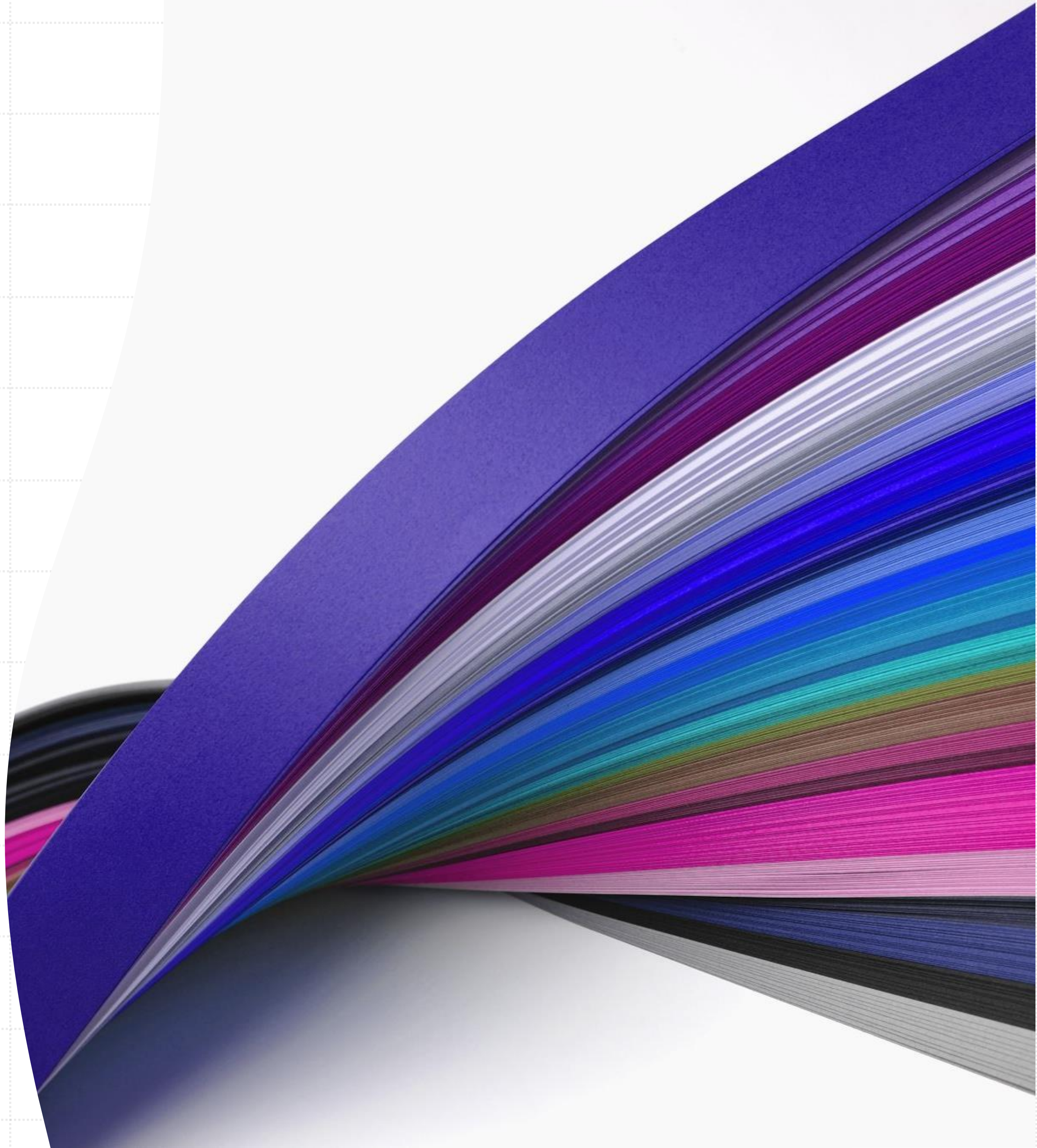
Observability

Logs, Tracing, Metrics, and Alerting with OpenTelemetry



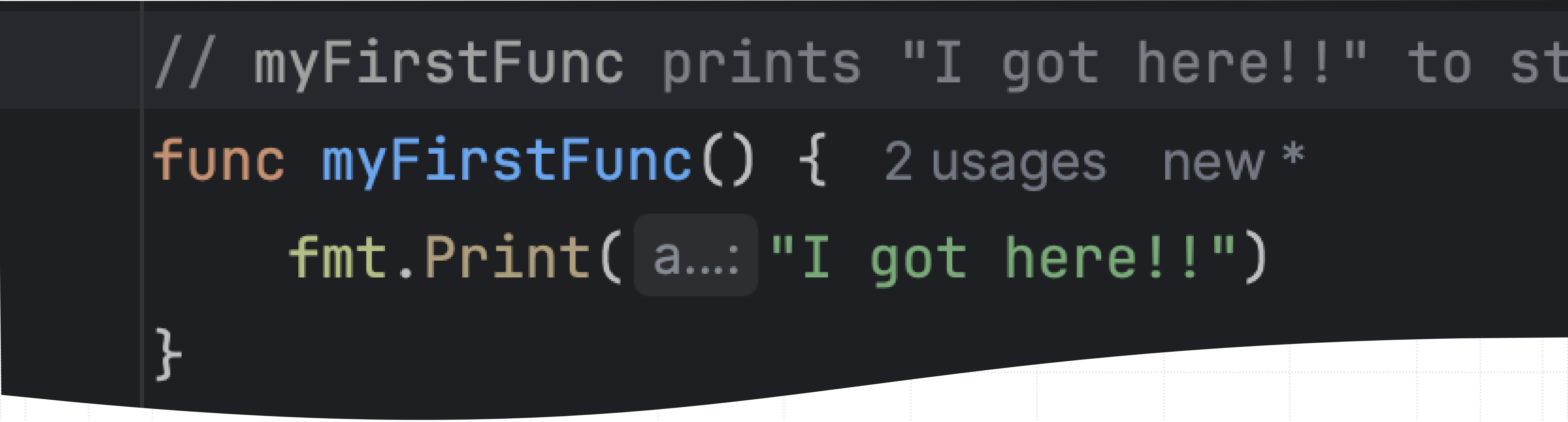
Agenda

Discuss Application Observability	
Introduce OpenTelemetry and its sub-projects	
(Un)Structured Logging and Log processing	▪ Enhance an OTel log processing pipeline
Instrumenting Applications with Distributed Tracing	▪ Hands on with distributed tracing with OTel and Jaeger
Instrumenting Applications with Metrics	▪ Hands on with OTel metrics and Prometheus
Alerting on Abnormalities	▪ Hands on with Prometheus Alert Manager and Pager Duty

Why should I care about observability?

What is OpenTelemetry?

- A specification for telemetry components
- A standard protocol that defines the shape of telemetry data
