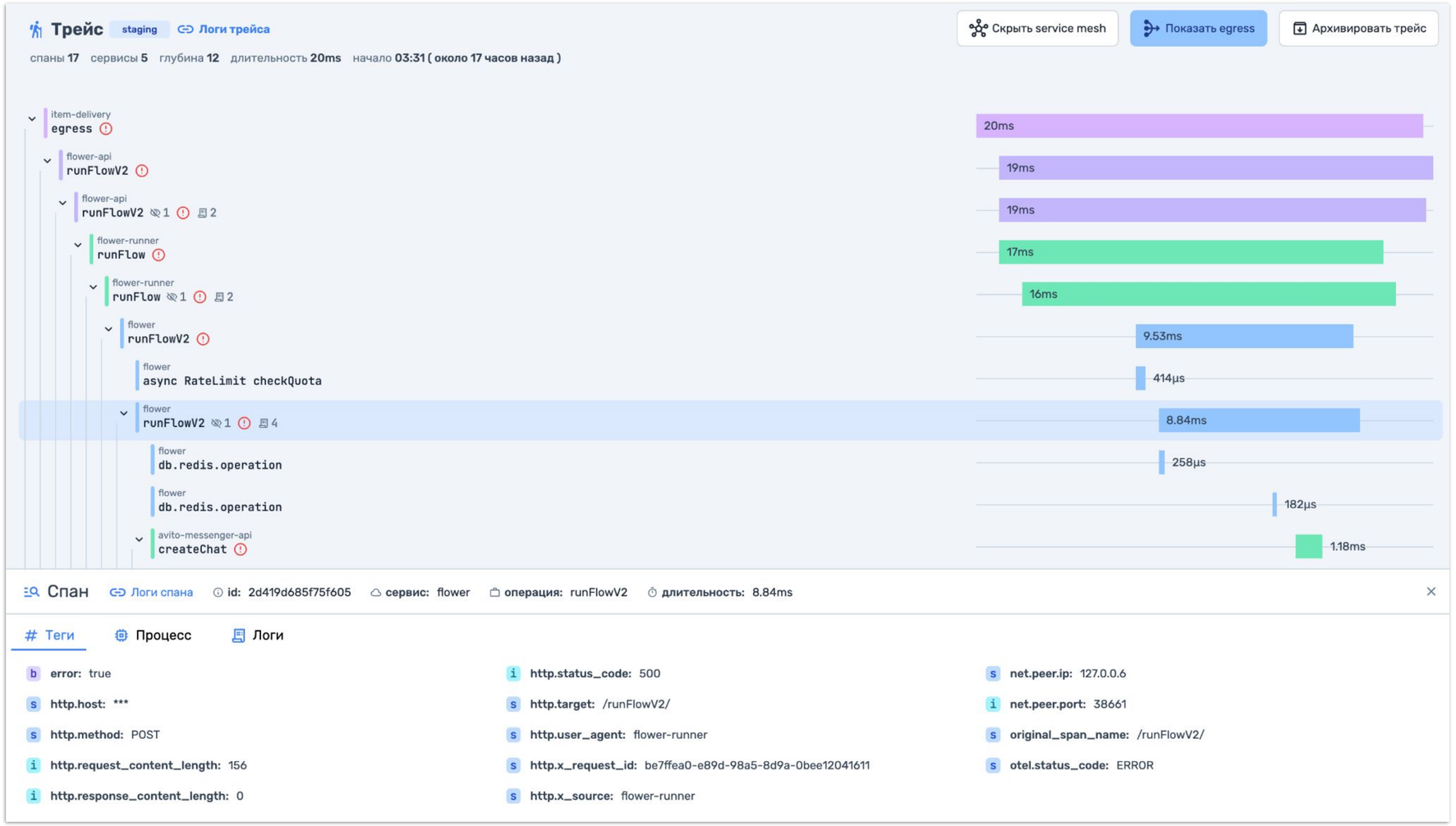
Логи!

Готовые инструменты – ELK, NewRelic, DataDog, Splunk, …

Observability платформа в Авито

Идея – **собрать разные отладочные данные в одном месте и предоставить интерфейс для быстрого поиска**

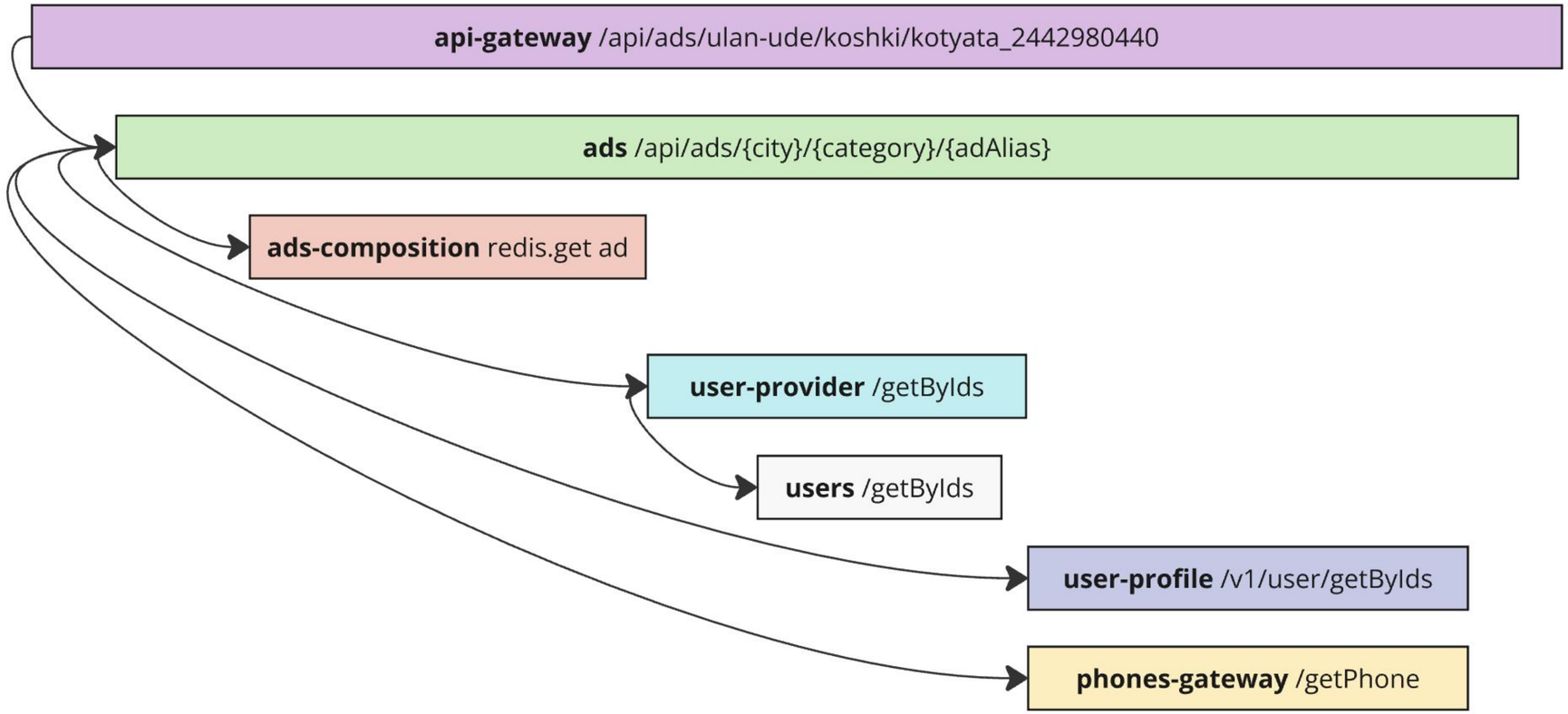
- ❏ Отладка микросервисного общения
- ❏ Верхнеуровневый анализ архитектуры
- ❏ Поиск первопричины инцидента



Про трейсинг

Трейс – информация о том, какие сервисы и СУБД участвовали в обработке конкретного пользовательского запроса.

~ добавить X-Request-Id к access logs всех приложений

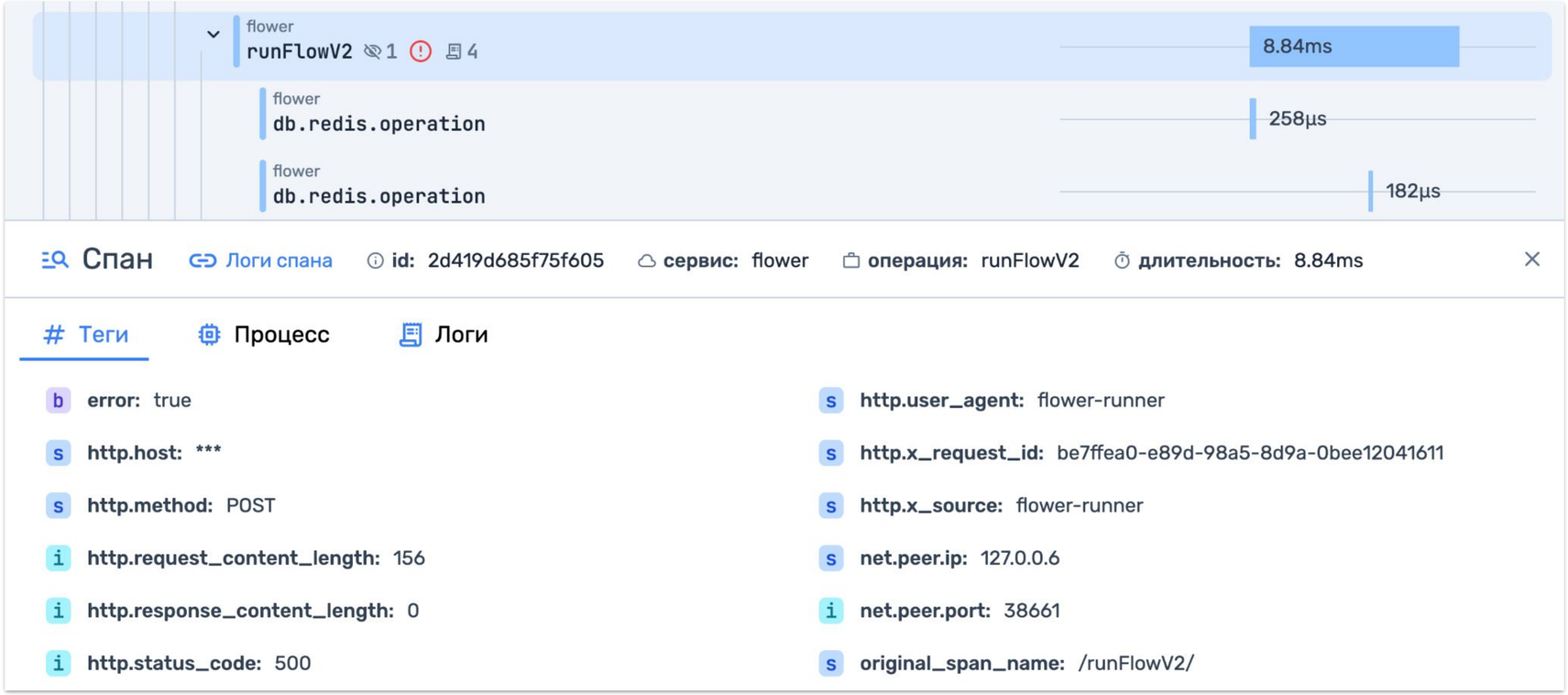


Спаны

Спан – промежуточная точка: HTTP-запрос, поход в СУБД, внутренняя функция…

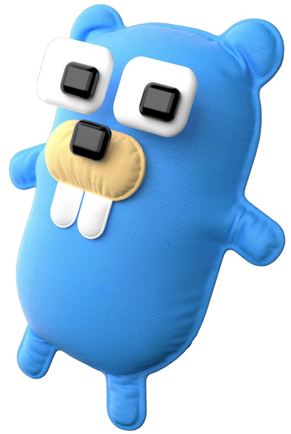
Спаны связываются с помощью Trace ID и образуют трейс

