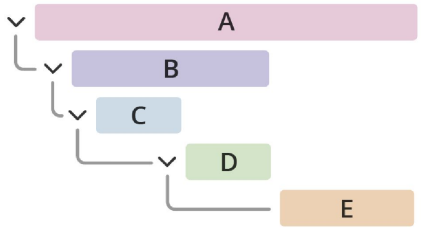
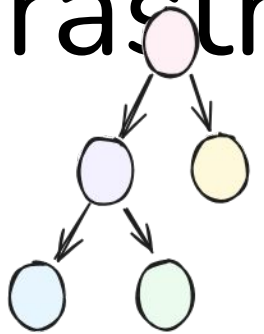
ENCOM 2024

Observabilidade Cloud-Native no Kubernetes: métricas, rastro e logs





Edson Célio
Emídio Neto

Edson Célio
Staff SRE @ Taller

CNCF Ambassador

Kubernetes Contributor

OpenTelemetry Contributor

DevOpsDays Organizer - Fortaleza

Emídio Neto
Staff SRE @ PicPay

OpenTelemetry Approver/Contributor

DevOpsDays Organizer - Natal

Mestre em Sistemas e Computação
UFRN

Já fui professor no IFRN

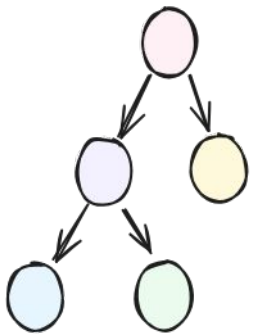
Agenda

Fundamentos

Observabilidade

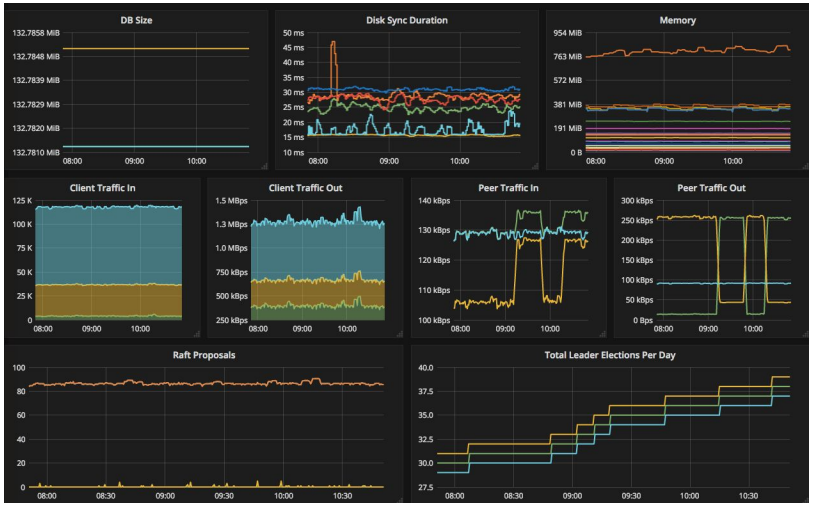
Instrumentação

Demo!



Fundamentos

Fundamentos



Vários termos...

1. Telemetria
2. Monitoramento
3. Observabilidade

Logs, métricas... Infinidade de termos...

Telemetria nada mais é do que um dado emitido por um sistema em relação ao seu comportamento em tempo de execução.

Fundamentos - Antes de começar

Monitoramento != Observabilidade

Monitoramento utiliza os dados de telemetria para responder perguntas conhecidas

> O tempo de resposta do meu sistema passou de 100ms?

> A taxa de erro do meu sistema passou de 5%?

Observabilidade utiliza os dados de telemetria para responder perguntas desconhecidas

> Por que alguns usuários estão enfrentando erros intermitentes que não são capturados pelos logs?

Fundamentos - Antes de começar

Por que devemos olhar pro monitoramento?

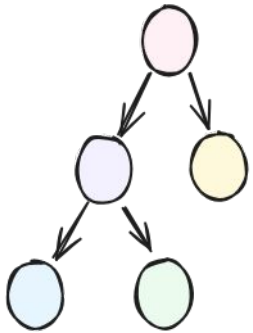
> Precisamos saber se nosso serviço funciona!

E o que devemos monitorar?

> 4 Golden Signals (Latência, Volume, Taxa de erro, Saturação)

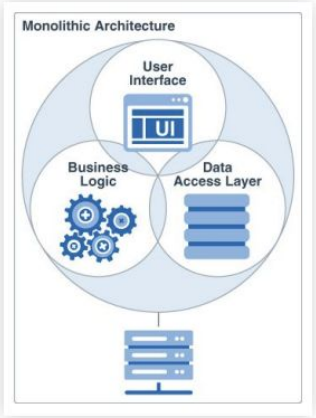
> Troubleshooting! precisamos de dados para debugar um problema.

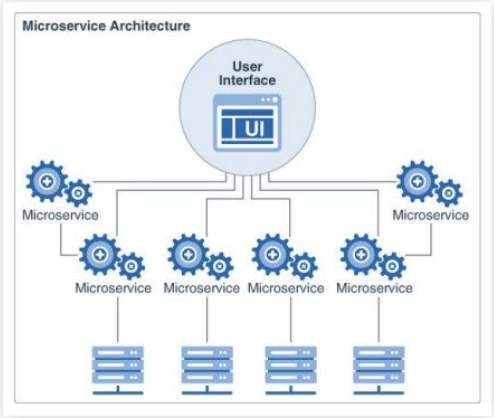
Novos paradigmas, novos desafios - Monitoramento, Troubleshooting, Dev

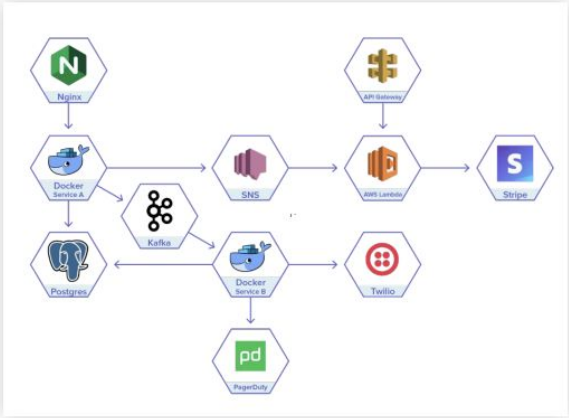


Observabilidade

Um pouco de contexto...







Host-based Monolithic → Host-based Distributed → Abstracted-host Highly Distributed

Observabilidade

Ajuda a entender e solucionar problemas de um sistema ao fornecer respostas para perguntas complexas e desconhecidas

Por que isso tá acontecendo?

R: Sei lá! tá tudo lento!

R: Problema silencioso. É o banco!

