



OpenTelemetry 101



Zameer Fouzan
Senior Developer Relations Engineer



Agenda

01 OpenTelemetry Background

02 Core Concepts of OpenTelemetry

03 Instrumentation - Hands-on

04 Open Telemetry Collector



**WORKED FINE
IN DEV**

OPS PROBLEM NOW

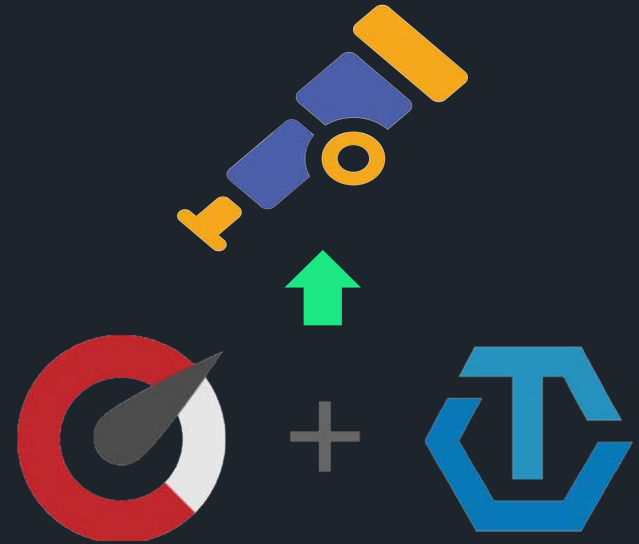
OpenTelemetry

Background

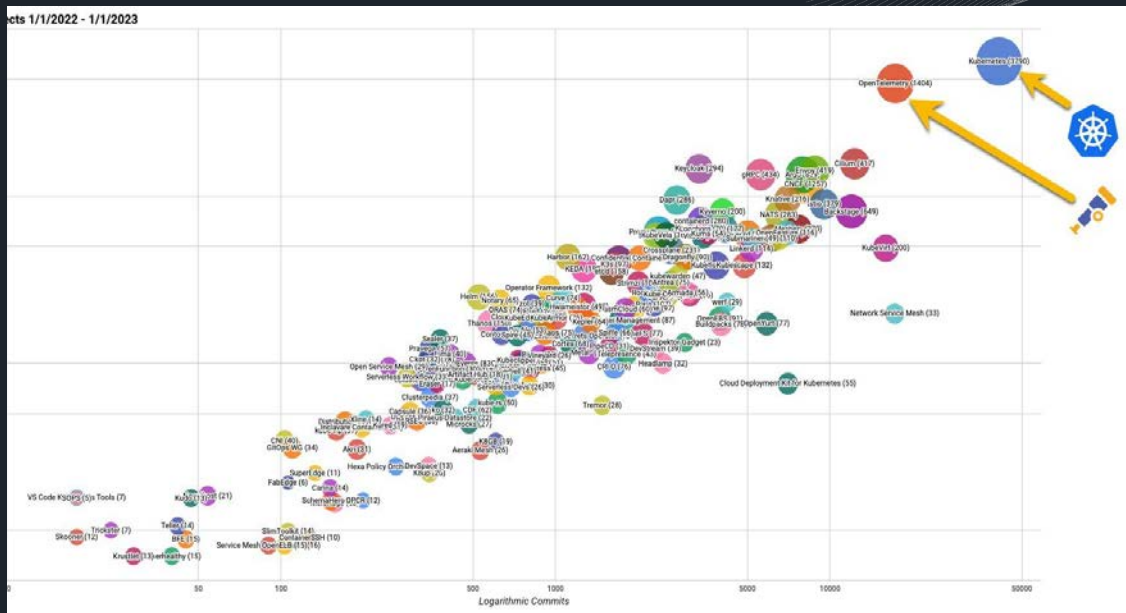
OpenTelemetry

Key Facts on OpenTelemetry

- OpenTelemetry is an **Incubating** project of CNCF.
- Formed through a merger of the **OpenTracing** and **OpenCensus** projects.
- **Vendor agnostic** - set of APIs, libraries, integrations, and a collector service for telemetry.
- **Standardizes** how you collect telemetry data from your applications and services.
- Send it to an **Observability platform** of your choice.



