模块十二：OpenTelemetry 可观测性入门

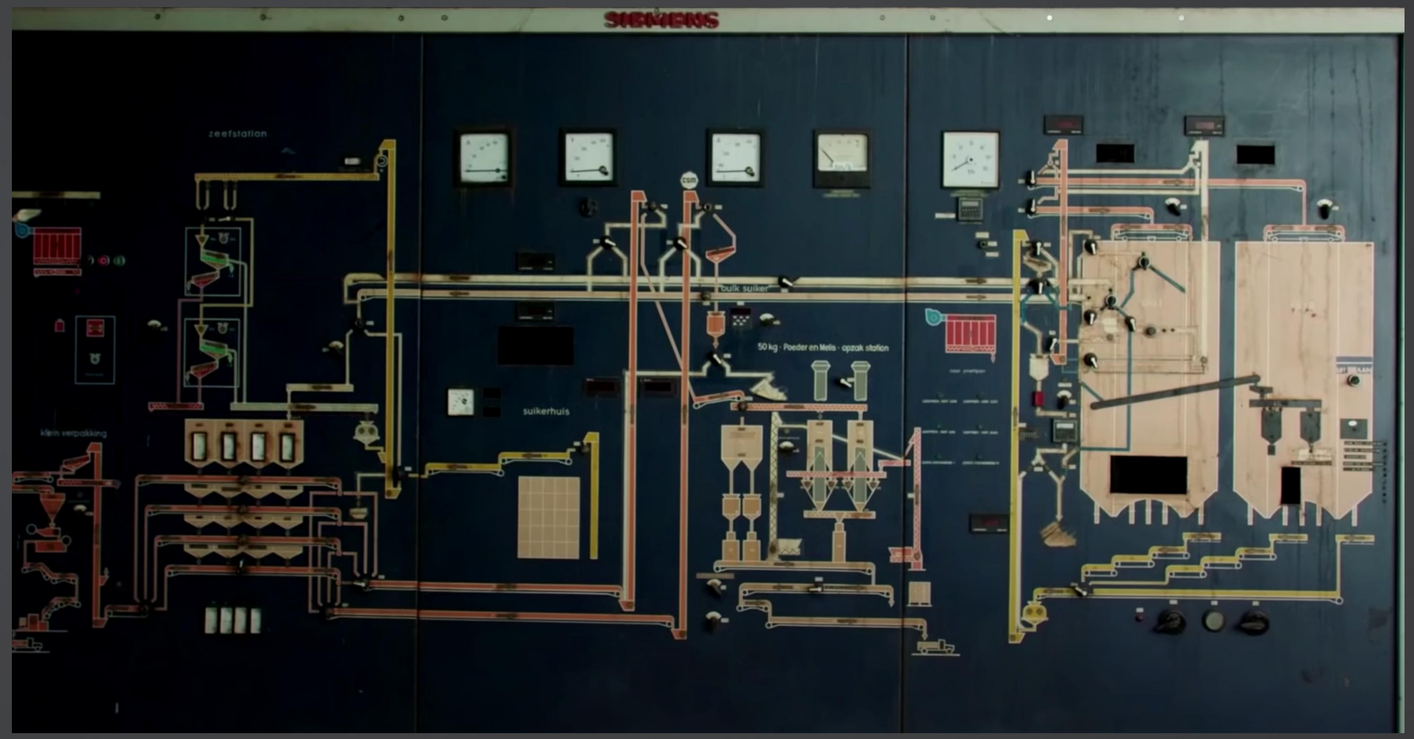
王炜/前腾讯云 CODING 高级架构师

目录

1. 可观测性发展历史

可观测性发展历史

1. 来源于系统控制理论科学
2. 可观测性最早是 1960 年 E·卡尔曼（Rudolf E. Kálmán）提出
3. 在工业制造领域使用最多，例如生产制造环节的“监控大盘”，可以实时查看系统各组件是否运行良好（例如：电机转速、阀门控制、压力、温度）

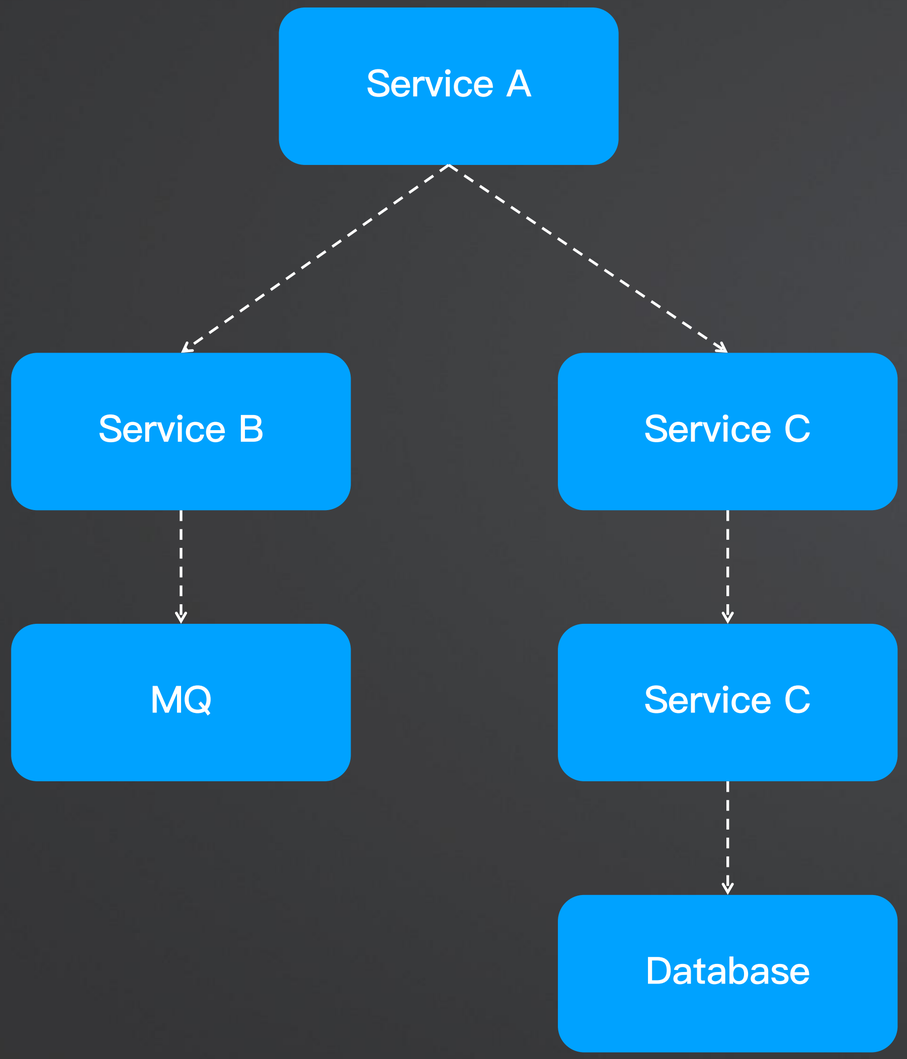


可观测性的本质

单体应用监控

- 通常以 Agent 的方式部署
- 结合日志文件进行告警
- 监控指标通常是系统级的，例如进程数、性能指标、响应时间、CPU、内存等
- 除了开源项目以外还有 APM（应用性能监控） SaaS 服务可选，例如早期的 New Relic、DataDog

微服务应用带来的挑战



- 难以追踪请求的完整链路，当发生性能瓶颈或错误时，难以定位到具体的服务或步骤
- 每个微服务生成自己的日志并分布在多个节点上，导致日志的收集、聚合和分析变得复杂
- 微服务间调用产生的网络延迟、服务依赖关系以及通信故障不易直接观察到
- 通过可观测性系统解决 Traces、Metrics、Logs 的问题