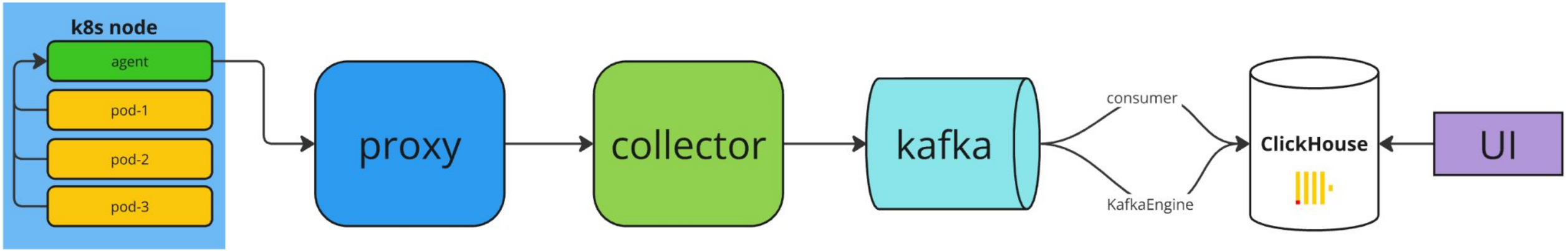
Спаны похожи на логи, но по отдельности ценности не несут



Характеристики и требования к платформе

- ❏ **> 17 миллионов** событий в секунду
- ❏ ~3000 сервисов
- ❏ Несколько дата-центров
- ❏ Latency – не так критично
- ❏ Терять данные можно

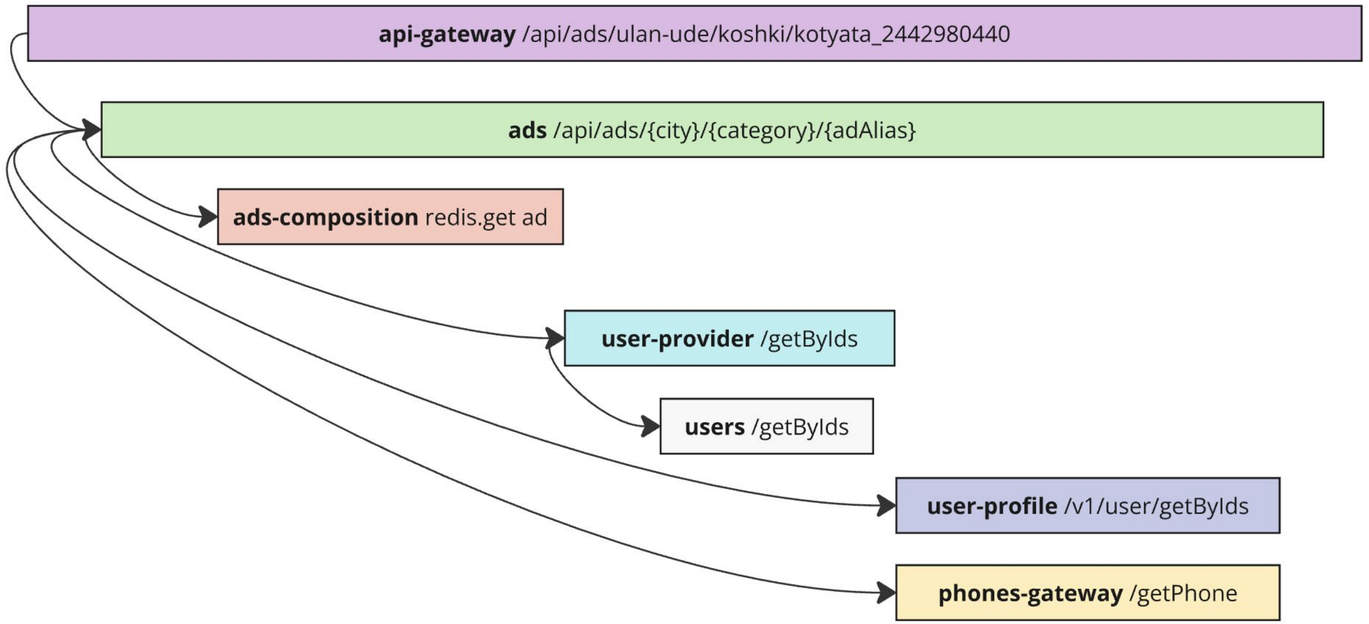


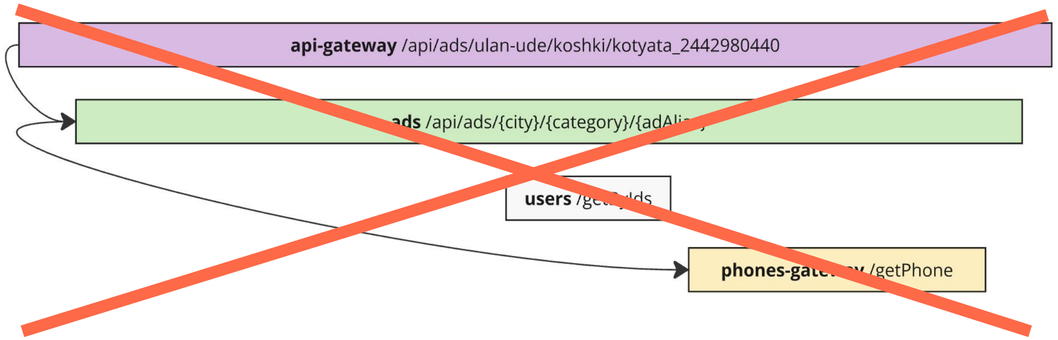


Терять данные можно, но не все

Не предъявляются строгие требования к гарантии доставки данных

Но нельзя доставлять лишь часть трейса: либо все, либо ничего

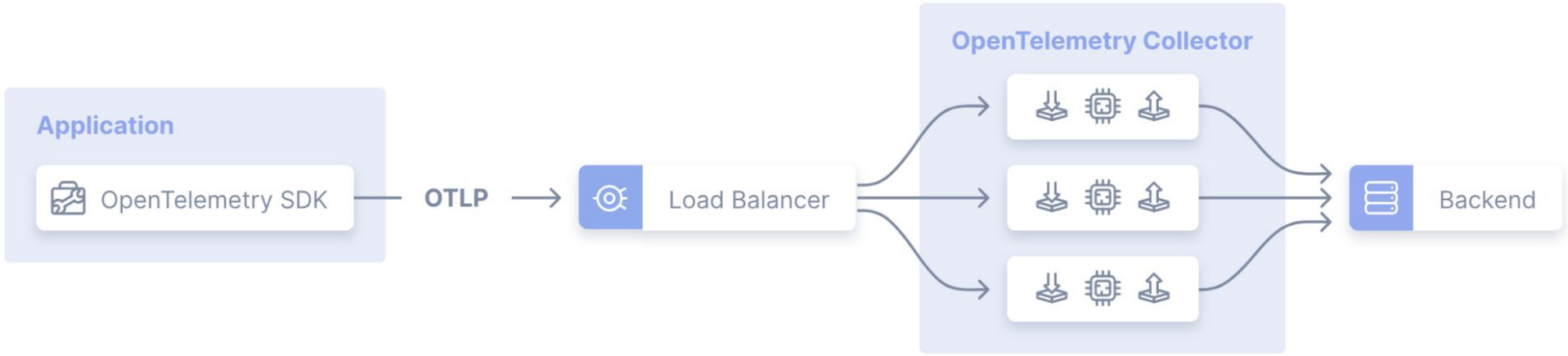




Архитектура

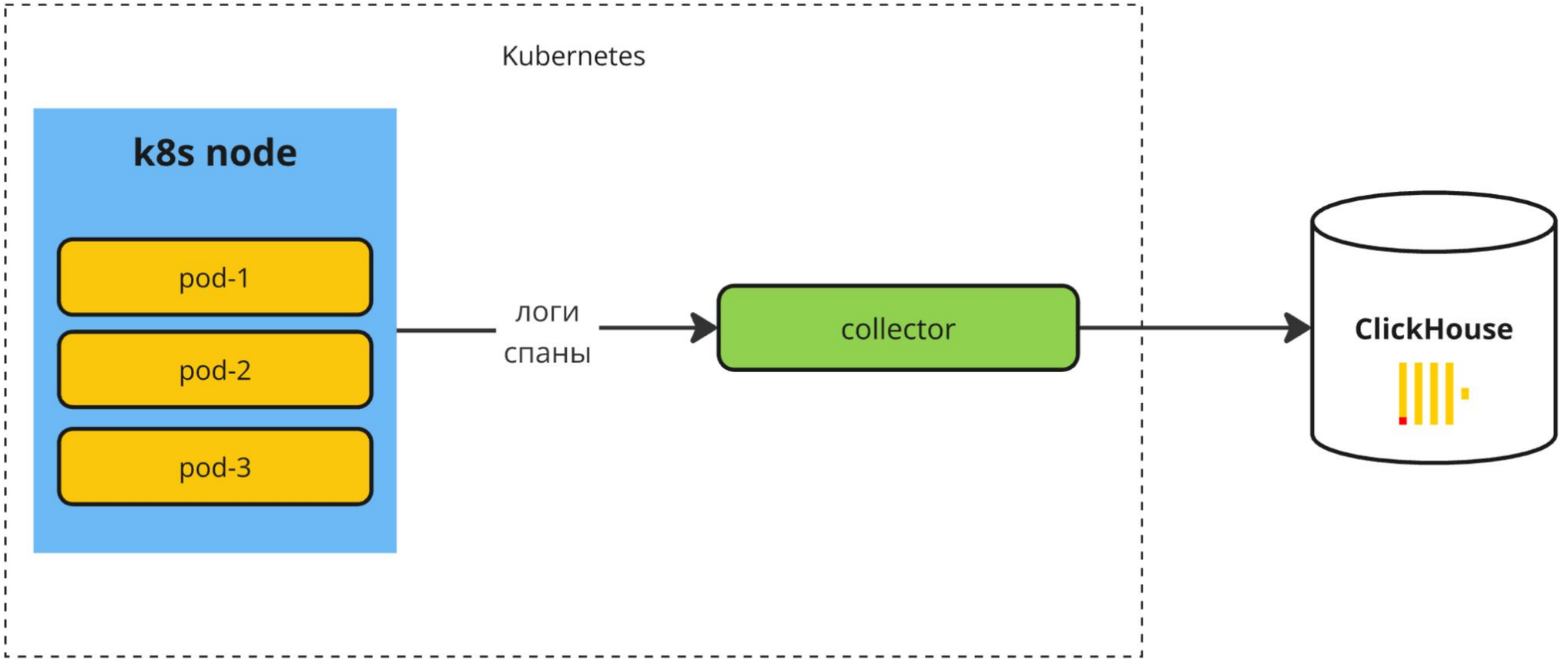
Типичная архитектура

OpenTelemetry – набор инструментов и стандарт для обработки телеметрии (трейсинг, логи, метрики)