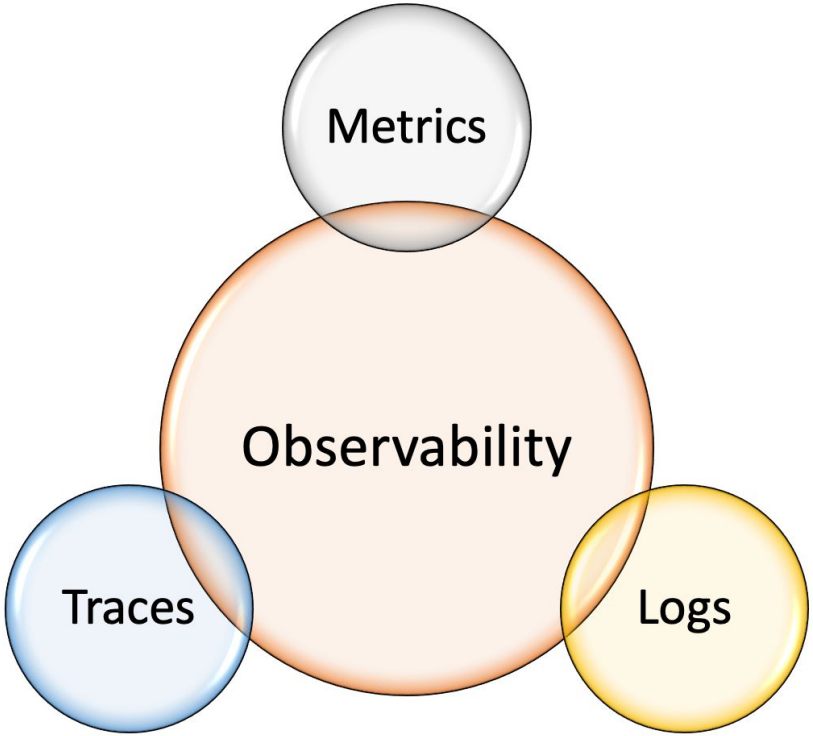
Observabilidade

Três pilares da Observabilidade

Métricas nos dizem o quê

Traces nos dizem onde

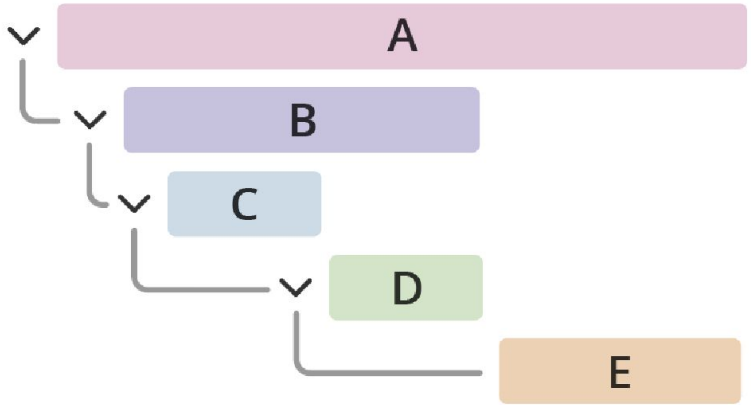
Logs nos dizem o porquê



Observabilidade

“Um trace nos conta toda a história de uma transação ao longo de sua propagação por um sistema.”

INSTRUMENTAÇÃO É O CAMINHO



Sua aplicação precisa emitir sinais de alguma forma

E se minha aplicação for apenas um monolito com um banco de dados?

R: tem que instrumentar igual

Observabilidade - Logs

```
192.168.1.1 - - [11/Oct/2018:11:36:39 +0200] "GET /index.php/apps
192.168.1.1 - - [11/Oct/2018:11:36:40 +0200] "GET /index.php/apps
192.168.1.1 - - [11/Oct/2018:11:36:42 +0200] "GET /index.php/apps
192.168.1.1 - - [11/Oct/2018:11:36:42 +0200] "GET /index.php/apps
192.168.1.1 - - [11/Oct/2018:11:36:44 +0200] "GET /index.php/apps
192.168.1.1 - - [11/Oct/2018:11:36:46 +0200] "GET /index.php/apps
192.168.1.1 - - [11/Oct/2018:11:36:46 +0200] "GET /index.php/apps
192.168.1.1 - - [11/Oct/2018:11:36:47 +0200] "GET /index.php/apps
```

Logs (o porquê)

- Registro de eventos importantes
- DEBUG, WARNING, INFO, ERROR
- Timestamp, descrição

OLD SCHOOL

Aplicação escreve log em um arquivo local

Sysadmin faz “greps” pra descobrir o problema

- Aplicação emite log pro STDOUT
- Coleta de logs e centralização
- Log estruturado (JSON)
- Logs canônicos
- Ferramentas profissionais de análise e armazenamento

Observabilidade - Métricas

Métricas (o quê)

- Representação numérica de um evento em um sistema
- Agregações e análises ao longo de dimensões

Contadores -> Monotônico (só sobe) ou sobe-desce -> Número de visualizações em tempo real

Gauges -> Valores “snapshot” -> Uso de disco

Histogramas -> Delta (inicio-fim) de um evento (requisição de um usuário) mapeado em buckets

- **RED** (Requests, Error, Duration)
- **USE** (Utilization, Saturation, Error)
- Métricas de infraestrutura
- Métricas de Aplicação

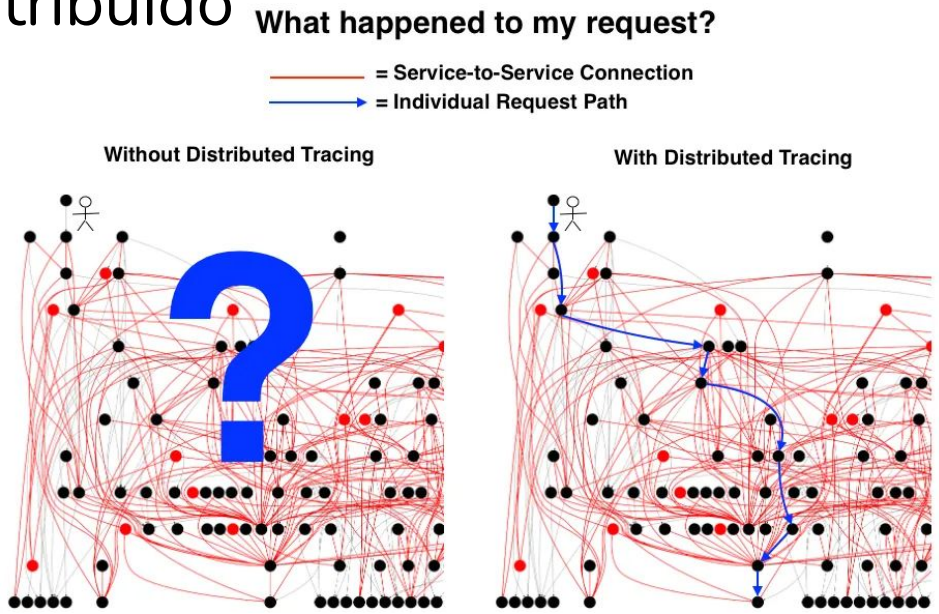
Do OLD SCHOOL para o CLOUD NATIVE

Prometheus

cAdvisor

Observabilidade - Tracing Distribuído

- Por quais serviços a requisição passou?
- Em quais serviços temos gargalos?
- Onde nosso código gasta mais tempo?
- Quanto tempo é gasto durante a comunicação entre serviços?
- Qual comportamento da requisição em cada serviço?