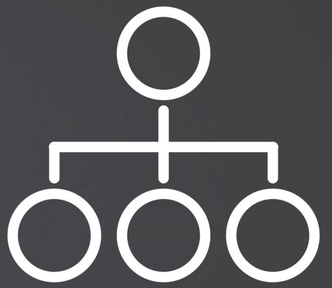
可观测性的三大基础

日志

数据库启动失败

指标

这周的用户平均
请求延迟比上周
慢了 30%



分布式追踪

微服务依赖关系和拓扑

可观测性、监控和 APM 的关系

监控

- 是可观测性的子集
- 收集并展示指标，通常也会一并配合告警

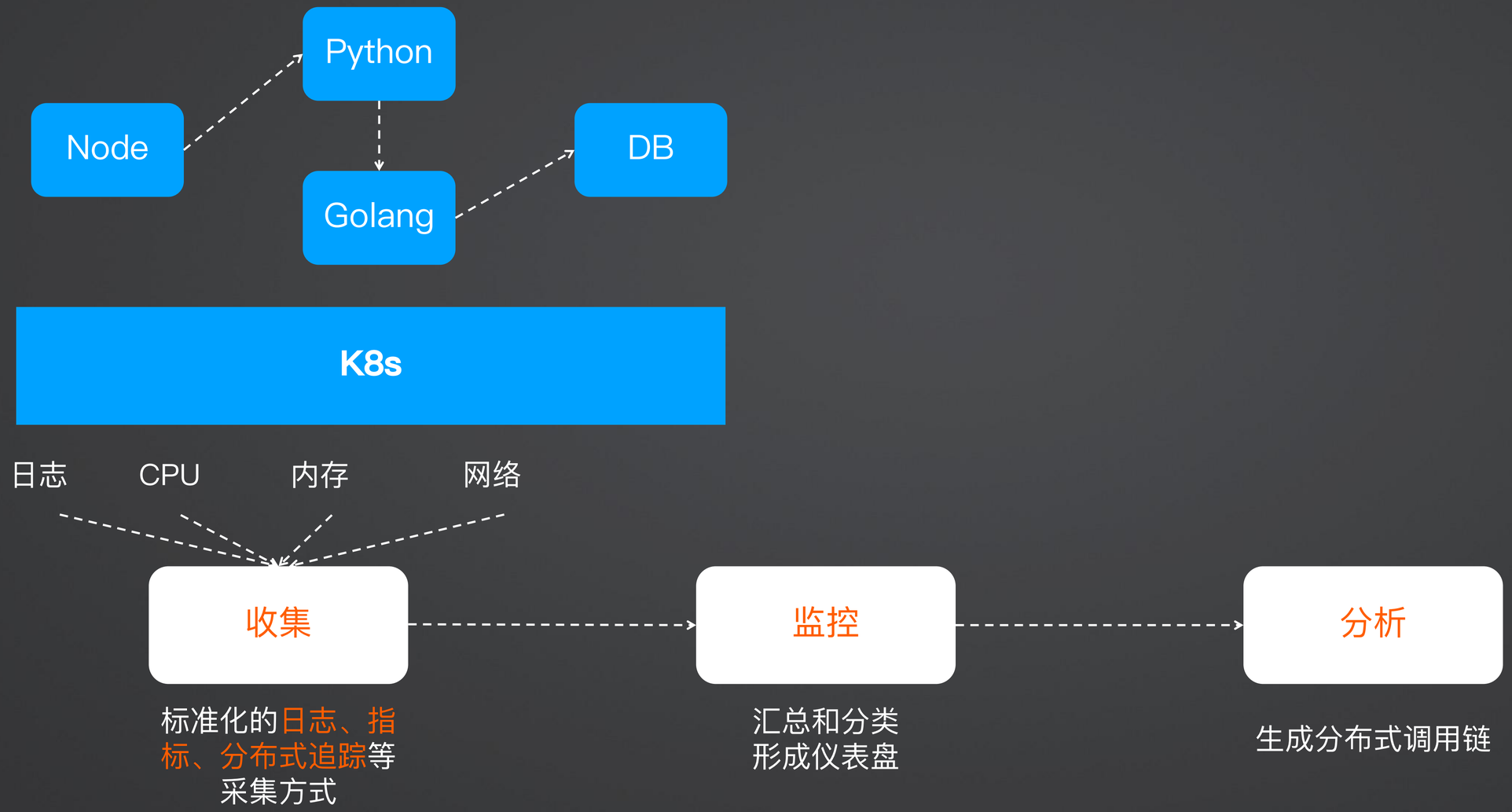
APM

- 单个服务的代码性能分析、事务跟踪、资源使用情况等

可观测性

- 涵盖了所有了解系统状态的工具和实践，将 APM 和监控提升到了下一个阶段
- 包括：日志、监控、分布式追踪
- 目的是提供应用整体的系统视图，尤其适合微服务架构

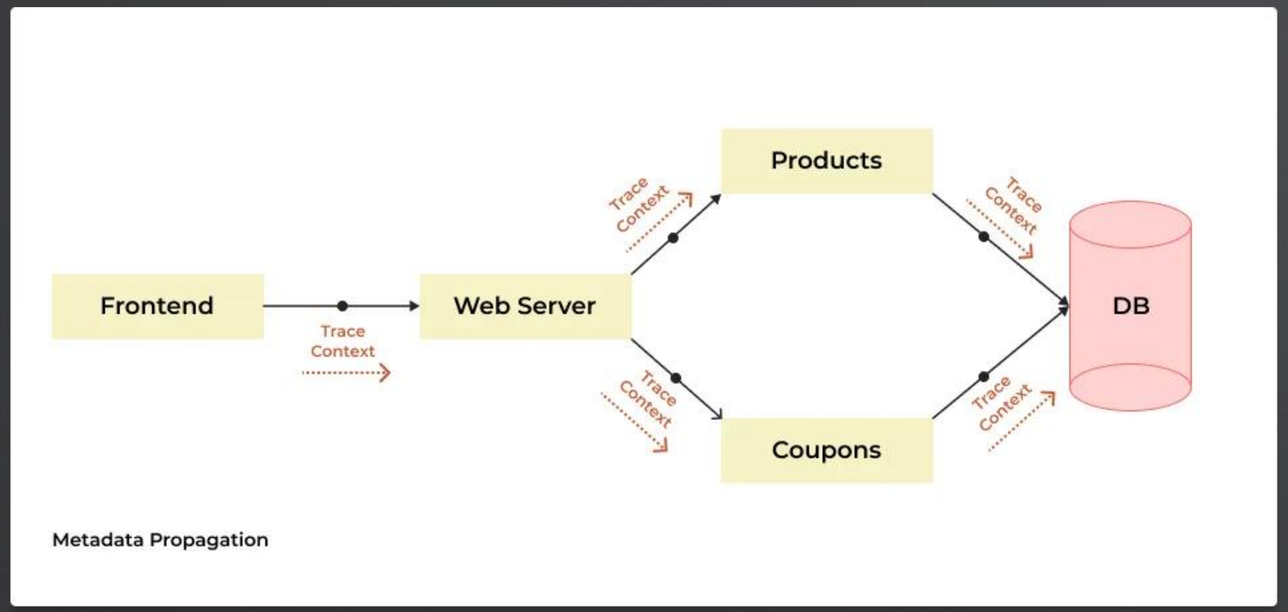
构建可观测性


Python
Node
DB
Golang
K8s
日志
CPU
内存
网络
收集
监控
分析
标准化的日志、指标、分布式追踪等采集方式
汇总和分类形成仪表盘
生成分布式调用链

分布式追踪（Tracing）

提供了一个请求从接收到处理整个生命周期的跟踪路径，一次请求一般会经过 N 个系统，因此也被称为分布式链路追踪

原理：Trace Context（追踪上下文）的传播



- 每个请求的链路中，都伴随着一种叫做 Trace Context 的元数据，作为追踪请求的标识符
- Trace Context 从最初的请求发起方（Frontend）开始生成，并通过链路传递到下游的每个微服务节点和数据库
- 这个追踪上下文记录每个微服务的执行信息，帮助识别和关联整个请求链中的各个操作

Trace Context 的携带方式

HTTP 请求

- 一般放在 HTTP 请求的 Header 内

gRPC

- 放在 Metadata 元数据内（类似于 HTTP 的 Header）

RPC

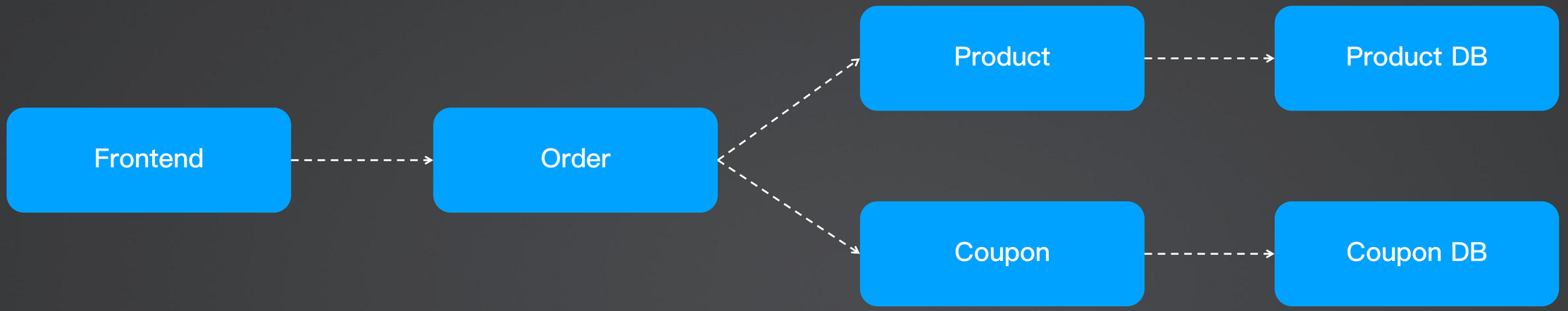
- 自定义字段

Jaeger

Uber 开源的分布式追踪工具，通过 uber-trace-id Header Key 传递上下文

- uber-trace-id 通常包含四个部分，使用（-）分隔
  - 例如 uber-trace-id: {trace-id}:{span-id}:{parent-span-id}:{flags}
- trace-id：唯一标识整个追踪链路的 ID，所有属于同一个请求的 Span 都共享相同的 trace-id
- span-id：每个独立的操作（Span）的唯一标识符，例如微服务访问了其他服务、访问了数据库、访问了消息队列
- parent-span-id：当前 Span 的父 Span 的 ID（对于根 Span，通常是 0 或不使用）
- flags：标志位，用于传递追踪选项，比如是否采样（sampled）

实际例子




```
Span 1 (Frontend HTTP Request) Trace，由 6 个 Span 组成
  └── Span 2 (Order Service Processing) Span 3、4、5、6 的父 Span
        ├── Span 3 (Product Service Query) Span 5 的父 Span
        │       └── Span 5 (Product Service DB Query)
        └── Span 4 (Coupon Service Check)  Span 6 的父 Span
                └── Span 6 (Coupon Service DB Query)
```

Zipkin

Twitter 开源的分布式追踪工具，通过 X-B3 Header Key 传递上下文

- X-B3-TraceId：唯一标识整个追踪链路的 ID，所有属于同一个请求的 Span 都共享相同的 trace-id
- X-B3-SpanId：每个独立的操作（Span）的唯一标识符，例如微服务访问了其他服务、访问了数据库、访问了消息队列
- X-B3-ParentSpanId：当前 Span 的父 Span 的 ID（仅当存在父级调用时才会存在这个 Header）
- X-B3-Sampled：采样标识，可以是 0（不采样） 和 1（采样）
- X-B3-Flags：标志位，是否为调试模式
- b3 (单一头)：Zipkin 还支持将所有 B3 追踪信息压缩到一个单一的 b3 头中，按照 traceid-spanid-parentspanid-sampled 组合

W3C Trace Context

W3C 标准

- traceparent：version–trace–id–span–id–flags 组成
  - version（2位）：表示 Trace Context 规范的版本号（当前为 00）
  - trace–id（32位）：一个唯一标识整个分布式追踪的 ID
  - span–id（16位）：当前请求的 span ID
  - flags（2位）：用来传递追踪相关的标志位
- tracestate：可选，用来携带额外的元数据，例如 key1=value1,key2=value2

Metrics 和 Logs

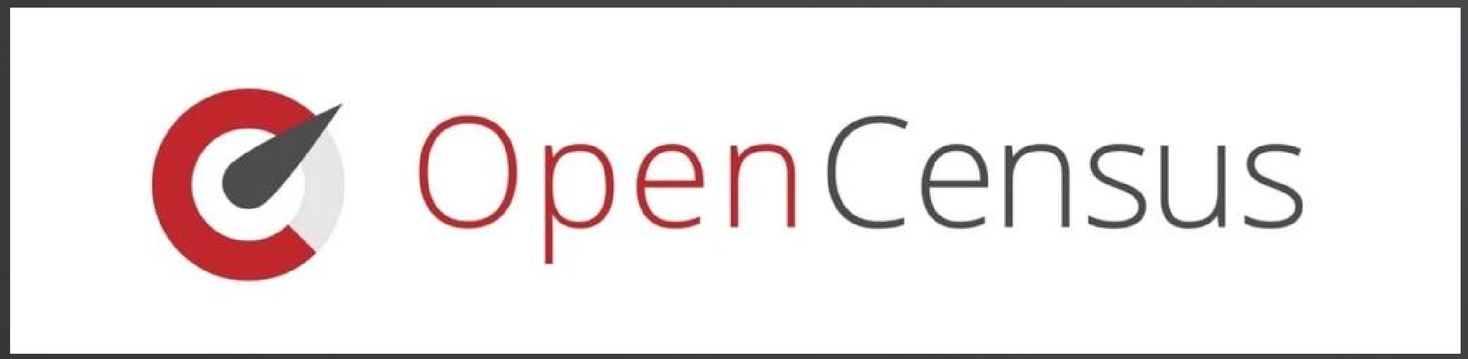
- Metrics：例如 CPU 使用率、请求延迟、用户访问数等的 Counter、Gauge、Histogram（直方图） 指标
  - 一般使用 Prometheus 采集、存储和查询指标
- Logs：服务输出的日志
  - 可选 EFK、ELK、Loki 等日志系统采集和存储日志