https://opentelemetry.io/docs/collector/deployment/gateway/

Наша простая архитектура



Трудность №1: неприхотливость хранилища

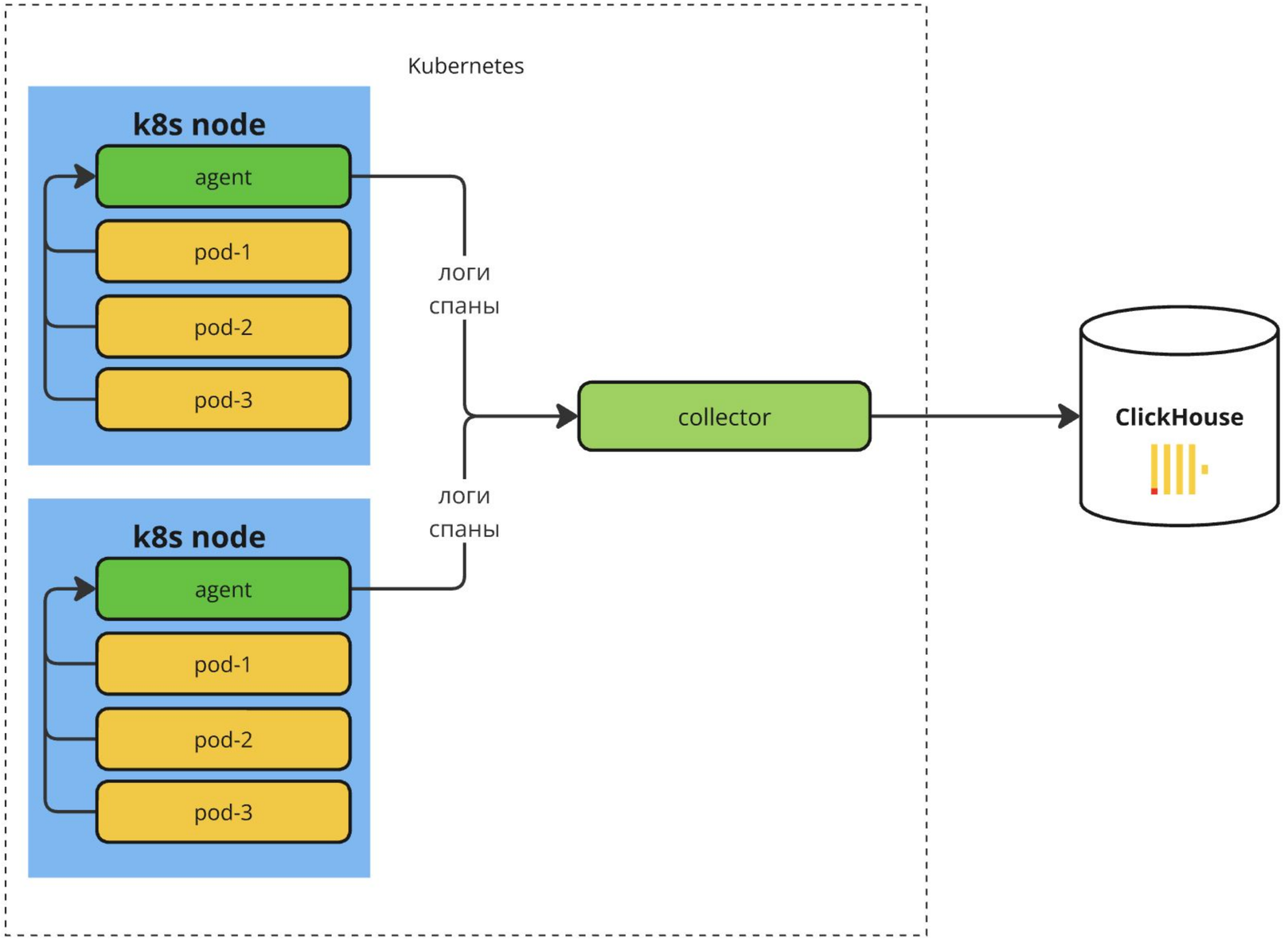
ClickHouse не любит много входящих запросов

Нужно сузить входной поток: Buffer Table, Reverse Proxy, Message Broker, …

Трудность №2: много соединений

1000 узлов * 80 подов = **80 0000** клиентов, генерирующих события

- ❏ Большая нагрузка на коллекторы
- ❏ Перезагрузка коллекторов – теряем много данных


Kubernetes
k8s node
agent
pod-1
pod-2
pod-3
k8s node
agent
pod-1
pod-2
pod-3
логи
спаны
логи
спаны
collector
ClickHouse

Агент

- ❏ Запущен на каждом k8s-узле
- ❏ Принимает спаны, вычитывает логи
- ❏ Первичная фильтрация и обогащение
- ❏ Троттлинг

Агент

- ❏ Запущен на каждом k8s-узле
- ❏ Принимает спаны, вычитывает логи
- ❏ Первичная фильтрация и обогащение
- ❏ Троттлинг
- ❏ **Батчи**
- ❏ **Буферизация**

Батчинг – это важно

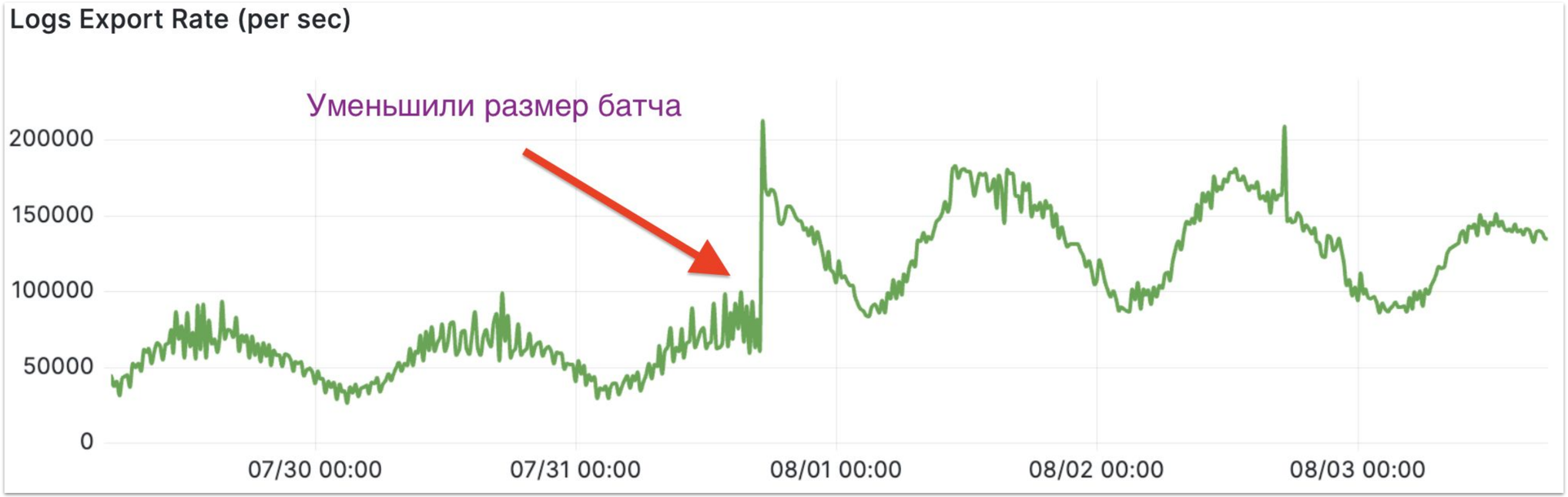
Чем меньше запросов – тем лучше

- Трейдофф между latency и производительностью

Поиск баланса

Слишком большие батчи:

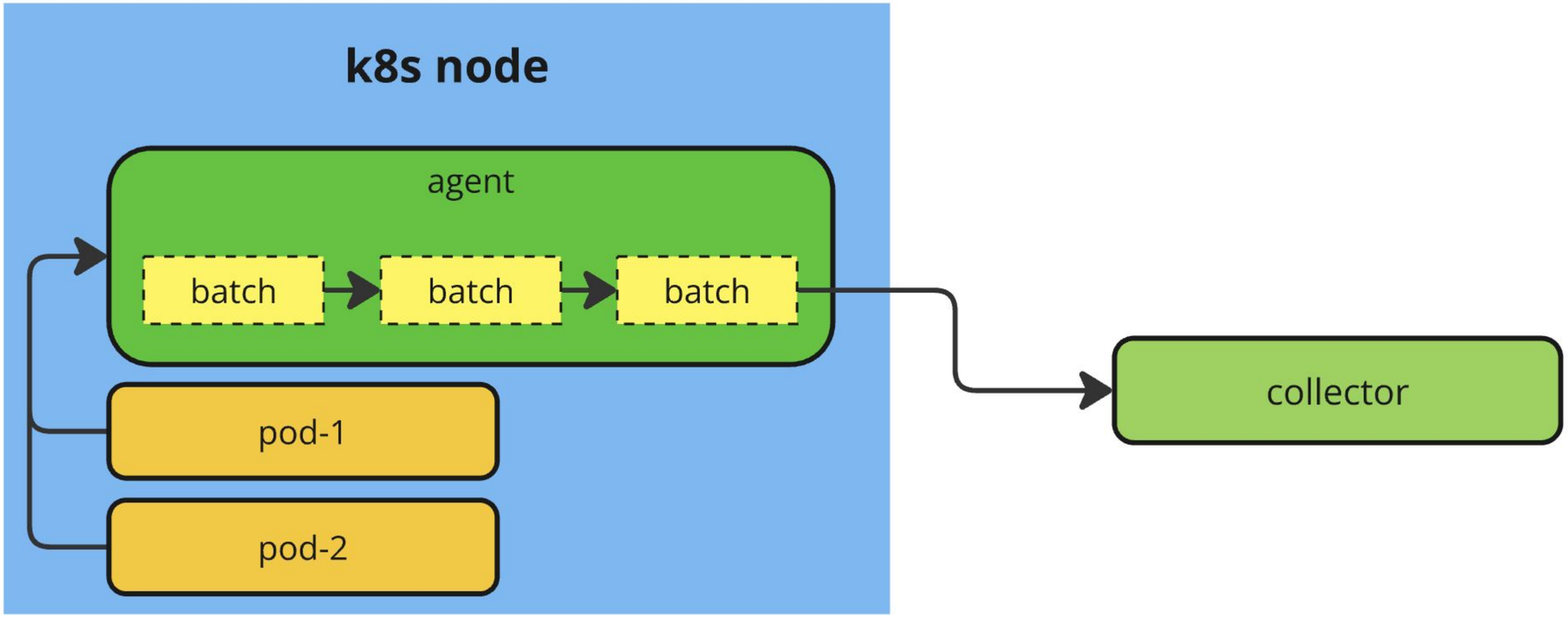
- ❏ Больше RAM
- ❏ Помним про Garbage Collector
- ❏ Дольше обрабатываются, из-за чего появляются пробки