Fonte: [1c]

Observabilidade - Tracing Distribuído

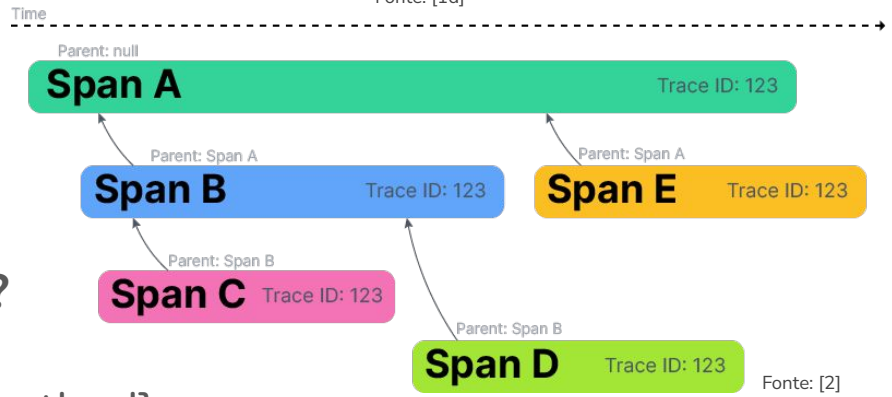
Spans

Operação com começo e fim (timestamps)

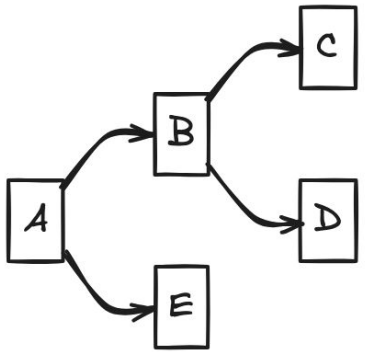
Parent Span, Child Span. Seria um grafo?

- Atributos->Metadados{user_id, http.method}
- Propagação de contexto

Fonte: [1d]

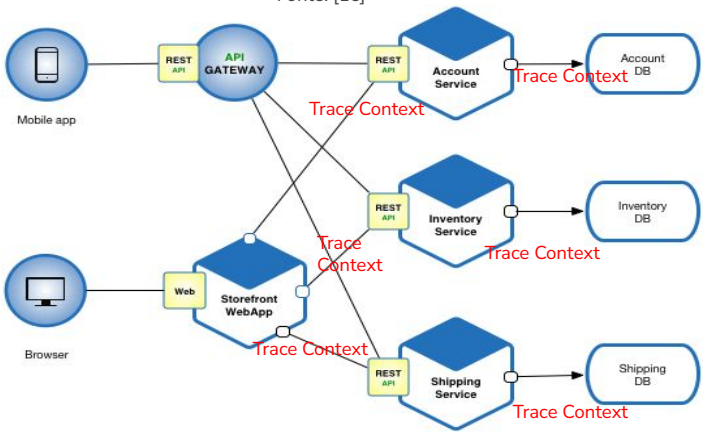


Fonte: [2]



Tracing Distribuído

Fonte: [1e]



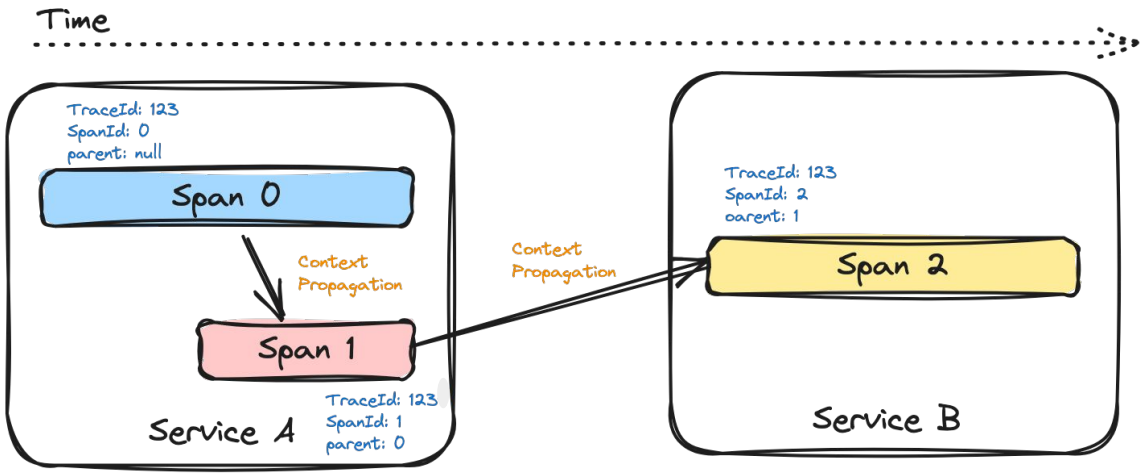
Propagação de contexto

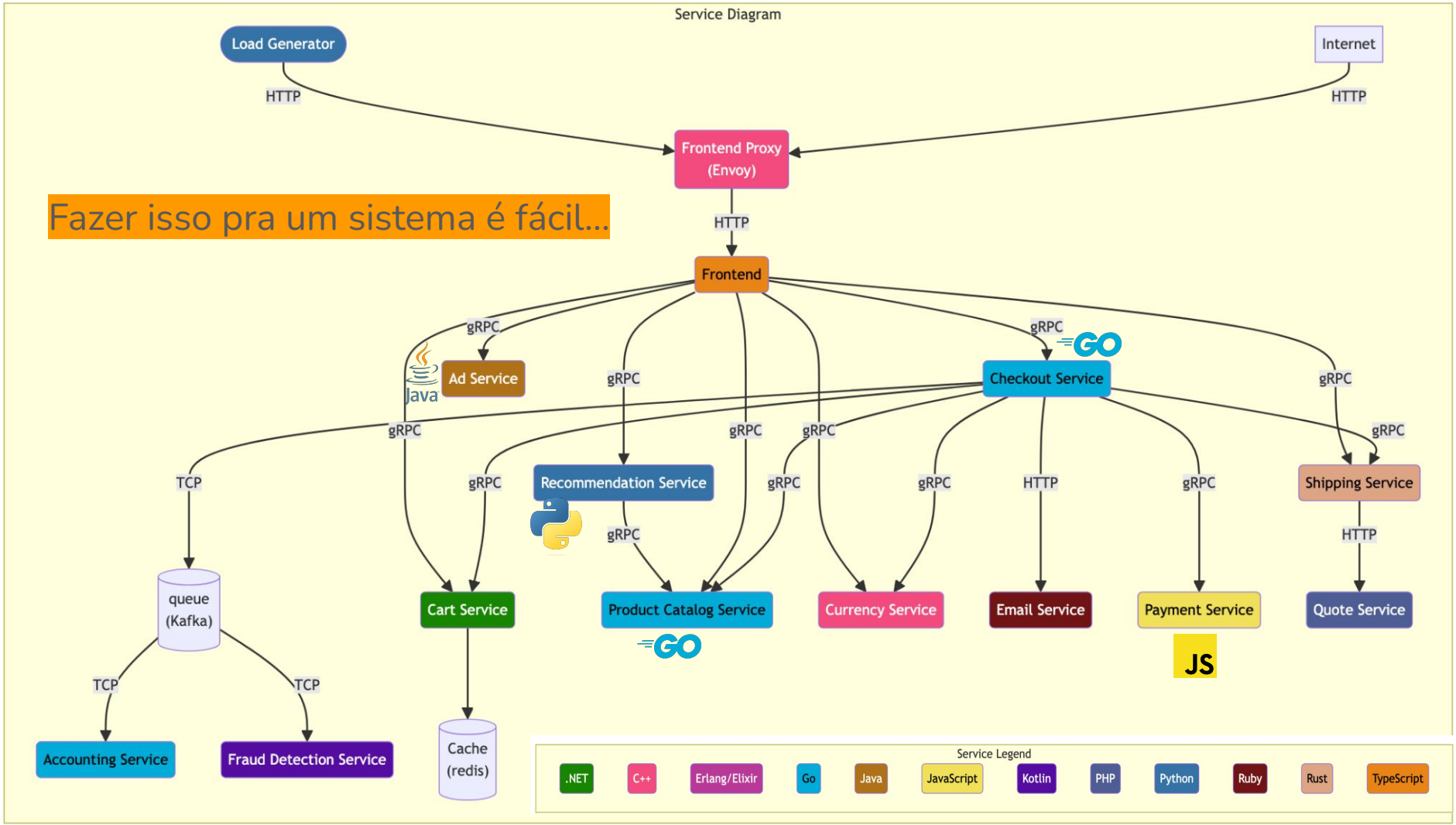
Correlação!

