Engineer

Observability

understanding a system from the outside, from what it emits

What it is

Observability is the degree to which a team can understand what a running system is doing from the outside, using only the data the system sends out, without opening it up or knowing in advance what went wrong. The OpenTelemetry project's primer puts it in terms of questions: an observable system lets you ask questions about it without knowing its inner workings, including questions nobody thought of when the software was written. The primer's shorthand for the question it should be able to answer is "Why is this happening?"

That makes observability a property of the system, not a product. A tool can store and display data, but the data has to come from the application first. OpenTelemetry's documentation is direct about it: for a system to be observable, its components must be instrumented, meaning their code emits signals. It then sets a practical bar. An application counts as properly instrumented when its existing signals let developers investigate an issue without first changing the instrumentation.

The data is called telemetry, and OpenTelemetry sorts it into signals:

- **Traces** record the path of one request through the system. A trace is made of spans, each a single unit of work with a name, a start and end time and attributes such as the HTTP method and route. The first span is the root and covers the request from start to finish.
- **Metrics** are measurements captured at runtime and aggregated over time: error rate, request rate, processor use.
- **Logs** are timestamped records of events. Many of them are not tied to any particular request.
- **Baggage** is contextual information passed along with a request so that other signals can use it.

The signals page also lists events, a specific type of log, and profiles, recordings of resource usage at the level of the code, as under development or at the proposal stage.

What turns three piles of data into something a team can question is correlation. OpenTelemetry propagates context between services, so a span in one service knows which trace it belongs to, and it can stamp the IDs of the active trace and span onto each log record. Every signal also carries a resource: attributes that identify the entity producing it, such as the service name, the host or the container. With those shared keys, a team can move from a metric to the requests behind it, and from a request to the log lines it produced.

Observability and monitoring overlap but are not the same thing. Monitoring watches for conditions someone decided in advance to care about and raises an alert when one appears. Observability is what the team has to work with once the alert has fired and the cause is something nobody planned for.

Use the concept in a decision

Put “What is observability?” 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

Modern systems fail in ways that are hard to predict. A request may pass through a gateway, several services, a queue and a database before it answers, and a fault in any one of them can show up as slowness somewhere else. The primer calls distributed tracing essential for such systems, because their problems are often nondeterministic or too complicated to reproduce on a developer’s machine. When a failure cannot be reproduced, the telemetry recorded while it happened is the only evidence there is.

Dashboards and alerts answer questions that were anticipated. That is valuable, and it covers the failures a team has seen before. The hard incidents are the new ones, where the question is not whether error rates rose but why requests from

one region, through one version, to one payment provider began to fail. If answering that means writing new code, deploying it and waiting for the problem to come back, the outage lasts as long as that round trip.

Observability also keeps the user in view. The primer defines reliability as whether the service is doing what users expect, and gives an example: a shop could be up all the time and still be unreliable if clicking to add black shoes to the cart does not always add black shoes. A host metric would never show that. Request-level telemetry can, and it is what service level indicators are built from: measurements taken from the user's point of view, which service level objectives then tie to what the business needs.

Finally, it changes the cost of an incident. A team that can follow a slow request to the exact call that held it up spends its time fixing, not guessing, and the postmortem afterwards can quote evidence rather than recollection.

What it looks like in practice

Take an invented online shop. Since 14:00, a small share of checkouts take more than two seconds. Nothing is down, no alert has fired, and no dashboard was built for this. Here is how a team with correlated signals would work through it. Every name, ID and number below is made up for the example.

The first step is the latency histogram for checkout, measured in seconds. In the OpenMetrics text format, a bucket can carry an exemplar after a `#` : a reference to data outside the metric, most commonly the ID of a trace that landed in that bucket. Two of the buckets read:

```
checkout_bucket{le="1.0"} 95
checkout_bucket{le="2.5"} 97 # {trace_id="3f1c9a77e2b04d18a6c5e0b9d4127f60"} 2.3
```

The second step follows that trace ID. The trace shows where the 2.3 seconds went, span by span:

```
POST /checkout                      2.30 s   service checkout
  SELECT cart                       0.04 s   service checkout
  POST /authorise                   2.21 s   service payments
    event: provider timeout, retrying
```

```
  INSERT order                    0.03 s   service checkout
```