Буферизация данных

Если коллектор недоступен, агент может накапливать данные у себя

- Логи могут оставаться на диске
- Спаны – только в памяти ⇨ может быть OOM



Семплирование

Данных слишком много, нужны не все

5% данных позволяют найти инцидент, но ресурсов требуется меньше

- ❏ Заранее нельзя понять, пригодится ли трейс
- ❏ В приоритете – ошибки

Данных слишком много, нужны не все

5% данных позволяют найти инцидент, но ресурсов требуется меньше

- ❏ Заранее нельзя понять, пригодится ли трейс
- ❏ В приоритете – ошибки

Head-based

Сохранить состояние в идентификаторе трейса

`TraceID: sample=0;550e8400e29b41d4a71`

\+ дешево избавляемся от ненужных данных
– сохраняем какие-то случайные данные

Tail-based

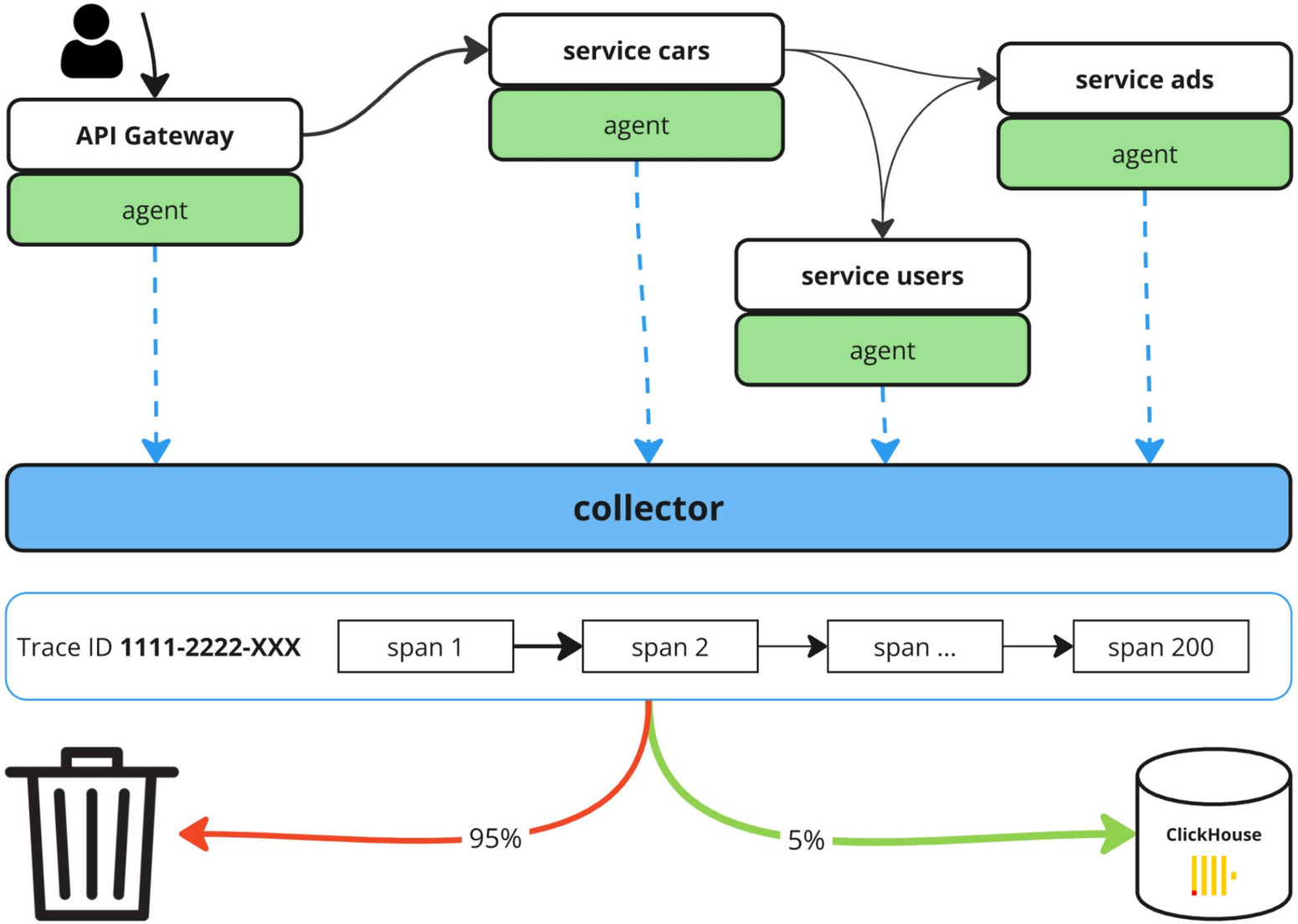
Собираем всю информацию по трейсу и делаем вывод, полезен ли этот трейс для нас

\+ сохраняем более качественные данные
– сложнее в реализации

Tail-семплирование

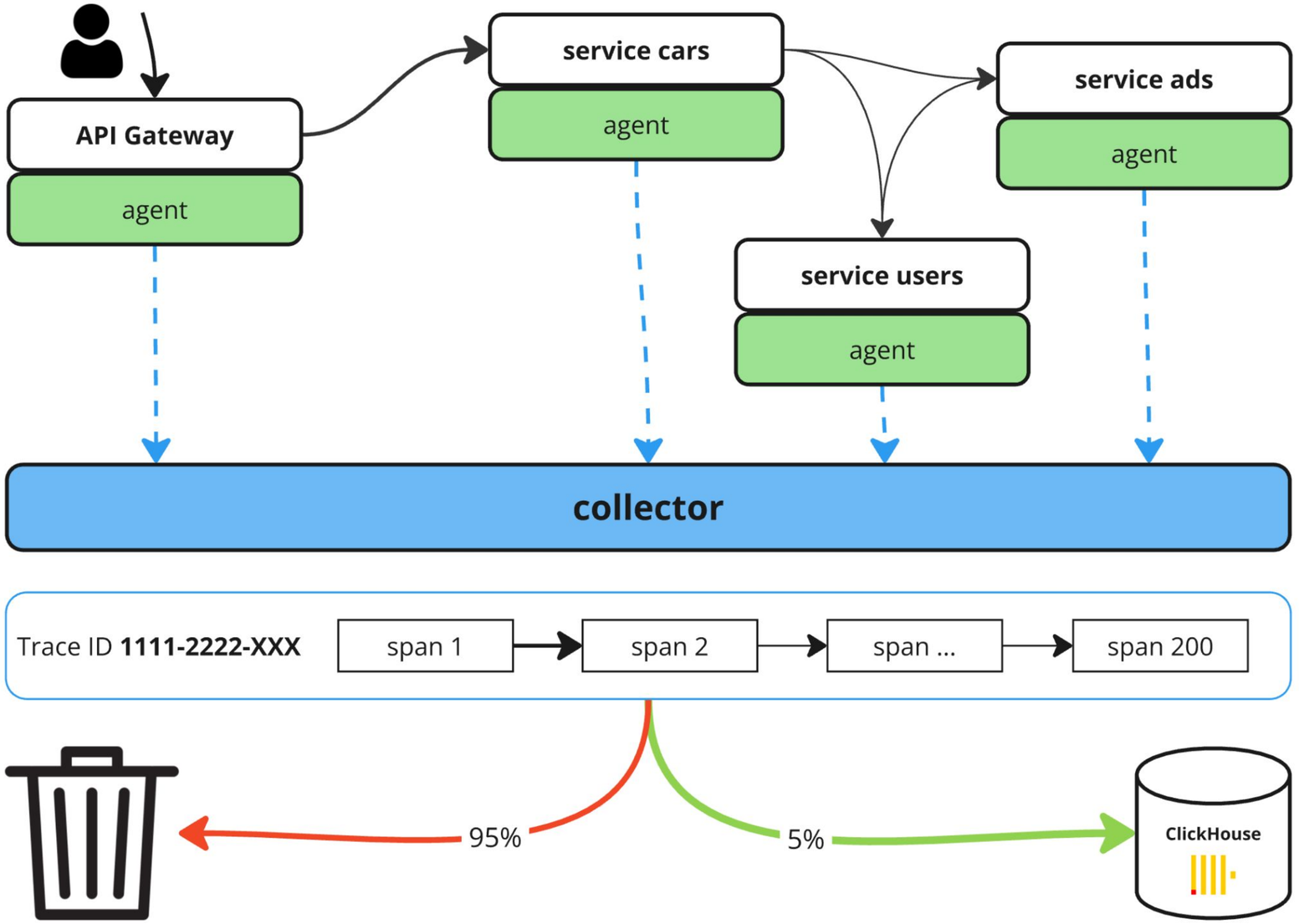
- Агент отправляет события
- Коллектор ждет *все** события по трейсу
- Решение о семплировании

* – коллектор накапливает события в течение N секунд после появления первого события по трейсу

