# Infrastruktur

- Repo vorbereiten

```
git clone --recurse-submodules https://github.com/frzifus/ContainerConf-Workshop-2024.git
```

- Jaeger / Datenbank / ‚other'-app
  - `cd ContainerConf-Workshop-2024/source`
  - `podman-compose up`

Red Hat

# OpenTelemetry Workshop

## Instrumentieren von Applikationen

Heiko Rupp & Benedikt Bongartz

# Agenda

- Erster Teil:
  - Ein wenig Theorie
  - Präparation Infrastruktur
  - Quarkus - REST - App  als Java-Beispiel
  - Python - REST - Server
  - ggfls: Asynchrone Kommunikation via Kafka
- Zweiter Teil:
  - Kubernetes / Weitere Themen

# Ein wenig Theorie

# Theorie: Warum Tracing / OpenTelemetry

- Verfolgen von Aufrufen innerhalb einer Applikation
  - Finden von Unregelmäßigkeiten
  - Performance Optimierung
- Micro-Services
  - Viele Komponenten
- Aber auch Monolithen
  - Es ist ja nie nur ein Binary alleine

Red Hat

# Theorie: Warum Tracing / OpenTelemetry

- In Container-Umgebungen (z.B. Kubernetes / OpenShift)
  - Immutable Container
    - Ich kann nicht einfach eine Shell aufmachen
    - Ich kann oft nicht einfach einen Debugger / Profiler anhängen
  - Hohe Parallelität
    - An welche Instanz müsste ich den Debugger anhängen?

Red Hat

# Theorie: Warum Tracing / OpenTelemetry

- Für Monolithen
  - Normal Cluster von 3+ Instanzen wegen HA
  - Datenbank
  - Anbindung an ERP / Payment / …
- => Also auch hier ein verteiltes System

# Theorie: Warum Tracing / OpenTelemetry

- ‚Passives' Verfolgen von Aufrufen
- Weiteres Tool neben Metriken und Logs
- Metriken:
  - Zustand des Servers ( CPU-Last, Speicher, …)
  - Keine Info zu einem Request
- Logs:
  - Geschwätzig. Schwer mit Logs von anderer App zu korrelieren

Red Hat

# Theorie: Warum Tracing / OpenTelemetry

## OpenTelemetry


- Herstellerunabhängige Telemetriedatenerfassung
- Sammlung von
  - Spezifikationen
  - APIs
  - Werkzeugen
- zum Sammeln von **Traces**, Logs und Metriken
- Sprachen- und Tool- unabhängig
- https://opentelemetry.io/

Red Hat

# Tools die wir nutzen

- Open-Source Distributed Tracing System
- Entwickelt von Uber, jetzt Teil der CNCF (Cloud Native Computing Foundation)
- **OTLP Native**: Jaeger 2.0 basiert auf dem OpenTelemetry Collector!
- **End-to-End Distributed Tracing**: Verfolgt Anfragen über Microservices hinweg
- **Performance Monitoring**: Identifiziert Latenzen und Flaschenhälse
- **Abhängigkeitsanalyse**: Veranschaulicht Service-Abhängigkeiten
- **Root Cause Analysis**: Unterstützt bei der Fehlersuche und Analyse von Performance-Problemen

Red Hat

# Tools die wir nutzen

- Open-Source Monitoring- und Alarmsystem
- Ursprünglich von SoundCloud entwickelt, jetzt Teil der CNCF (Cloud Native Computing Foundation)
- **OTLP Native**: Seit Prometheus 3.0
- **Zeitreihen-Datenbank**: Speichert Metriken als Zeitreihen-Daten (z.B. CPU-Auslastung, Anfragen pro Sekunde)
- **Pull-basierte Datenerfassung**: Abfrage von Metriken durch Abholen (Pull) von definierten Endpunkten
- **Flexible Abfragesprache (PromQL)**: Leistungsfähige Sprache zur Abfrage und Analyse von Metriken

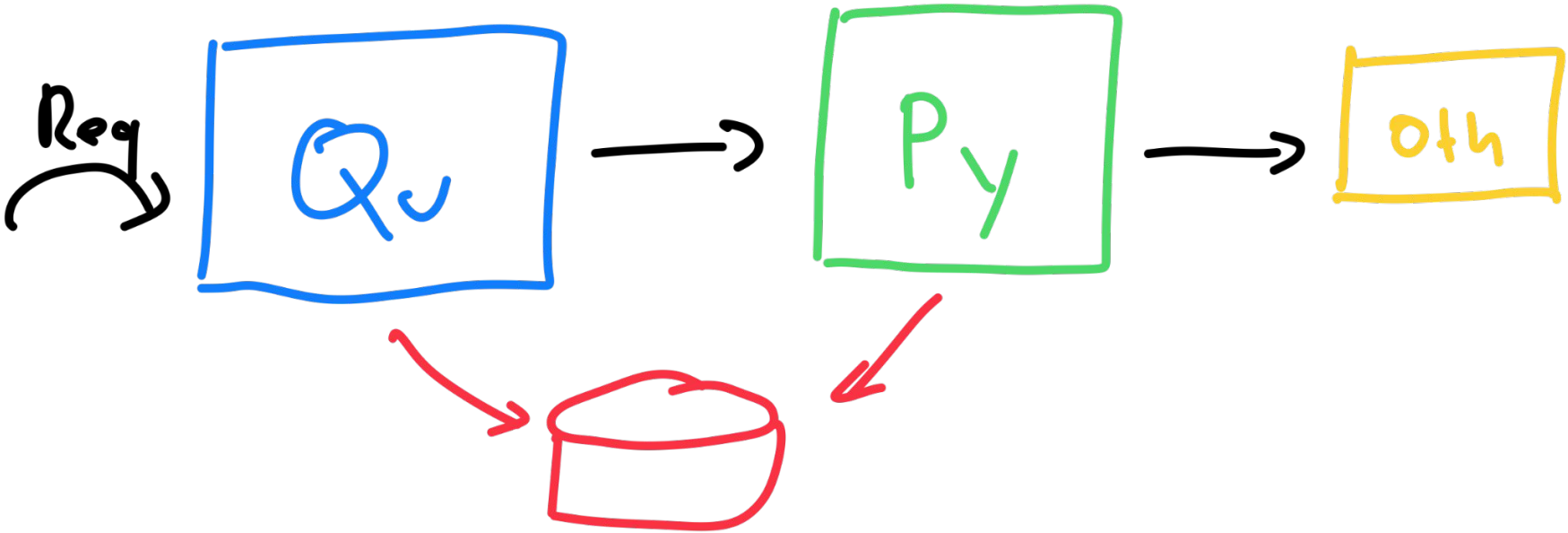
Red Hat

# OBSERVABILITY: TNG


Red Hat

# Theorie: Trace / Span - was ist das?

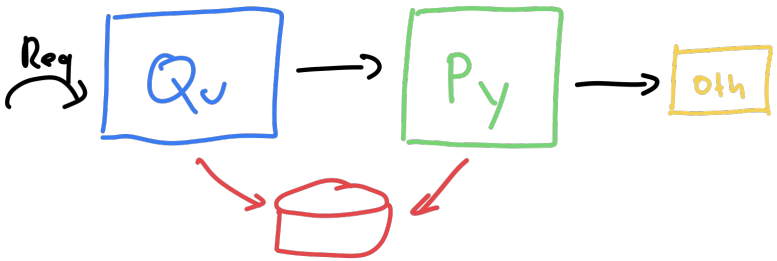
## Die Replikator-Applikation

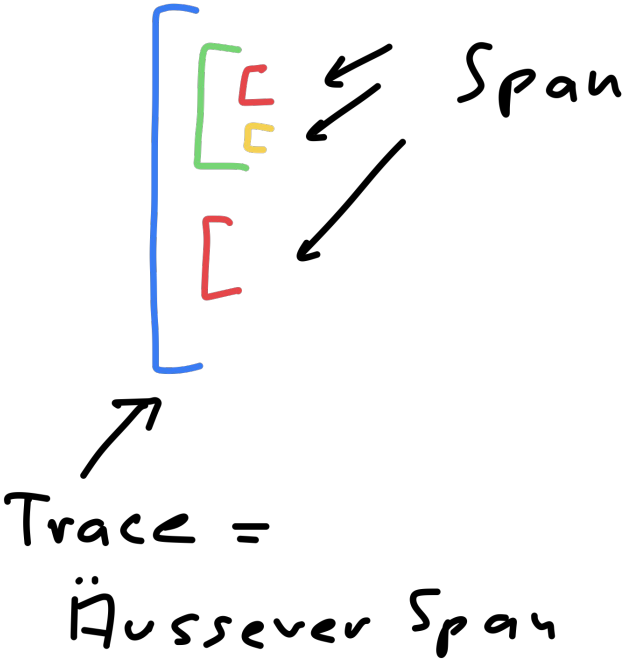



Red Hat

# Theorie: Trace / Span - was ist das?

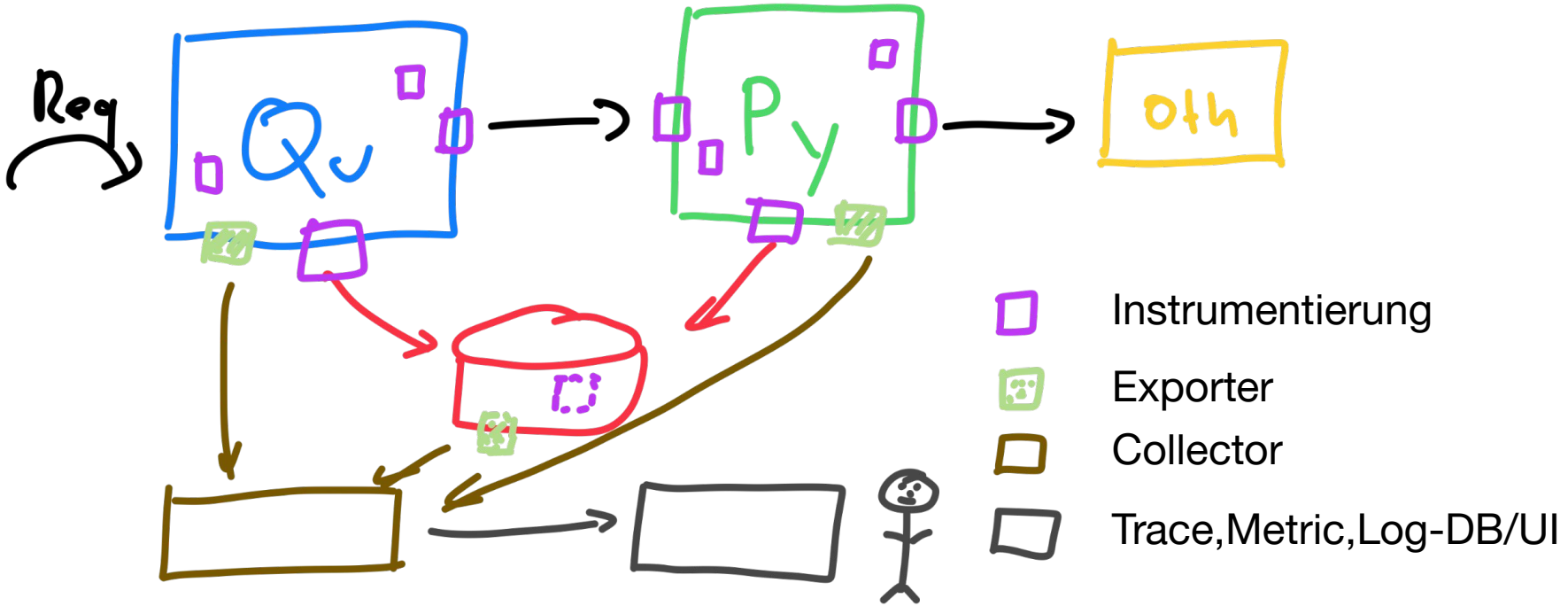
## Aufrufe verfolgen…




Red Hat

# Theorie: Tracing-Architektur




Red Hat

# Präparation Infrastruktur

# Infrastruktur

- Repo vorbereiten

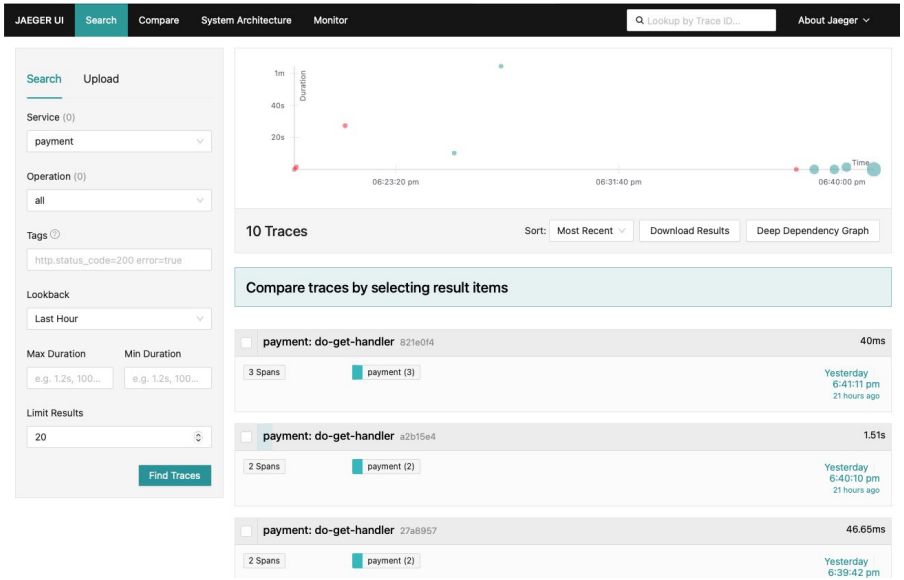
```
git clone --recurse-submodules https://github.com/frzifus/ContainerConf-Workshop-2024.git
```