The Rise of OpenTelemetry



OpenTelemetry is second in CNCF - strong interest in modern **Observability**.

Enhanced support for **Open Standards**, including OpenTelemetry, eBPF, and Grafana - Gartner MQ 2022

The Rise of OpenTelemetry



Ubiquity

Promotes better coverage
for instrumentation



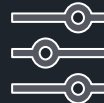
Vendor Neutral

Provides flexibility to
change backend



Interoperable

End-to-end visibility with
standard instrumentation



Configurable

Pick and choose from the
pieces what is needed

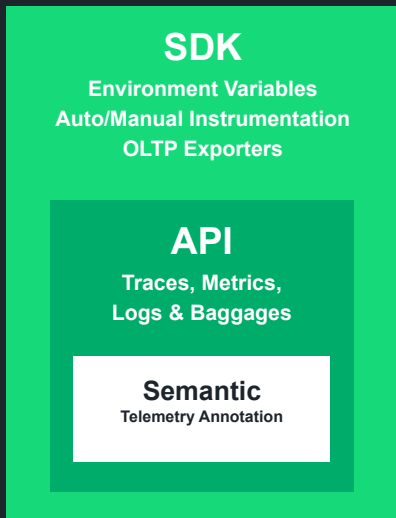
By 2025, **70%** of new cloud-native application monitoring will use open source instrumentation

⁽¹⁾ Source: Gartner Magic Quadrant 2021 for Application Performance Monitoring - [link](#)

Core Concepts

The building blocks

OpenTelemetry - Building blocks



Core Concepts on Instrumentation

- **Semantic Conventions** - annotate telemetry with attributes specific to the represented operation, such as HTTP calls.
- **API** - data types for tracing, metrics, and logging data.
- **SDK** - language-specific implementation of the API.
- SDKs incorporate automatic instrumentation for common libraries and frameworks for your application.
- **OpenTelemetry Protocol (OTLP)** - used to send data to your backend Observability platform of choice.