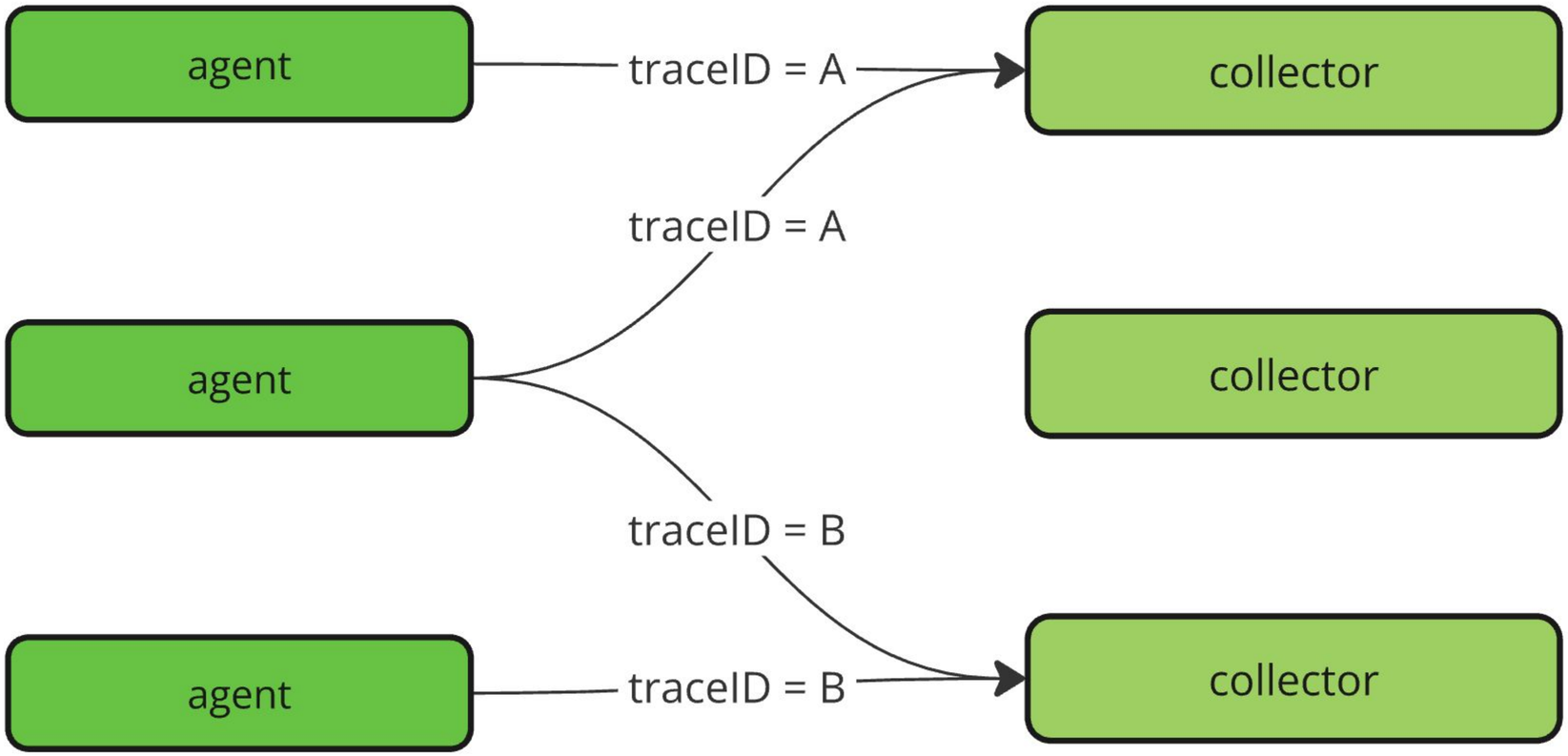


Tail-семплирование

- ❏ Требуется больше памяти
- ❏ Разрывы долгих трейсов: асинхронные и отложенные операции
- ❏ Affinity-балансировка



События по одному Trace ID должны собираться на одном и том же коллекторе

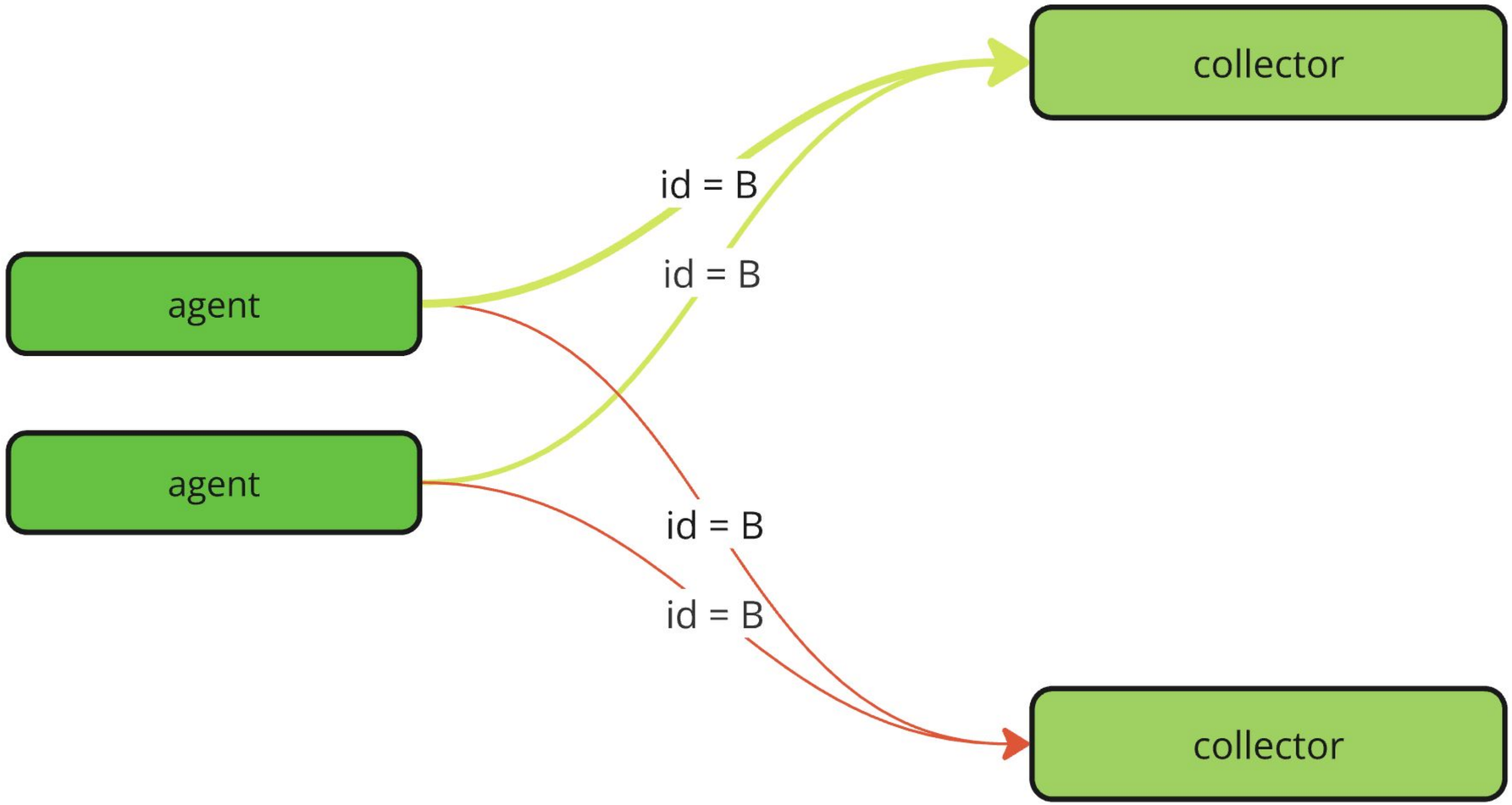


Consistent hashing – не всегда хорошо

+ Приспособлен к изменению набора хостов: меньше потерь данных
– Неравномерность распределения нагрузки
– Сложнее

В нашем случаи – ребалансировка случается редко
hash(id) % N – наш выбор

Affinity и ретраи



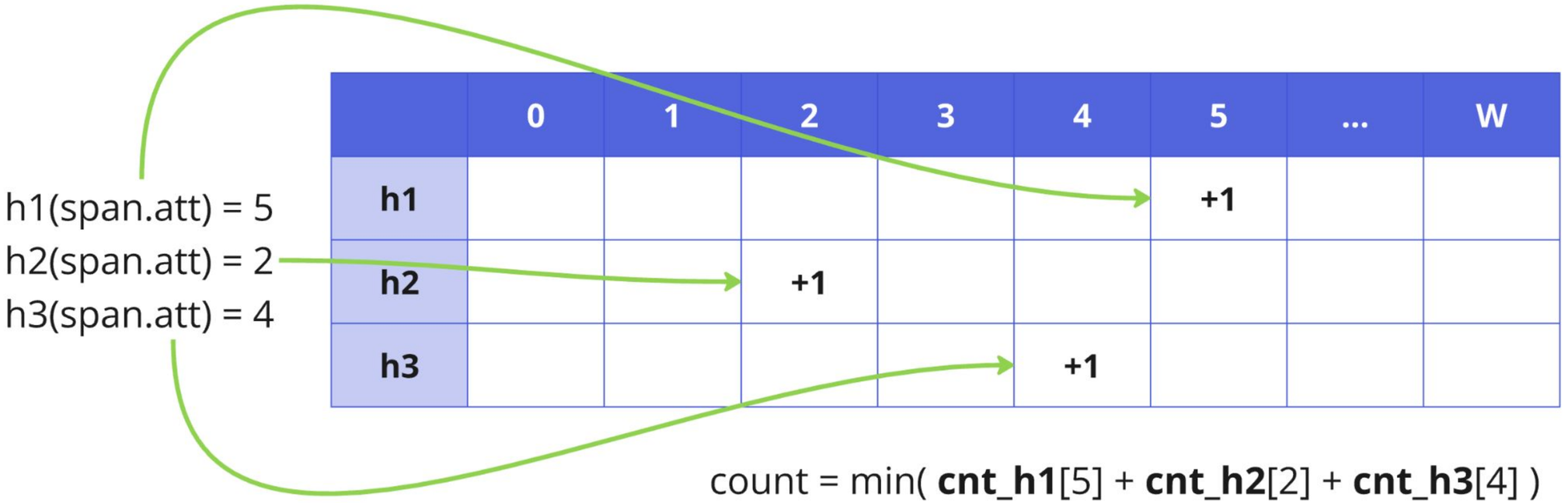
Умное семплирование

Хочется сохранять трейсы из событий с редкими атрибутами (встречаются раз в час)

- ❏ Не требуется много места
- ❏ Растет процент запросов, которые можно отследить