- Jaeger / Datenbank / ‚other'-app
  - `cd ContainerConf-Workshop-2024/source`
  - `podman-compose up`

Red Hat

# Jaeger

http://localhost:16686/search




Red Hat

# Quarkus-REST-Applikation

# Setup

- `cd source/quarkus-replicator`
- `./mvnw quarkus:dev`

# Simpler Server

## replicator/TeaResource.java

```
@Path("/tea")
public class TeaResource {
@GET
@Produces(MediaType.TEXT_PLAIN)
public String makeTea(@QueryParam("kind") String kind)
     throws Exception {
  return "Here is your " + kind + " tea - enjoy!";

}
```

http://localhost:8080/

# Sehen Sie, Sie sehen nichts

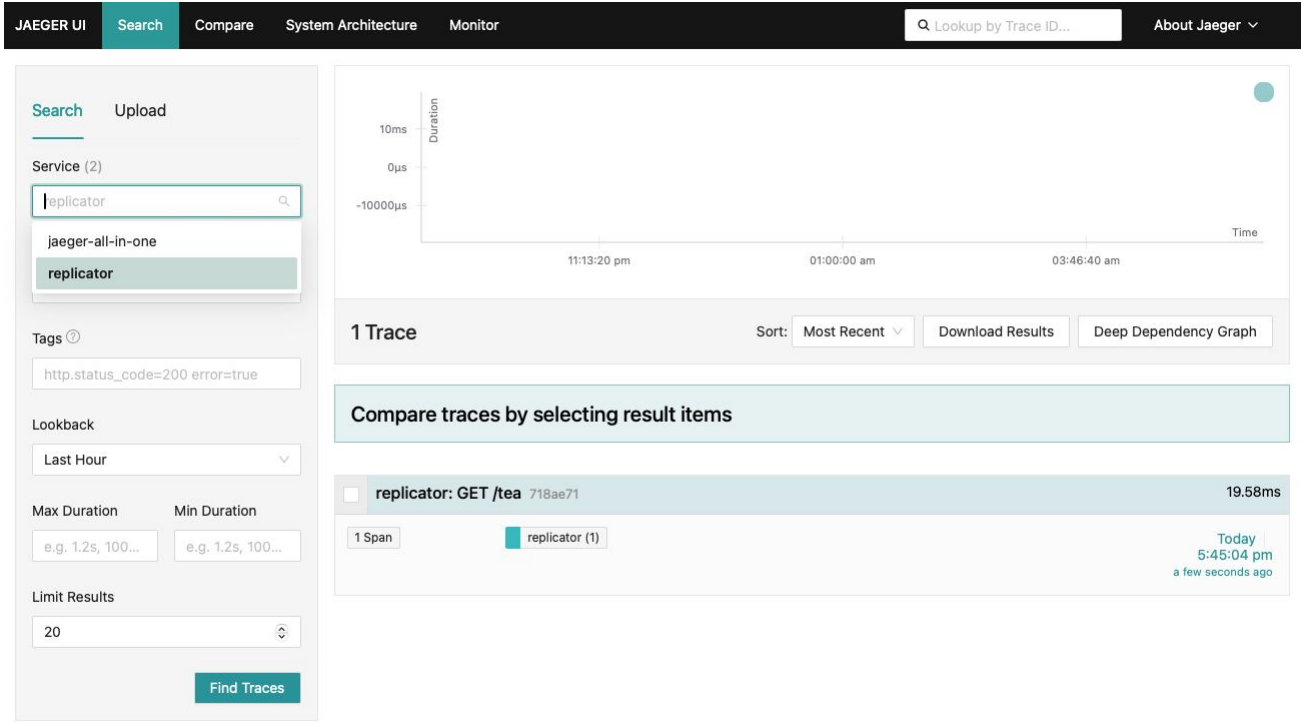
## Wir müssen die Instrumentierung anschalten

pom.xml

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-opentelemetry</artifactId>
</dependency>
```

application.properties

```
quarkus.application.name=replicator
quarkus.otel.exporter.otlp.endpoint=http://${otel.host:localhost}:4317
```




http://localhost:16686/search

Red Hat

# Tees via Datenbank validieren

## TeaResource.java

```
String name = kind.toLowerCase(Locale.ROOT);


Tea tea = Tea.findByName(name);
if (tea == null) {
    throw new NotFoundException("No such tea " + kind);
}
```

# JDBC-Instrumentierung

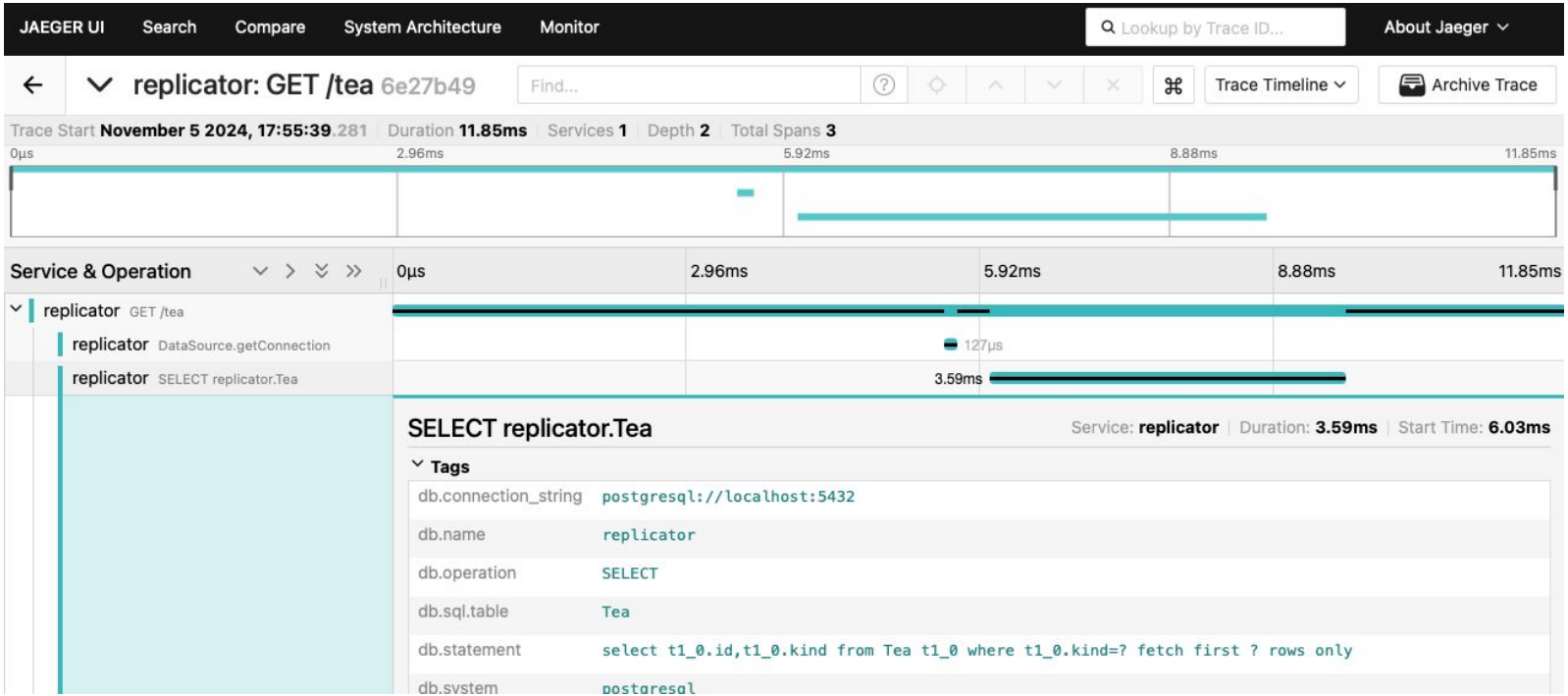
pom.xml

```
<dependency>
    <groupId>io.opentelemetry.instrumentation</groupId>
    <artifactId>opentelemetry-jdbc</artifactId>
</dependency>
```

application.properties

```
quarkus.datasource.jdbc.telemetry=true
```

Red Hat




http://localhost:16686/search

Red Hat

# Tee aufbrühen

## TeaResource.java

```
@Inject
Brewery brewery;

brewery.brewTea(kind);
```

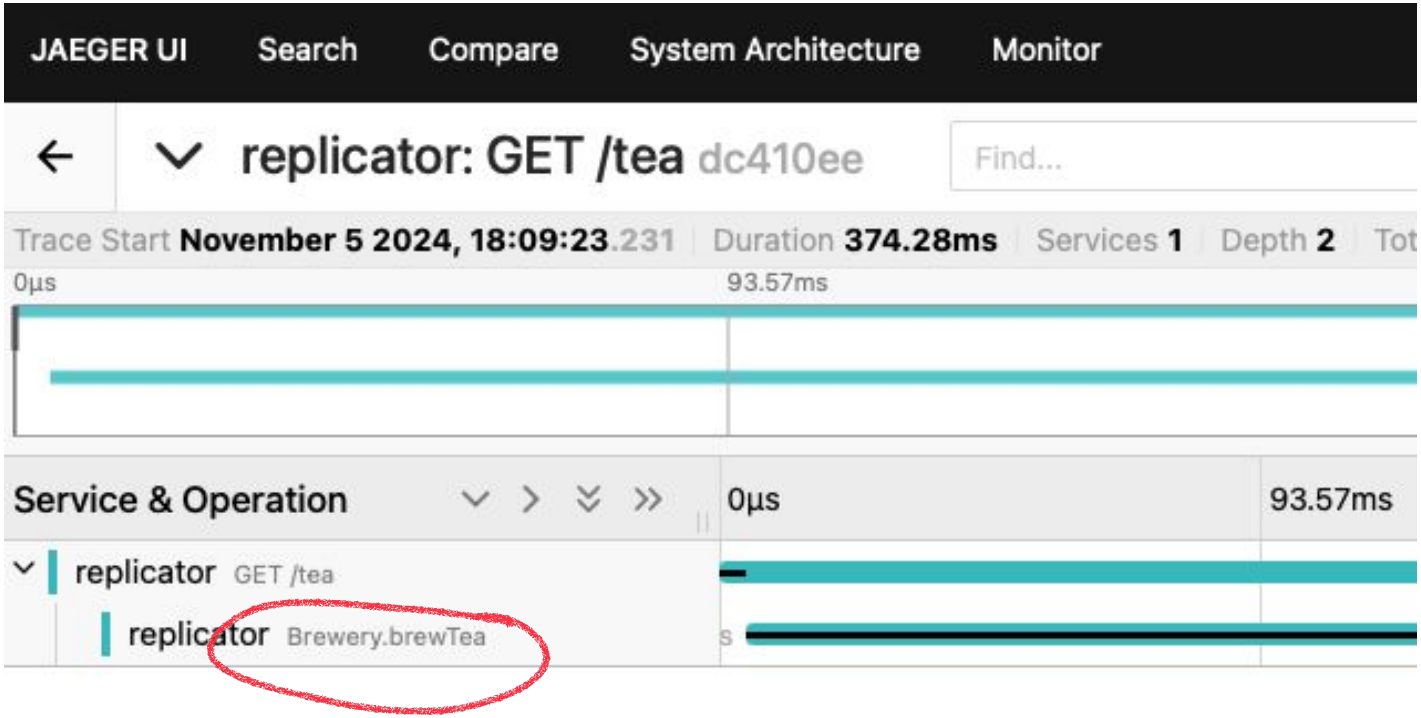

Red Hat

# Tee aufbrühen - eigener Span

## Brewery.java