- Contexto->{trace_id, parent_id}

Propagação de metadados:

- Em um mesmo processo
- Entre processos




Service Diagram
Load Generator
Internet
HTTP
HTTP
Frontend Proxy (Envoy)
Fazer isso pra um sistema é fácil...
HTTP
Frontend
gRPC
Ad Service
Java
gRPC
gRPC
Checkout Service
GO
gRPC
gRPC
gRPC
gRPC
gRPC
gRPC
gRPC
gRPC
TCP
Recommendation Service
gRPC
HTTP
gRPC
Shipping Service
HTTP
queue (Kafka)
Cart Service
Product Catalog Service
GO
Currency Service
Email Service
Payment Service
JS
Quote Service
TCP
TCP
Accounting Service
Fraud Detection Service
Cache (redis)
Service Legend
.NET
C++
Erlang/Elixir
Go
Java
JavaScript
Kotlin
PHP
Python
Ruby
Rust
TypeScript

Observabilidade - Desafios

O primeiro desafio é a **instrumentação do software!**

Você precisa gerar SINAIS de telemetria para começar a ter utilidade

1. Instrumentar cada chamada no seu sistema (http, queries no banco, etc)
2. Criar spans para cada request e response
3. Adicionar o contexto em cada Span (de onde tá saindo)
4. Fazer Injeção e extração de IDs, atributos nas chamadas ao longo da propagação do contexto (de serviço em serviço)

É MUITA COISA PRA FAZER

Observabilidade - Desafios

Um sistema só pode ser considerado devidamente instrumentado se:

Não é necessário adicionar mais instrumentação para solucionar algum problema, pois temos as informações para responder qualquer pergunta

Tracing Distribuído

- Como e onde armazenar os dados gerados? *Coletores de dados e backends*
- O que fazer com os dados brutos do trace? Como visualizar os dados de traces? Ferramentas de visualização e Análise

JAEGER

ZIPKIN

Grafana Tempo

Tracing Distribuído

É fácil de compreender, porém complexo na prática devido à quantidade de componentes envolvidos para suportar a implementação:

- O software pode não estar adequado para aceitar facilmente a instrumentação dos métodos
- Ninguém consegue parar para fazer instrumentação manual. Envolve muito esforço operacional e não há espaço para isso nas sprints

Falta de um padrão sobre como instrumentar código e enviar dados de telemetria para um backend de observabilidade

O **OpenTelemetry** ajuda com isso!

OpenTelemetry - Observabilidade

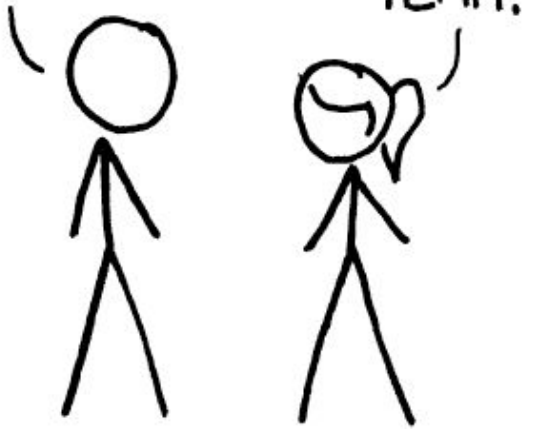
Que tal fazer uma lib pra gente usar aqui na empresa?

E se a gente fizer um padrão de como as instrumentações devem acontecer para um sistema ser observável?

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)

SITUATION:
THERE ARE
14 COMPETING
STANDARDS.

14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD
THAT COVERS EVERYONE'S
USE CASES.



SOON:

SITUATION:
THERE ARE
15 COMPETING
STANDARDS.

OpenTelemetry - Em 3 slides

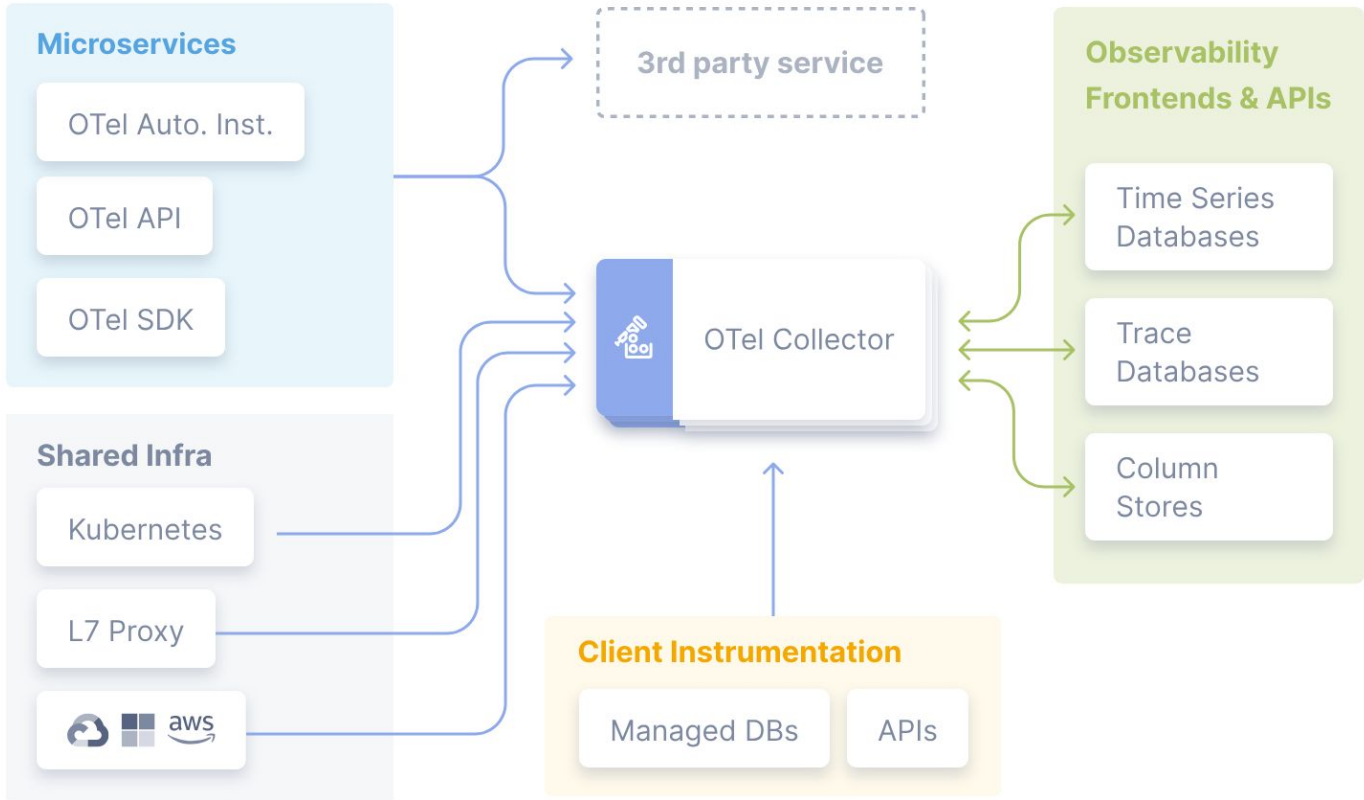
OpenTelemetry é o meio pelo qual o software é instrumentado para se tornar observável.