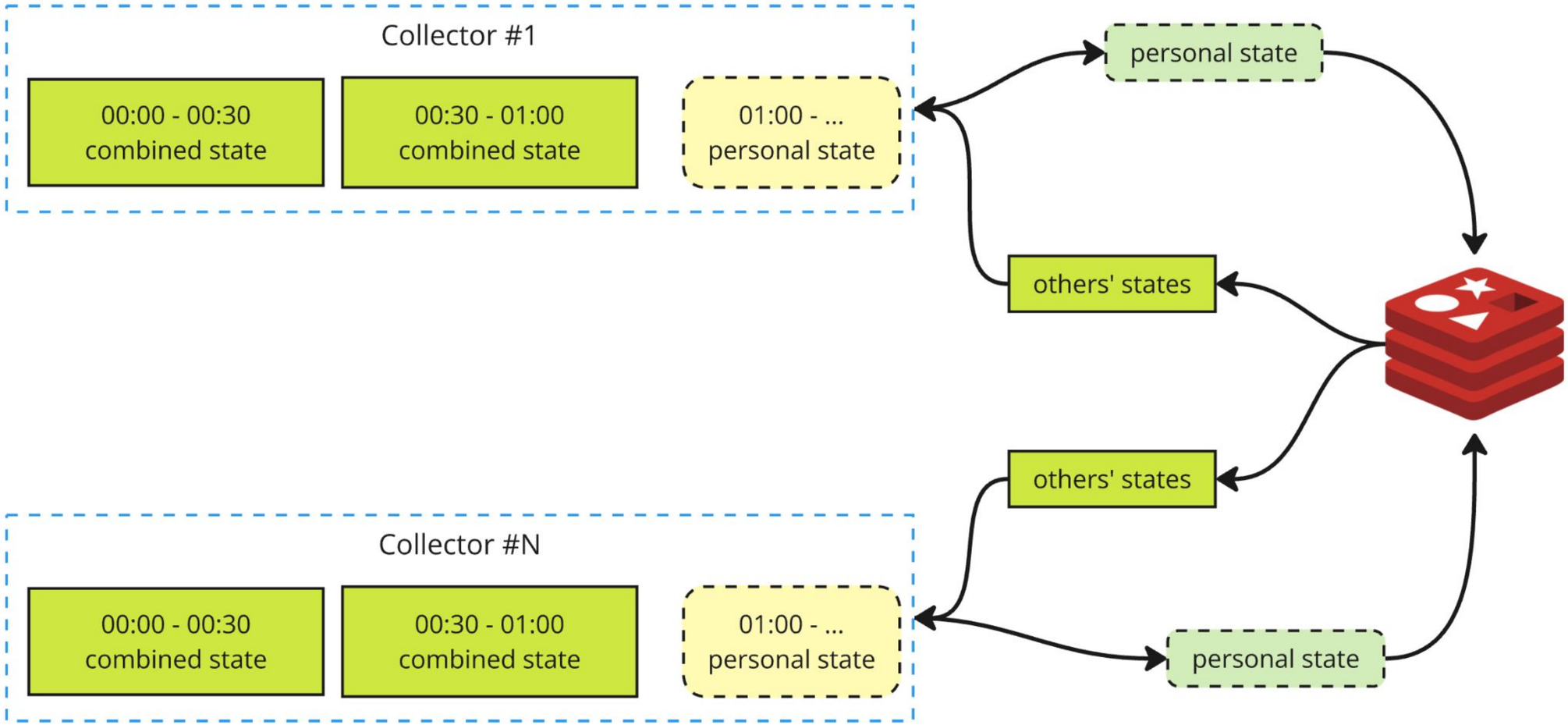
Count-Min Sketch

- Считает число вхождений элемента
- Вероятностный
- Необходимо синхронизировать состояние между коллекторами



Distributed Count-Min Sketch

- Время поделено на фреймы по N минут
- По завершению фрейма коллекторы обмениваются состояниями



Маршрутизация данных

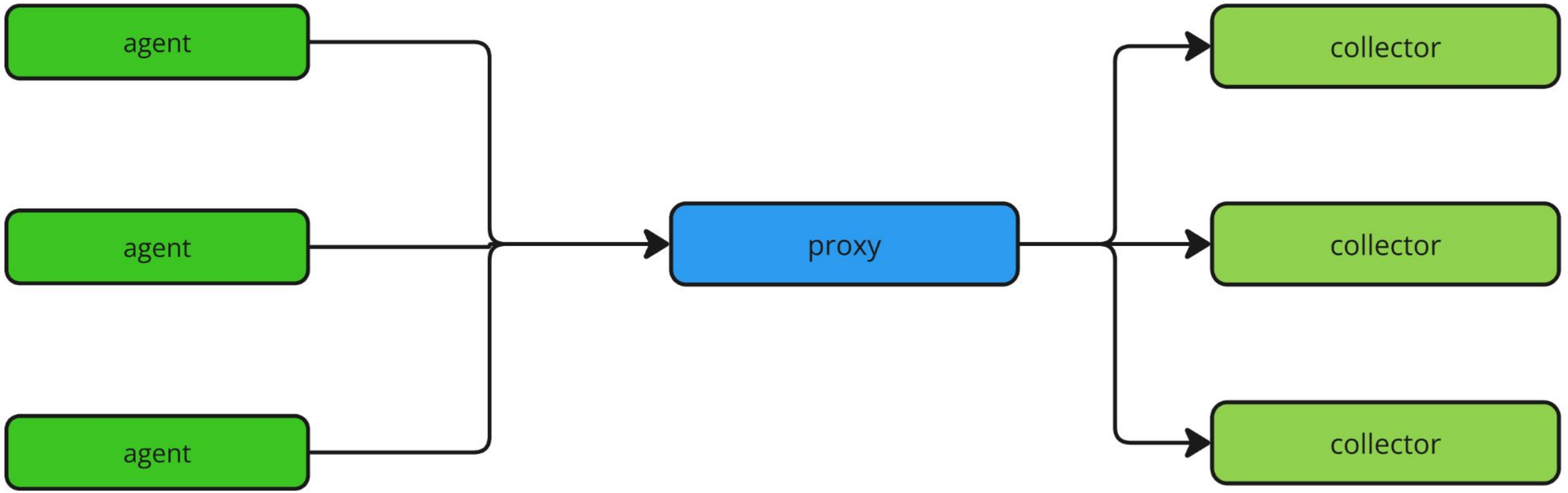
Service discovery

В начале – все агенты знали адреса всех экземпляров коллекторов

- Сложная эксплуатация
- Отказоустойчивость?

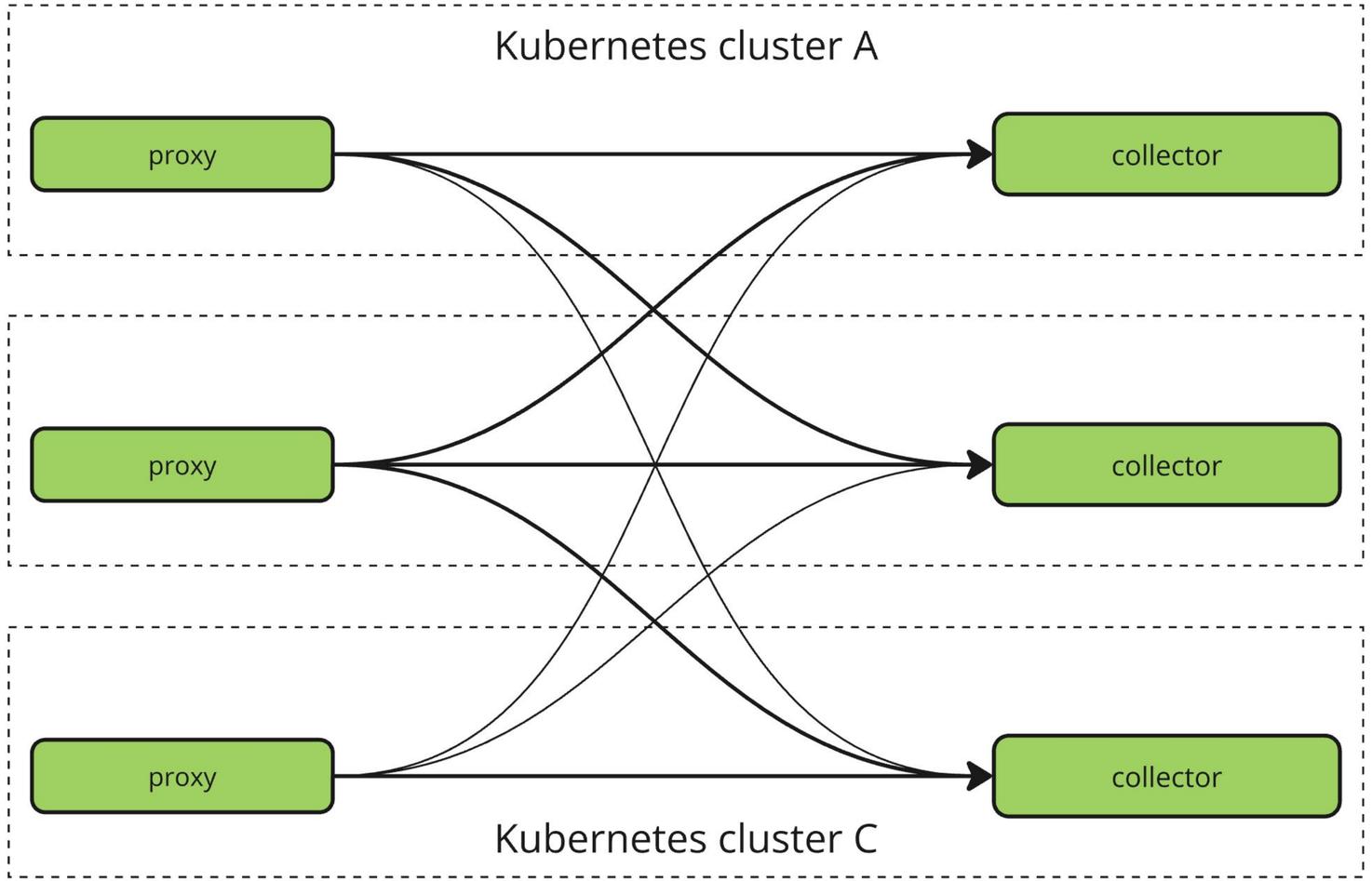
Промежуточный слой

- ❑ Dynamic discovery
- ❑ Health checks
- ❑ Буферизация данных
- ❑ **Незначительный рост потребления ресурсов**



Локальность данных по датацентрам

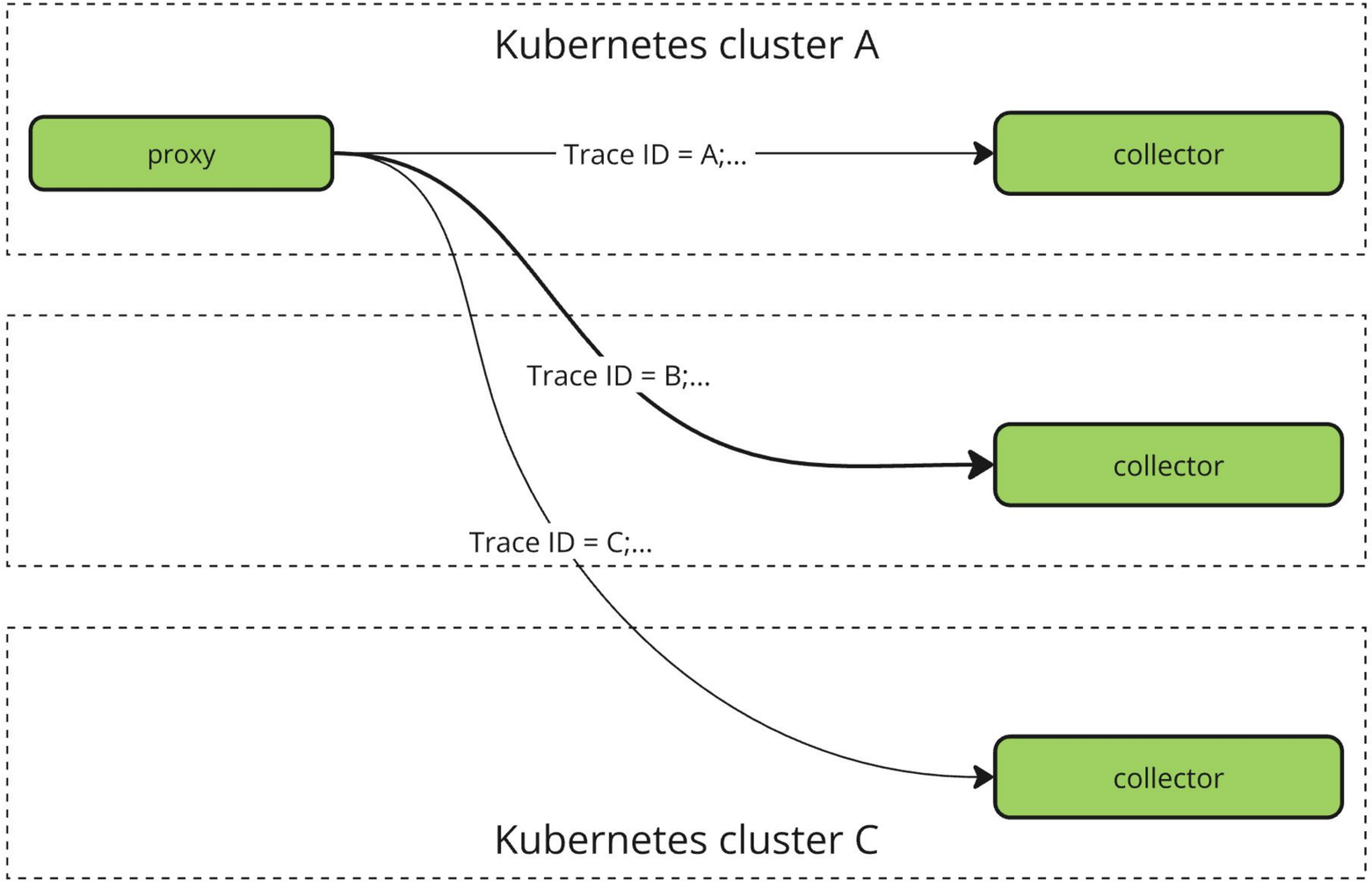
Спаны генерируются в разных дата-центрах, но есть локальность