```
@ApplicationScoped
public class Brewery {

    @WithSpan()
    void brewTea(@SpanAttribute(value = "kind
```

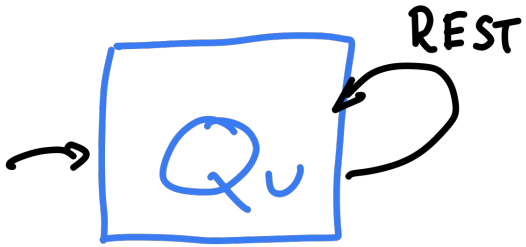
Red Hat


JAEGER UI
Search
Compare
System Architecture
Monitor
replicator: GET /tea dc410ee
Find...
Trace Start November 5 2024, 18:09:23.231
Duration 374.28ms
Services 1
Depth 2
0µs
93.57ms
Service & Operation
0µs
93.57ms
replicator GET /tea
replicator Brewery.brewTea

Red Hat

# „Externen“ PaymentService aufrufen

**TeaResource.java**

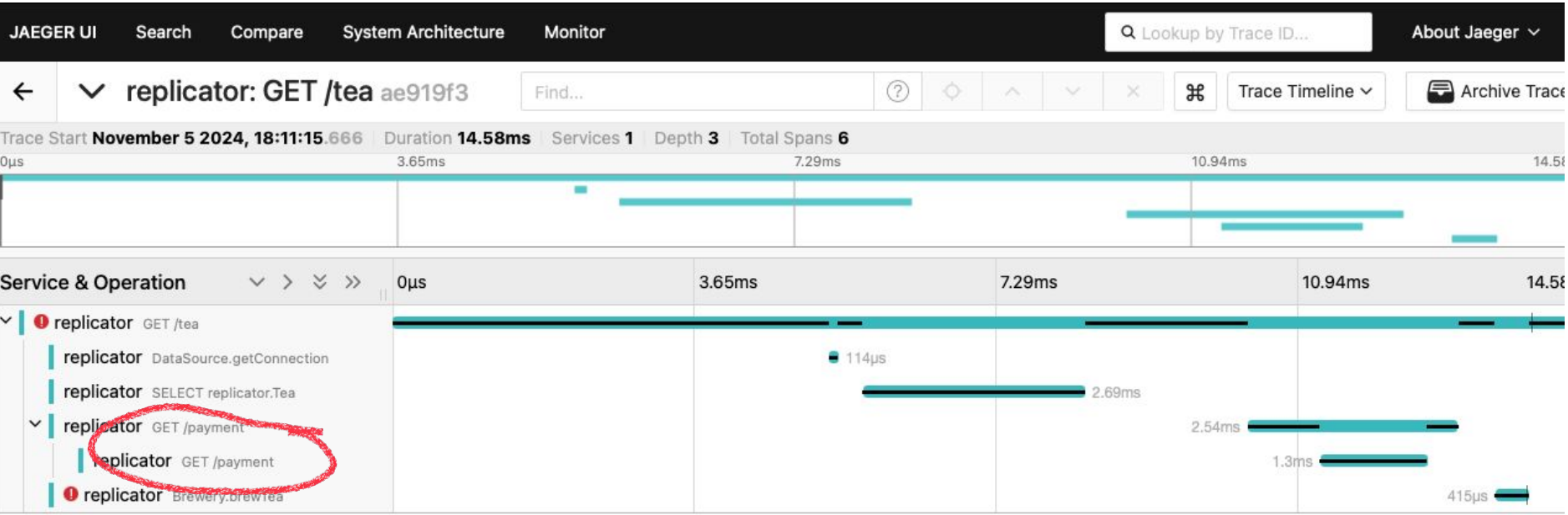



```
@RestClient
PaymentService paymentService;
```

```
boolean paid = checkPayment(kind);
if (!paid) {
    throw new NotPaidException(kind);
}
```

```
private boolean checkPayment(String kind) {
    return Boolean.parseBoolean(paymentService.isPaid(kind));
}
```

Red Hat


JAEGER UI
Search
Compare
System Architecture
Monitor
Lookup by Trace ID...
About Jaeger
replicator: GET /tea ae919f3
Find...
Trace Timeline
Archive Trace
Trace Start November 5 2024, 18:11:15.666
Duration 14.58ms
Services 1
Depth 3
Total Spans 6
0µs
3.65ms
7.29ms
10.94ms
14.58
Service & Operation
0µs
3.65ms
7.29ms
10.94ms
14.58
replicator GET /tea
replicator DataSource.getConnection
114µs
replicator SELECT replicator.Tea
2.69ms
replicator GET /payment
2.54ms
replicator GET /payment
1.3ms
replicator Brewery.brewTea
415µs

Red Hat

# Geschafft für den Moment…


Red Hat

# Python-REST-Applikation

# Setup

- Neues Terminal-Fenster
- `$ cd source/python-pay`
- Virtual env anlegen: `$ python3 -m venv ./venv`
- Aktivieren: `$ . venv/bin/activate`
- Pakete installieren: `$ pip install -r requirements.txt`

Red Hat

# Start mit minimalem Server

## server-minimal.py

❯ python3 server-minimal.py

Server started on port 8787

❯ curl localhost:8787

true%

127.0.0.1 - - [05/Nov/2024 22:17:06] "GET / HTTP/1.1" 200 -

Red Hat

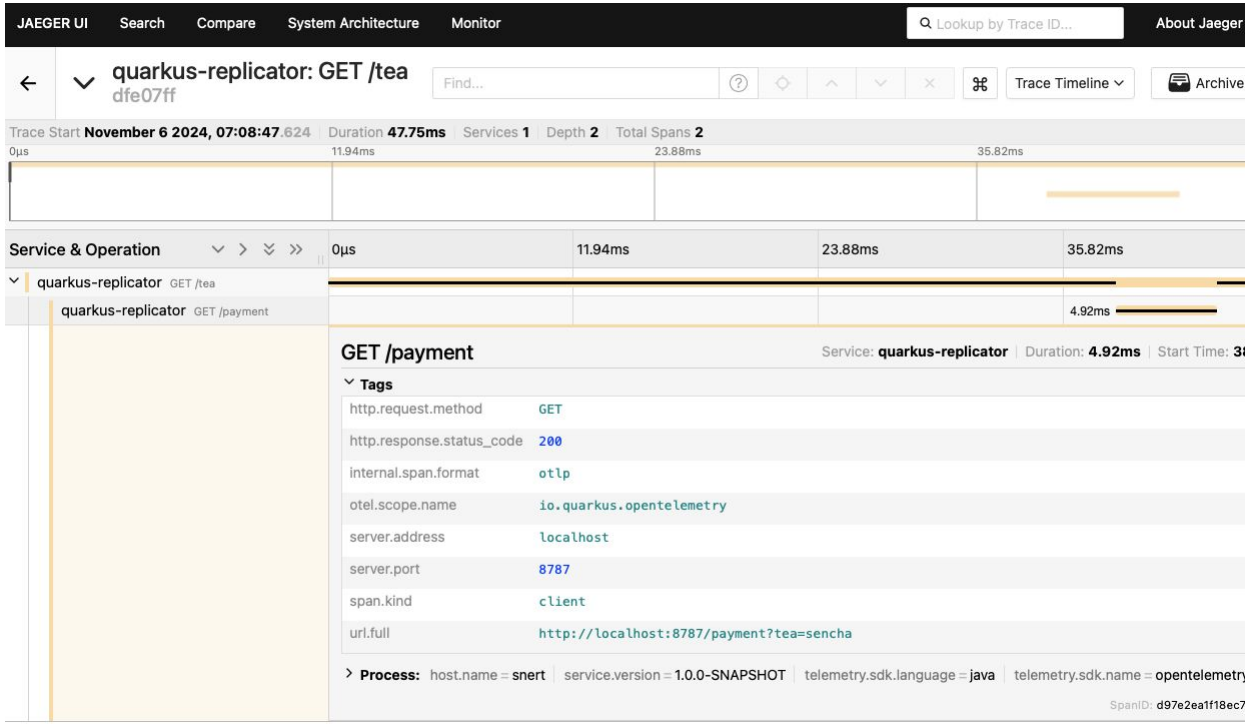
# Replicator mit externen Payment

```
# The remote payment service we consult
quarkus.rest-client."de.bsd.replicator.PaymentService".url= \
http://${payment.hostport:localhost:8080}
```

```
$ mvn quarkus:dev -Dpayment.hostport=localhost:8787
```

Red Hat

# Wir sehen noch nix in Jaeger




Red Hat

# Trace-Exporter

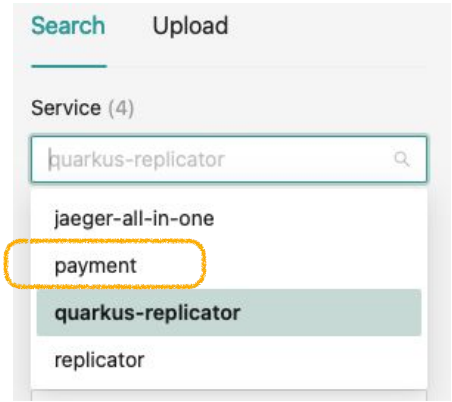
## server1.py

```
# Set up exporting
resource = Resource(attributes={
    SERVICE_NAME: "payment"
})

# Configure the provider with the service name
provider = TracerProvider(resource=resource)
# We need to provide the /v1/traces part when we use the http-exporter on port 4318
# For the grpc endpoint on port 4317, this is not needed.
processor = BatchSpanProcessor(OTLPSpanExporter(endpoint="http://localhost:4317"))
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)

tracer = trace.get_tracer(__name__)
```

# Instrumentierung von Hand

## server1-1.py