大一统的前期

- Tracing 无统一标准，采集需要集成单独的 SDK 和工具
- Tracing 的标准也不同
- 2016 年 11 月 OpenTracing 进入 CNCF，希望能统一 Tracing 的标准
- 不必被具体的追踪系统（如 Jaeger、Zipkin 或 LightStep）所绑定

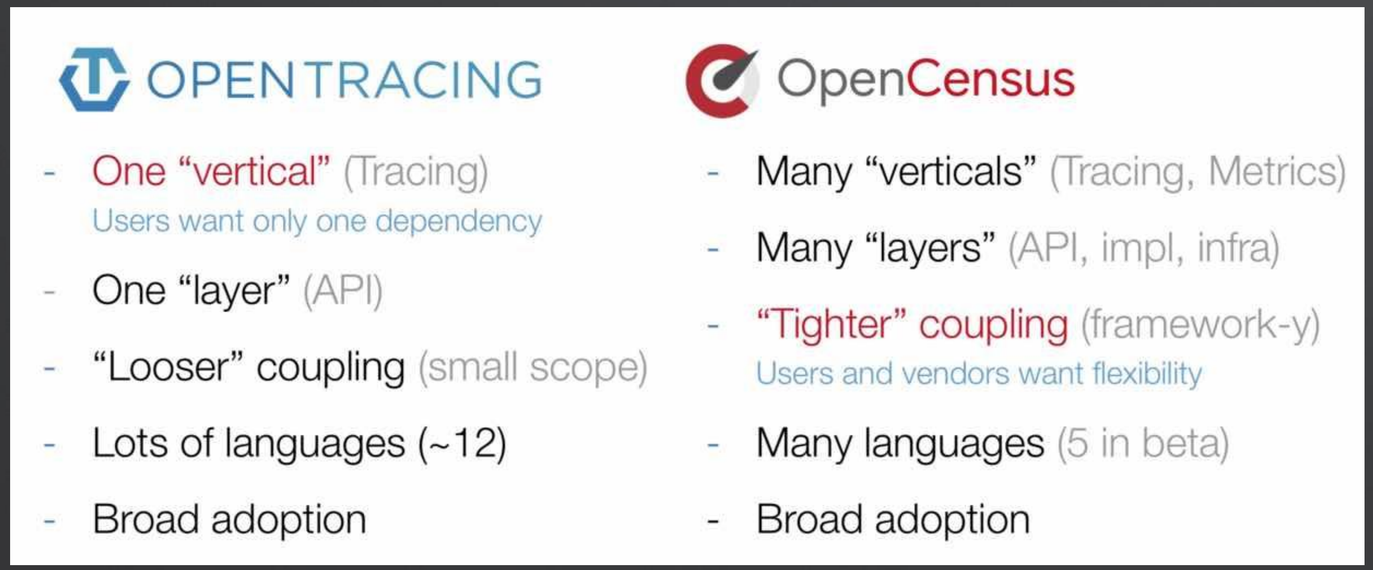
OpenCensus

- Google 开源
- 希望能统一 Tracing 和 Metrics 标准
- 微软随后加入了项目



OpenTracing VS OpenCensus

- 维度：OpenTracing 只专注追踪，OpenCensus 涵盖追踪和指标
- 单层 VS 多层：OpenTracing 主要专注于一个API 层，不关注实现细节。OpenCensus 则引入了多层架构，不仅包含 API 层，还包含实现层和基础设施层
- 松耦合 VS 紧耦合：OpenTracing 是一种松耦合的设计，允许开发者选择不同的追踪实现（Jaeger、Zipkin）。OpenCensus 则更紧耦合，它提供了一个更“框架化”的解决方案
- 语言支持：OpenTracing 支持大部分语言，OpenCensus 则支持较少的语言



进入大一统时代：OpenTelemetry

- OpenTracing 和 OpenCensus 存在功能重合和竞争，过于消耗开发资源，并且对社区不友好
- 为了统一标准，节约开发资源，Google 和微软共同提出合并这两个项目，于是诞生了新的 OpenTelemetry 项目
- 自此实现了 Traces、Metrics、Logs 的大统一时代

2. OpenTelemetry 简介

什么是 OpenTelemetry

- 是一个开源的、供应商中立的可观测性框架，旨在提供应用程序的分布式追踪、指标和日志的统一收集和导出功能
- OpenTelemetry 提供了统一的API 和 SDK，它的核心目标是简化可观测性数据的跨语言和跨平台收集

设计目标：

- 跨语言支持：它为多种编程语言提供 SDK，允许在不同技术栈中实现一致的数据采集
- 后端无关性：数据可以发送到各种后端系统，如 Jaeger、Prometheus、Grafana、Elastic 等，用户可以自由选择后端平台
- 自动化和上下文传播：自动采集分布式系统中的上下文信息，通过 HTTP headers 等机制传递追踪信息，减少开发者手动管理的复杂度