Локальность данных по датацентрам

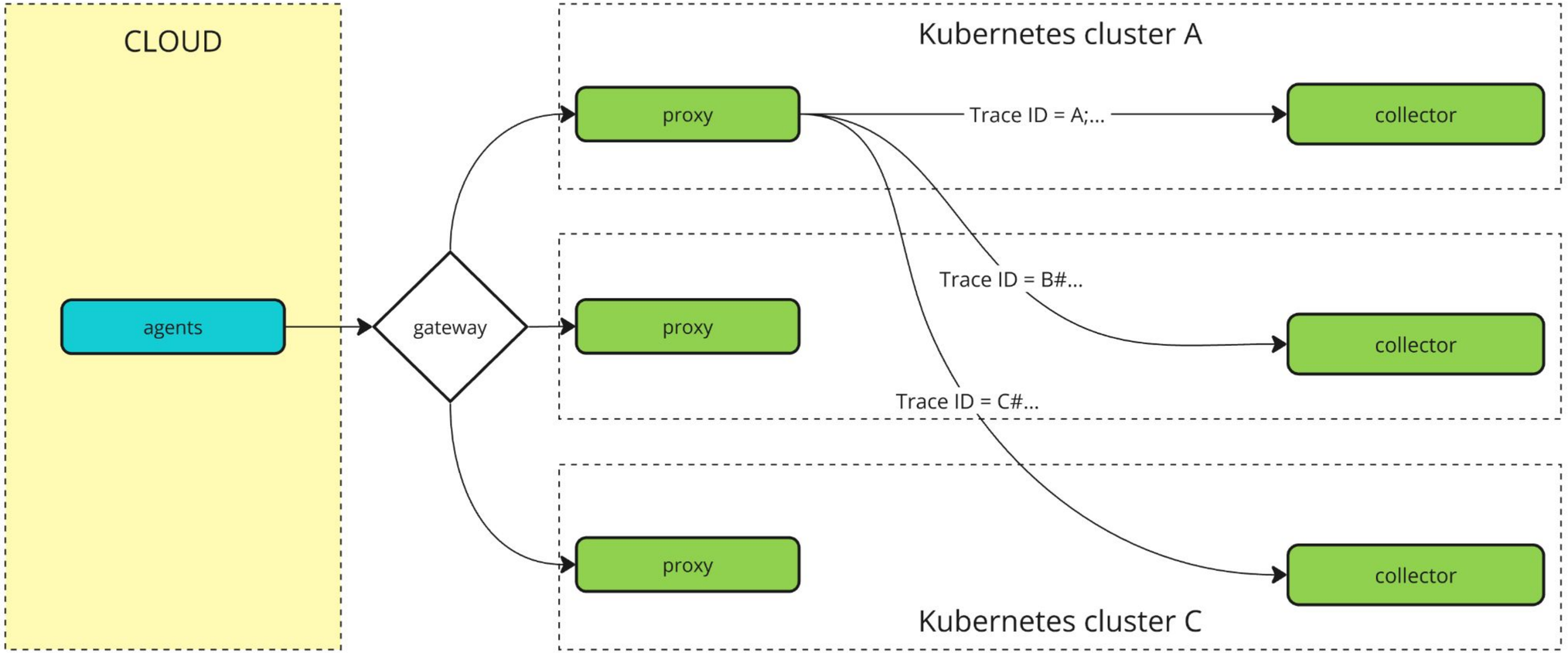
Спаны генерируются в разных дата-центрах, но есть локальность

- ❑ Запишем дата-центр в Trace ID
- ❑ Спаны агрегируются на коллекторах из одного ДЦ
- ❑ **> 90% трафика не выходит из ДЦ**



Локальность данных из облака

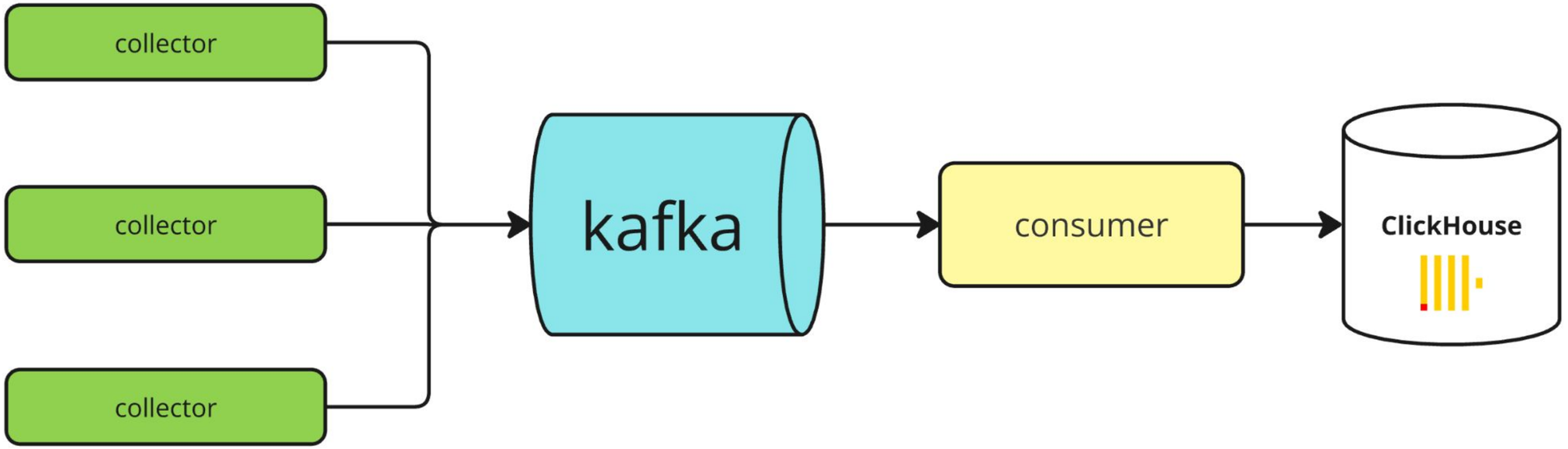
- Трафик дороже
- Вычислительные мощности дорогие



Отправка данных в ClickHouse

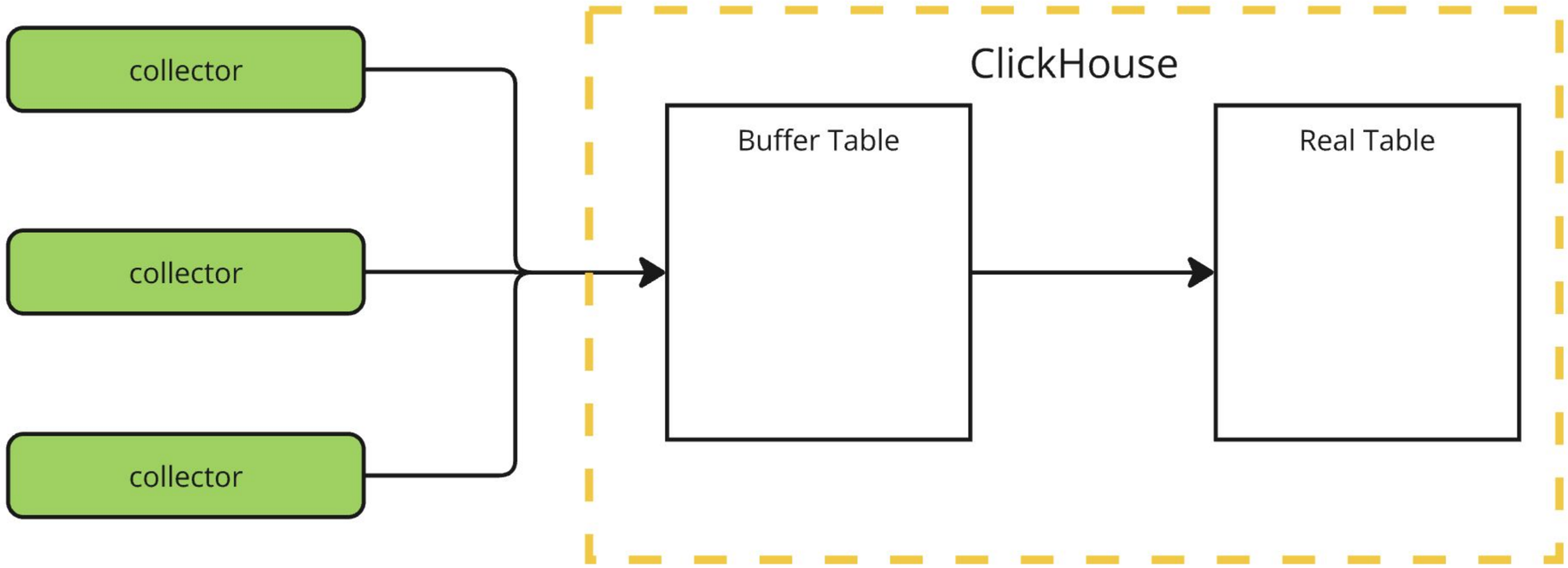
Worker-consumer

- Популярный паттерн
- Воркеры не знают про состояние друг друга
- Сложно регулировать частоту отправки