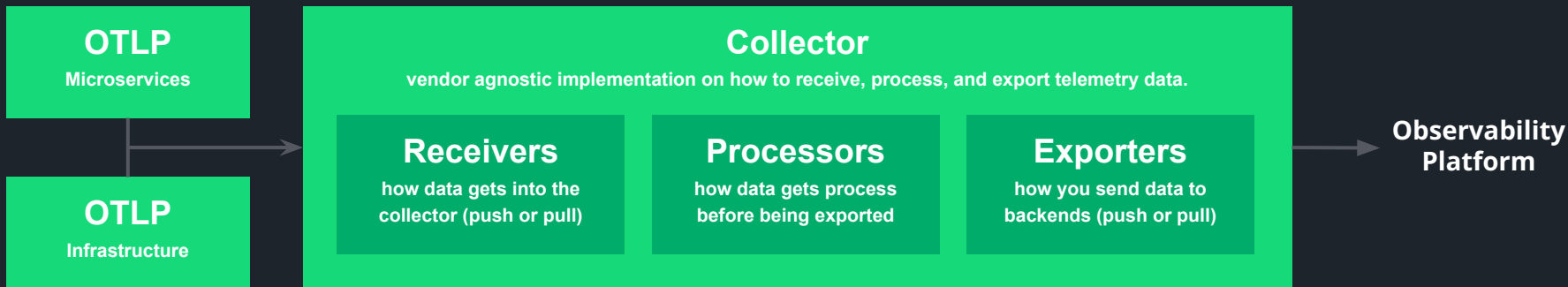
Application Instrumentation

Hands-On+Demo

Infrastructure

The Collector

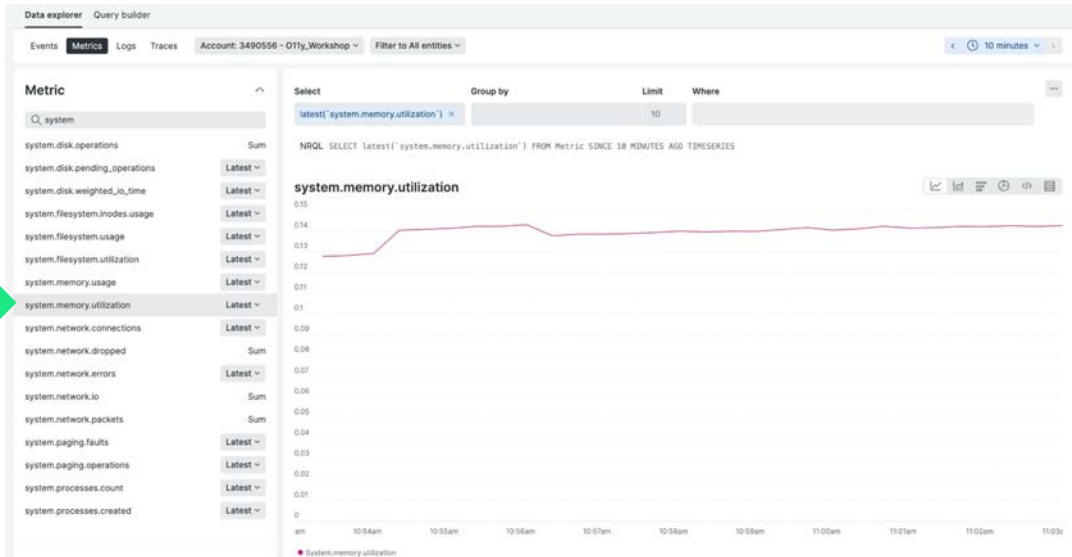
OpenTelemetry - Collector



a **proxy** that process multiple telemetry formats, via an **agent** or a **gateway** (e.g., OTLP, Jaeger, Prometheus, as well as many commercial/proprietary tools)

OpenTelemetry - Collector

```
1
2 receivers:
3   hostmetrics:
4     collection_interval: 20s
5   scrapers:
6     cpu:
7       metrics:
8         system.cpu.utilization:
9           enabled: true
10      load:
11      memory:
12        metrics:
13          system.memory.utilization:
14            enabled: true
15      disk:
16      filesystem:
17        metrics:
18          system.filesystem.utilization:
19            enabled: true
20      network:
21      paging:
22        metrics:
23          system.paging.utilization:
24            enabled: true
25      processes:
26      otlp:
27      protocols:
28      grpc:
29
30 processors:
31   batch:
32   cumulative_delta:
33     metrics:
34       - system.network.io
35       - system.disk.operations
36       - system.network.dropped
37       - system.network.packets
38       - process.cpu.time
```



Collect additional infrastructure metrics for OpenTelemetry

OpenTelemetry - Collector

Storing all tracing data is **costly**. Most of the time, you **ONLY NEED THE RIGHT DATA**, when outages are occurring, or patterns are emerging.

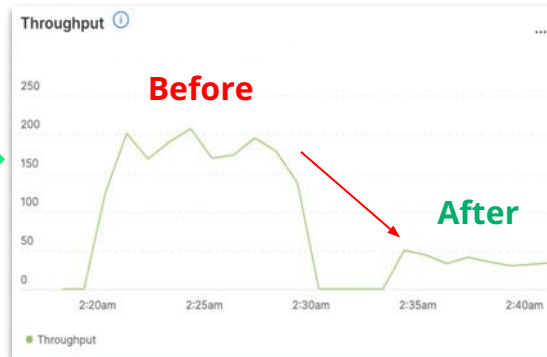


Head based sampling works well for an overall statistical sampling of requests through a distributed system.

Tail based sampling is best to decide what to keep, based on isolated, independent portions of the trace data.