```
def do_GET(self):
    ctx = {}
    with tracer.start_as_current_span("do-get-handler",
        context=ctx) as span:
        self.send_response(200)
...
```

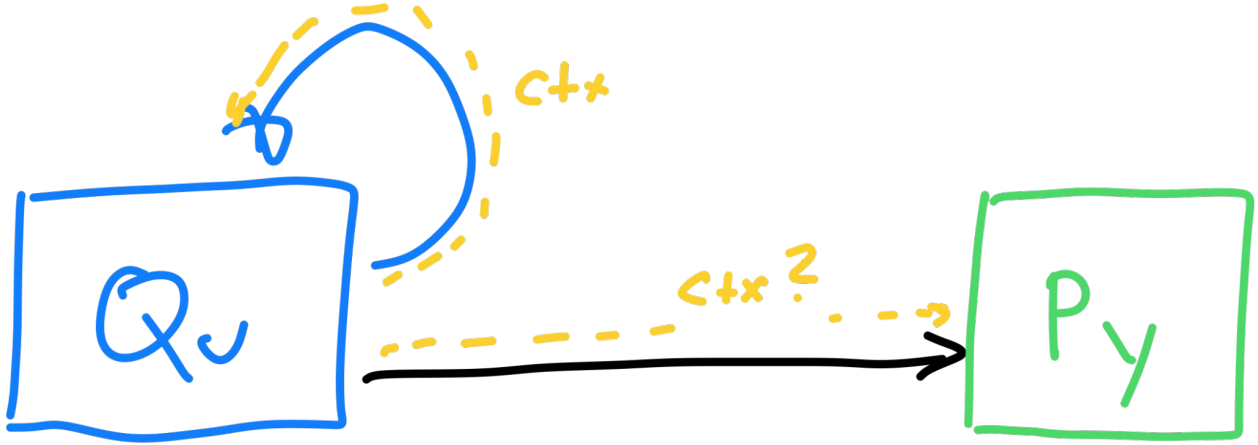



Red Hat

# Warum tut das nicht?

## Beim Payment in Quarkus hat es doch auch getan?

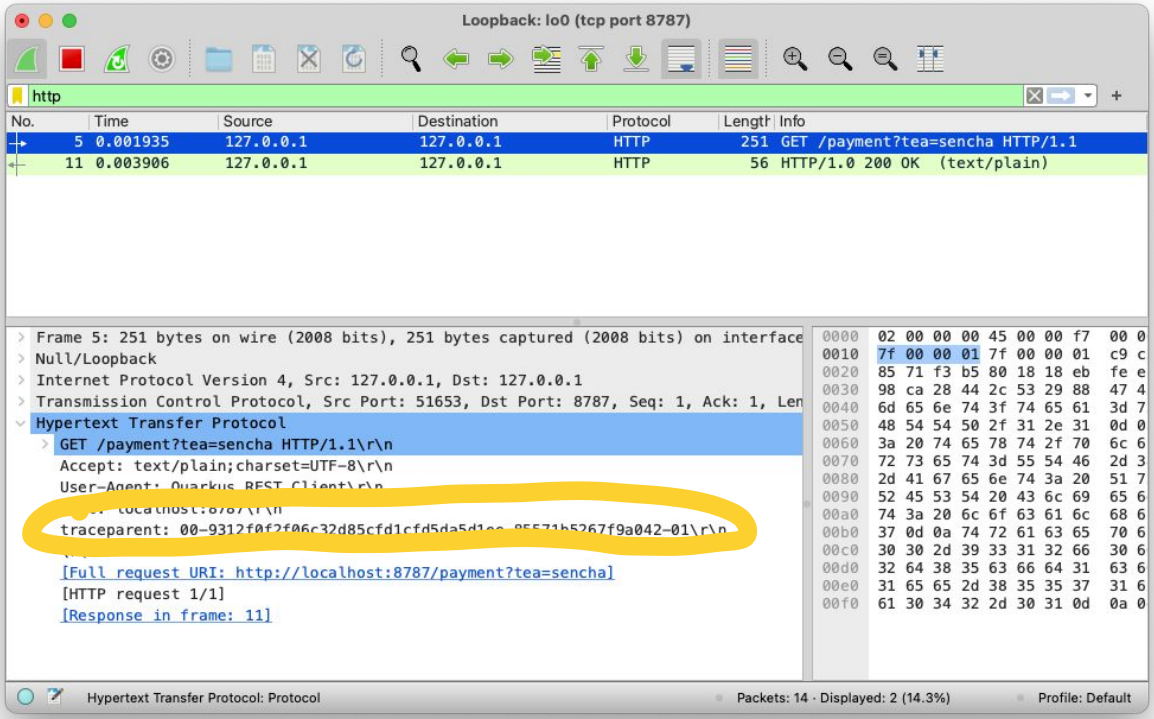
- Quarkus hat sowohl REST-Sever als auch -Client instrumentiert
- Wir müssen den Kontext weiter geben




Red Hat

# Einschub: Kontext weiterleiten

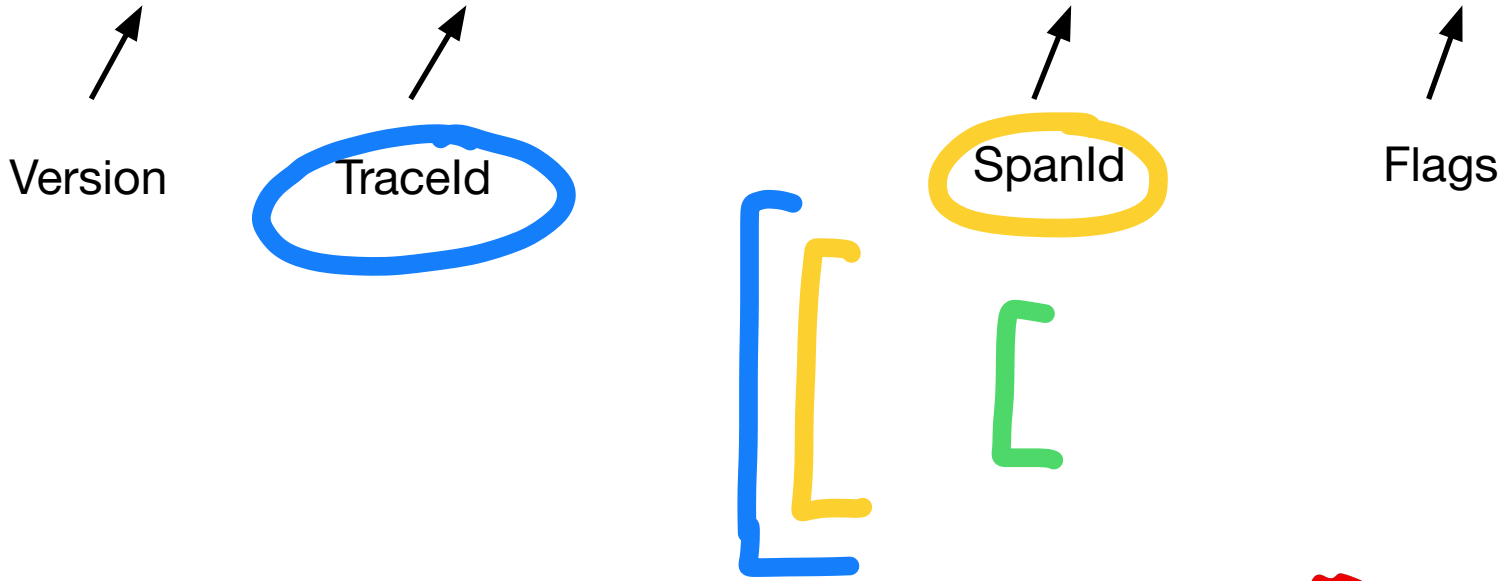
## Header anschauen in WireShark




Red Hat

# traceparent-http—header

traceparent: 00-9312f0f2f06c32d85cfd1cfd5da5d1ee-85571b5267f9a042-01



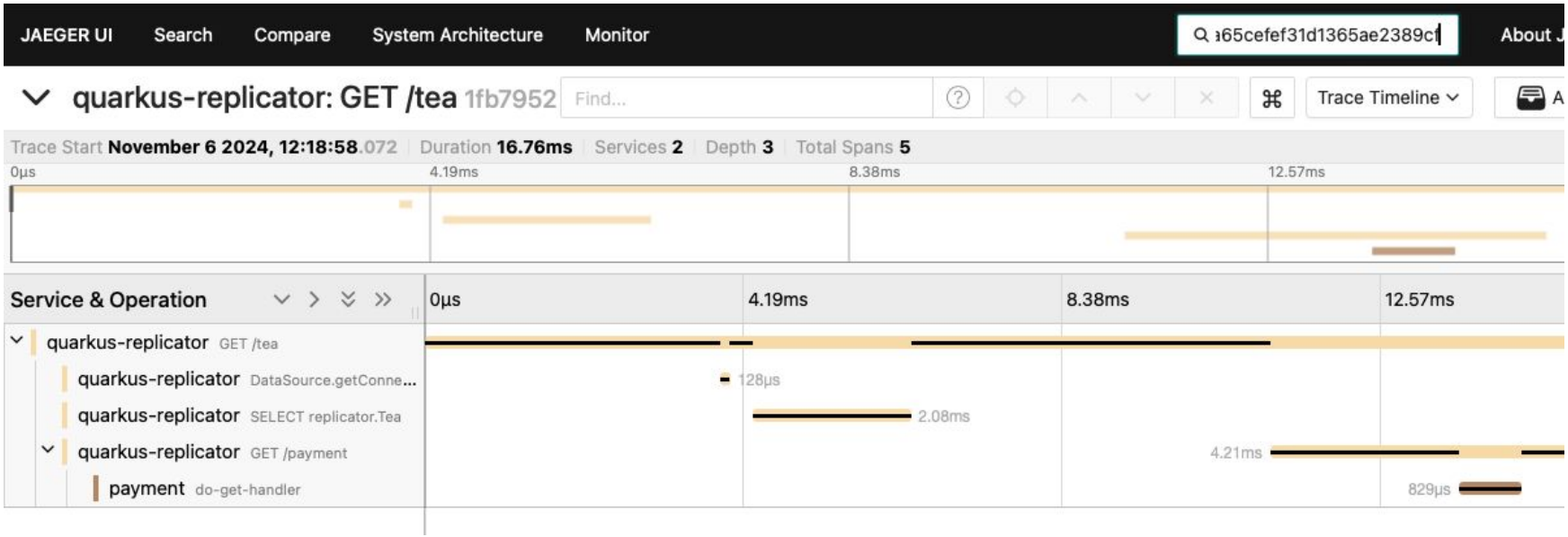

Red Hat

# Traceparent in Code

## server2.py

```
inc_trace = self.headers["traceparent"]
ctx = {}
span_context = None
if inc_trace is not None:
    print(inc_trace)
    span_context = extract_trace_data(inc_trace)
    ctx = trace
        .set_span_in_context(NonRecordingSpan(span_context))

with tracer.start_as_current_span("do-get-handler",
     context=ctx) as span:
```

Red Hat


JAEGER UI
Search
Compare
System Architecture
Monitor
Trace Timeline
quarkus-replicator: GET /tea 1fb7952
Find...
Trace Start November 6 2024, 12:18:58.072
Duration 16.76ms
Services 2
Depth 3
Total Spans 5
0µs
4.19ms
8.38ms
12.57ms
Service & Operation
quarkus-replicator GET /tea
quarkus-replicator DataSource.getConne...
128µs
quarkus-replicator SELECT replicator.Tea
2.08ms
quarkus-replicator GET /payment
4.21ms
payment do-get-handler
829µs

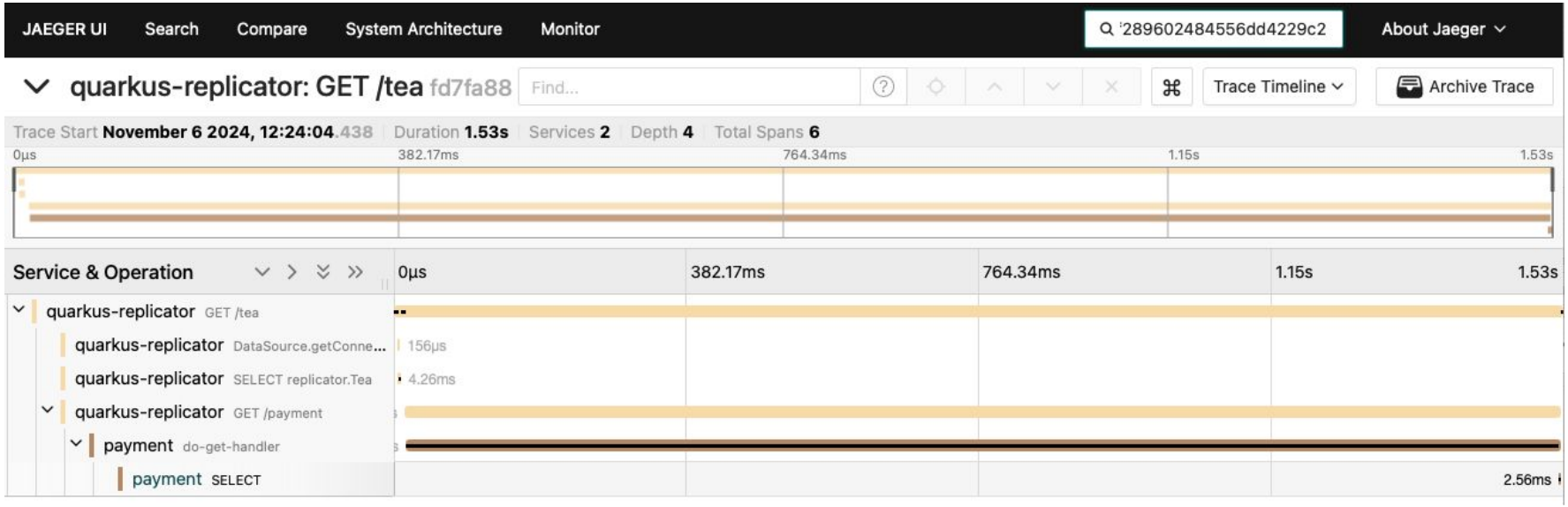
Red Hat

# Ist der Tee bezahlt?

## server3.py

```
Psycopg2Instrumentor().instrument(enable_commenter=True,
commenter_options={})

connection = psycopg2.connect(database="replicator",
                              user="demo",
                              password="lala7",
                              host="localhost",
                              port=5432)

cursor = connection.cursor()

cursor.execute("SELECT * FROM tea WHERE kind=(%s);", (tea, ) )
```

Red Hat


JAEGER UI
Search
Compare
System Architecture
Monitor
289602484556dd4229c2
About Jaeger
quarkus-replicator: GET /tea fd7fa88
Find...
Trace Timeline
Archive Trace
Trace Start November 6 2024, 12:24:04.438
Duration 1.53s
Services 2
Depth 4
Total Spans 6
0µs
382.17ms
764.34ms
1.15s
1.53s
Service & Operation
quarkus-replicator GET /tea
quarkus-replicator DataSource.getConne...
156µs
quarkus-replicator SELECT replicator.Tea
4.26ms
quarkus-replicator GET /payment
payment do-get-handler
payment SELECT
2.56ms

Red Hat

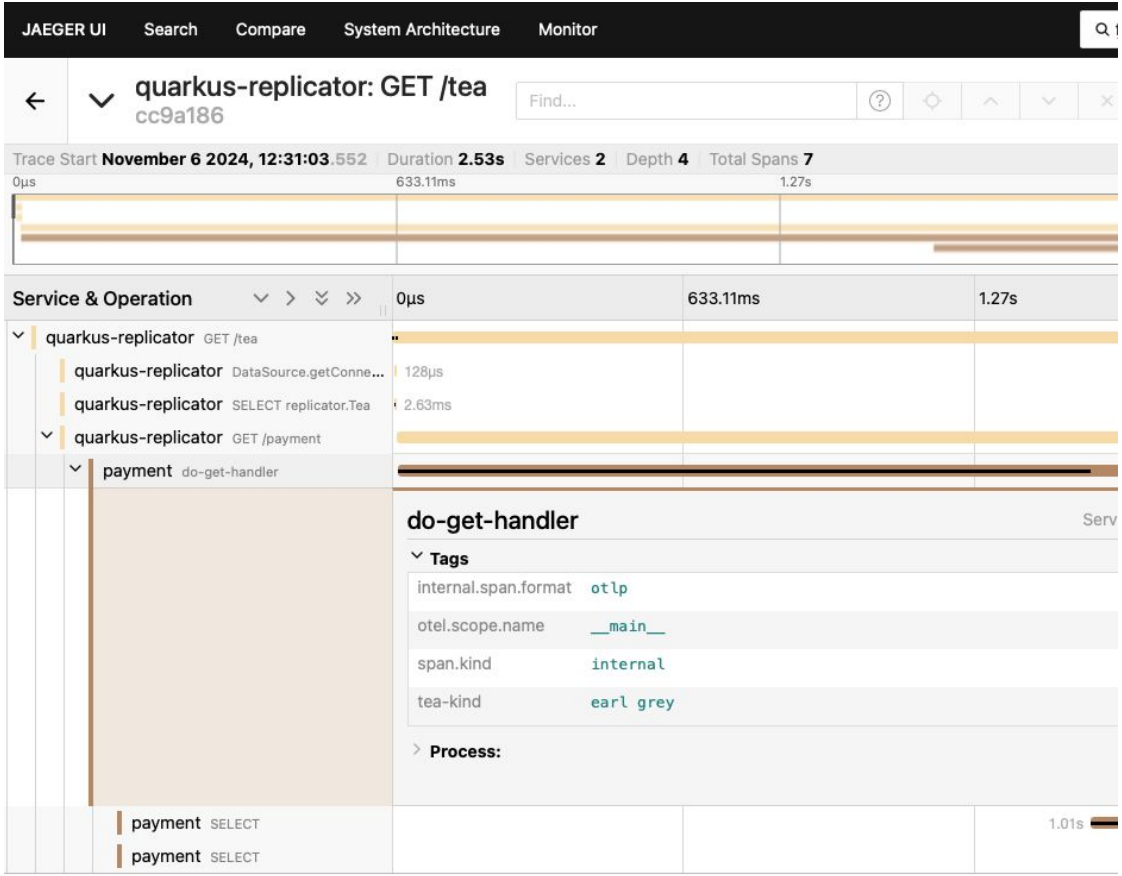
# Tee-Sorte im Span festhalten

**server4.py**