- APIs that define how to generate telemetry data
- A library ecosystem that implements instrumentation for common libraries and frameworks
- Automatic instrumentation components that generate telemetry data without requiring code changes
- Language SDKs that implement the specification, APIs, and export of telemetry data
- The OpenTelemetry Collector, a proxy that receives, processes, and exports telemetry data



```
// myFirstFunc prints "I got here!!" to st
func myFirstFunc() {  2 usages  new *
    fmt.Print( a...: "I got here!!")
}
```

Logging

You've been logging for as long
as you've been building software.

What data should be in a log entry?

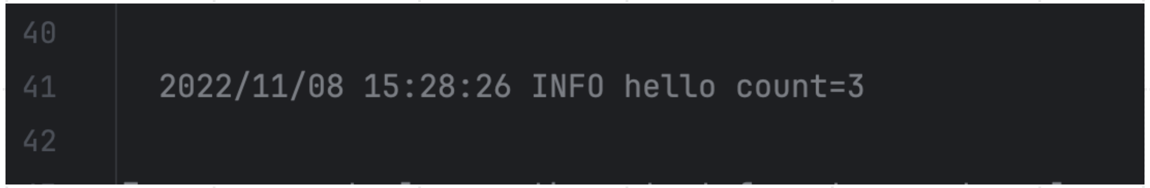
TIMESTAMP

SEVERITY

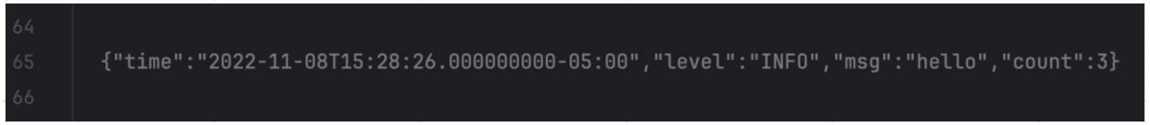
MESSAGE

CONTEXT

How should logs be formatted?