OpenTelemetry 不考虑数据如何去使用、存储、展示、告警等

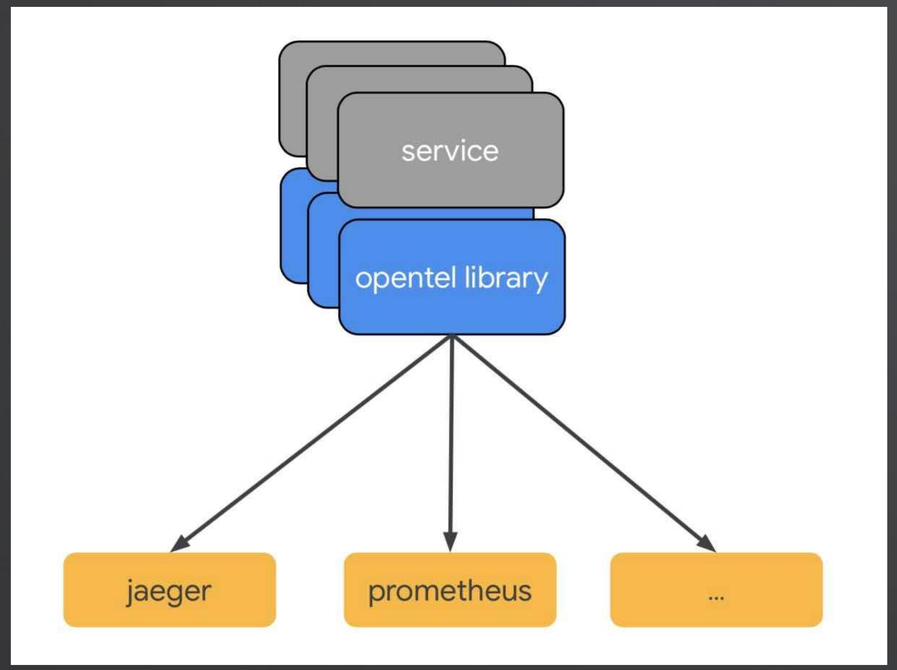
一般需搭配的项目

- Metrics：Prometheus + Grafana
- Logs：Loki、EFK
- Tracing：Jaeger、Zipkin、Tempo

Tips：使用 Grafana 全家桶（Tempo、Prometheus、Loki、Grafana），可实现在一套系统里查询追踪、指标和日志

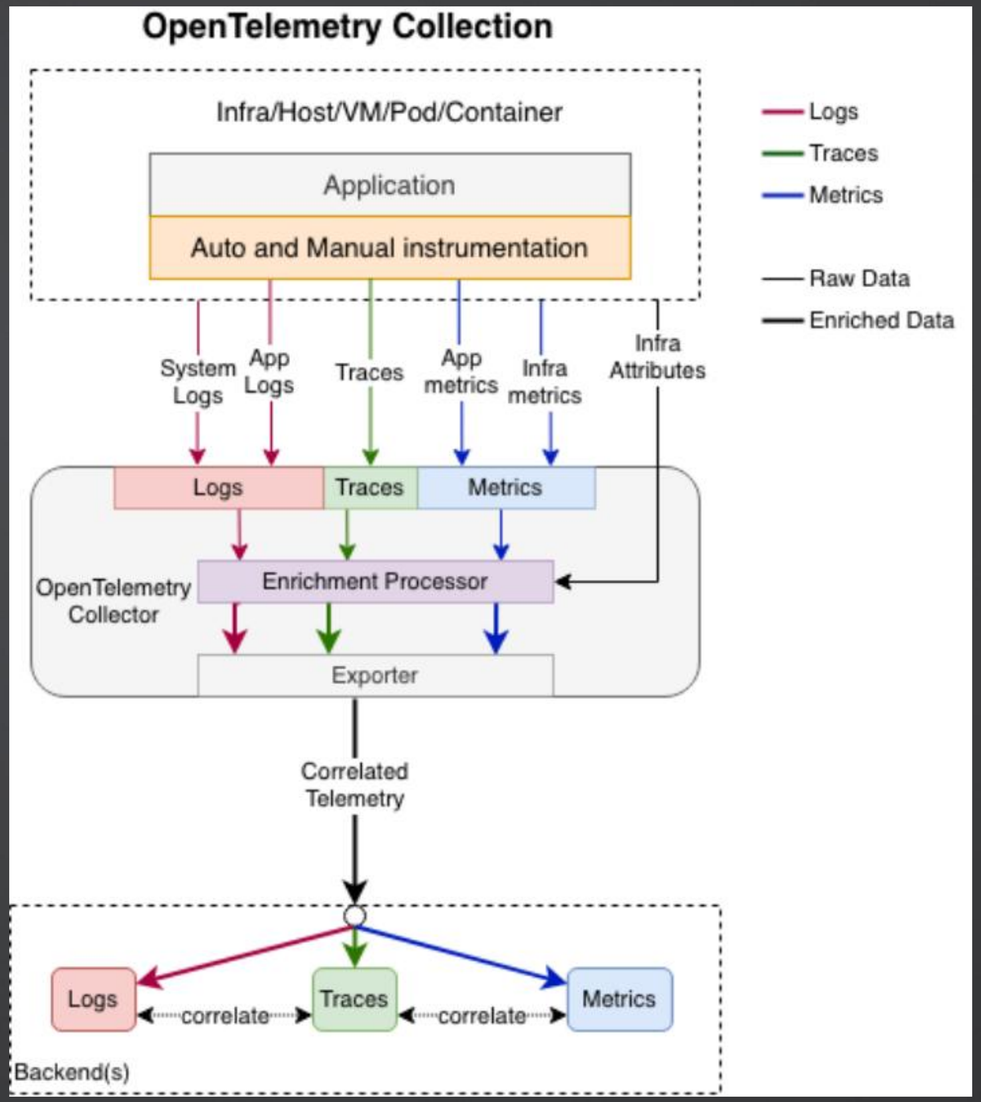
将可观测数据发送至后端

- 由服务集成 OTel SDK，并将可观测数据直接发送至后端
- 后端如：Jaeger、Prometheus 等



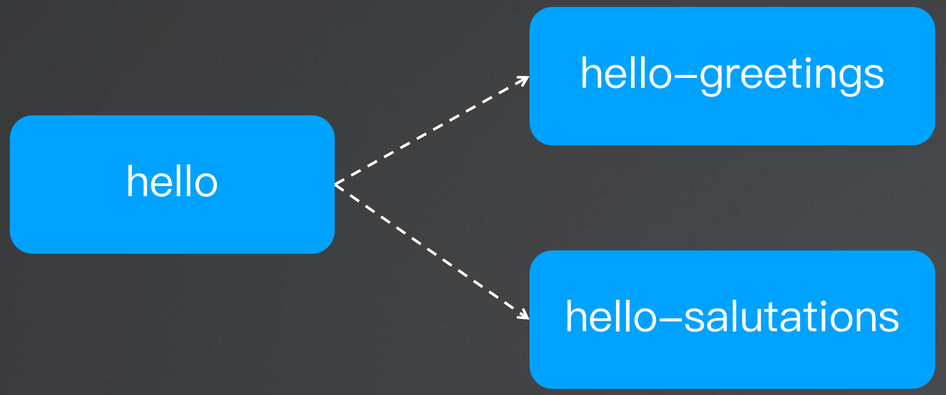
通过 Collector 发送至后端

- 借助自动集成或集成 SDK 的方式，将数据直接发送至 OpenTelemetry Collector 进行处理、转化和导出
- 导出至 Jaeger、Prometheus 等后端



3. OpenTelemetry 数据格式

Traces 数据格式（Span）




```
{
  "name": "hello",
  "context": {
    "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "051581bf3cb55c13"
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "Guten Tag!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```


- context
- attributes
- events
- links
- status


```
{
  "name": "hello-greetings",
  "context": {
    "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "5fb397be34d26b51"
  },
  "parent_id": "051581bf3cb55c13",
  "start_time": "2022-04-29T18:52:58.114304Z",
  "end_time": "2022-04-29T22:52:58.114561Z",
  "attributes": {
    "http.route": "some_route2"
  },
```



```
{
  "name": "hello-salutations",
  "context": {
    "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "93564f51e1abe1c2"
  },
  "parent_id": "051581bf3cb55c13",
  "start_time": "2022-04-29T18:52:58.114492Z",
  "end_time": "2022-04-29T18:52:58.114631Z",
  "attributes": {
    "http.route": "some_route3"
  },
```

Context

- Context 是每个 Span 的不可变对象，包括以下内容：
  - TraceID
  - Span ID
  - 跟踪标志，包含有关跟踪信息
  - 跟踪状态，可以携带特定供应商的跟踪信息的兼职对


```
{
  "name": "hello",
  "context": {
    "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "051581bf3cb55c13"
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "Guten Tag!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```

attributes

- attributes（属性）是包含元数据的键值对，用来携带追踪的操作信息，例如 route、port、target、host 等：
- 可以在 Span 创建期间或创建之后添加属性，属性具有以下规则：
  - 键必须是非空字符串值
  - 值必须是非空字符串、布尔值、浮点值、整数或这些值的数组

```
"attributes": {
  "net.transport": "IP.TCP",
  "net.peer.ip": "172.17.0.1",
  "net.peer.port": "51820",
  "net.host.ip": "10.177.2.152",
  "net.host.port": "26040",
  "http.method": "GET",
  "http.target": "/v1/sys/health",
  "http.server_name": "mortar-gateway",
  "http.route": "/v1/sys/health",
  "http.user_agent": "Consul Health Check",
  "http.scheme": "http",
  "http.host": "10.177.2.152:26040",
  "http.flavor": "1.1"
},
```