```
span.set_attribute("tea-kind", the_tea)
```

quarkus-replicator: GET /tea cc9a186

7 Spans | payment (3) | quarkus-replicator (4)




Red Hat

# Ein wenig Abrunden, Fehler verhindern..

## server5.py

```
# We get a request like 'GET /payment?tea=sencha HTTP/1.1',
#      let's parse it down
req = self.requestline
if not '?' in req:
    span.set_attribute('request_line', req)
    self.send_response(400)
    self.end_headers()
    # This must come after sending the headers
    span.set_status(StatusCode.ERROR, 'Bad request line')
    return
```

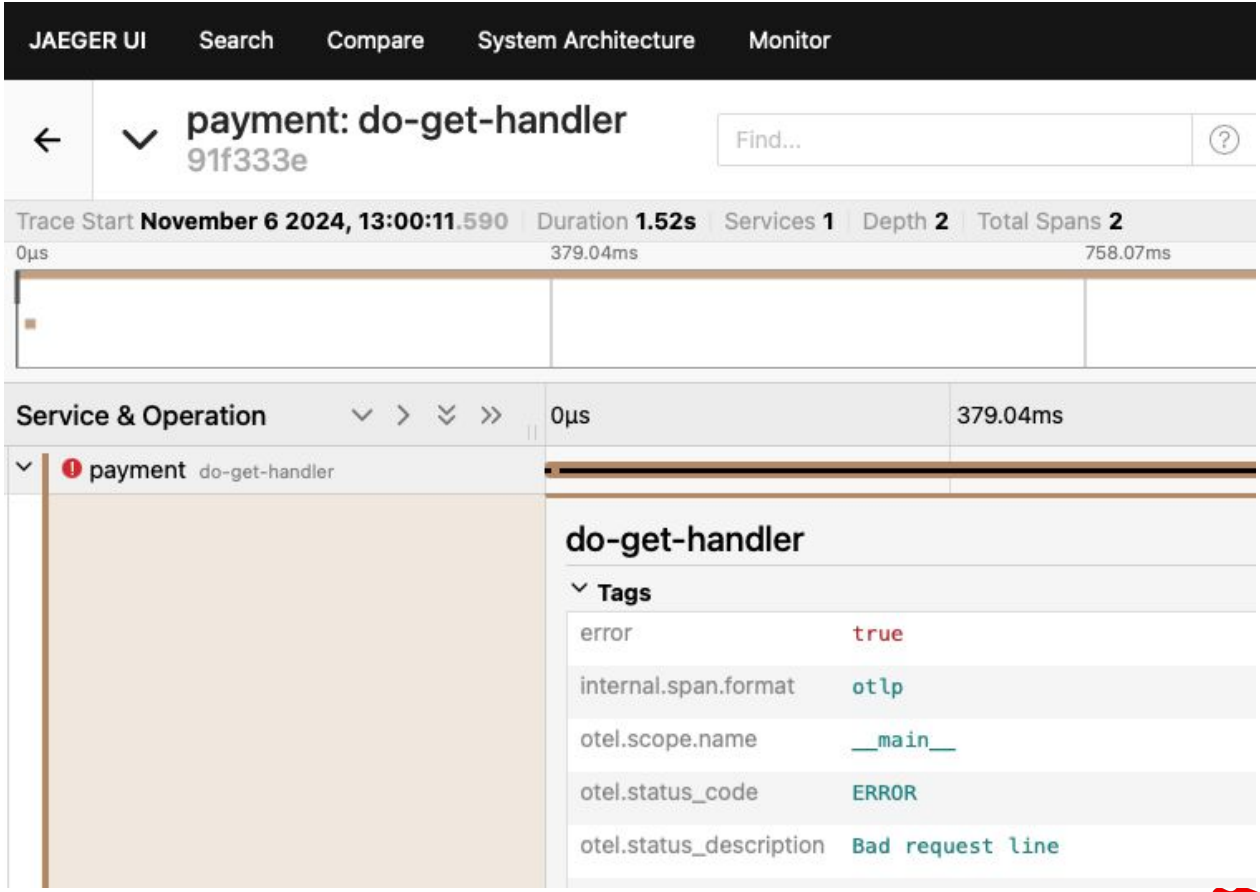
Red Hat

# … weiteren Server aufrufen

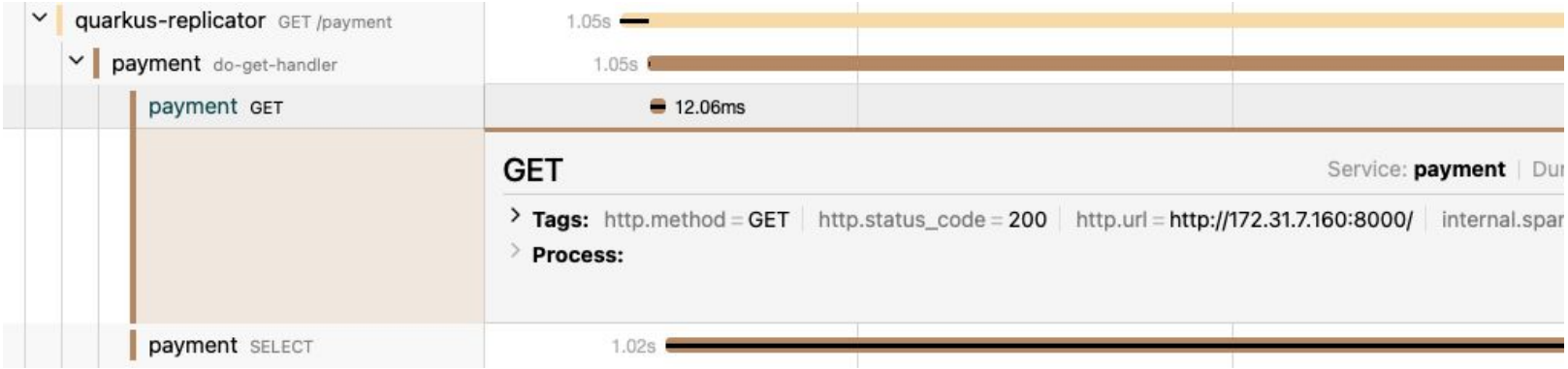
**server5.py**

```
    r = requests.get('http://' + args.other_hostport)
    print(r.status_code)


    RequestsInstrumentor().instrument()
```

```
› curl -i localhost:8787
```


JAEGER UI
Search
Compare
System Architecture
Monitor
payment: do-get-handler
91f333e
Find...
Trace Start November 6 2024, 13:00:11.590
Duration 1.52s
Services 1
Depth 2
Total Spans 2
0µs
379.04ms
758.07ms
Service & Operation
0µs
379.04ms
payment do-get-handler
do-get-handler
Tags
error true
internal.span.format otlp
otel.scope.name __main__
otel.status_code ERROR
otel.status_description Bad request line

Red Hat


quarkus-replicator GET /payment
1.05s
payment do-get-handler
1.05s
payment GET
12.06ms
GET
Service: payment
Tags: http.method = GET
http.status_code = 200
http.url = http://172.31.7.160:8000/
Process:
payment SELECT
1.02s

Red Hat

# Umgebungs“variablen“

## 12-Faktor-App

```
python server5.py \
   -pg.host=localhost \
   -otel_host=localhost \
   -other_hostport=localhost:8000
```

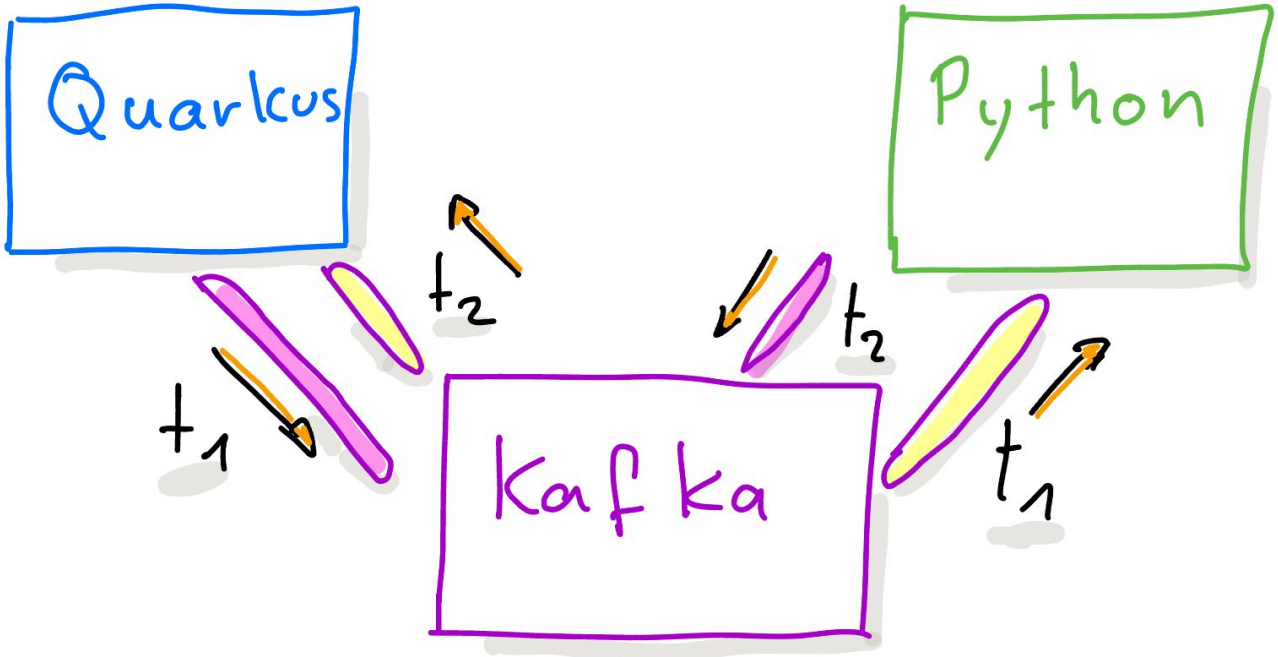
Red Hat

# tl;dr


Red Hat

# Wie geht das asynchron?

## zB via Apache Kafka?




Red Hat

# Quarkus

## TeaResource.java

```
@Inject
LoggerService loS;


loS.sendLog(kind, paid);
```

```
public class LoggerService {

    @Inject
    @Channel("topic1")
    Emitter<String> emitter;

    public void sendLog(String tea,
        boolean paid) {
        String message = tea + ':'
+ paid;
        emitter.send(message);
    }
```

# Kafkacat

```
$ kcat -C -t topic1 -b localhost:9092 -J
{  "topic":"topic1",
 "Headers":["traceparent","00-8533576e564e74dca5015e3e0391f11b-131d95f9275598ad-01"],
"Key":null,
"payload":"matcha:true"}
```

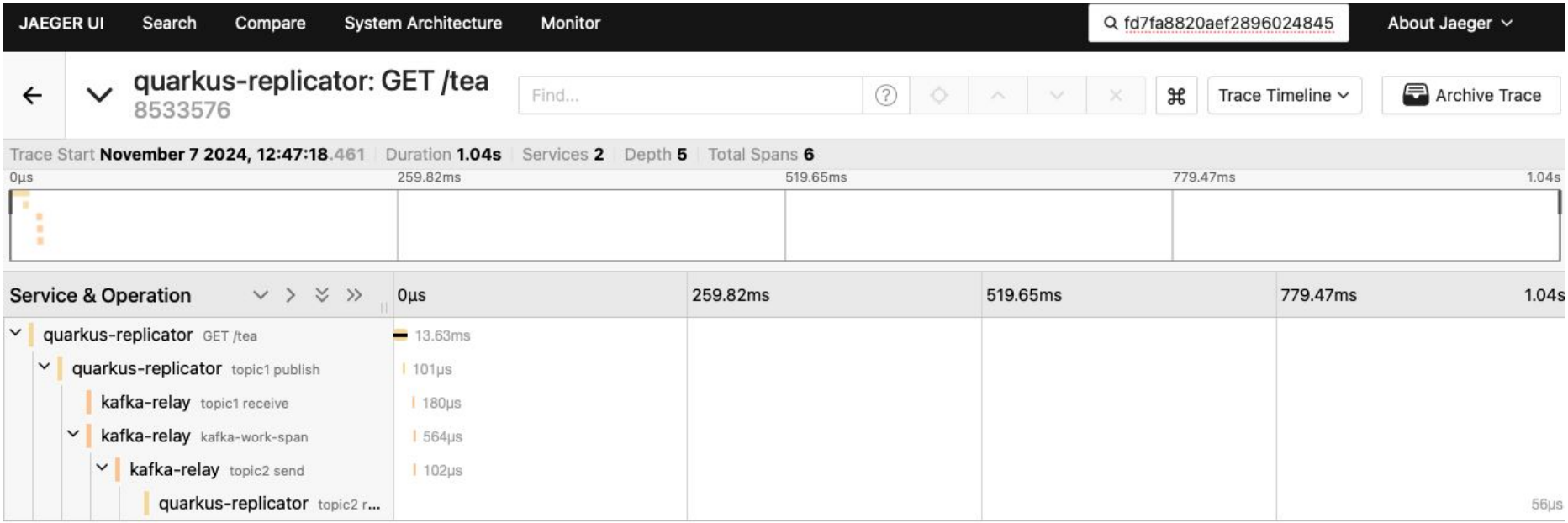
Red Hat

# Python

## kaf-relay.py