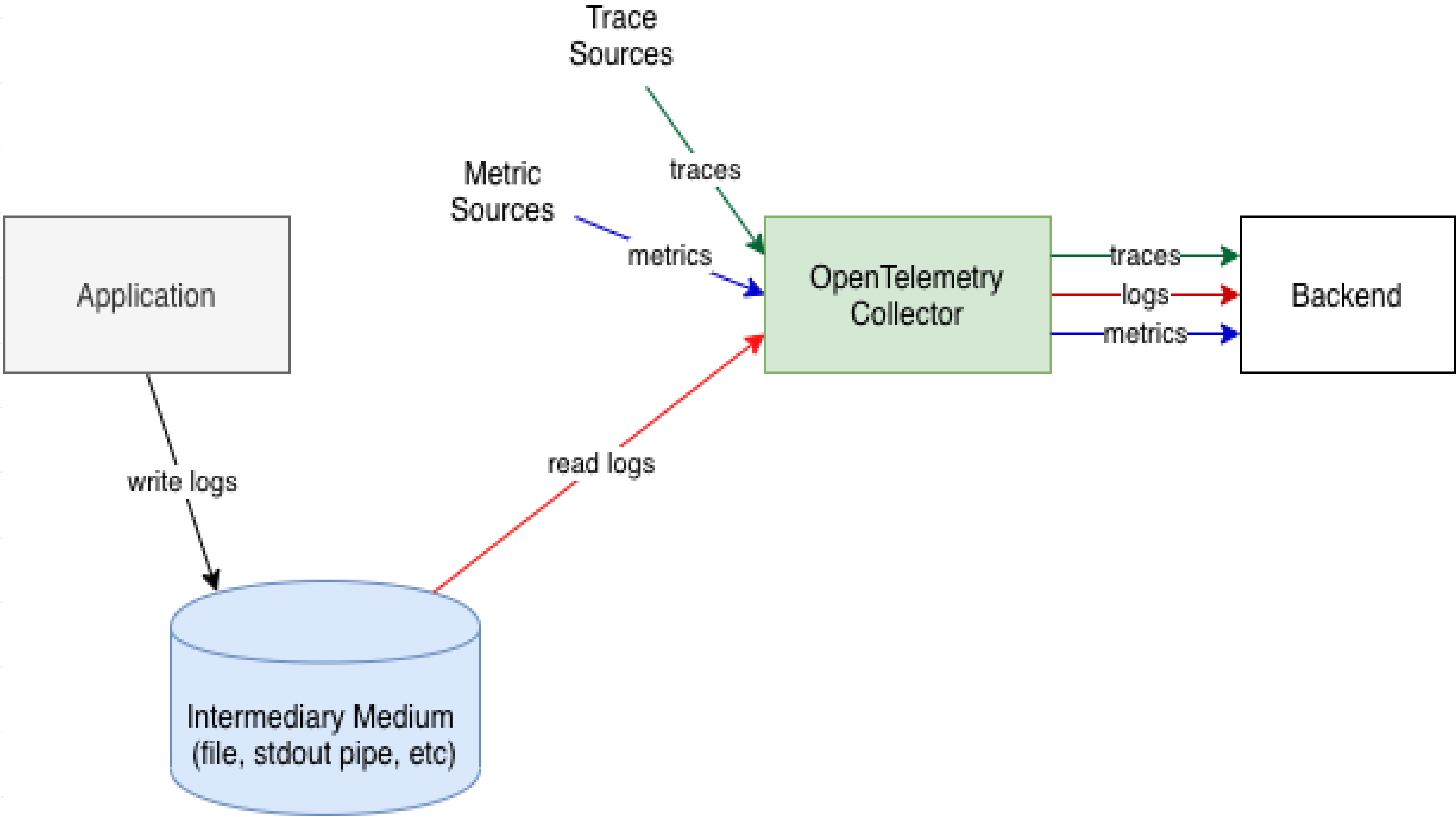
```
2022/11/08 15:28:26 INFO hello count=3
```



```
{"time":"2022-11-08T15:28:26.000000000-05:00","level":"INFO","msg":"hello","count":3}
```

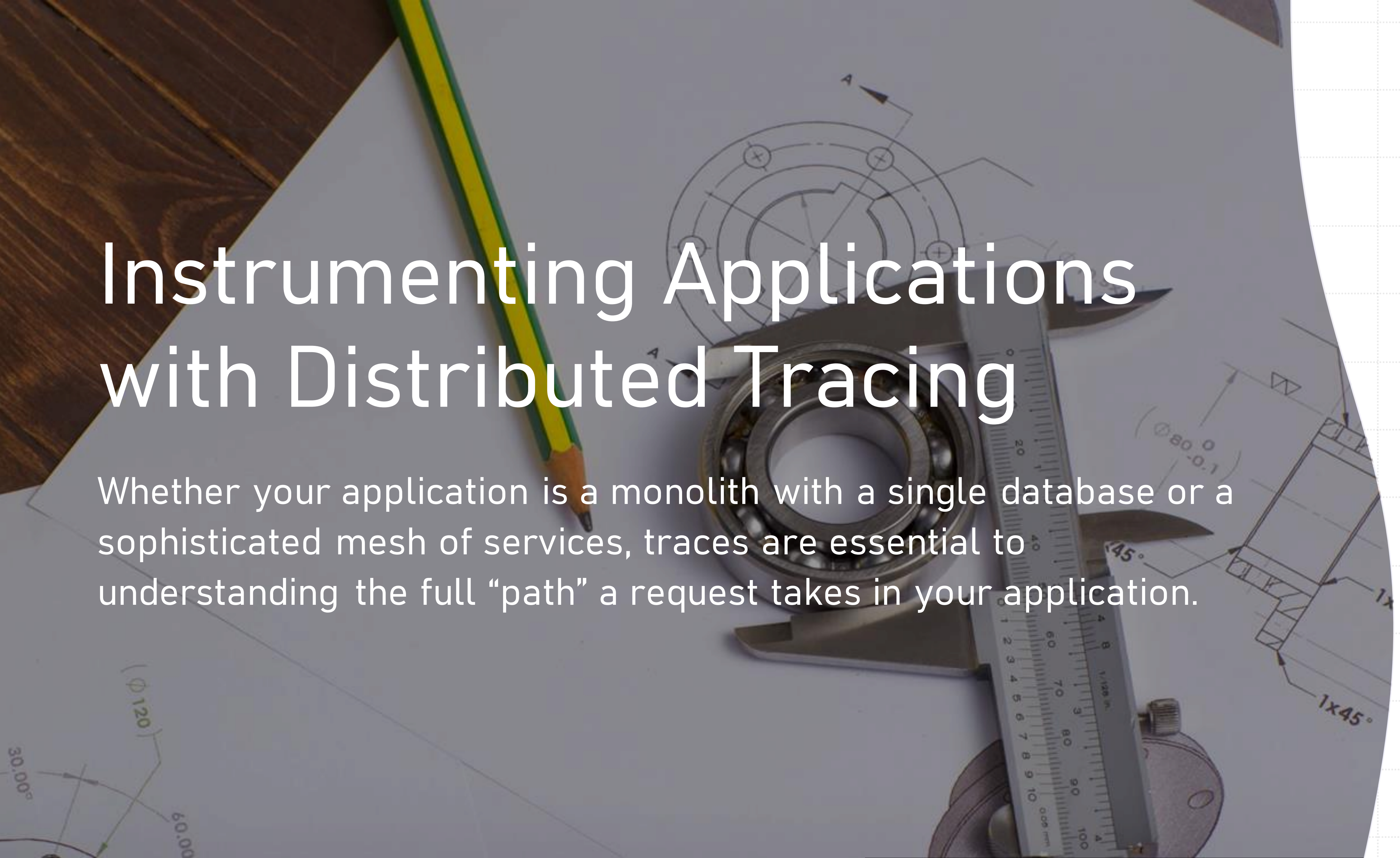
Where should logs be written?