CLOUD NATIVE COMPUTING FOUNDATION

OpenTelemetry

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

Fonte: [4]



Fonte: [4]

OpenTelemetry - Em 3 slides

OTel **não é um backend** de observabilidade, como Jaeger, Prometheus, etc

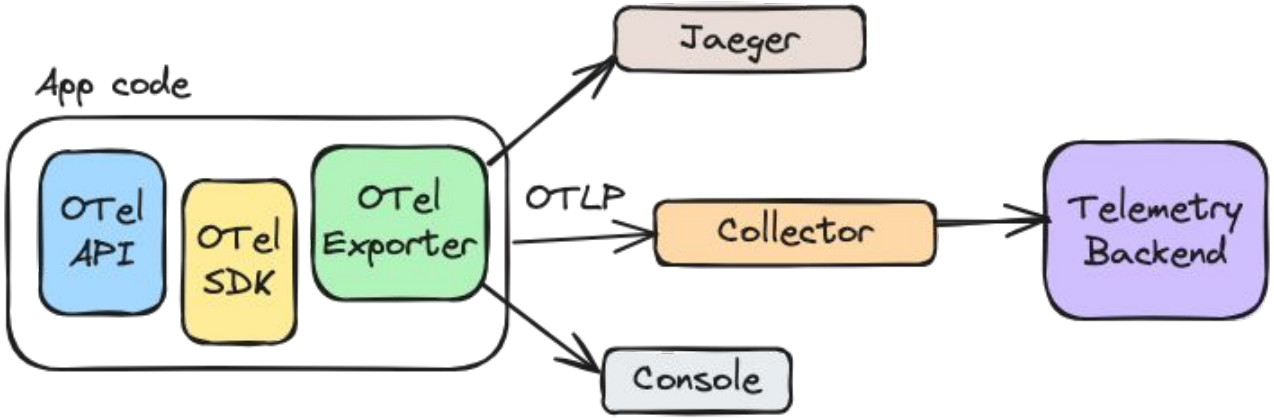
Define padrões, especificações e convenções de como dados de telemetria são gerados e gerenciados

OpenTelemetry API / SDK

Convenções Semânticas

OTLP

Bibliotecas de instrumentação



OpenTelemetry - Em 3 slides

Code-based via APIs e SDK para linguagem da aplicação + Libs de instrumentação

Zero-code (auto-instrumentation)

Podemos usar os dois métodos em conjunto!

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

OpenTelemetry

OTel **não é um backend** de observabilidade, como Jaeger, Prometheus, etc

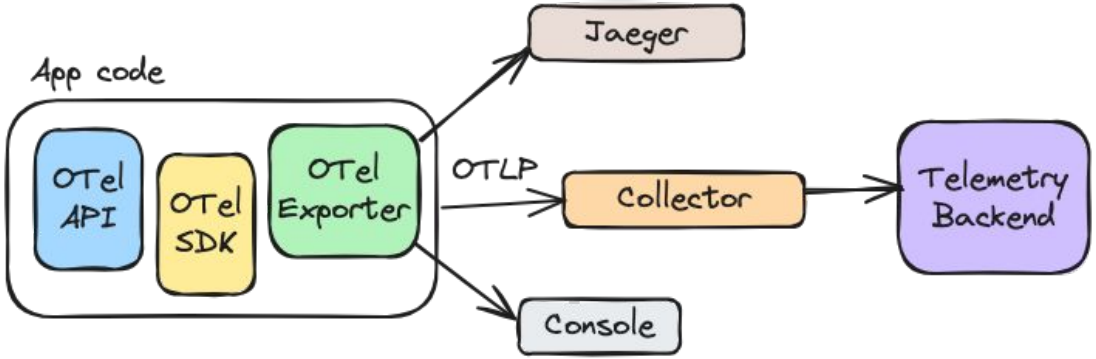
Define padrões, especificações e convenções de como dados de telemetria são gerados e gerenciados

OpenTelemetry API / SDK

Convenções Semânticas

OTLP

Bibliotecas de instrumentação



OpenTelemetry - Instrumentação

Code-based via APIs e SDK para linguagem da aplicação + Libs de instrumentação

- Trabalho de instrumentação manual ❌
- Instrumentação de métodos de forma customizada ✅

Zero-code (auto-instrumentation)

- Esforço apenas de configuração ✅
- Pouca customização ❌

Podemos usar os dois métodos em conjunto!

OpenTelemetry - Instrumentação

Code-based

```python
from flask import Flask
from opentelemetry.instrumentation.flask import FlaskInstrumentor

app = Flask(__name__)

FlaskInstrumentor().instrument_app(app)

@app.route("/")
def hello():
    return "Hello!"

if __name__ == "__main__":
    app.run(debug=True)
```

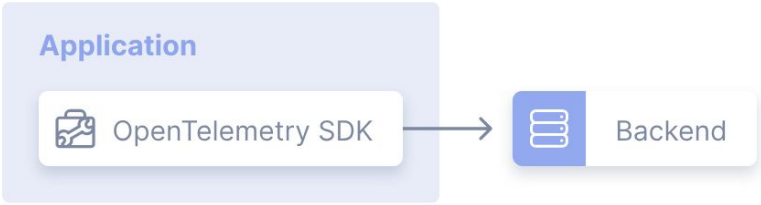
```python
# tracing.py
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    ConsoleSpanExporter,
    SimpleExportSpanProcessor,
)

trace.set_tracer_provider(TracerProvider())
trace.get_tracer_provider().add_span_processor(
    SimpleExportSpanProcessor(ConsoleSpanExporter())
)

tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("foo"):
    with tracer.start_as_current_span("bar"):
        with tracer.start_as_current_span("baz"):
            print("Hello world from OpenTelemetry
Python!")
```

OpenTelemetry - Instrumentação

Zero-code

```
OTEL_SERVICE_NAME=service-example
OTEL_TRACES_EXPORTER=console,otlp
OTEL_METRICS_EXPORTER=console
OTEL_EXPORTER_OTLP_TRACES_ENDPOINT=0.0.0.0:4317

$ opentelemetry-instrument python myapp.py
```

OpenTelemetry



Fonte: [4]

Por onde começar?