```
consumer = KafkaConsumer('topic1')
producer = KafkaProducer()

# Loop over incoming messages, process then and forward to topic2
for msg in consumer:
   trace_parent = get_trace_parent_header(msg)

   # Create a SpanContext object from the header value
   span_context = extract_trace_data(trace_parent)

   # Use this SpanContext as parent
   ctx = trace.set_span_in_context(NonRecordingSpan(span_context))

   with tracer.start_as_current_span("kafka-work-span", context=ctx)
        as span:
      # do the work
      body = msg.value.decode('utf-8')
      body = body + '  from Python'
      # and send it off to topic2
      producer.send('topic2', body.encode('utf-8'))
```

Red Hat


JAEGER UI
Search
Compare
System Architecture
Monitor
fd7fa8820aef2896024845
About Jaeger
quarkus-replicator: GET /tea
8533576
Find...
Trace Timeline
Archive Trace
Trace Start November 7 2024, 12:47:18.461
Duration 1.04s
Services 2
Depth 5
Total Spans 6
0µs
259.82ms
519.65ms
779.47ms
1.04s
Service & Operation
quarkus-replicator GET /tea
13.63ms
quarkus-replicator topic1 publish
101µs
kafka-relay topic1 receive
180µs
kafka-relay kafka-work-span
564µs
kafka-relay topic2 send
102µs
quarkus-replicator topic2 r...
56µs

Red Hat

# Geschafft für den Moment…


Red Hat

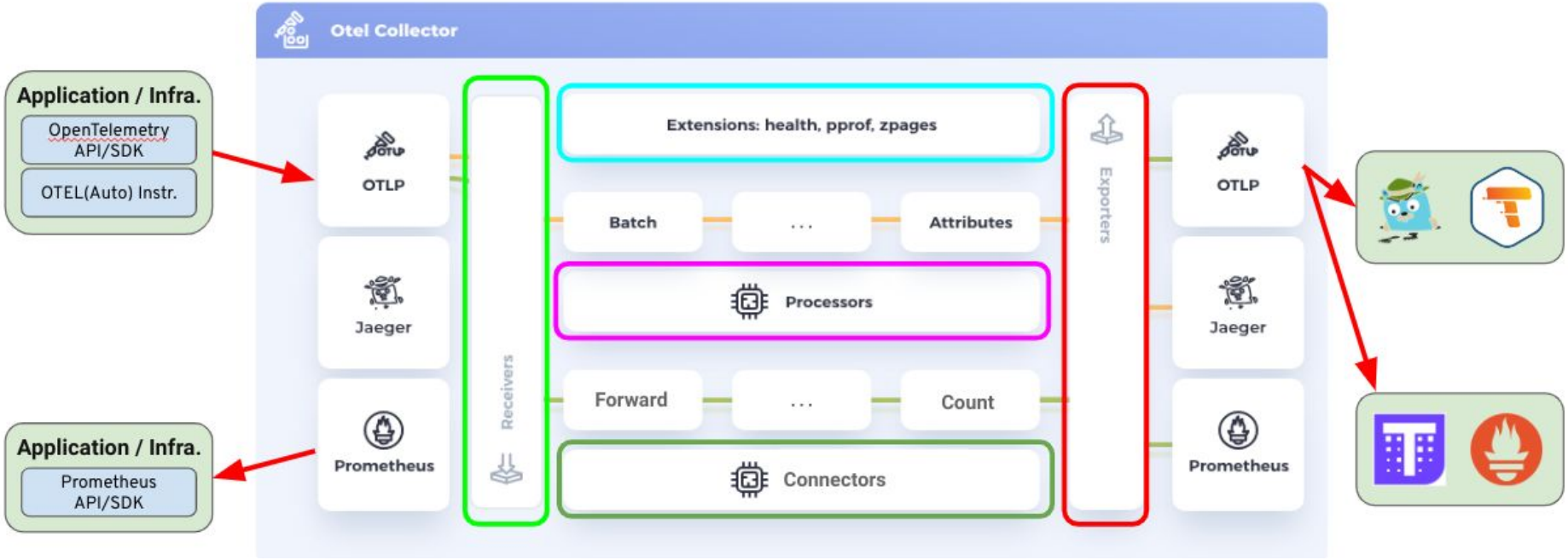
# Datenerfassung mit OpenTelemetry in der Kubernetes/OpenShift Umgebung

- Überblick über das OpenTelemetry-Projekt
  - Architektur und Einsatz des OpenTelemetry Collectors
  - Konfiguration und Deployment des Collectors
- Datenerfassung in Kubernetes und OpenShift
  - Einführung des Operators: Konfiguration und Deployment
  - Integration mit Service- und PodMonitors

Red Hat

# OpenTelemetry Collector

## Architektur




Red Hat

# OpenTelemetry Collector

## Konfiguration

```yaml
receivers:
  otlp:
    protocols:
      grpc:
      http:

exporters:
  prometheus:
    endpoint: "0.0.0.0:8889"

  otlp:
    endpoint: jaeger:4317
    tls:
      insecure: true

connectors:
  spanmetrics:

processors:
  batch:

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [spanmetrics, otlp]
    metrics/spanmetrics:
      receivers: [spanmetrics]
      exporters: [prometheus]
```

(Optional)

```yaml
receivers:
    prometheus:
      config:
        scrape_configs:
          - job_name: 'otel-collector'
            scrape_interval: 5s
            static_configs:
              - targets: ['0.0.0.0:8888']
```

Red Hat

# OpenTelemetry Collector

## Simple - Konfiguration

```
---
receivers:
  otlp:
    protocols:
      http:
        endpoint: 0.0.0.0:4318
processors:
  batch:

exporters:
  debug:

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

https://opentelemetry.io/docs/collector/configuration/

# OpenTelemetry Collector

## Konfiguration - Lokal testen

```
podman run --rm -it --network=host
ghcr.io/open-telemetry/opentelemetry-collector-contrib/telemetrygen traces --otlp-insecure
--otlp-http --duration 45s --rate 1 --otlp-endpoint="127.0.0.1:4318"
```

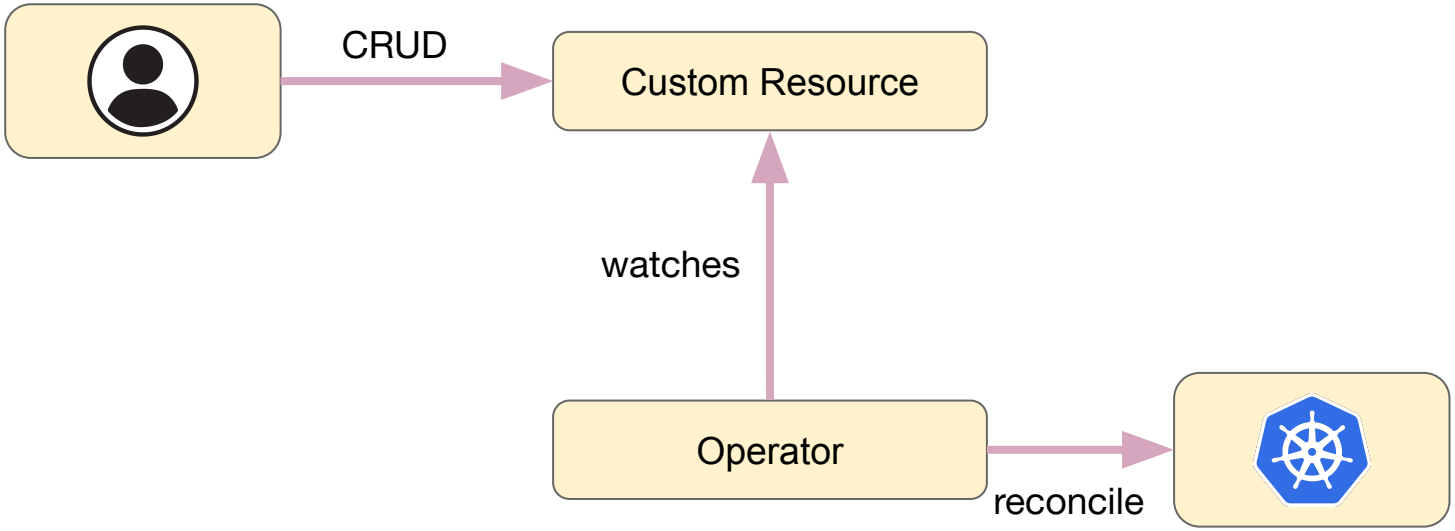
```
podman run --rm -it --network=host -v $(pwd)/config.yaml:/etc/otelcol-contrib/config.yaml:z
ghcr.io/open-telemetry/opentelemetry-collector-releases/opentelemetry-collector-contrib:0.113.0
```

```
...
2024-11-10T22:06:51.127Z  info    service@v0.113.0/service.go:261    Everything is ready. Begin running and processing data.
2024-11-10T22:07:02.752Z  info    Traces   {"kind": "exporter", "data_type": "traces", "name": "debug", "resource spans": 1, "spans": 2}
2024-11-10T22:07:03.755Z  info    Traces   {"kind": "exporter", "data_type": "traces", "name": "debug", "resource spans": 1, "spans": 2}
2024-11-10T22:07:05.759Z  info    Traces   {"kind": "exporter", "data_type": "traces", "name": "debug", "resource spans": 1, "spans": 2}
...
```

Red Hat

# OpenTelemetry Collector

## Operator




Red Hat

# OpenTelemetry Operator

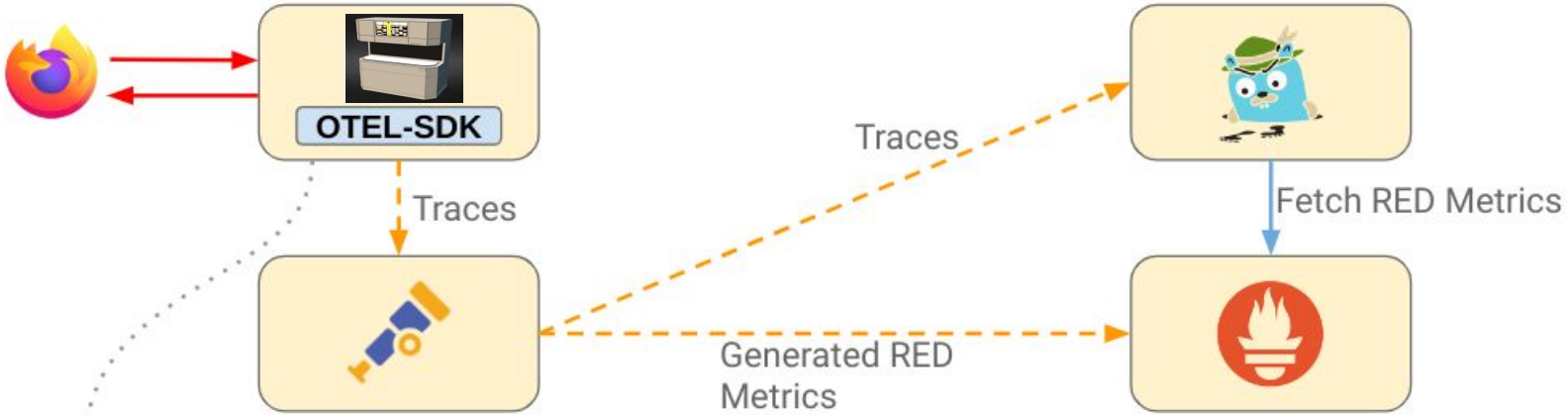
- Stellt schnittstellen bereit:
  - **opentelemetry.io/v1beta1**
    - **OpenTelemetryCollector**
  - opentelemetry.io/v1alpha1
    - OpenTelemetryCollector
    - Instrumentation
    - OpAMPBridge
    - Sampling (in devlopment)