Buffer Table

- Позволяют делать INSERT часто
- Но недостаточно часто…

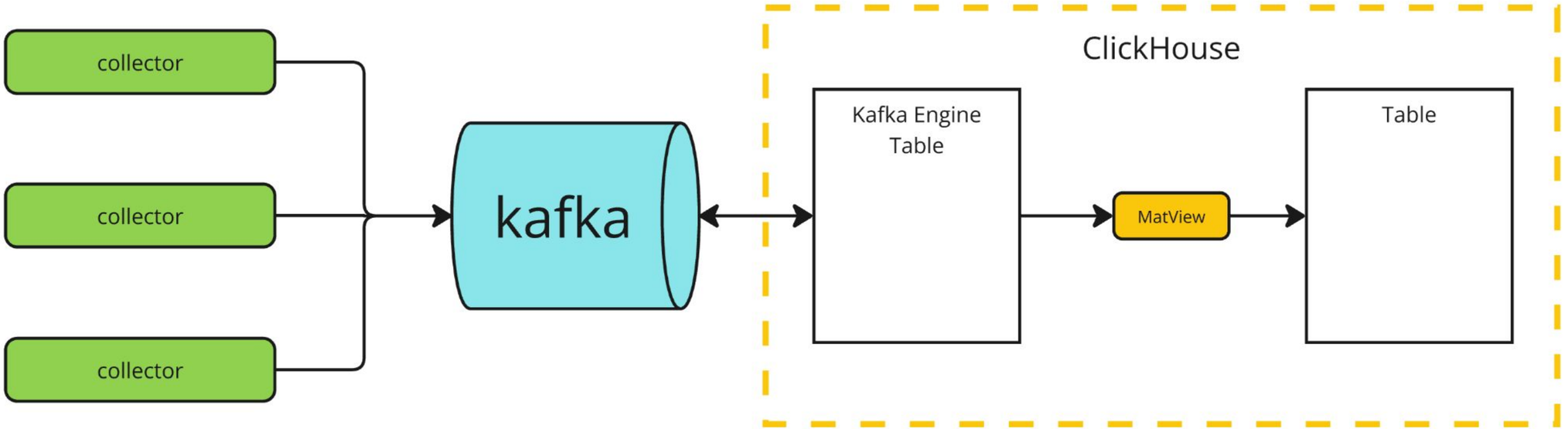


Kafka Engine

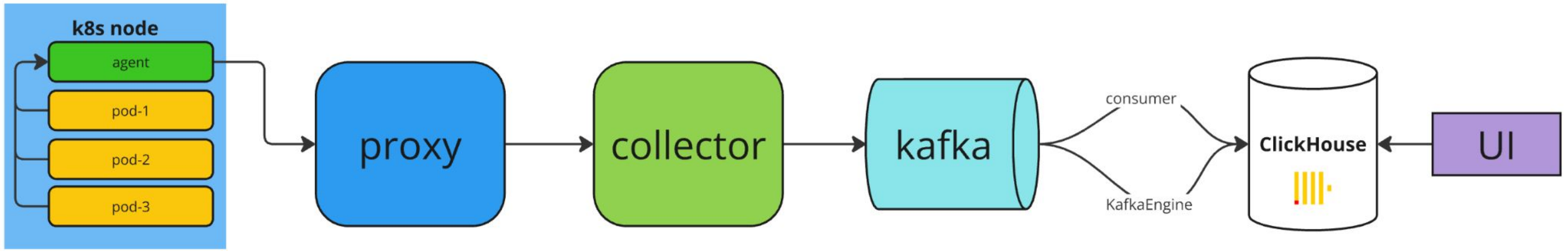
ClickHouse сам прочитает данные с нужной скоростью

Важные параметры:

- `kafka_thread_per_consumer`
- `kafka_num_consumers`



Архитектура целиком

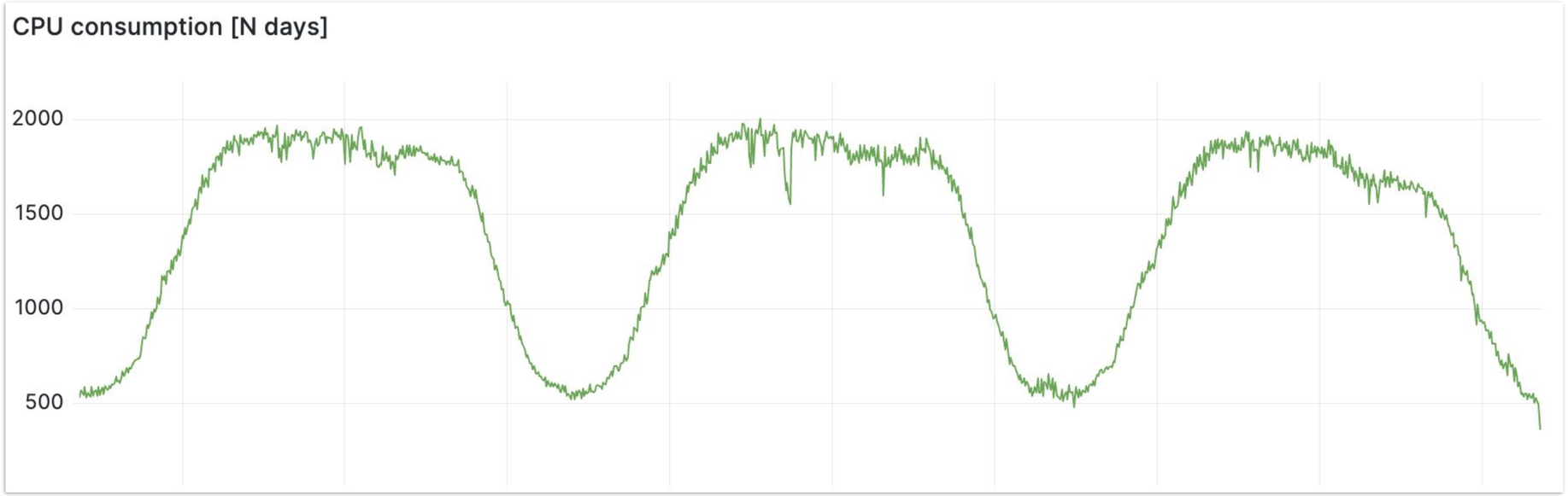


Железо

Без учета хранилища:

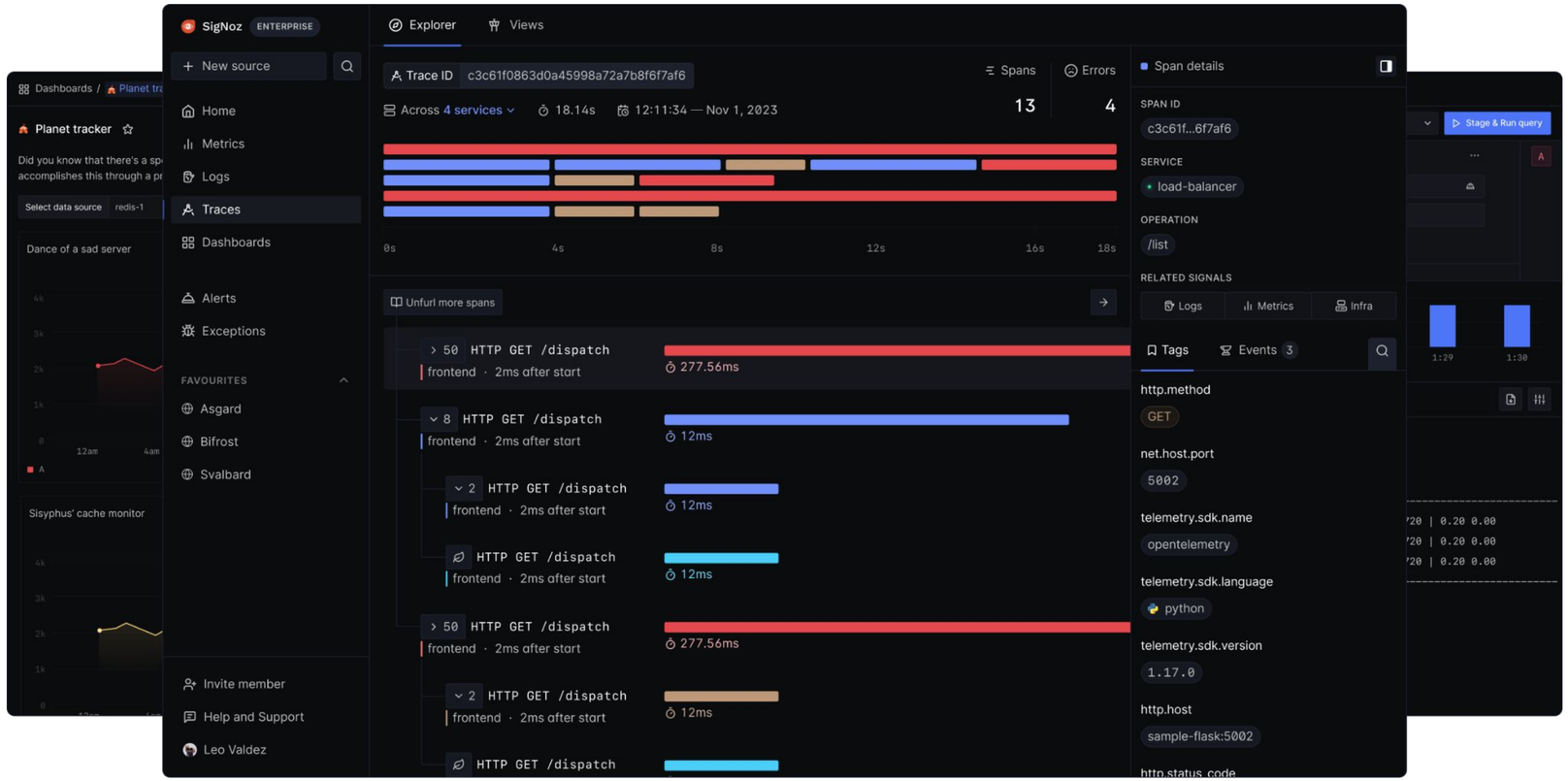
- 4.5 TB RAM
- 2000 CPU

SaaS-продукты стоили бы кратно дороже



DIY: как собрать тоже самое?

- OpenSource инструментов хватит для старта
- OpenTelemetry:
  - github.com/open-telemetry/opentelemetry-collector
  - github.com/open-telemetry/opentelemetry-collector-contrib
- ClickHouse 👍👍👍
- Jaeger, Kibana, Grafana, SigNoz



Выводы

- Бывают системы, где терять данные можно
- Идентификатор может хранить полезную информацию
- Добавление новых компонентов – не всегда плохо

Спасибо!

Что можем обсудить?

- ❏ Особенности оптимизации Kafka Engine
- ❏ Что делать, когда один из слоев не может принять данные?
- ❏ Как выстроить механизм ограничения нагрузки
- ❏ Гибкий retention: кому-то нужен месяц, а кому-то – сутки :)

Игорь Балюк
@lodthe