Log Processing with OpenTelemetry


Trace Sources
traces
Metric Sources
metrics
Application
OpenTelemetry Collector
traces
logs
metrics
Backend
write logs
read logs
Intermediary Medium (file, stdout pipe, etc)

Hands on with OTel Log Processing

- Build a pipeline for processing local log data using OpenTelemetry
- Stretch goal
  - Add a log exporter to send logs to a log indexing service rather than to STDOUT.
  - Enhance the log output with a new attribute or body field.



Instrumenting Applications with Distributed Tracing

Whether your application is a monolith with a single database or a sophisticated mesh of services, traces are essential to understanding the full “path” a request takes in your application.

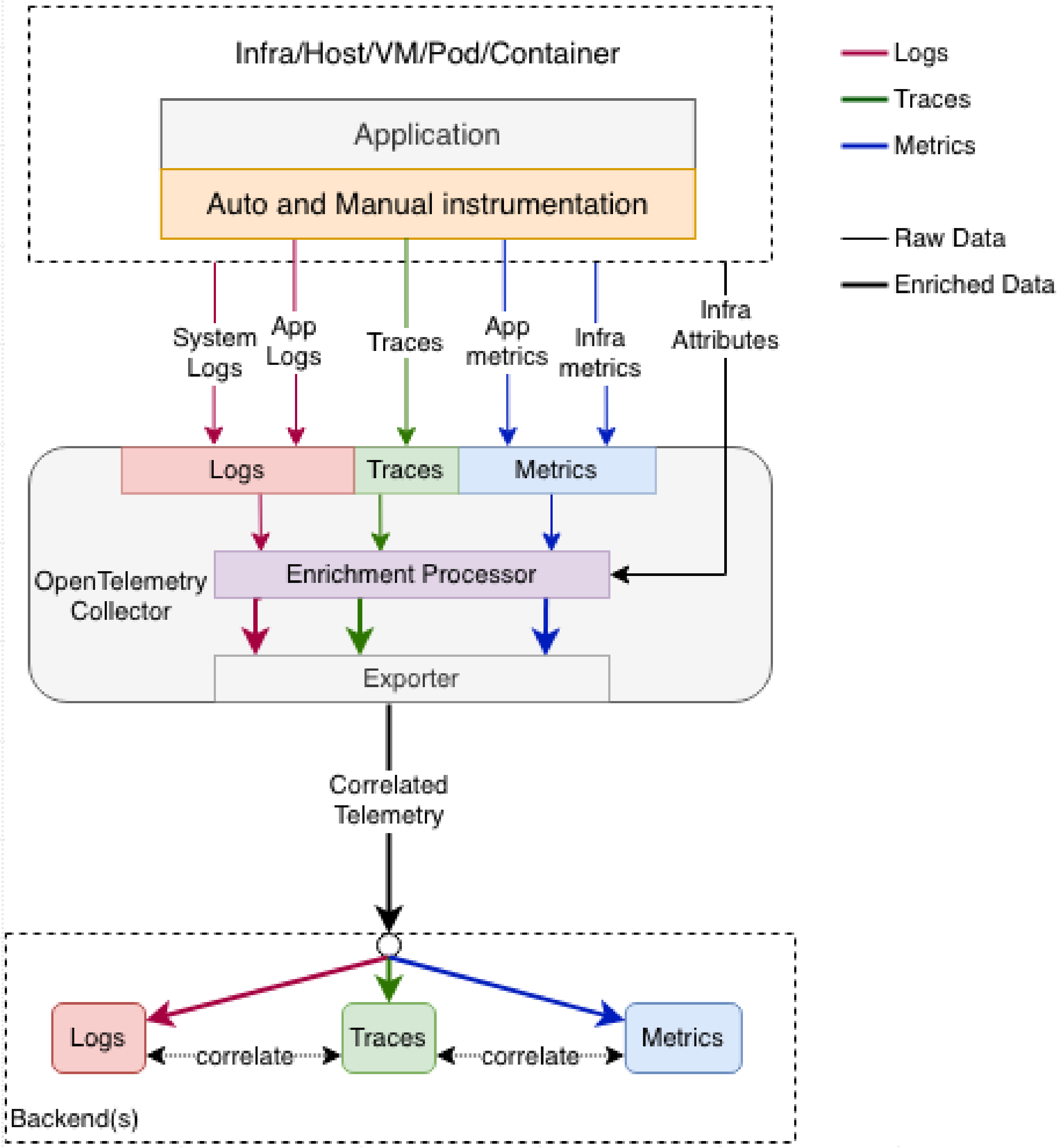
Spans

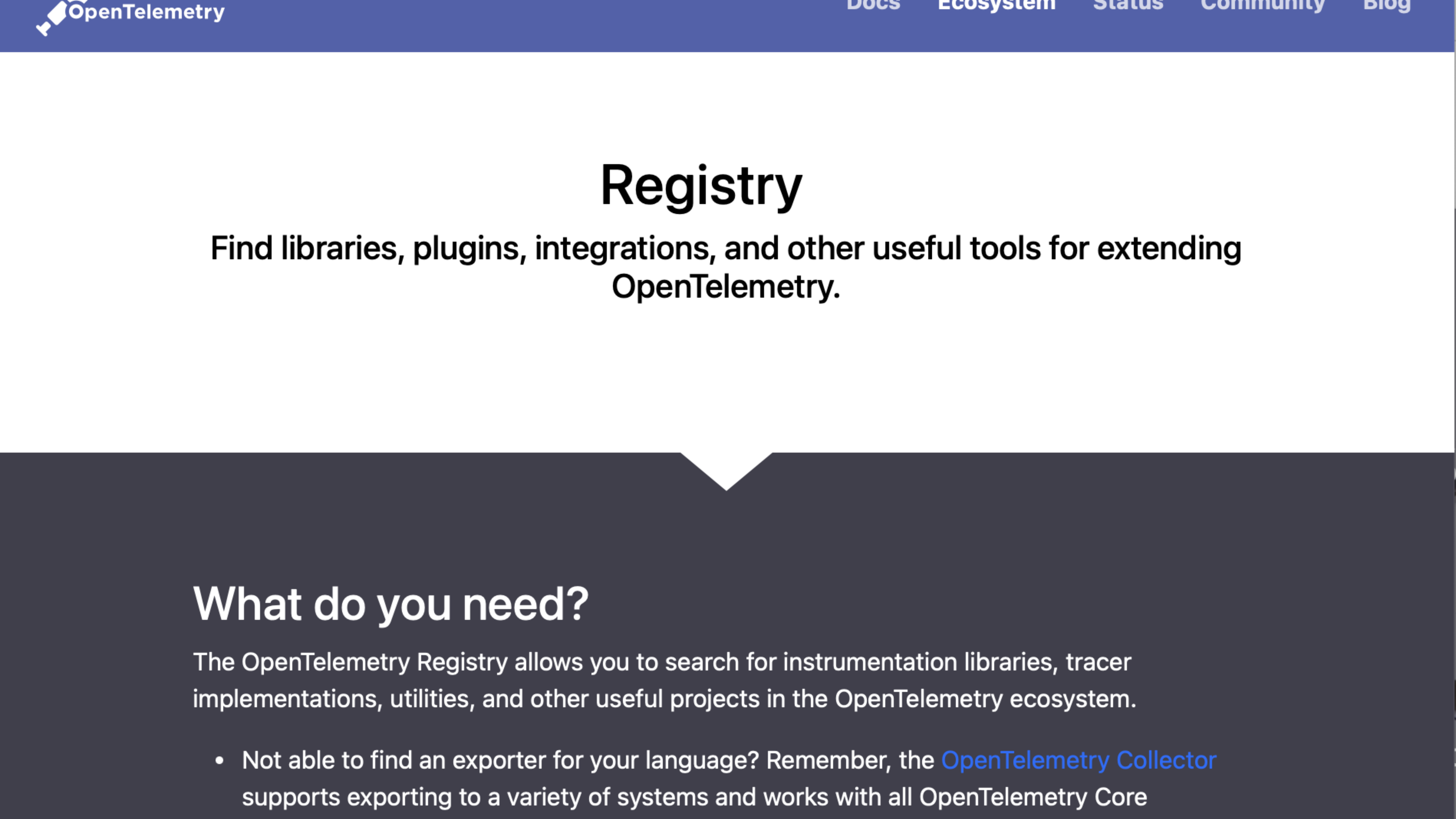
```
{
  "name": "hello",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x051581bf3cb55c13"
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114687Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "Guten Tag!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```