# OpenTelemetry Operator

- Wechsel zu Repo

  https://github.com/frzifus/ContainerConf-Workshop-2024/blob/main/30-otel-k8s.md

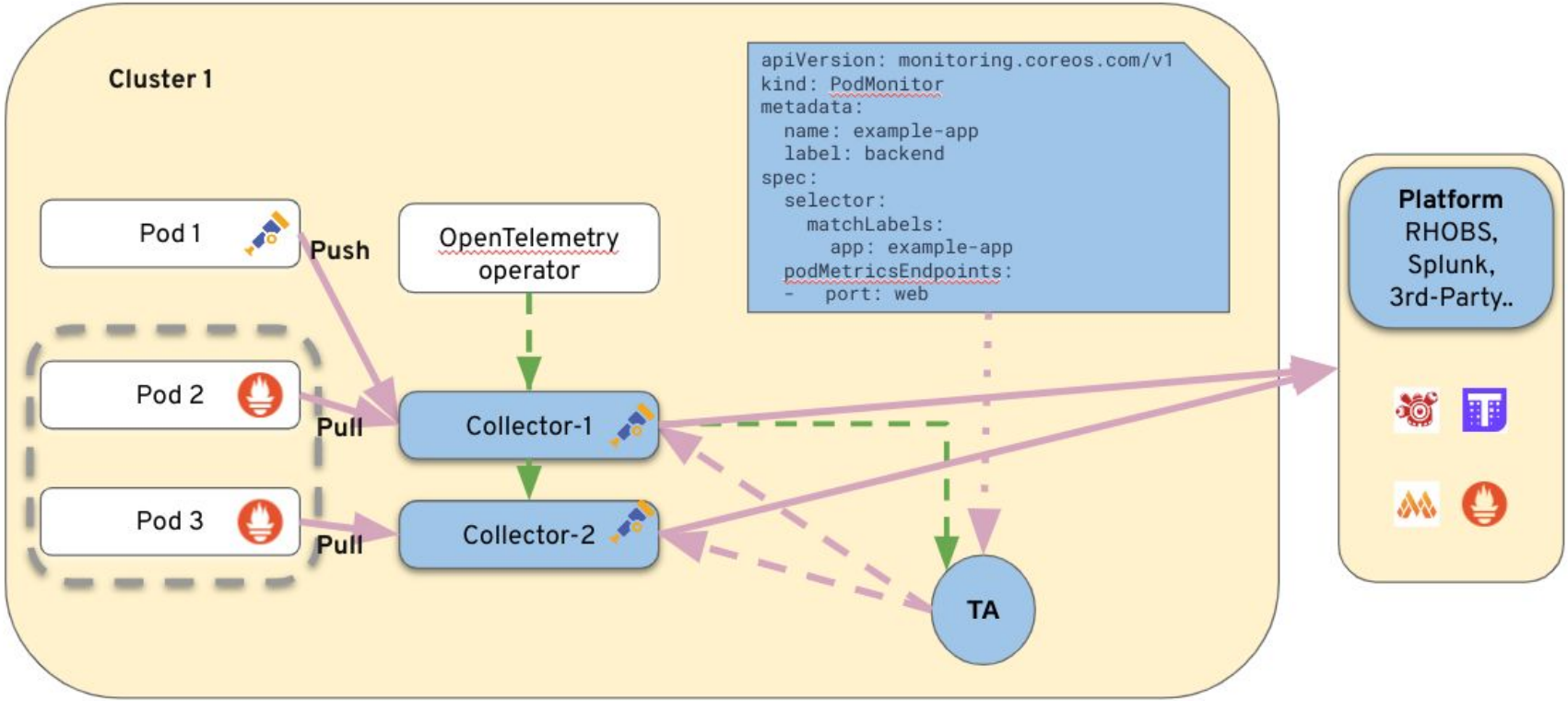
# OpenTelemetry Collector




```
with tracer.start_as_current_span("do-get-handler",
    context=ctx) as span:
```

Red Hat

# OpenTelemetry Collector




Red Hat

# Geschafft für den Moment…

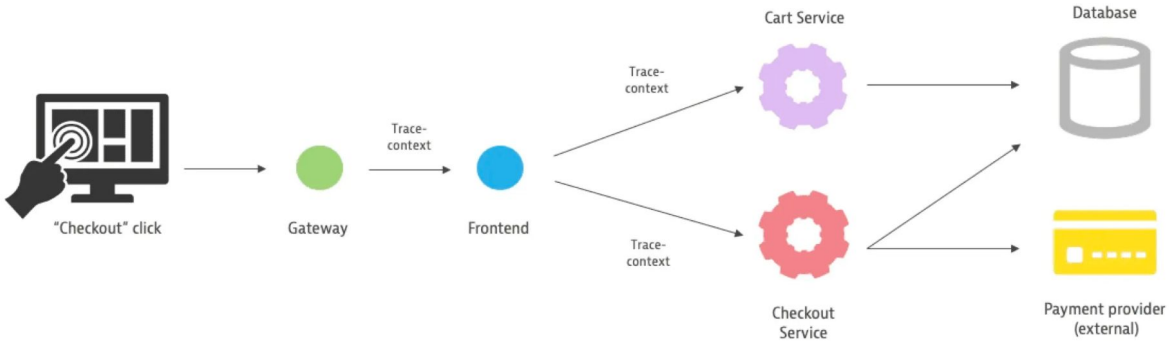

Red Hat

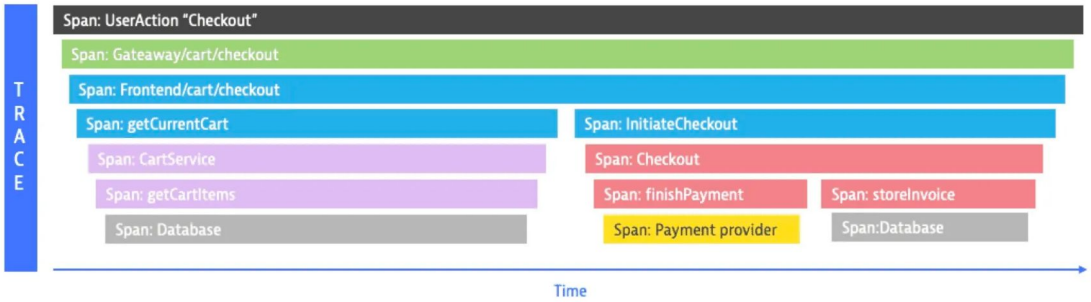
# Sampling und Filtering: Optimierung der Observability in verteilten Systemen

- Finale Theorierunde
  - Was ist sampling und warum sollte es uns interessieren?
  - Konfiguration des Tailbased sampling processors
    - Probleme und Lösungen
  - Demo setup

Red Hat

# Sampling Theorie

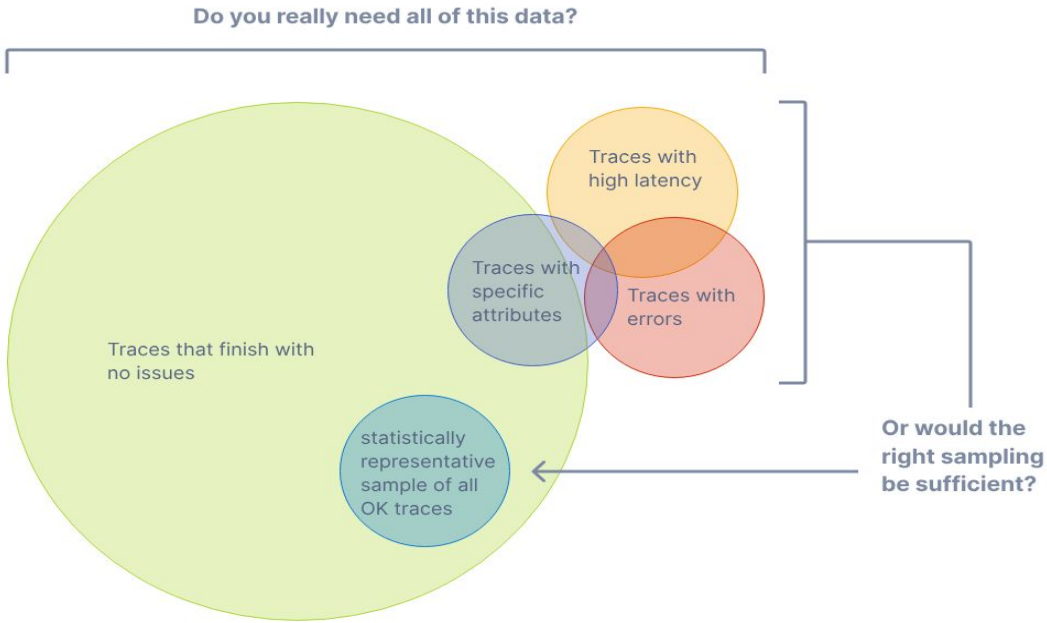



https://henesgokdag.medium.com/distributed-tracing-9300d55e7245

Red Hat

# Sampling Theorie




https://opentelemetry.io/docs/concepts/sampling/

Red Hat

# Sampling Theorie

- Setup (production environment, telco workloads)
  - ~30 Microservices, ~110 Nodes, ~2350 Pods, ~940 CPU

- Kostenberechnung bei der speicherung aller Telemetriedaten
  - 1.100.000 traces/min -> 1.100.000 * 60 * 24 * 30 * **100%** = 47.500.200.000
  - = 237.600 $ (27.01.2024, region=eu-west-1)

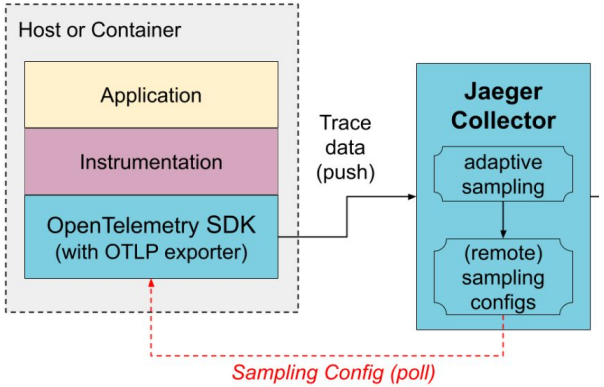
- Wie lassen sich die Unverhältnismäßig hohe kosten reduzieren?
  Wünschenswert wäre eine reduktion auf die **~0.1%** relevanten daten.

Red Hat

# Sampling Theorie

## Head-based sampling

- Sampling entscheidung wird bei **beginn** des Traces getroffen und über die services hinweg propagiert.'
- Effizient, einfach zu verstehen und zu konfigurieren
- Konfigurationsoptionen sind in SDKs verfügbar.
  - Manuell z.B. über Umgebungsvariablen
  - Jaeger Remote Sampling-Erweiterung (docs)
- Alternativer Probabilistischer Probenehmer-Prozessor (docs)

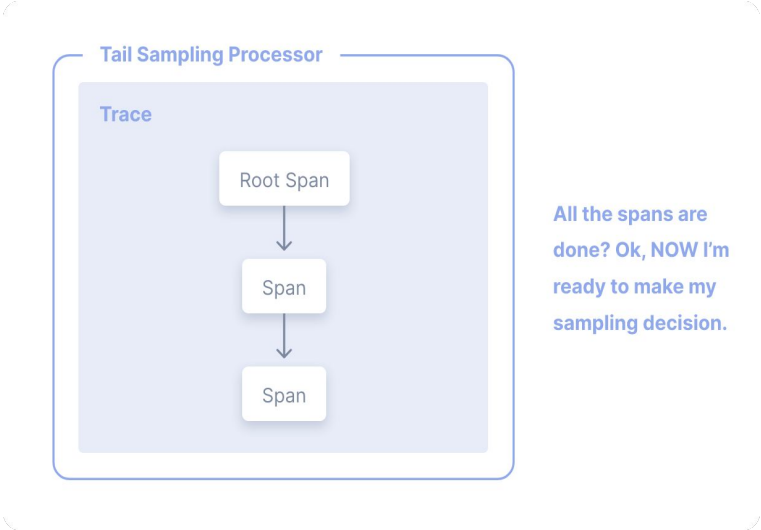



Red Hat

# Sampling Theorie

## Tail-based sampling

- Die Entscheidung über Sample wird am **Ende** eines traces getroffen
- Eine Informierte entscheidung kann nur getroffen werden, wenn alle spans zu einem trace vorliegen
- Ermöglicht komplexe Richtlinien
- **Verbraucht zusätzliche Ressourcen**
- Verwende tail-basiertes Sampling, wenn seltene oder extreme Fälle untersucht werden sollen, welche erhebliche Auswirkungen haben könnten oder besondere Aufmerksamkeit erfordern




https://opentelemetry.io/docs/concepts/sampling#tail-sampling

Red Hat

# Sampling Theorie

## Kostenreduktion

- Calculated cost telo/xray setup - 1.1 Mio traces per Minute
  - **100%** sampling => 237.600 $
  - **0.1%** sampling => 237 $ + (sampling cost)

- Zusätzliche Kosten für Tailbased Sampling
  - (Cluster resource limits 15000Mi + 6500m CPU)

  - Instanz optionen
    - Type: t2.xlarge  per H: 0,1856 USD  Cores: 4  Mem: 16 GiB
      - Xray(237 $) + t2.xlarge(133 $) = **370 $**
    - Type: t2.2xlarge  per H: 0,3712 USD  Cores: 8  Mem: 32 GiB
      - Xray(237 $) + t2.2xlarge(267 $) = **504 $**

.