The third step pulls the log lines that carry the same trace ID. OpenTelemetry's specification for logs written in formats other than its own protocol asks that the IDs be recorded as top-level fields named `trace_id` and `span_id` in JSON, lowercase and hex-encoded:

```
{
    "timestamp": 1793110051.402,
    "severity": "WARN",
    "body": "provider timeout after 2000 ms, retrying",
    "provider": "acquirer-b",
    "trace_id": "3f1c9a77e2b04d18a6c5e0b9d4127f60",
    "span_id": "a91c04e7d2f3b658"
}
```

Three hops, and the question nobody planned for has an answer: one of two payment providers began timing out, and every checkout routed to it waited for a retry. The same attribute, `provider`, can now be grouped on across all recent traces to confirm the pattern before anyone touches the code.

None of that works if the services do not identify themselves. OpenTelemetry SDKs set the `service.name` resource attribute to `unknown_service` by default, and the documentation recommends setting it explicitly, in code or through an environment variable. Other resource attributes can be added the same way:

```
env OTEL_SERVICE_NAME=checkout \
    OTEL_RESOURCE_ATTRIBUTES=deployment.environment.name=production \
    ./checkout
```

On this site, three portfolio entries show observability being built up in steps. One describes a company's first observability dashboard, which collected system metrics such as CPU, memory and disk from remote Linux servers over SSH; a second version was later built on Grafana and Prometheus. Work to cut operational risk with Grafana and Prometheus instrumented the key services and chose metrics that meant something over vanity numbers. A later entry, first-party error monitoring and tracing, added OpenTelemetry tracing over OTLP, so

that a slow request can be followed across the frontend, the API and the database rather than guessed at, alongside an error pipeline that sanitises payloads before they are stored.

Where it goes wrong

Buying it instead of building it in. A new tool with nothing useful flowing into it is an empty room. Observability comes from instrumentation in the code and the libraries, so budget the engineering time to add spans, attributes and log fields, not only the licence.

Three signals, three islands. Metrics in one tool, logs in another and traces in a third, with no shared key between them, leave the team copying timestamps from one screen to another. Put the trace and span IDs into every log record, attach exemplars to latency histograms where the stack supports them, and give every signal the same resource attributes.

Every service called unknown_service. It is the SDK default, and a fleet of services sharing it cannot be told apart. Set `OTEL_SERVICE_NAME` for every service. The resource is attached when the tracer and metric providers are created and cannot be changed afterwards, so it has to be right at start-up.

Only the edges are instrumented. Zero-code instrumentation is a good start and records what happens at the edges of an application: in its libraries and the environment it runs in. The questions in an incident are usually about the business: which customer tier, which provider, which feature flag value. Those need attributes added in code. OpenTelemetry's documentation calls code-based instrumentation a complement to zero-code, and the two can run at once.

Instrumenting during the incident. If the answer required a new log line and a deploy, the system was not properly instrumented by the primer's own measure. Make "what did we have to add?" a standing question in every incident review, and treat each answer as a follow-up item.

Unbounded detail in the wrong signal. User IDs, full URLs and request bodies belong on spans and log records, not in metric labels, where every new value creates another time series. The OpenMetrics standard caps the label names and values on an exemplar at 128 characters in total, precisely to stop exemplars

being used to carry span data, and lets an ingestor discard exemplars altogether. Check that the links survive the whole path from the exporter to the screen before relying on them.

Telemetry that leaks. Spans and logs record what requests contained, and that can include addresses, names and tokens. Decide which attributes may be recorded, strip the rest before storage, and protect telemetry with the same care as the database it describes.

Further reading

- Observability primer — the OpenTelemetry project's definition, with reliability, SLIs and SLOs, spans and distributed traces.
- Signals — traces, metrics, logs and baggage, and the signals still under development.
- Instrumentation — code-based and zero-code instrumentation, and how context propagation ties signals together.
- Resources — resource attributes, the `unknown_service` default and the environment variables that set them.
- Trace Context in non-OTLP Log Formats — the stable specification for recording trace and span IDs in syslog, plain text and JSON logs.
- OpenMetrics 1.0 — the exposition format standard, including exemplars and its advice on cardinality.

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 observability?” 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/observability/