```
{
  "name": "hello-greetings",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x5fb397be34d26b51"
  },
  "parent_id": "0x051581bf3cb55c13",
  "start_time": "2022-04-29T18:52:58.114304Z",
  "end_time": "2022-04-29T22:52:58.114561Z",
  "attributes": {
    "http.route": "some_route2"
  },
  "events": [
    {
      "name": "hey there!",
      "timestamp": "2022-04-29T18:52:58.114561Z",
      "attributes": {
        "event_attributes": 1
      }
    },
    {
      "name": "bye now!",
      "timestamp": "2022-04-29T18:52:58.114585Z",
      "attributes": {
        "event_attributes": 1
      }
    }
  ]
}
```

Correlation is the key to causality

OpenTelemetry Collection





Registry

Find libraries, plugins, integrations, and other useful tools for extending OpenTelemetry.

What do you need?

The OpenTelemetry Registry allows you to search for instrumentation libraries, tracer implementations, utilities, and other useful projects in the OpenTelemetry ecosystem.

- Not able to find an exporter for your language? Remember, the OpenTelemetry Collector supports exporting to a variety of systems and works with all OpenTelemetry Core

Hands on with OTel Tracing

- Build a client and server application which propagates traces over HTTP between the services.
- View the distributed traces in a local Jaeger instance.
- Stretch goal
    - Add an error to a span and find it in Jaeger.
    - Change the tracing to be stochastic rather than always recording.

Instrumenting Applications with Metrics

Metrics are measure of attributes of an application at a point in time. They are most often numeric.

Anatomy of OTel Metrics

Measure

Describes describes the type of the individual values recorded by a library.

Measurement

Describes a single value to be collected for a Measure.

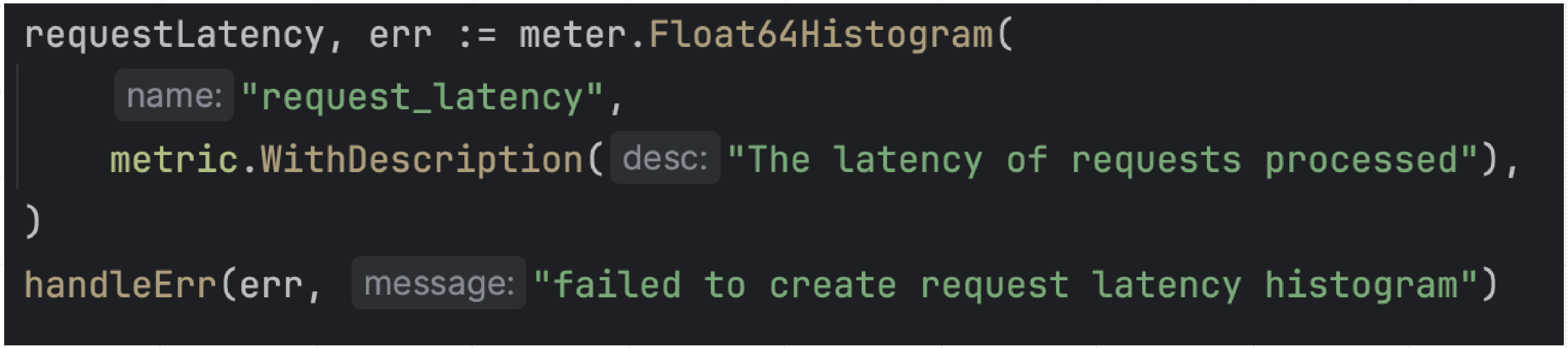
Pre-aggregation

Counter metrics for adding or decrementing a value.

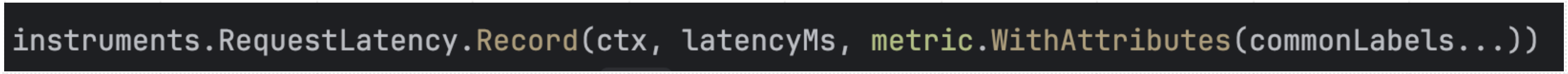
Gauge metrics for recording an instantaneous measurement of a value.

Histogram metrics for bin’ing measured values

Example Metric



```
requestLatency, err := meter.Float64Histogram(
        name: "request_latency",
        metric.WithDescription( desc: "The latency of requests processed"),
)
handleErr(err,  message: "failed to create request latency histogram")
```