events

- event 可以认为是 span 的结构化日志消息，通常用来表示 Span 持续时间内的单一时间点事件
- 例如：
  - 页面何时变为可交互的


```
{
  "name": "hello",
  "context": {
    "trace_id": "5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "051581bf3cb55c13"
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "Guten Tag!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```

links

- 将一个 span 与另一个 span 关联起来，注意和 Parent Span ID 区分，links 主要面对异步场景
- Parent Span ID 用来构建层级（父子关系），而 links 用来建立 span 之间的相关性
- 例如：
  - 消息队列场景：当消息队列中的一条消息由多个不同的消费者消费时，可以通过 Span Links 来表达这些消费者之间的关联性，而不是 Parent Span ID 的父子关系

status

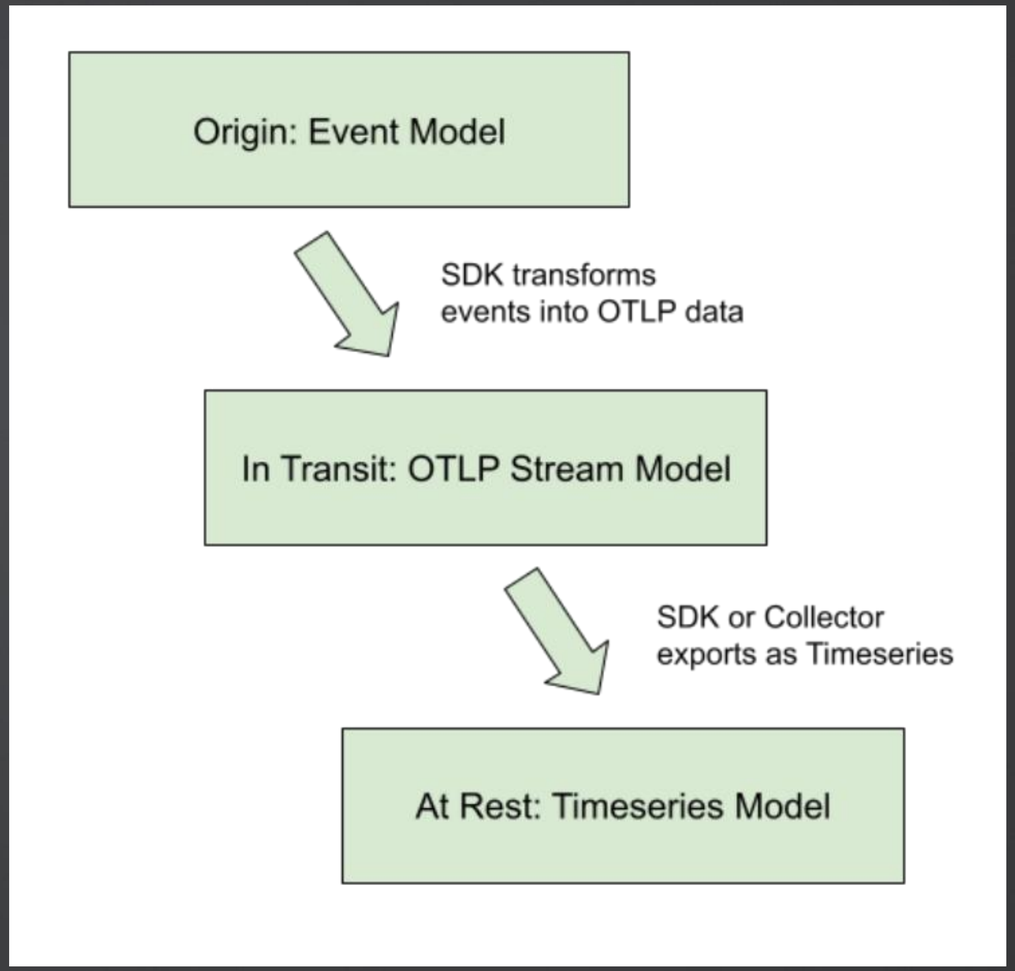
- 每一个 span 都有一个状态，三个可能的值为
  - Unset：默认值为 Unset，表示没有成功完成并且没有错误
  - Error：表示跟踪过程产生了错误，例如处理请求的过程产生了 500 错误
  - OK：意味着被开发人员明确标记为无错误

Metric Instruments

- 名称
- 种类
  - Counter：只增加
  - Asynchronous Counter：每次导出时收集一次，通过聚合访问
  - UpDownCounter：累计值，可以增加和减少
  - Asynchronous UpDownCounter：每次导出时收集一次，通过聚合访问
  - Gauge：度量当前值的指标，衡量某个时刻的状态，异步
  - Histogram：直方图，例如请求延迟