OpenTelemetry - Collector

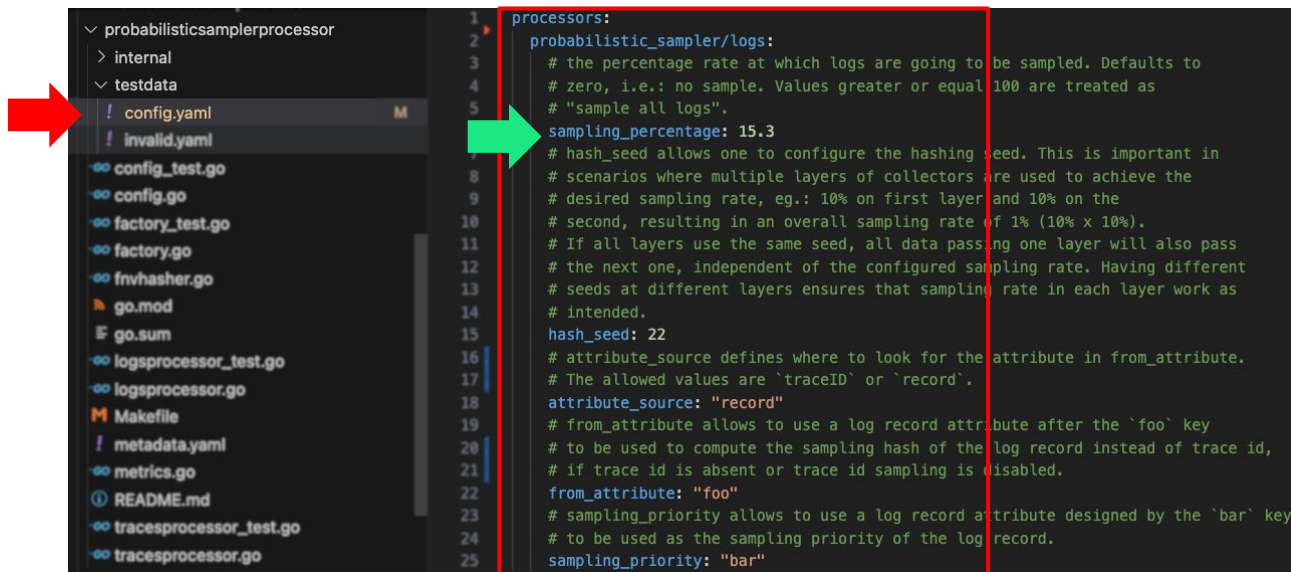
```
30 processors:
31   tail_sampling:
32     decision_wait: 15s
33     num_traces: 100
34     expected_new_traces_per_sec: 10
35     policies:
36     [
37       {
38         name: test-policy-1,
39         type: always_sample
40       },
41       {
42         name: test-policy-2,
43         type: latency,
44         latency: {threshold_ms: 1000}
45       },
46     ]
```



Effects from Tail Based Sampling

Apply processing decision in the Collector, before shipping data out

OpenTelemetry - Collector



The image shows a file explorer on the left and a code editor on the right. A red arrow points to the 'testdata' directory in the file explorer, and a green arrow points to the 'processors' section in the code editor. The code editor displays the configuration for the 'probabilistic_sampler/logs' processor, which is highlighted with a red border. The configuration includes a 'sampling_percentage' of 15.3 and a 'hash_seed' of 22. The code also includes comments explaining the purpose of these settings and the 'attribute_source' field.

```
1 processors:
2   probabilistic_sampler/logs:
3     # the percentage rate at which logs are going to be sampled. Defaults to
4     # zero, i.e.: no sample. Values greater or equal 100 are treated as
5     # "sample all logs".
6     sampling_percentage: 15.3
7     # hash_seed allows one to configure the hashing seed. This is important in
8     # scenarios where multiple layers of collectors are used to achieve the
9     # desired sampling rate, eg.: 10% on first layer and 10% on the
10    # second, resulting in an overall sampling rate of 1% (10% x 10%).
11    # If all layers use the same seed, all data passing one layer will also pass
12    # the next one, independent of the configured sampling rate. Having different
13    # seeds at different layers ensures that sampling rate in each layer work as
14    # intended.
15    hash_seed: 22
16    # attribute_source defines where to look for the attribute in from_attribute.
17    # The allowed values are `traceID` or `record`.
18    attribute_source: "record"
19    # from_attribute allows to use a log record attribute after the `foo` key
20    # to be used to compute the sampling hash of the log record instead of trace id,
21    # if trace id is absent or trace id sampling is disabled.
22    from_attribute: "foo"
23    # sampling_priority allows to use a log record attribute designed by the `bar` key
24    # to be used as the sampling priority of the log record.
25    sampling_priority: "bar"
```