```
instruments.RequestLatency.Record(ctx, latencyMs, metric.WithAttributes(commonLabels...))
```

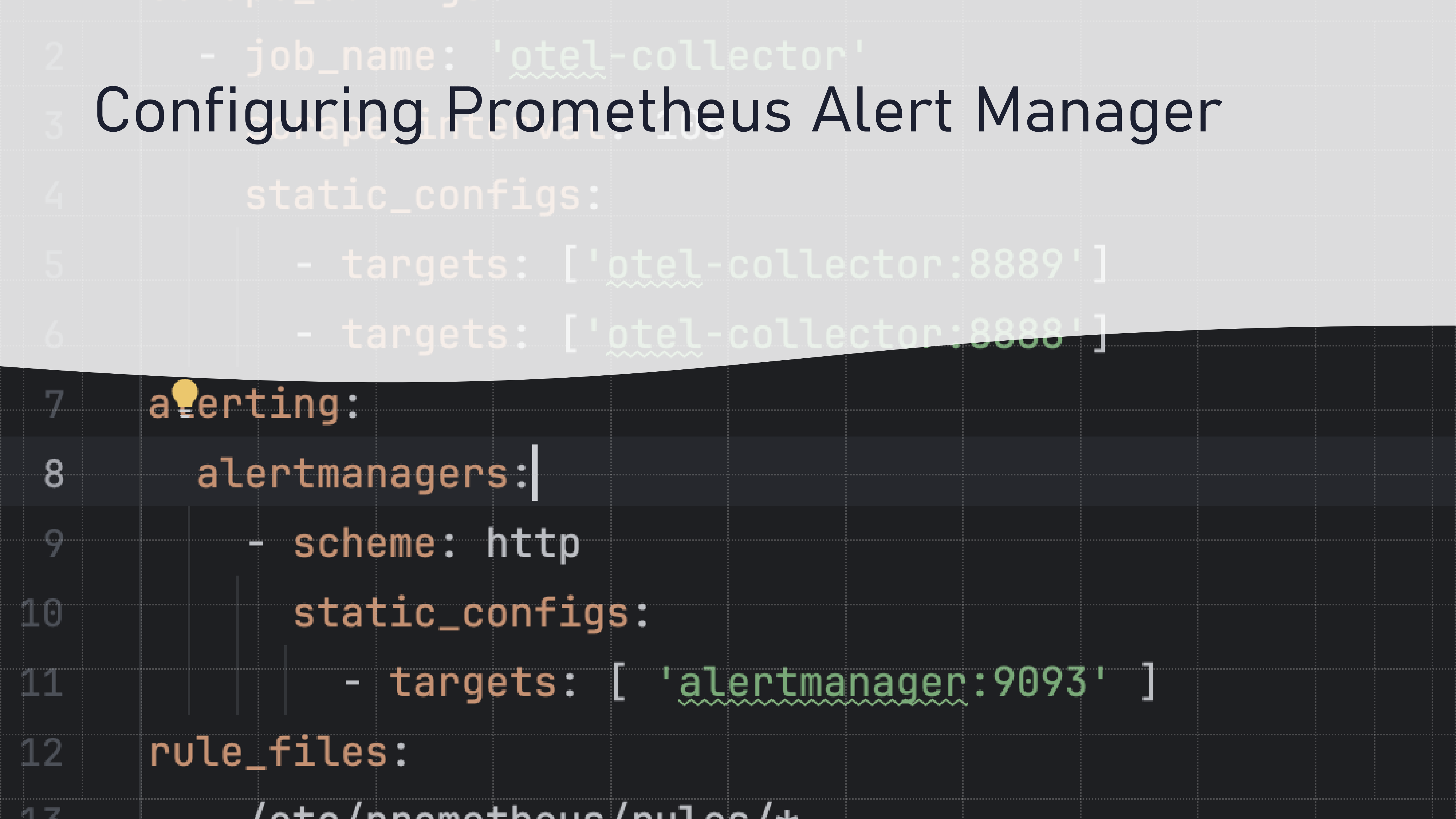
Hands on with OTel Metrics

- Build a client and server application which defines measures, records metrics, sends metrics to the OTel collector.
- Use the OTel collector to provide a scape endpoint for Prometheus.
- Write some PromQL to visualize your metrics.
- Stretch goal
  - Add a new measure and recordings to the application.
  - Create a bucketed histogram or other advanced PromQL query.

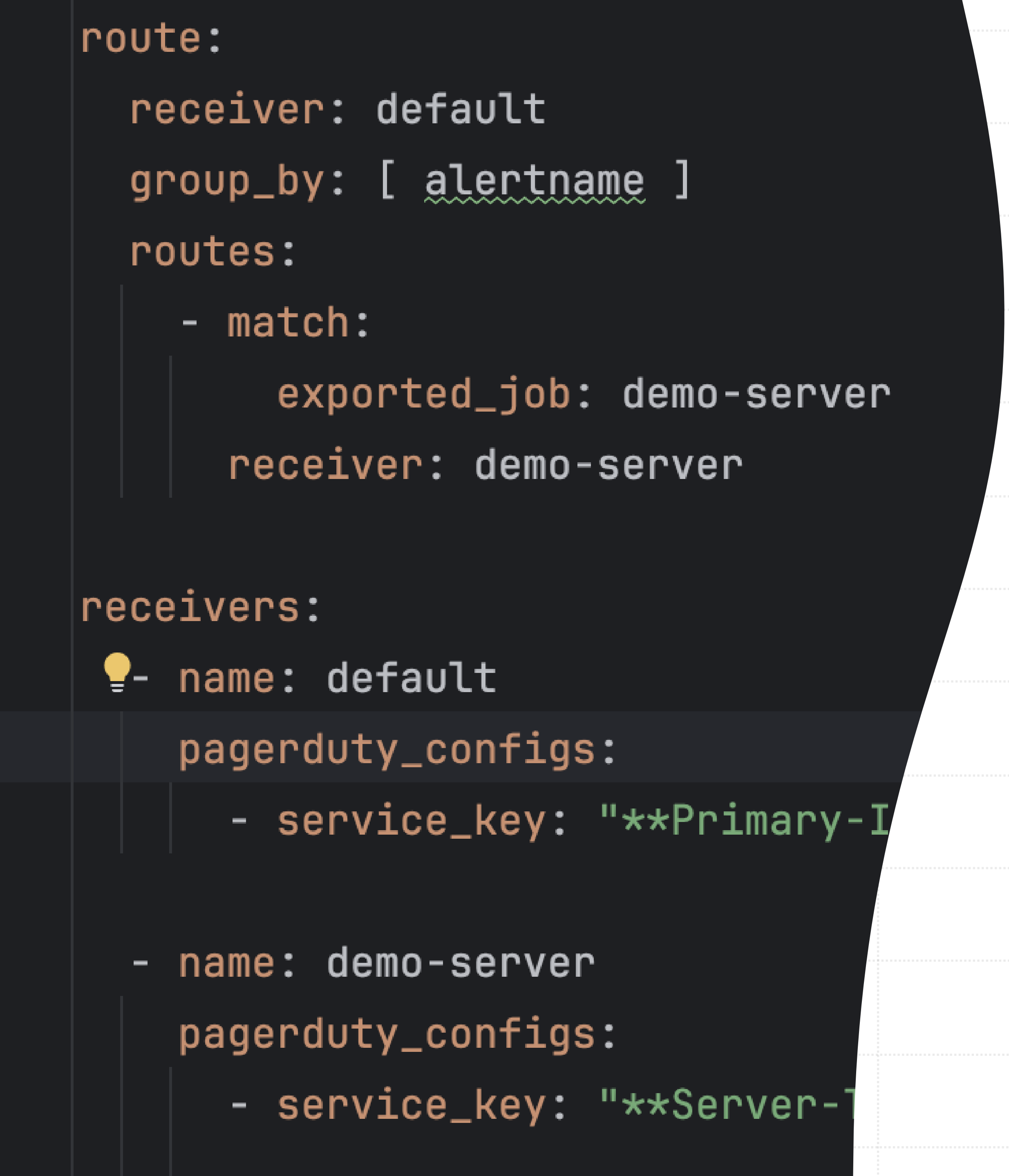
Alerting on Abnormalities

Get notifications your service is misbehaving rather than waiting for your customers tell you.

Configuring Prometheus Alert Manager