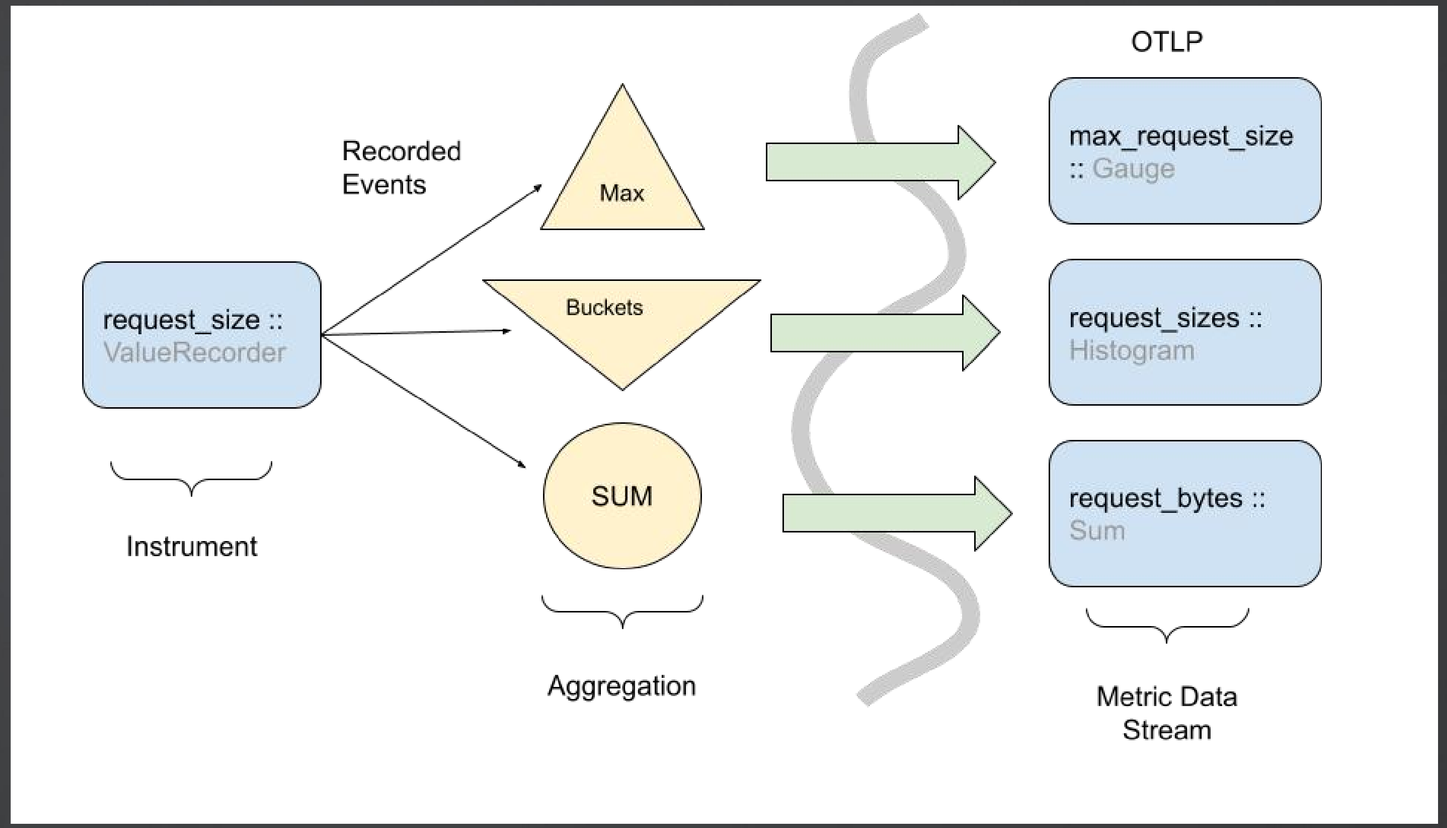
Metric 数据模型

- Origin: Event Model （事件模型）
  - 事件模型是数据的最初形态
- In Transit: OTLP Stream Model（OTLP 流模型）
  - OTLP 是 OpenTelemetry 的通信协议，这个阶段数据被打包为一种流式数据模型，并传输到监控系统
- At Rest: Timeseries Model（时间序列模型）
  - 时间序列的形式存储，是数据在存储时的最终形态



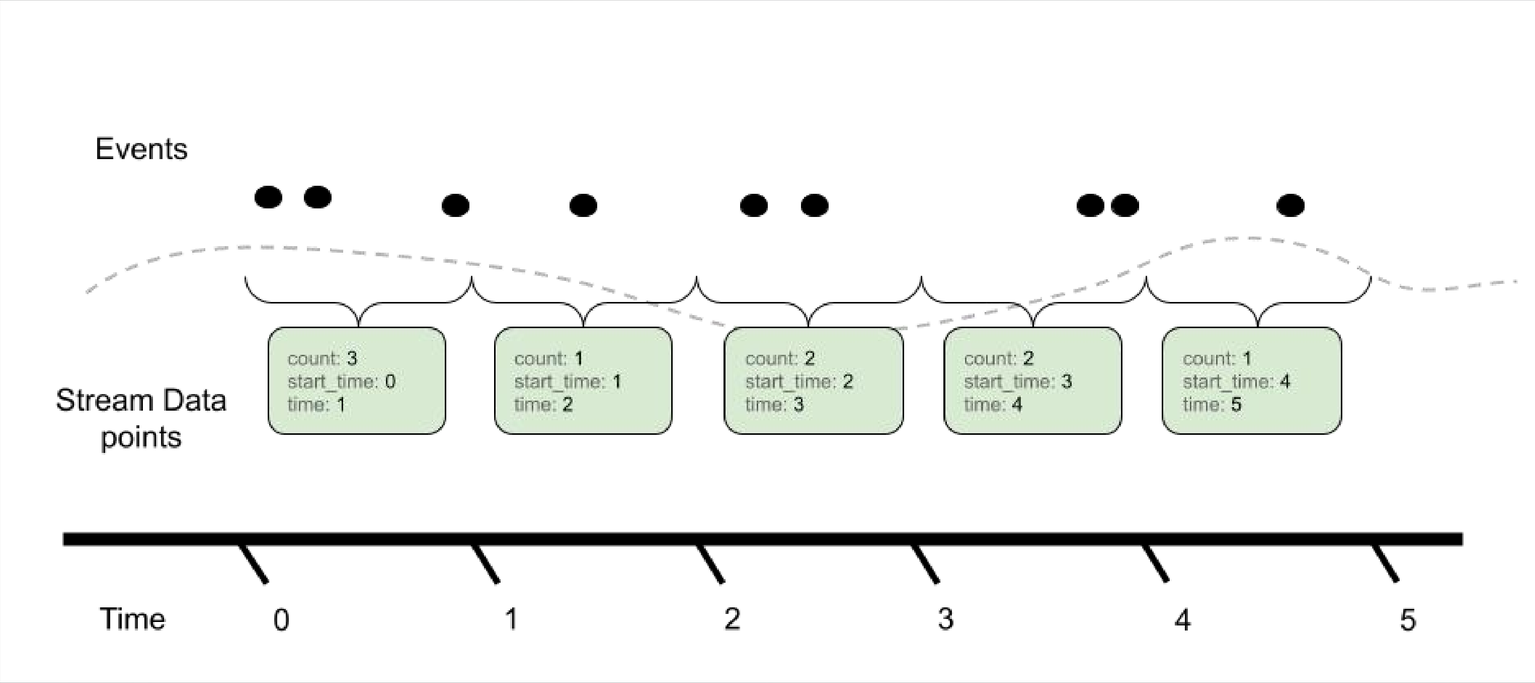
事件模型

- OpenTelemetry 的 ValueRecorder 捕获指标
- 进行不同方式的统计和处理
  - Max：提取最大值
  - Buckets：分桶
  - Sum：求和
- 不同的统计方式会生成不同的指标
  - max_request_size :: Gauge
  - request_sizes :: Histogram
  - request_bytes :: Sum



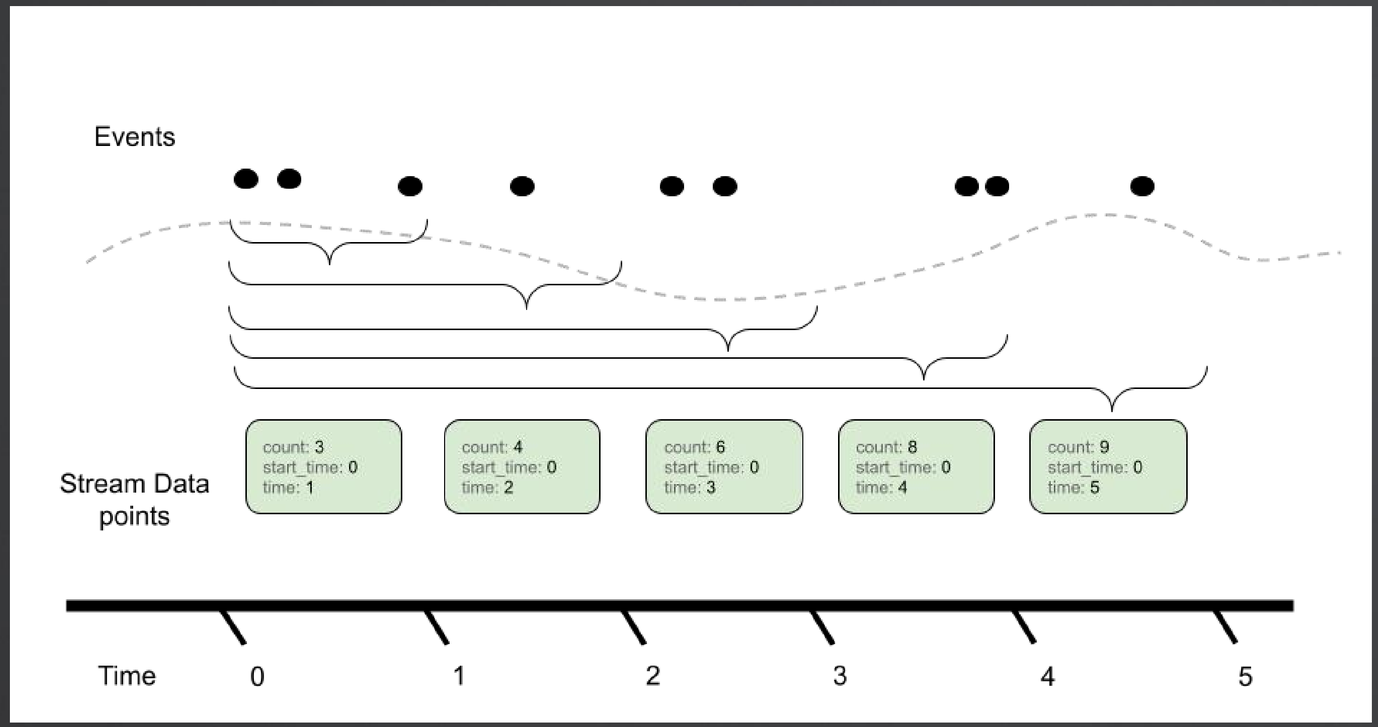
Sum： Delta

- 计算一段窗口的总和
- (start, end] 作为时间窗口，计算时时间窗口不会产生重叠



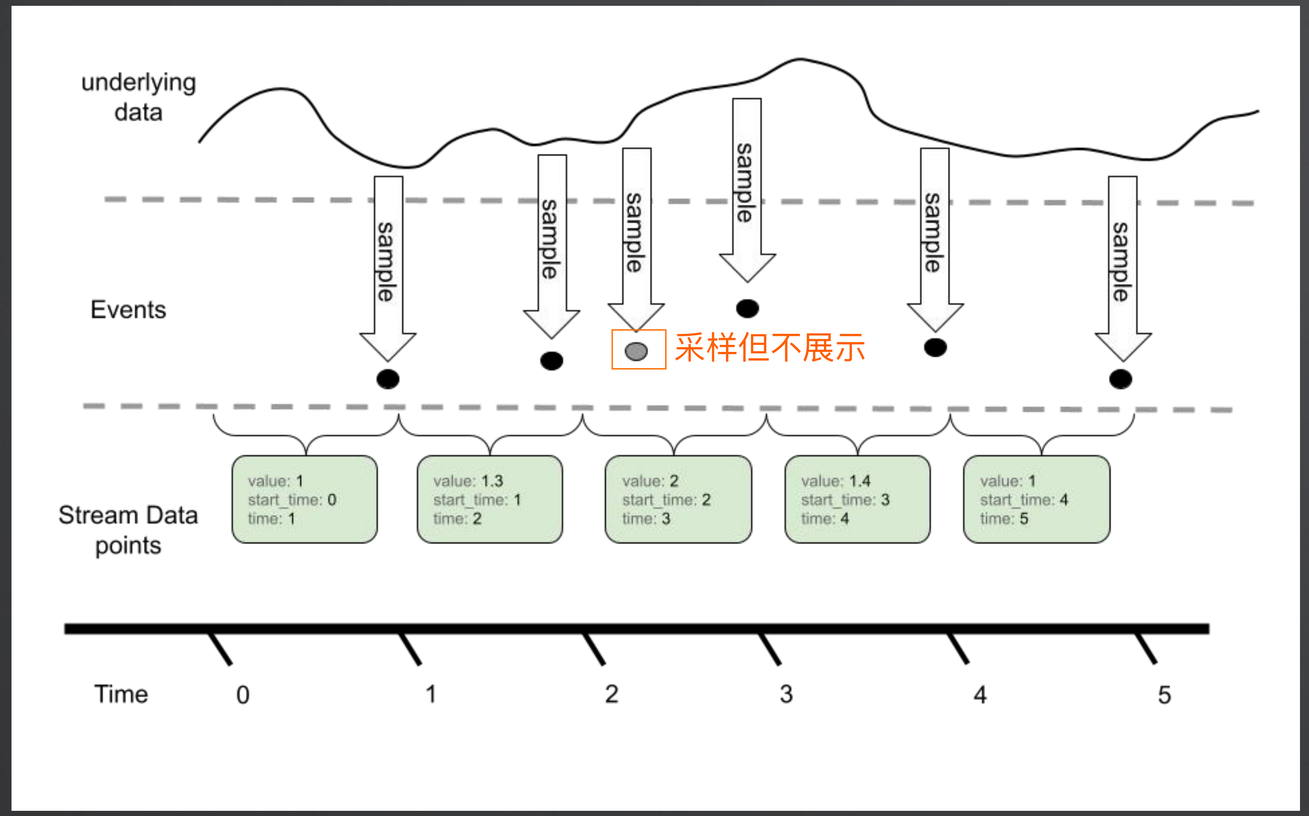
Sum：累计聚合

- 从“开始”计算总和（通常意味着进程/应用程序启动）



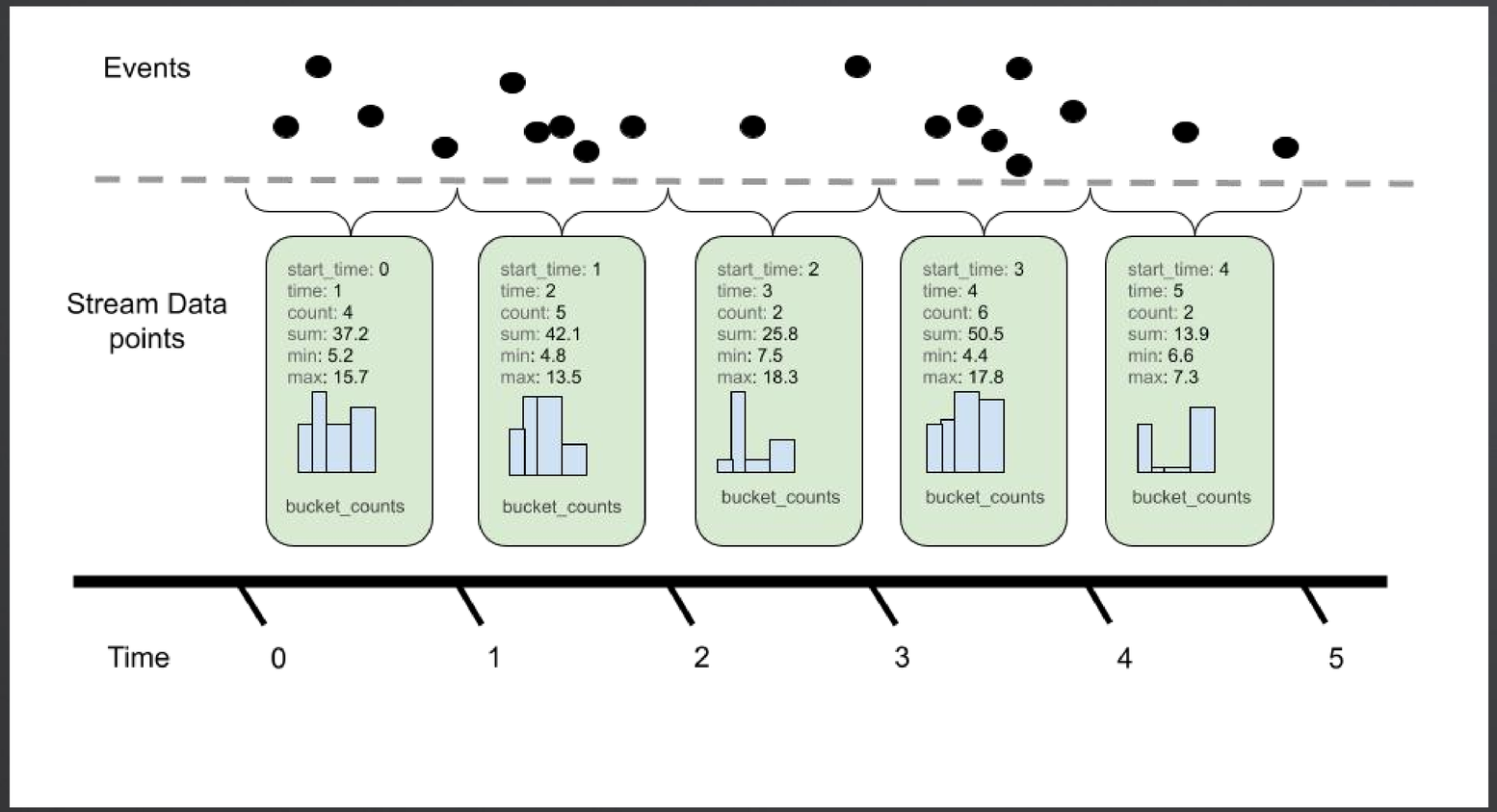
Gauge

- 采集给定时间窗口的最后采样事件
- 包含一组独立的属性名称–值对、采样值和时间戳组成



Histogram

- 每个数据点包括：一组独立的属性名称–值对、(start, end] 时间窗口、count、sum、min、max



Metrics 如何与 Traces 数据关联?

- 通过 trace id 和 span id 和 Traces 关联
- 在 Prometheus 抓取指标时可以通过“注释”的形式进行关联