Probabilistic Based Sampling

Severless

AWS Lambda

Lambda Instrumentation

```

  "opentelemetry/instrumentation-aws-lambda": {
    enabled: true,
    disableAwsContextPropagation: true,
    requestHook: (span, { event, context }) => {
      span.setAttribute("faas.name", context.functionName);

      if (event.requestContext && event.requestContext.http) {
        span.setAttribute(
          "faas.http.method",
          event.requestContext.http.method
        );
        span.setAttribute(
          "faas.http.target",
          event.requestContext.http.path
        );
      }
    },
    responseHook: (span, { err, res }) => {
      if (err instanceof Error)
        span.setAttribute("faas.error", err.message);
      if (res) {
        span.setAttribute("faas.http.status_code", res.statusCode);
      }
    },
  },
},

```

- Official Lambda library
- Request Hooks
- Response Hooks
- Custom Attributes

```
NODE_OPTIONS: "--require ./otel-wrapper"
```

Lambda Instrumentation

The screenshot displays the New Relic APM interface for a service named 'otel-sls-sdk-dev-api'. The main view shows a trace with a total duration of 59.74 ms. A specific span for 'GET /api/weather' is highlighted, with a duration of 20.15 ms. This span is broken down into several sub-spans, including middleware and a request handler. A detailed view of the 'GET /api/weather' span is shown on the right, listing various attributes such as http.host, http.method, http.route, http.scheme, http.status_code, http.status_text, http.target, http.url, http.user_agent, id, and instrumentation.provider. Three green arrows point to the 'GET /api/weather' span in the main view, the 'http.route' attribute in the details view, and the 'http.target' attribute in the details view.

| Span Name | Duration |
|-------------------------------|----------|
| otel-sls-sdk-dev-api | 59.74 ms |
| GET /api/weather | 20.15 ms |
| middleware - urlencodedParser | 0.02 ms |
| middleware - <anonymous> | 0.03 ms |
| router - /api/weather | 0.01 ms |
| request handler - / | 0.00 ms |
| middleware - query | 0.00 ms |
| middleware - expressInit | 0.05 ms |
| middleware - jsonParser | 0.03 ms |
| GET | 18.28 ms |
| Uninstrumented time | 8.58 ms |

| Attribute | Value |
|--------------------------|---|
| http.host | 3.230.230.121 |
| http.method | GET |
| http.route | /api/weather |
| http.scheme | http |
| http.status_code | 200 |
| http.status_text | OK |
| http.target | /api/weather?location=bangalore |
| http.url | http://3.230.230.121/api/weather?location=bangalore |
| http.user_agent | axios/1.6.1 |
| id | 304ba16ee5677b62 |
| instrumentation.provider | opentelemetry |

Recap and Highlights

Exciting times for Open Source Observability!

OpenTelemetry is growing and being adopted at a rapid pace.

Be mindful with your maturity, and plan ahead on the adoption of OpenTelemetry.

Just having telemetry is **NOT** observability. Instrumentation should include contextual **traces, logs, or metrics** to improve observations.

Deployment of the OpenTelemetry collector is easier to gather Telemetry.

Gather telemetry from your pipelines to measure your MTTI, MTTD, MTTR

Actively investing in OpenTelemetry, and help engineers work based on **"data, not opinions"**.

Resources

Further Reading

Open Telemetry Resources

- [Application Instrumentation](#)
- [Open Telemetry on Serverless](#)
- [Open Telemetry in CI/CD](#)
- [Collector](#)
 - [Sampling Strategies](#)
- Collector on Kubernetes
 - Configuration - <https://opentelemetry.io/docs/collector/configuration/>
 - Helm - <https://opentelemetry.io/docs/kubernetes/helm/collector/>
 - Operator - <https://opentelemetry.io/docs/kubernetes/operator/>
 - Automatic instrumentation - <https://opentelemetry.io/docs/kubernetes/operator/automatic/>

\$whoami



Zameer Fouzan

Senior Developer Relations Engineer