- Bastante simples de implementar
- Útil se você precisa coletar dados de dispositivos IoT

Tradeoffs

- Requer alterações de código se a coleta, o processamento ou a ingestão de traces forem alteradas (e.g., mudou o backend)
- Você vai ter que implementar toda lógica para envios em batch, retries, filtros, atributos. Não é recomendado para ambientes produtivos

OpenTelemetry

Agora que temos instrumentação, precisamos resolver outro problema:

Como coletar os sinais de telemetria das aplicações?

OTel também nos ajuda a simplificar isso com o **OTel Collector** (otelcol)

OpenTelemetry - Collector

Fonte: [4]

OpenTelemetry - Collector

Receivers

- OTLP (http, grpc)
- Jaeger, Zipkin
- Prometheus
- Kafka

Exporters

- Jaeger, Zipkin
- OTLP
- Serviços gerenciados

Processors

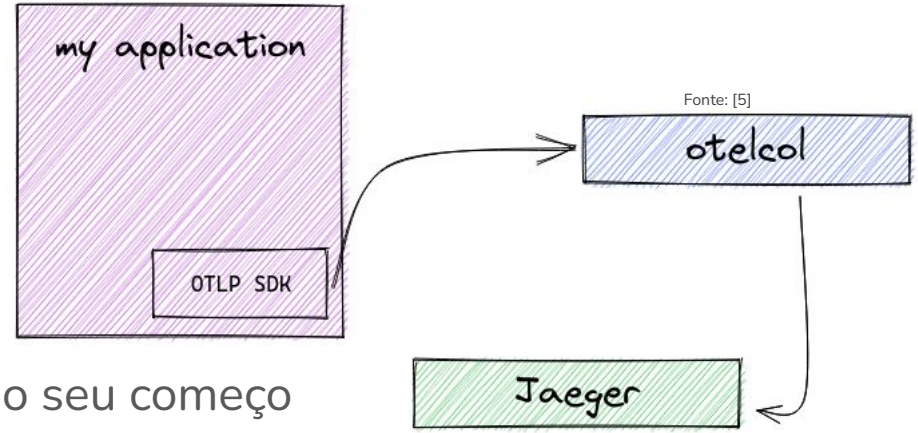
- Sampling
- Atributos
- Envios em batch
- Memory limiter
- Normalização

E como é que eu rodo isso?

```yaml
receivers:
  otlp:
    protocols:
      grpc:
        endpoint: 0.0.0.0:4317
      http:
        endpoint: 0.0.0.0:4318

processors:
  batch:

exporters:
  jaeger:
    endpoint: jaeger:4317
    insecure: true

service:
  extensions: []
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [jaeger]
```

docker run otel/opentelemetry-collector:0.96.0

OpenTelemetry - Collector



Fonte: [5]

Implantação Agent

- Se você não tem nada, esse método é o seu começo
- Desacoplamento entre o app e o backend
- Testar backends, fazer PoCs e mostrar o valor
- Você pode facilmente mudar o exporter sem precisar mudar nada em código

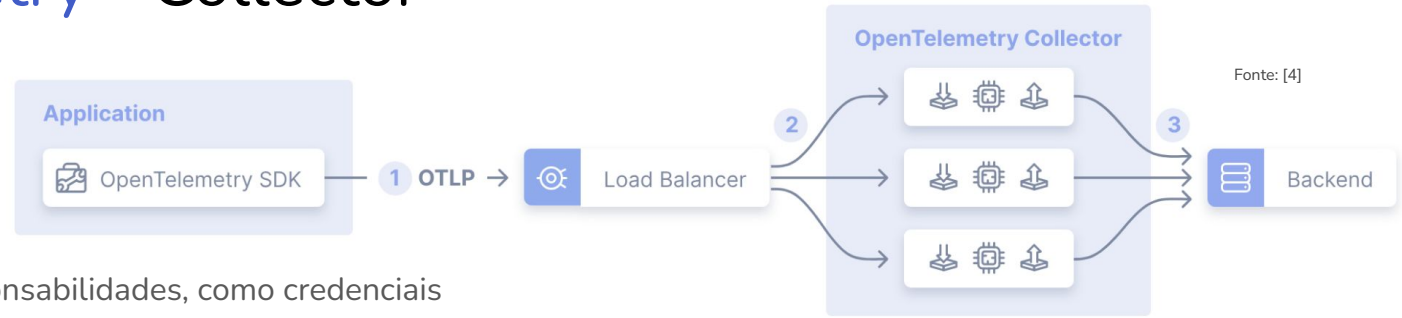
Tradeoffs

- É aquele negócio, é mais uma coisa para manter!

```yaml
service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [jaeger]
```

OpenTelemetry - Collector

Implantação Gateway



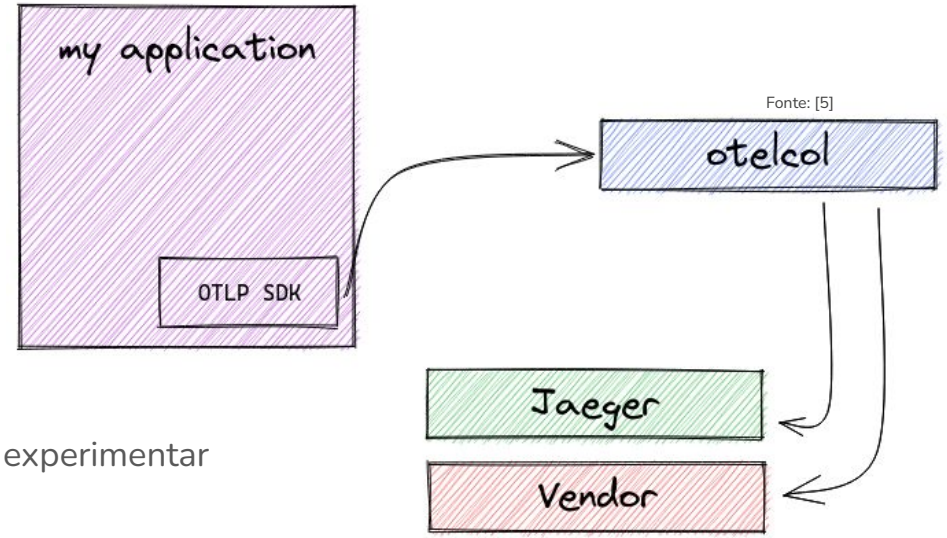
Fonte: [4]

- Separação de responsabilidades, como credenciais
- Gerenciamento centralizado de políticas (por exemplo, filtragem de determinados logs ou amostragem)
- Alguns processor só vão funcionar bem se todos spans de um mesmo trace forem enviados para o mesmo coletor (se torna stateful)

Tradeoffs

- Custo
- Não é necessário se você quer apenas um balanceador de carga sem se importar com a visão completa do trace

OpenTelemetry - Collector



Fonte: [5]