```
fastapi_requests_duration_seconds_bucket{app_name="opentelemetry-python",le="0.05",method="GET",path="/chain"} 116.0 #
{TraceID="816b665ef13eb2a601ba89754c9d8568"} 0.038773452999521396 1697551994.764819
fastapi_requests_duration_seconds_bucket{app_name="opentelemetry-python",le="0.075",method="GET",path="/chain"} 203.0 #
{TraceID="bec85c922f770f9b5bcce071bd4ca604"} 0.07377643600011652 1697551993.7234154
fastapi_requests_duration_seconds_bucket{app_name="opentelemetry-python",le="0.1",method="GET",path="/chain"} 222.0 #
{TraceID="d47bf6c6decd4898392d14acfd29d510"} 0.07596021700010169 1697551923.2636392
```

Logs 如何与 Traces 数据关联?

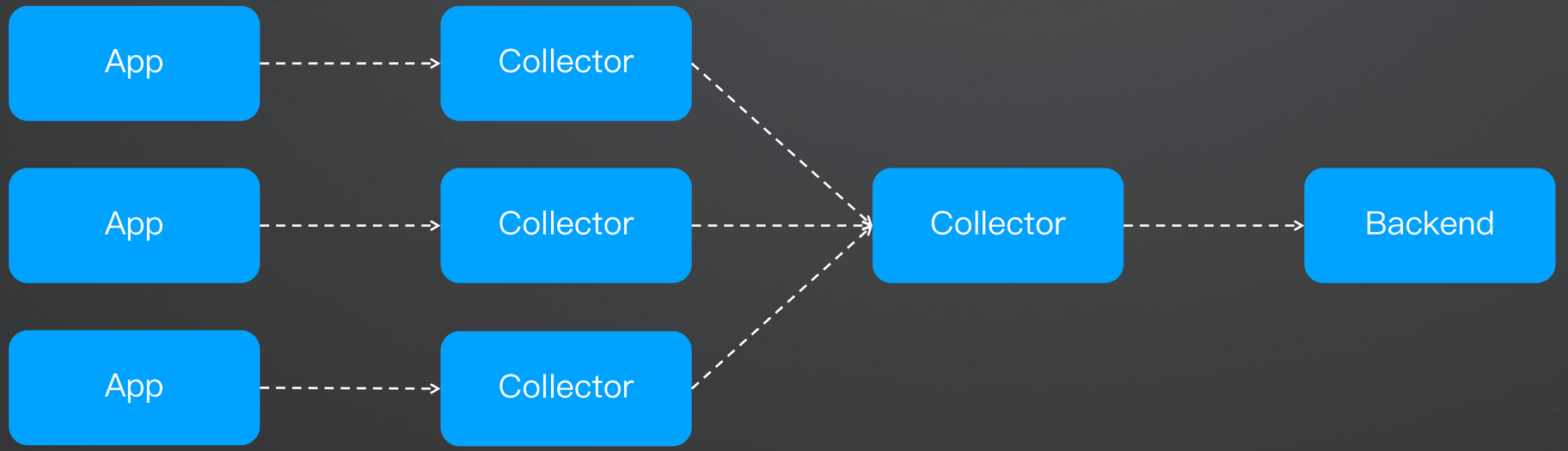
- 输出日志的时候把 trace_id 和 span_id 连同日志一起打印出来