Red Hat

# Sampling Theorie

## Konfiguration

```yaml
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
 name: sampling-col
spec:
 mode: deployment
 replicas: 5
 resources: ...
 config: |
   receivers:
     otlp:
   processors:
     memory_limiter: ...
     batch/traces: ...
     tail_sampling: ...
   exporters:
     otlp/tempo: ...
   service:
     pipelines:
       traces:
         receivers: [otlp]
         processors: [memory_limiter, batch/traces, tail_sampling]
         exporters: [otlp/tempo]
```

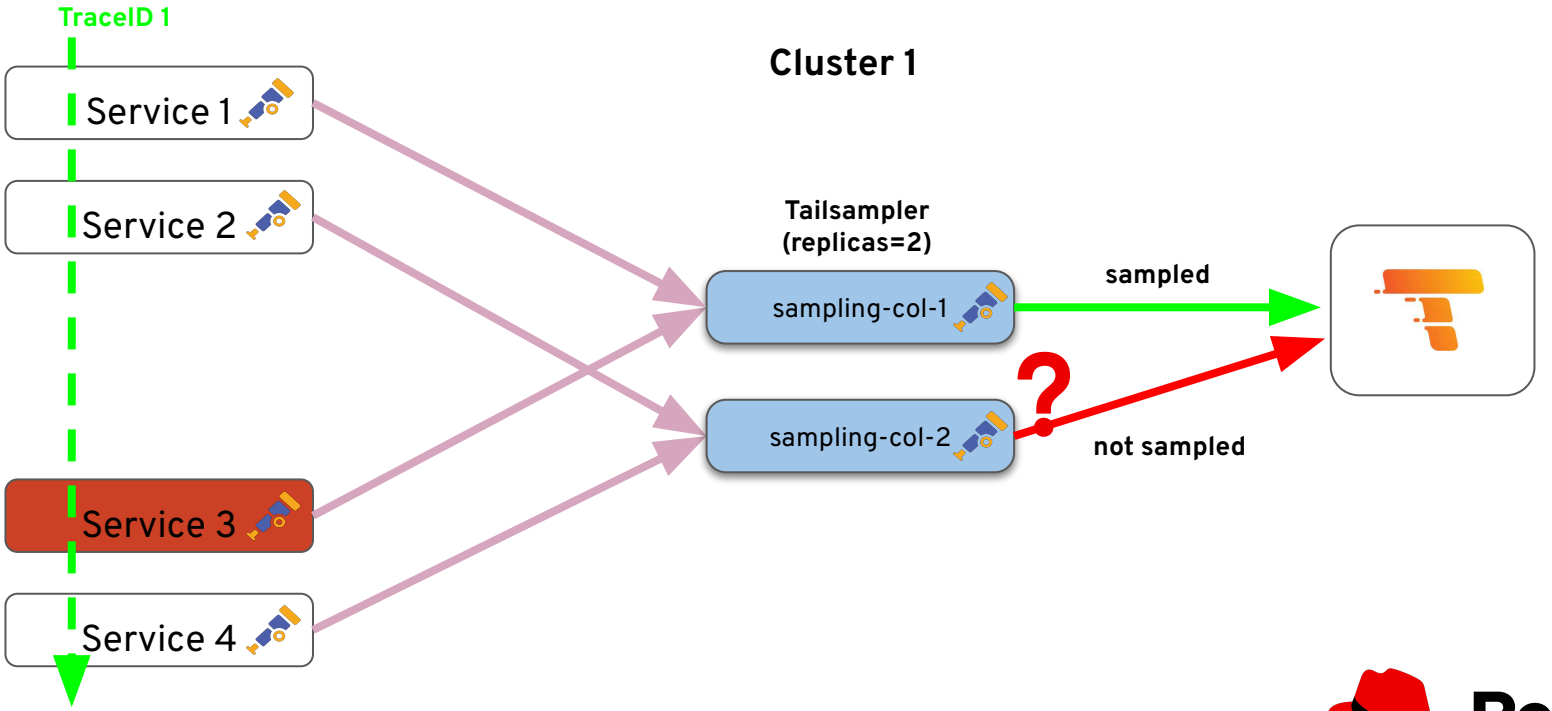
```yaml
resources
  requests:
    memory: "3000Mi"
    cpu: "1300m"
  limits:
    memory: "3000Mi"
    cpu: "1300m"
```

```yaml
tail_sampling:
  decision_wait: 10s
  num_traces: 100000
  policies
    - name: policy-errors-retain
      type: status_code
      status_code: {status_codes: [ERROR]}
    - name: policy-probabilistic
      type: probabilistic
      probabilistic:
        sampling_percentage: 10
```

Red Hat

# Sampling Theorie

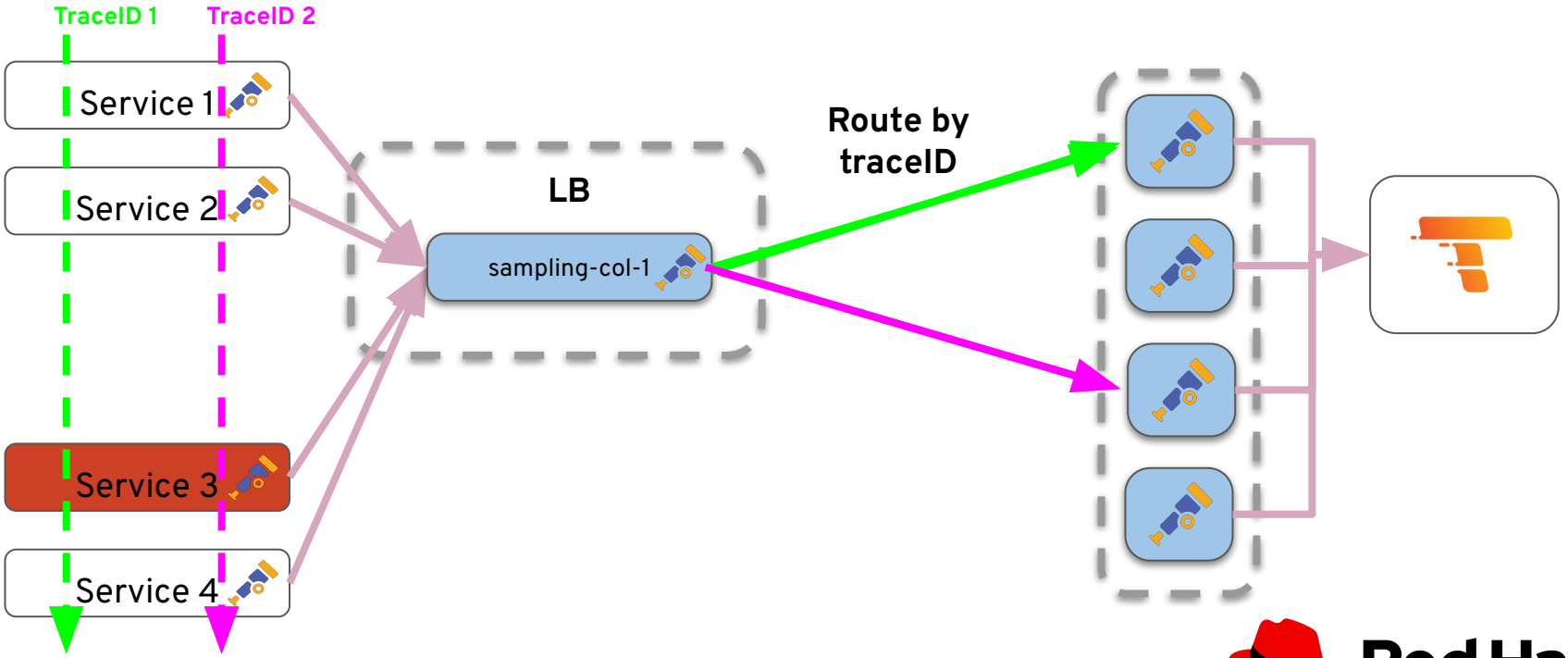
## Probleme




Red Hat

# Sampling Theorie

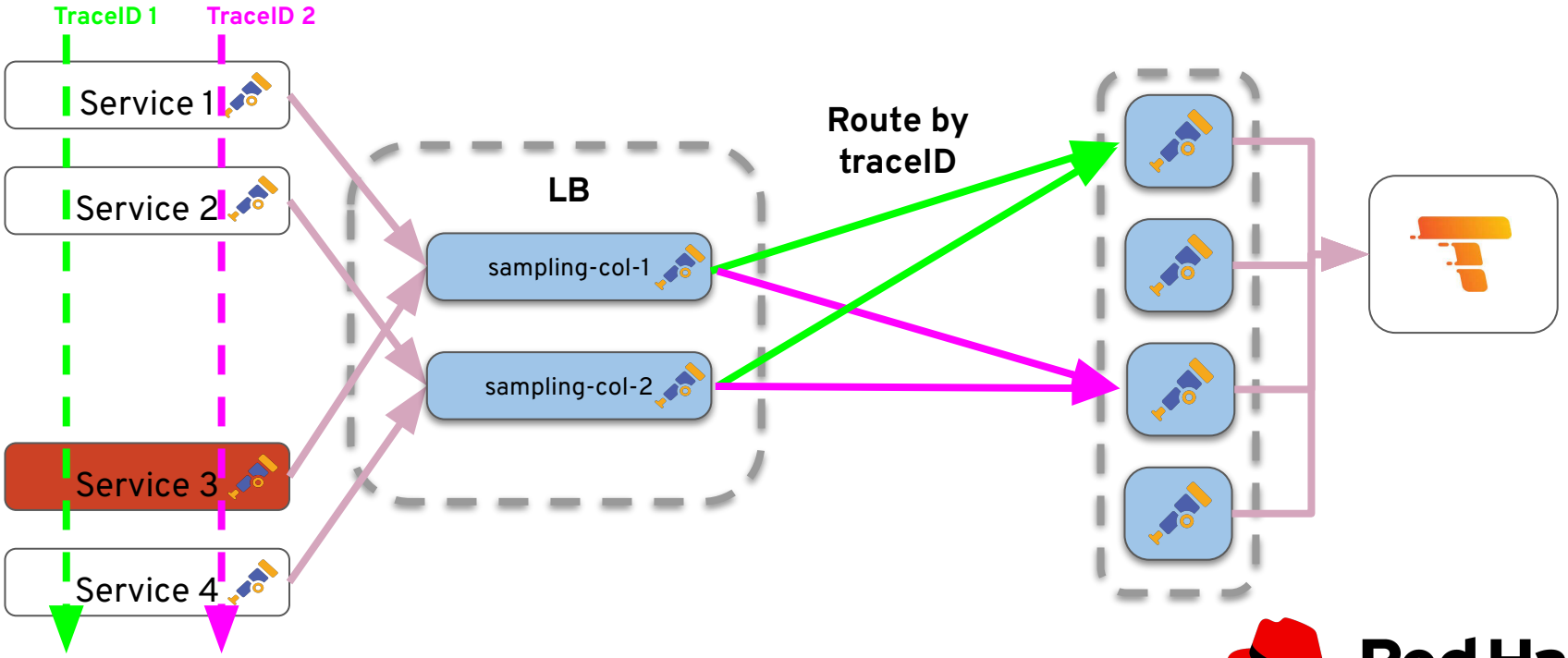
## Scaling




Red Hat

# Sampling Theorie

## Scaling




Red Hat

# Sampling Theorie

## Load-balancing exporter

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
 name: telemetry-lb
spec:
 mode: deployment
 replicas: 2
 config: |
   exporters:
     loadbalancing:
       routing_key: traceID
       protocol:
         otlp:
           sending_queue
             queue_size: 4000
       resolver:
         dns:
           hostname: sampling-collector-headless.observability-backend.svc.cluster.local
           port: 4317
   service:
     pipelines:
       traces:
         exporters: [loadbalancing]
```

```
resolver:
     k8s:
       service: lb-svc.kube-public
       ports:
         - 15317
         - 16317
resolver
  static:
    hostnames:
    - backend-1:4317
    - backend-2:4317
    - backend-3:4317
    - backend-4:4317
```

Red Hat

# Geschafft für den Moment…

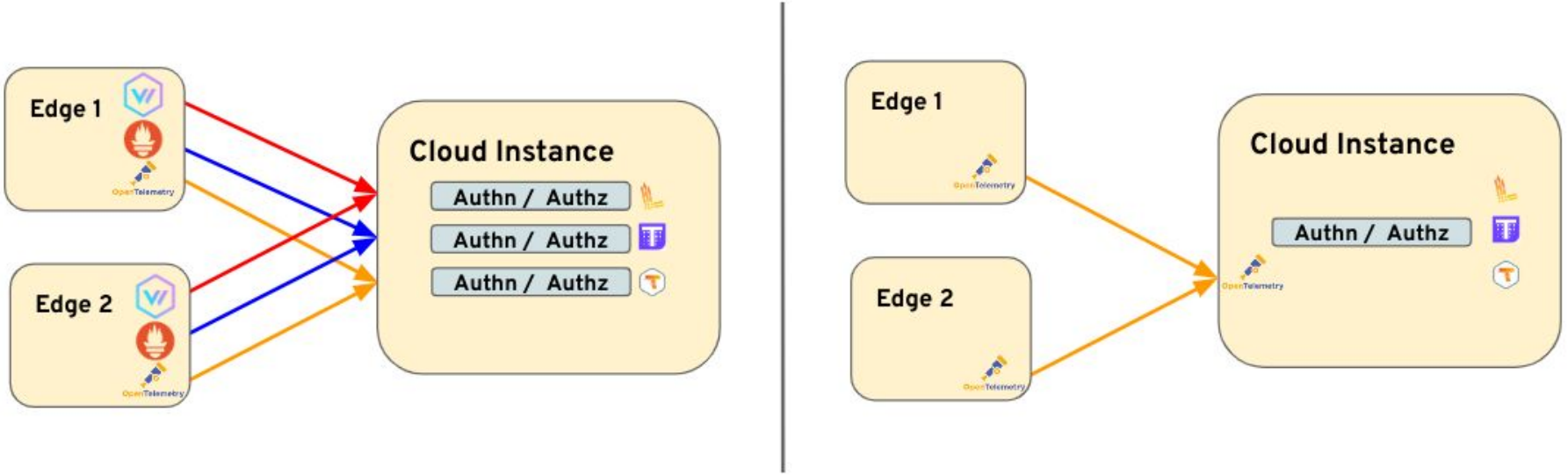

Red Hat

# Teilnehmer Projekt

- TBD

# OpenTelemetry Collector

## Edge




Red Hat