```yaml
  - job_name: 'otel-collector'
    scrape_interval: 10s
    static_configs:
      - targets: ['otel-collector:8889']
      - targets: ['otel-collector:8888']
alerting:
  alertmanagers:
    - scheme: http
      static_configs:
        - targets: [ 'alertmanager:9093' ]
rule_files:
    /etc/prometheus/rules/*
```

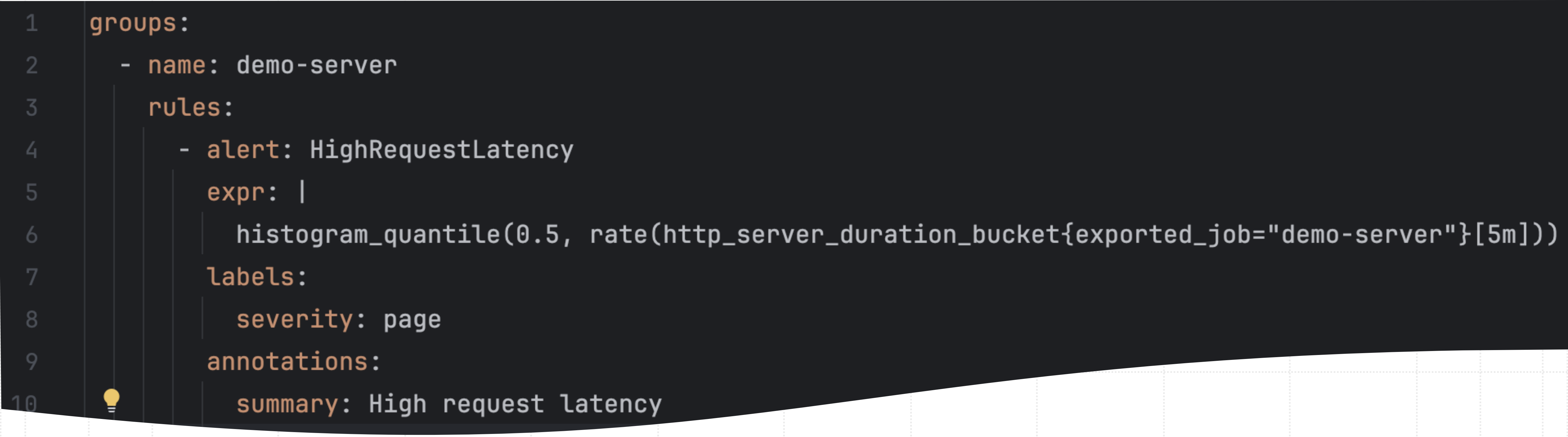


```
route:
  receiver: default
  group_by: [ alertname ]
  routes:
    - match:
        exported_job: demo-server
      receiver: demo-server

receivers:
- name: default
  pagerduty_configs:
    - service_key: "**Primary-I

- name: demo-server
  pagerduty_configs:
    - service_key: "**Server-T
```

Configuring Alert Manager



```yaml
groups:
  - name: demo-server
    rules:
      - alert: HighRequestLatency
        expr: |
          histogram_quantile(0.5, rate(http_server_duration_bucket{exported_job="demo-server"}[5m]))
        labels:
          severity: page
        annotations:
          summary: High request latency
```

Configuring Alerting Rules

Hands on with Prometheus Alerting

- Use the metrics and tracing application.
- Deploy the Prometheus Alert Manager with a rule to alert on mean response times of over 200ms.
- Stretch goal
  - Get a trial for https://www.pagerduty.com and receive a text alert / open incident when Prometheus Alert Manager is triggered.
  - Add another alert rule to trigger based on some other metric that is being collected.