Pattern Fanout

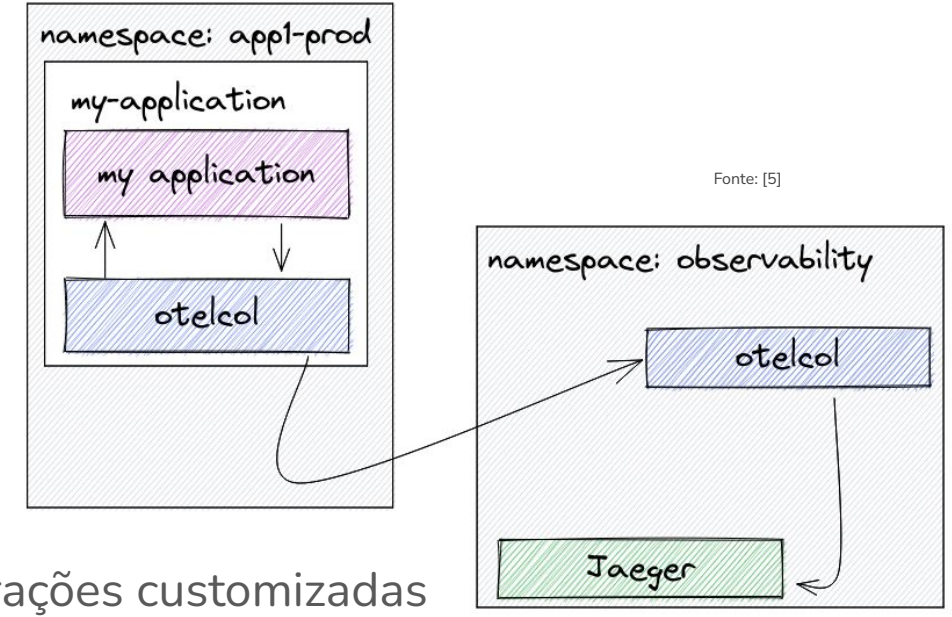
- Se você já tem alguma coisa implementada e quer experimentar
- Mostrar o valor de solução X ou Y
- Você tem o seu cenário estável + um cenário experimental coexistindo, por exemplo

Tradeoffs

- Aqui o ponto de atenção é o sizing que tu tem hoje x preço do vendor
- 2 soluções rodando = maior custo

```
service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [jaeger, vendor]
```

OpenTelemetry - Collector



Fonte: [5]

Kubernetes Sidecar + Deployment

- Sidecar resolve os casos de configurações customizadas
- O deployment resolve o caso para centralizar tudo a nível do ambiente e scale up e down com facilidade

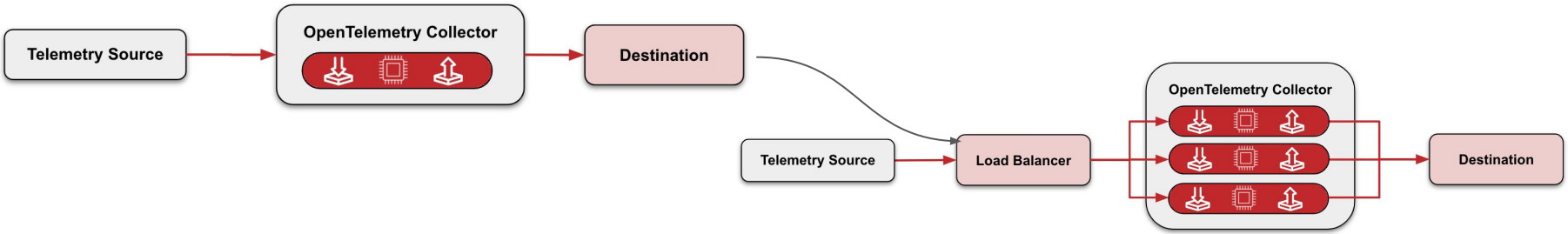
Também é possível implantar como **DaemonSets** ou **Statefulsets!**

OpenTelemetry - Collector

Implantação do collector: agent ou gateway. Qual usar e quantos?

1. Um collector agent ao lado de cada aplicação ou ambiente;
2. Quando alcançar maturidade e for implementar processors stateful (e.g., tail-sampling) -> gateway

Por que não os dois?

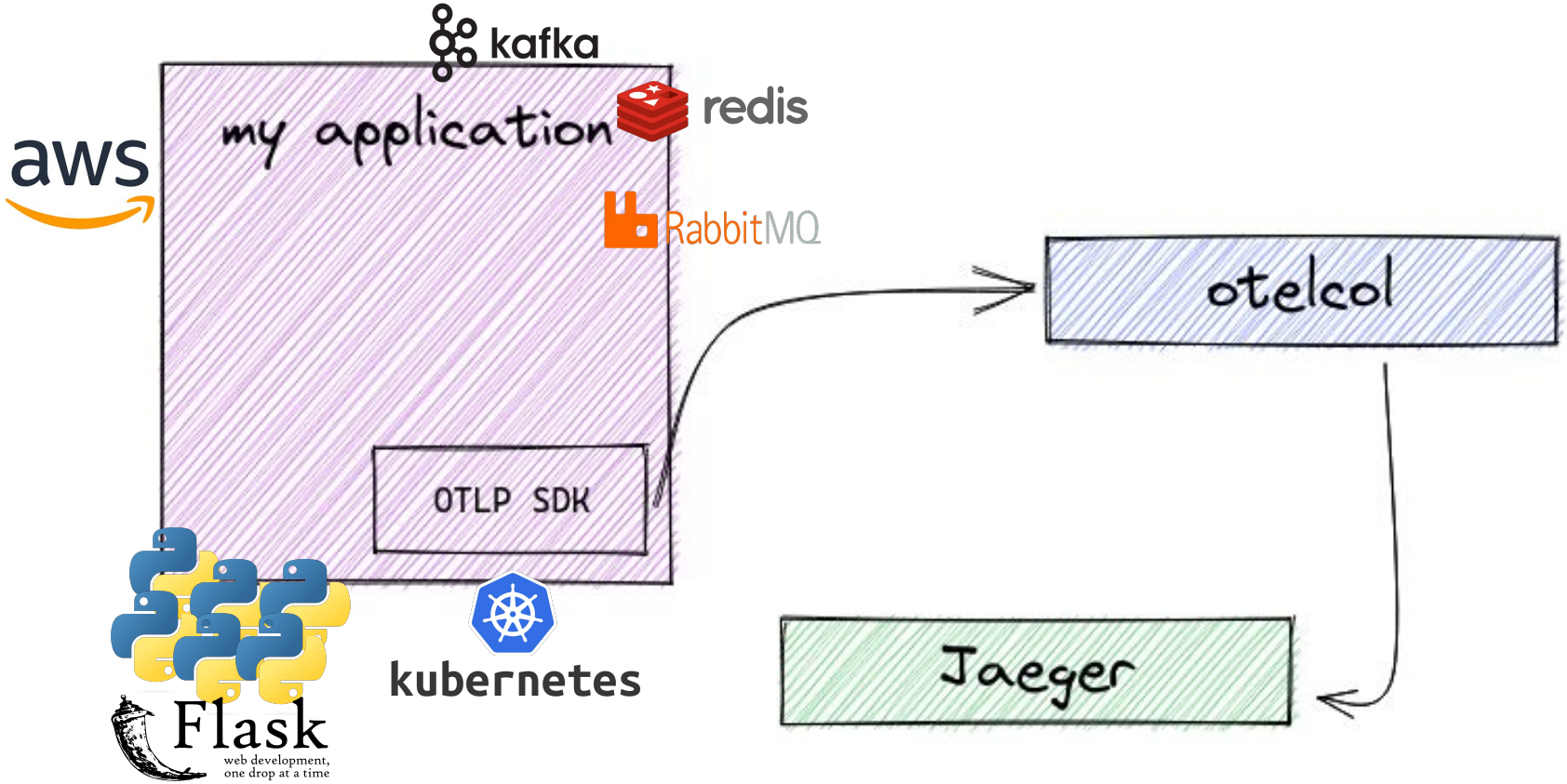


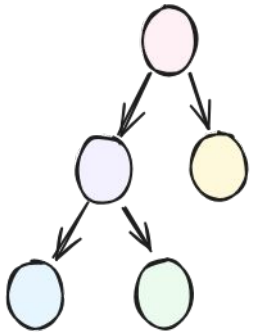
Fonte: [4]