```go
func logWithTracing(ctx context.Context, message string) {
    // 获取当前的 SpanContext
    spanContext := trace.SpanContextFromContext(ctx)
    if spanContext.IsValid() {
        log.Printf("trace_id=%s, span_id=%s, message=%s",
            spanContext.TraceID().String(),
            spanContext.SpanID().String(),
            message,
        )
    } else {
        log.Printf("message=%s", message)
    }
}
```

4. OpenTelemetry 组件

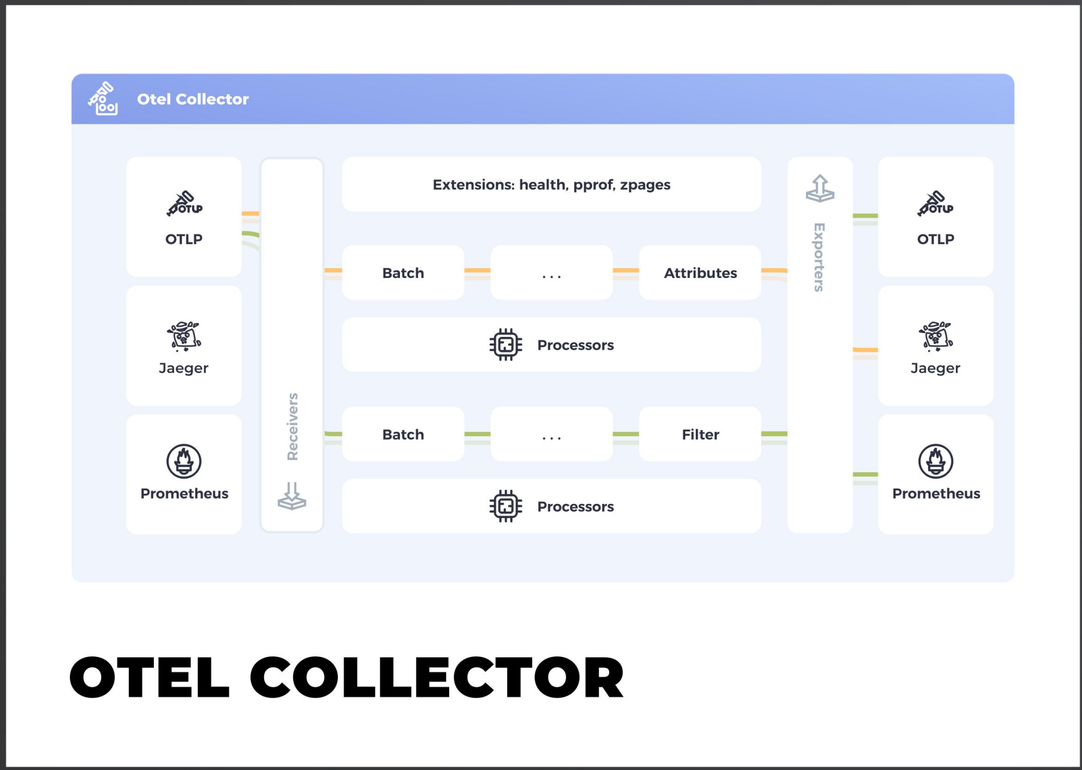
Collector

- OpenTelemetry Collector 是一个与供应商无关的代理，可以接收、处理和导出遥测数据，注意它不是集成 OpenTelemetry 所必须的
- 一个场景是借助 Collector 管道来处理敏感数据
- 可以用来构建更加复杂的可观测性系统（例如多层的 Collector 网关）



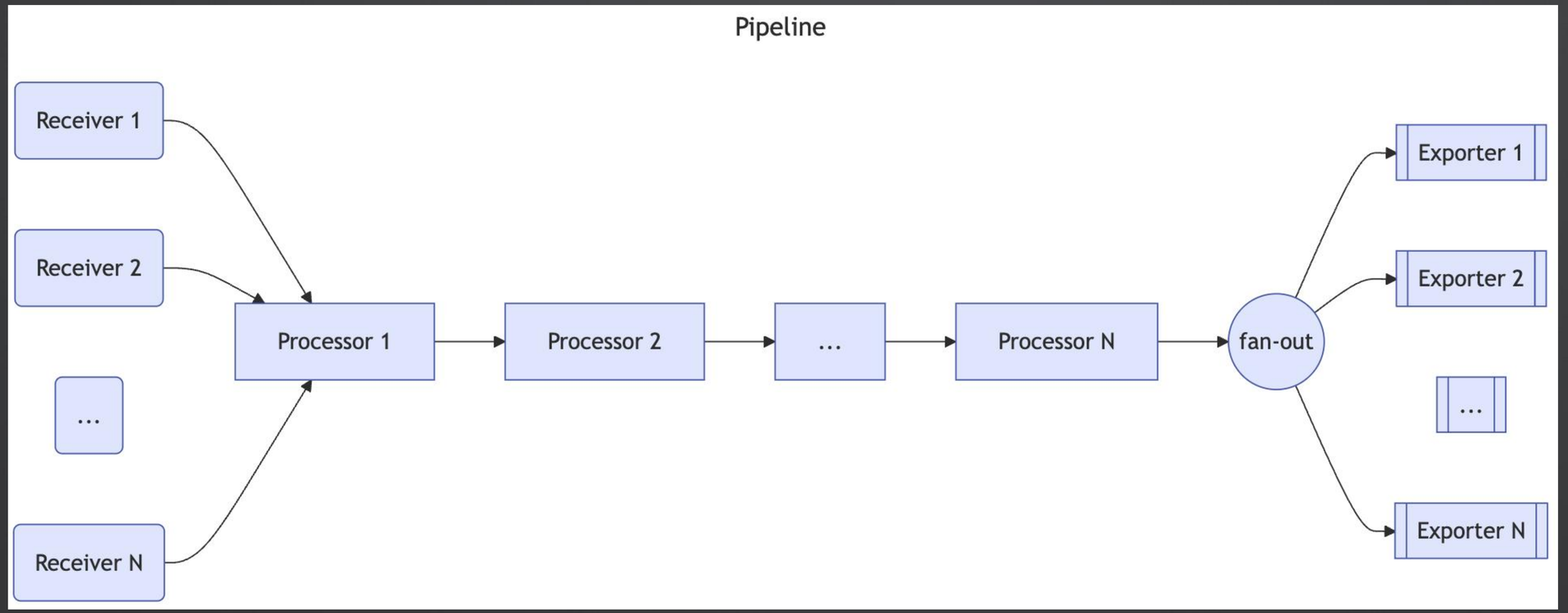
Collector 架构

- Collector 可以包含一个或多个 Pipeline，每一个 Pipeline 包含 Receiver、Processor、Exporter 三部分组成
- Pipeline 负责具体的接受数据、处理和导出数据：可以对 Traces、Metrics、Logs 进行处理



Pipeline

- Pipeline（管道）可以定义 N 个接收器、N 个处理器和 N 个导出方式
- 所有接收器接受到的数据会先进入到第一个处理器，然后再依次进入其他的处理器，这个过程可能会根据过滤规则丢弃数据
- 最后导出采用“扇出”的方式将数据副本导出到不同的后端

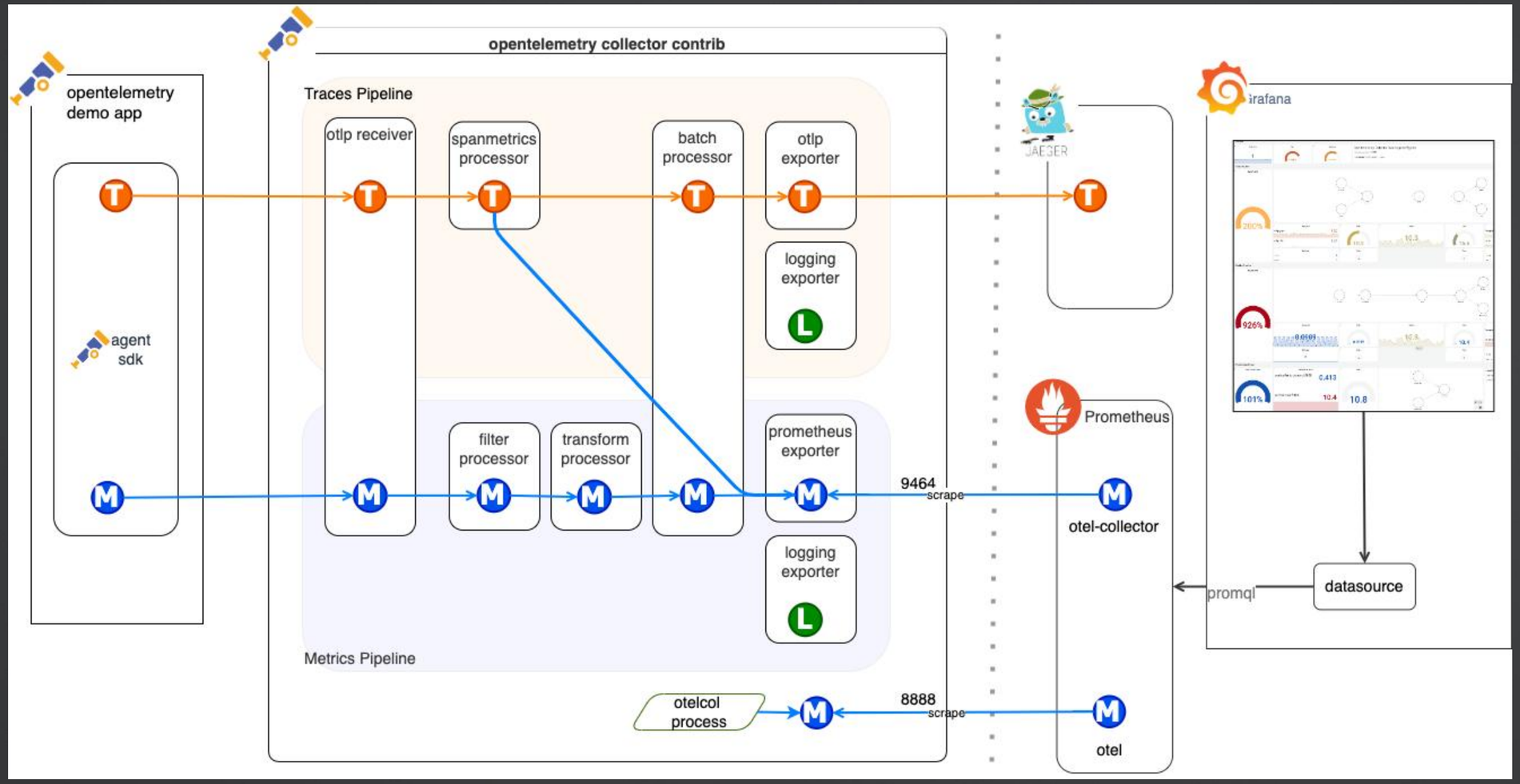


客户端 SDK

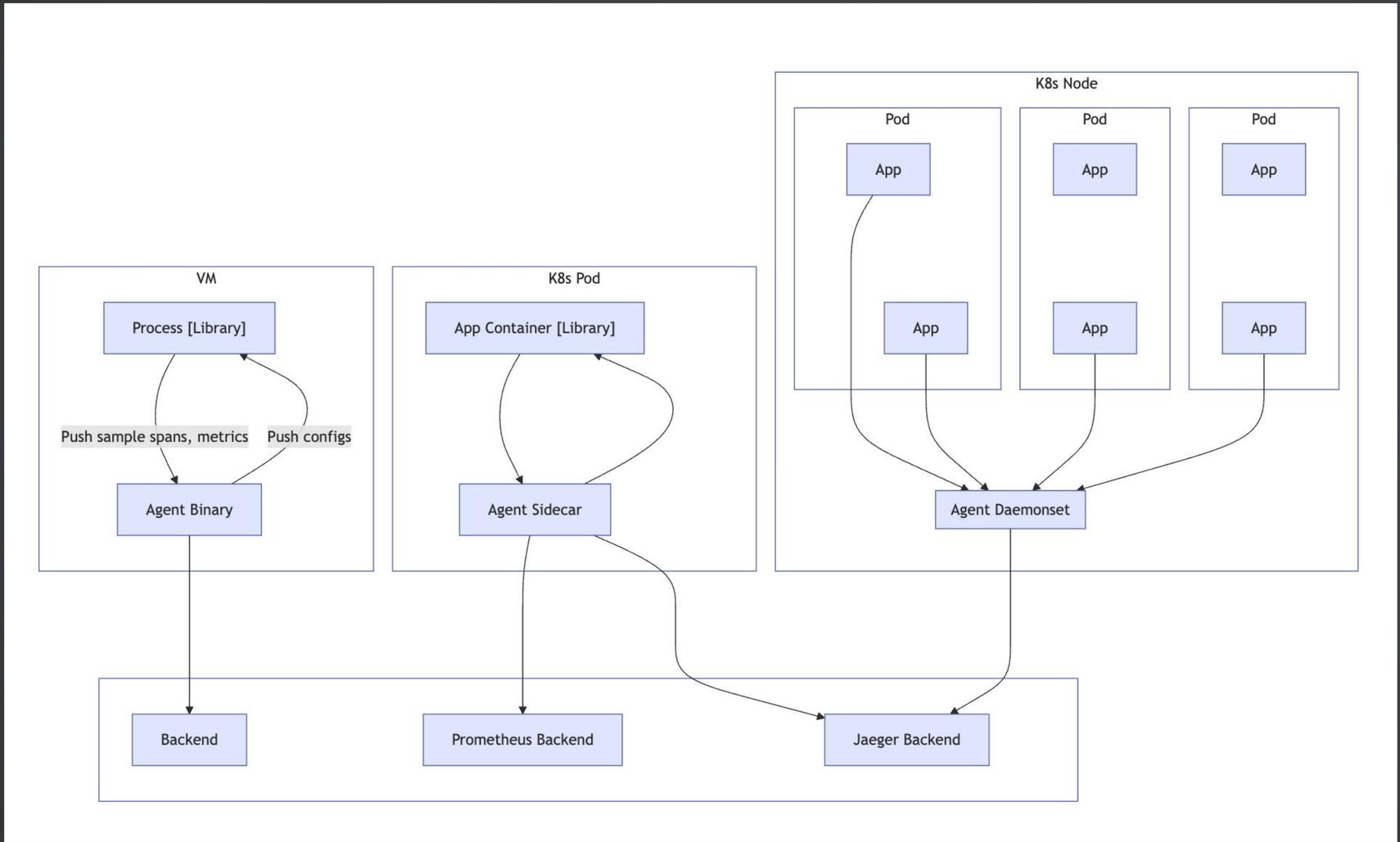
Language	Traces	Metrics	Logs
C++	Stable	Stable	Stable
C#/.NET	Stable	Stable	Stable
Erlang/Elixir	Stable	Development	Development
Go	Stable	Stable	Beta
Java	Stable	Stable	Stable
JavaScript	Stable	Stable	Development
PHP	Stable	Stable	Stable
Python	Stable	Stable	Development
Ruby	Stable	Development	Development
Rust	Beta	Alpha	Alpha
Swift	Stable	Development	Development

5. OpenTelemetry 数据流

数据流


opentelemetry collector contrib
opentelemetry demo app
Traces Pipeline
otlp receiver
spanmetrics processor
batch processor
otlp exporter
logging exporter
agent sdk
filter processor
transform processor
prometheus exporter
logging exporter
Metrics Pipeline
otelcol process
9464 scrape
8888 scrape
JAEGER
Prometheus
otel-collector
otel
irafana
promql
datasource

混合部署数据流


K8s Node
Pod
App
App
Pod
App
App
Pod
App
App
VM
Process [Library]
Push sample spans, metrics
Push configs
Agent Binary
K8s Pod
App Container [Library]
Agent Sidecar
Agent Daemonset
Backend
Prometheus Backend
Jaeger Backend

6. 如何集成 OpenTelemetry

1. 零侵入

- 插桩模式，根据不同语言实现不同，可能会用到例如字节码、monkey patching、eBPF 技术将 OpenTelemetry SDK 注入到程序里
- 优点：不需要开发
- 目前支持的语言：.NET、Go、Java、JavaScript、PHP、Python
- 能自动插桩的对象：请求和响应、数据库调用、消息队列调用等
- 不能自动插桩的对象：例如业务指标无法通过插桩的方式自动生成

2. SDK

Language	Traces	Metrics	Logs
C++	Stable	Stable	Stable
C#/.NET	Stable	Stable	Stable
Erlang/Elixir	Stable	Development	Development
Go	Stable	Stable	Beta
Java	Stable	Stable	Stable
JavaScript	Stable	Stable	Development
PHP	Stable	Stable	Stable
Python	Stable	Stable	Development
Ruby	Stable	Development	Development
Rust	Beta	Alpha	Alpha
Swift	Stable	Development	Development

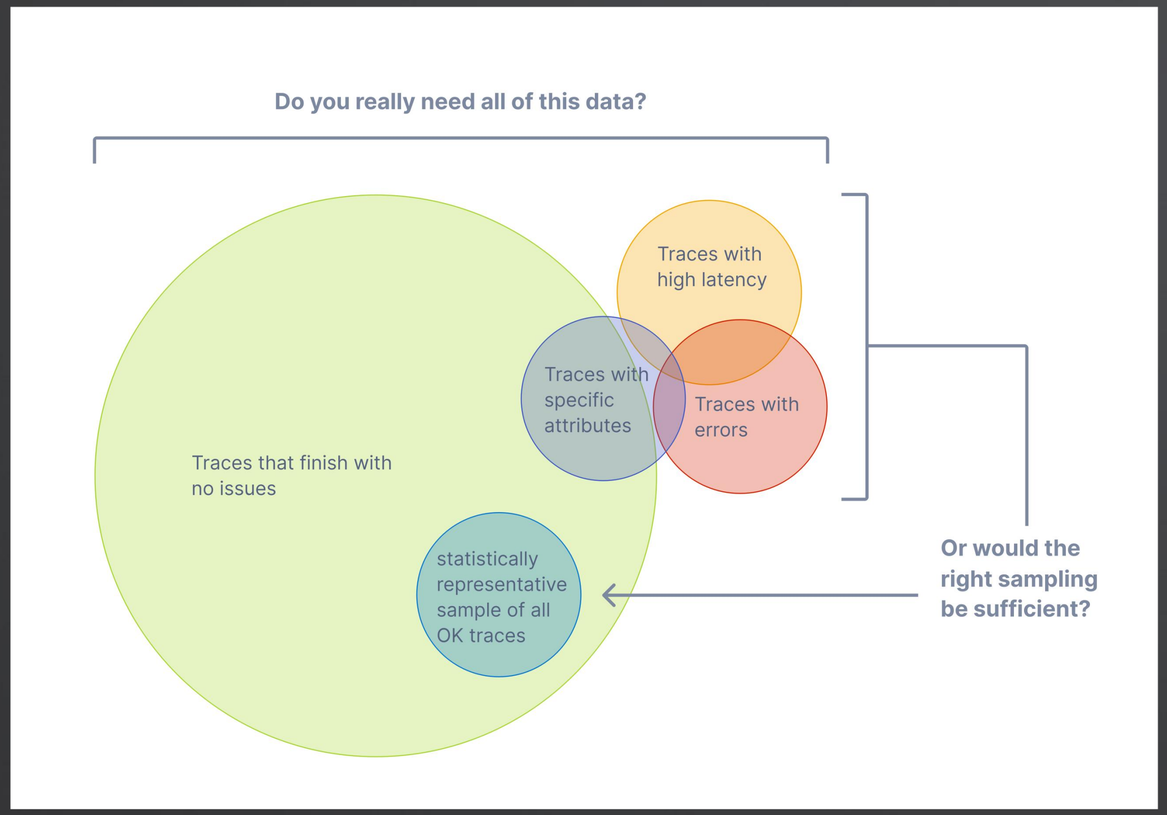
- 支持大部分语言
- 生产环境集成 OpenTelemetry 的推荐方式
- 缺点：需要额外的开发工作

3. OpenTelemetry API 库

- 直接调用 OTel API 的封装库，一般不使用
- 二次开发可以考虑

采样问题

- 大部分请求是正常的，将所有请求都采样成本太高
- 1% 的采样率是比较常见的



采样和不采样

什么时不采样?

- 生成的数据很少，比如每秒只有几十条
- 不允许丢弃任何追踪数据

什么时候应该采样?

- 每秒生成几百上千条追踪
- 大多数请求是正常的

采样方式

头部采样（OpenTelemetry Collector 支持）

- 在请求的入口就决定是否需要采样，例如根据采样百分比来决定
- 如果采样，那么记录完整的可观测信息，如果不采样，则丢弃
- 缺点：不能根据追踪结果进行动态采样，例如只希望采样有请求错误的场景；可能丢失有价值，重要的可观测数据

尾部采样（OpenTelemetry Collector 支持，大部分 SDK 不支持）

- 在请求退出时决定是否采样
- 可以根据完整的可观测信息决定是否采样，实现更加智能的采样策略
- 缺点：复杂度较高，延迟较高

THANKS