Engineer

OpenTelemetry

one vendor-free way to send telemetry

What it is

OpenTelemetry is an open-source standard, with libraries and tools to match, for producing traces, metrics and logs from running software and handing them to whatever system stores and displays them. Most engineers call it OTel. It is a project of the Cloud Native Computing Foundation, formed when two earlier efforts, OpenTracing and OpenCensus, merged into one.

The most useful thing to know about it is what it leaves out. OpenTelemetry is not an observability backend: it keeps no data, draws no dashboards and sends no alerts. Jaeger, Prometheus and the commercial platforms do that part. So the question “OpenTelemetry or Prometheus?” compares a pipe with a destination, and the usual answer is both: OpenTelemetry where the data is produced, a backend of the team’s choosing where it is kept.

The project is several pieces that fit together:

- a **specification** that defines the API, the SDK and the data, across every language;
- **OTLP**, the protocol that carries telemetry over gRPC or HTTP, encoded with Protocol Buffers;
- **semantic conventions**, shared names for shared things, so an HTTP route or a database statement is labelled the same way in Go, Java and Python;
- an **API and SDK per language**, with instrumentation libraries and automatic instrumentation for popular frameworks;
- the **Collector**, a separate process that receives telemetry, processes it and exports it onward.

Of the signals it carries, traces take the most explaining. A trace follows one request through every service it touches. Each step is a span: a name, a start time, a duration, a set of attributes, and the identity of the span that caused it. When the request crosses from one service to the next, that identity travels with it in a header, by default the one W3C Trace Context defines:

```
traceparent: 00-a0892f3577b34da6a3ce929d0e0e4736-f03067aa0ba902b7-01
```

The four fields are the format version, the trace ID shared by every span of the request, the ID of the calling span, and a set of flags, where a final `01` marks the trace as sampled. Metrics and logs complete the picture, and a fourth signal, baggage, carries small key-value facts along with the request so that a service three hops away can still see them.

Use the concept in a decision

Put “What is OpenTelemetry?” to work in a real decision. We can help relate the concept to the system you are building or buying. Monitoring and alerts that speak up while a fault is still small, and the figures that show how often it stays that way.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Built the company's first observability dashboard — the company's first view of its own systems

Built the company's first observability dashboard Get in touch →

Why it matters

Storage is the visible cost of observability. The stickier one is instrumentation: someone has to add the calls to every service, in every language, and keep them current as the code changes. When each vendor shipped its own agent and its own API, changing vendor meant doing all of that again. With OpenTelemetry the code calls one API, and the choice of backend moves into configuration, either an exporter in the SDK or a route in the Collector. The project states the principle outright: the data a system generates belongs to whoever runs it.

That ownership is practical as well as contractual. The Collector sits between the applications and the backend, which makes it the place to filter, sample, rename or strip attributes before anything leaves, and to send the same stream to two backends while a migration is under way. Moving from one vendor to another becomes a change to a Collector configuration rather than to a hundred services.

Volume is the cost to plan for. Each request can produce dozens of spans, and at real traffic the bill for keeping all of them can outgrow the service being watched. The SDK's default sampler, `ParentBased(root=AlwaysOn)`, samples every new trace and follows the caller's decision for the rest, so out of the box everything is kept. How much to keep is a decision a team makes on purpose.

Maturity differs by signal and by language. OTLP is stable for traces, metrics and logs, while each language's SDK moves at its own pace, and the project's status page says where each one stands. Read it before committing a service written in a less common language.

And it is not always worth it. A single application with structured logs and a handful of metrics gains little from distributed tracing. OpenTelemetry pays for itself when a request crosses process boundaries, through a frontend, an API, a queue and a database, and the question "where did the time go?" takes a meeting to answer.

What it looks like in practice

On one platform the monitoring was built rather than bought, and OpenTelemetry is its tracing half. Tracing runs over OTLP, so a slow request is followed across the frontend, the API and the database rather than guessed at. Beside it runs an in-house error pipeline, an errmon service and an ingest service, that sanitises payloads before they are stored, groups errors into recurring problems, detects spikes and regressions with a cooldown so one bad deploy does not page anyone forty times, symbolicates minified frontend stack traces, and runs a synthetic heartbeat to prove that the pipeline itself is alive.

The results land in the platform's own database as error events, error groups, an inbox, API latency and stack samples. Operators work from 11 views over that data, one of them for service level objectives. The page is plain about the price: building cost real time and added one more thing to maintain, and a bought tracker would have been running the same afternoon. In return, sensitive data stays on the platform. The alerting rules, too, were written for this platform rather than a generic one.

Where it goes wrong

The trace stops at a boundary. Context crosses HTTP calls easily, because the instrumentation libraries write and read the header on both sides. A message queue, a scheduled job or a proxy that strips unknown headers breaks the chain unless someone carries the context across by hand. The trace splits in two, and the slow half looks unrelated to the request that caused it. Put the context in the message metadata, and check that every service runs the same propagator.

Attributes leak or explode. A span attribute is easy to add, and the easiest values to add are the wrong ones: email addresses, full URLs with tokens in the query string, SQL with the literal values still in it. On metrics the danger is cardinality. An attribute with millions of distinct values, a user ID for instance, multiplies the number of time series and the bill along with it. Follow the semantic conventions, keep identifiers off metrics, and redact in the Collector as a second line of defence.

Sampling happens by accident. Head sampling decides at the start of a request, before anyone knows whether it will be slow or fail, so keeping 1% of traces also keeps about 1% of the rare slow ones. Tail sampling decides after seeing a trace's spans together and can keep every error and every slow request, but the component doing it must hold all that data while it waits. It is a stateful service with capacity of its own to plan and to monitor.

The pipeline is trusted blindly. OTLP acknowledges delivery one hop at a time; end-to-end delivery is outside its scope. A Collector that runs short of memory or falls behind on its export queue drops telemetry, and the applications sending to it carry on as if all were well. Watch the Collector's own metrics, run more than one, and treat a gap in the traces as a fault to explain rather than a quiet hour.

Further reading

- What is OpenTelemetry? — the project's own definition, its components, and what it deliberately is not.
- OTLP Specification — the wire protocol: transports, encoding, compression and acknowledgement.
- Trace Context — the W3C Recommendation behind the `traceparent` header.
- Context propagation — how a trace follows a request across process boundaries.

- Semantic conventions — the shared names for attributes, metrics and resources.
- Sampling — head and tail sampling, and what each one costs to run.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is OpenTelemetry?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/opentelemetry/