OpenTelemetry - Simplificando

- Instrumentação -> OTel API & SDK ✅
- Propagação de contexto -> OTel API & SDK ✅
- Sampling, batching, retries, normalização -> OTel Collector ✅
- Centralização e entrega para o backend -> OTel Collector ✅
- Tracing UIs & Backends -> e.g., jaeger -> Vamos ver na demo!

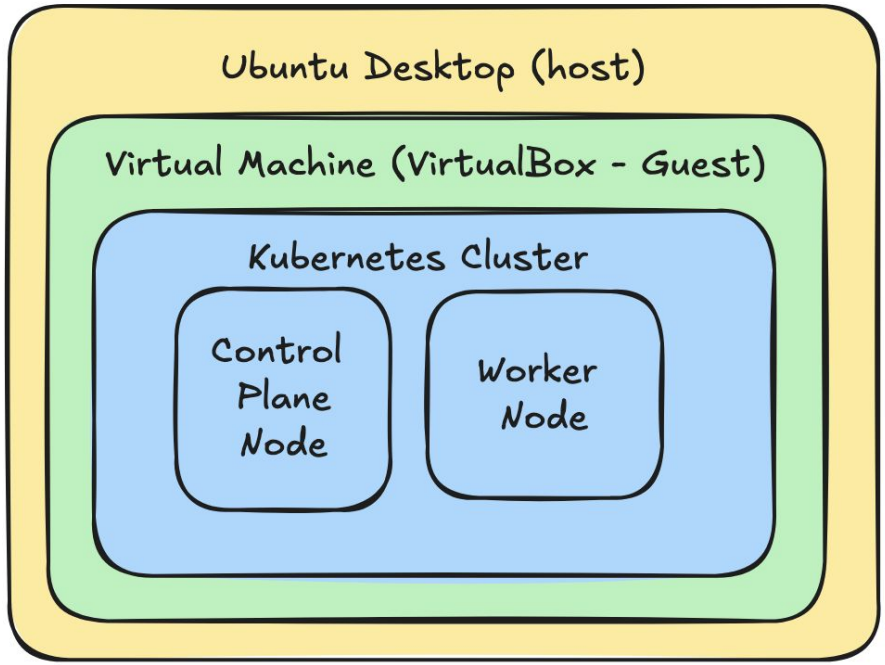
Simplificando





Demo

Demo!



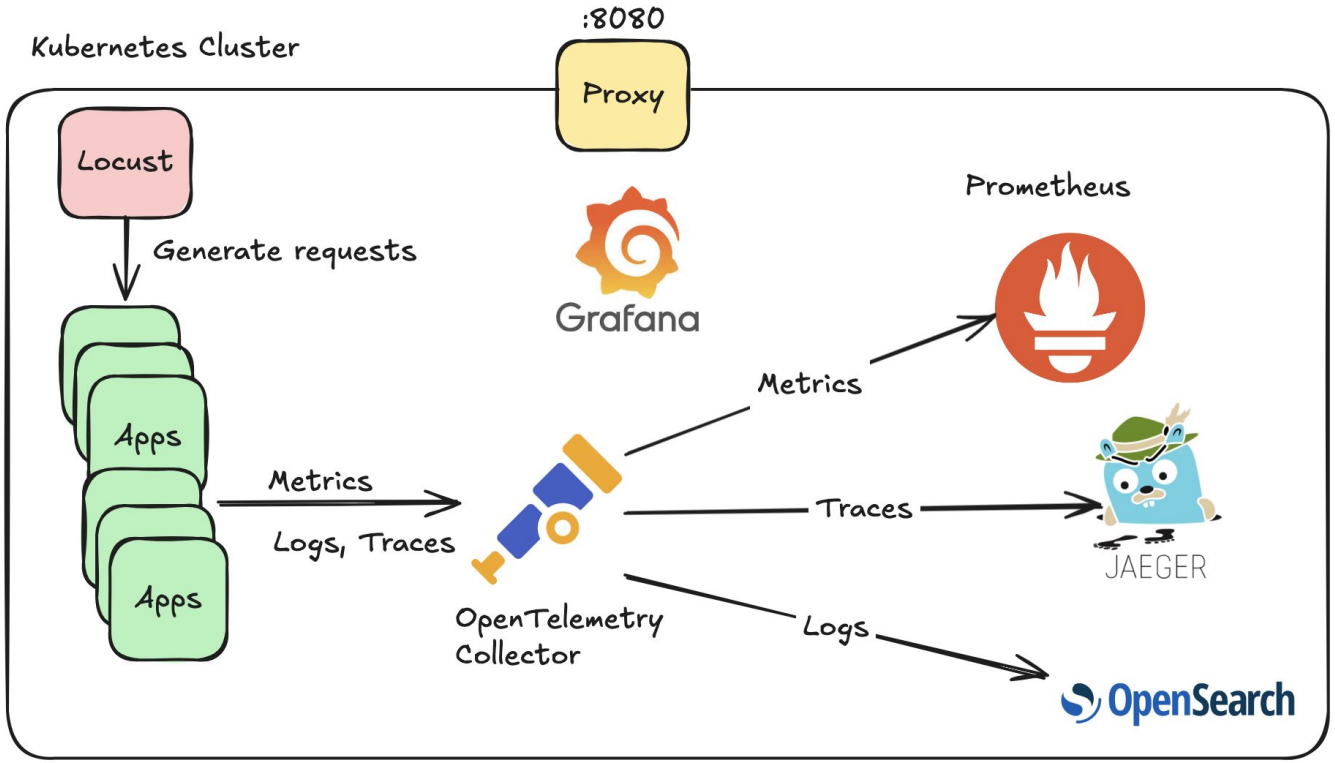
Máquina Virtual

Acesso via SSH pelo Terminal do Ubuntu Desktop

Subir Cluster Kubernetes com 2 nós

Instalar os componentes da Demo!

Demo! https://github.com/cloudnativern/o11y-workshop



Obrigado por participar!

Contato

Emídio Neto, Edson Célio

@emdneto, @edsoncelio

ENCOM 2024

Referências e Imagens

[1] https://docs.splunk.com/observability/en/apm/apm-spans-traces/traces-spans.html

[2] https://microservices.io/

[3] https://encore.dev/

[4] https://opentelemetry.io/

[5] https://github.com/jpkrohling/opentelemetry-collector-deployment-patterns