OpenTelemetry Primer

Observability from First Principles

Matthew Reider

OpenTelemetry Primer

Contents

28. Profiling

1. Telemetry

A running program can do one of two things when something goes wrong: fail silently, or tell you what happened. Programs that tell you what happened do so by emitting data about their own behavior. This data is called **telemetry**.

Without telemetry, a service is a closed box. You know it received a request and you know what it returned, but you cannot see what happened inside. With telemetry, the service describes its own internal state as it runs.

```
14:32:01  checkout  received request    /checkout
14:32:01  checkout  queried database    12ms
14:32:02  checkout  sent response       201 Created  387ms
```

2. Signals

Telemetry is not one undifferentiated stream. OpenTelemetry organizes it into three kinds of data, each designed to answer a different question. These kinds are called **signals**.

Metric
: A number measured over time. Answers *how much?* or *how often?*

Trace
: A record of one request's path through services. Answers *where did the time go?*

Log
: A record of a specific event. Answers *what exactly happened?*

The rest of this primer explores each signal, then shows how they connect.

3. Metrics

Imagine ten thousand events per second scrolling past. No one can read them all. A **metric** solves this by counting, averaging, or summarizing those events into a single number over a period of time.

Ten thousand requests become "422 per second." A thousand response times become **percentiles**. A percentile tells you what percentage of values fell below a given number. The p99 (99th percentile) means 99 out of 100 requests were faster than this value – only the slowest 1% were slower. The p50 is the median, the typical experience. The p95 captures most users.

```
10,241 requests    →  "422 req/s"
 9,712 durations     →  "p50: 45ms, p95: 210ms, p99: 820ms"
   238 failures      →  "error rate: 2.3%"
```

4. Metric Types

Not all measurements work the same way. OpenTelemetry defines three types of metric, each suited to a different shape of question.

Counter

A value that only goes up. Total requests served, total bytes sent, total errors. You read it by looking at the rate of increase.

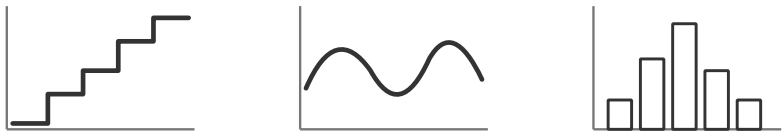
Gauge

A value that goes up and down. Current memory usage, active connections, queue depth. A snapshot of the present moment.

Histogram

A distribution of values sorted into ranges called buckets. How many requests took 0–100ms? How many took 100–500ms? A histogram is what makes percentiles possible – by counting how

many values land in each bucket, you can calculate the p50, p95, or p99.

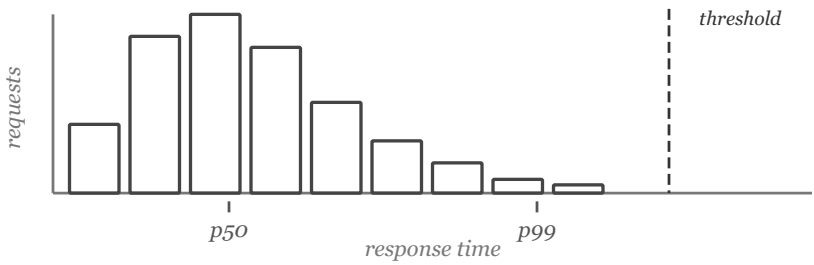


counter *gauge* *histogram*

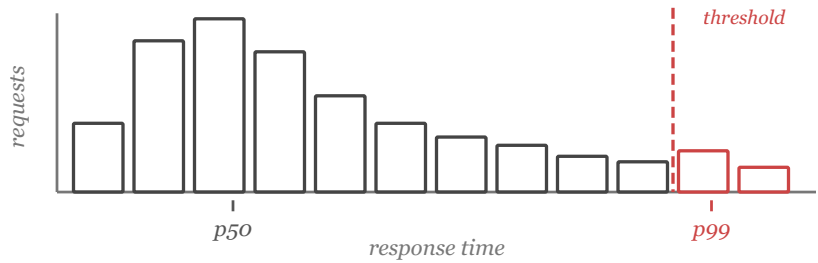
5. **Outliers**

An average can hide problems. If 95 requests take 50ms and 5 take 3,000ms, the average is 197ms – a number that describes nobody's actual experience. The fast majority masks the slow few.

Outliers are those slow few. Percentiles expose them where averages cannot. The p99 separates the worst 1% from the rest. Here, most requests cluster fast and the tail is short – nothing unusual:



Now something goes wrong. The p50 barely moves, but the tail stretches past the threshold – a small subset of requests is suffering:

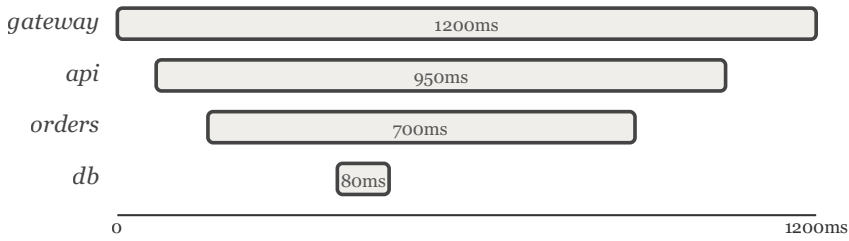


This is why teams set alert thresholds on percentiles rather than averages. An alert that fires when p99 latency crosses a threshold catches degradation that an average would smooth away.

6. Traces

Metrics can tell you that latency is high, but not which service is responsible. When a user's request passes through five services, which one caused the delay?

A **trace** follows that single request from its entry point to its final response, recording the time spent at each service along the way. If a checkout request passes through gateway, api, orders, and database, the trace shows exactly where the time went:



Now you can see that the orders service is the bottleneck.

7. Spans

A trace is made of smaller pieces called **spans**. Each span represents one operation: handling an HTTP request, querying a database, calling another service.

Every span records a start time, a duration, and a status. Spans nest inside each other, forming a tree. The topmost span represents the entire request. Its children represent the steps within it. By reading the tree, you can see which operation took the longest and which one failed. Here is the orders span from the trace above:

```
span:
  name       = GET /api/orders
  trace_id   = 4bf92f3577b34da6
  span_id    = 00f067aa0ba902b7
  parent_id  = a3ce929d0e0e4736
  start      = 14:32:01.234
  duration   = 700ms
  status     = OK
```

Spans can also carry timestamped events and links to spans in other traces – covered in entries 24 and 25.

8. Logs

Metrics tell you *that* something is wrong. Traces show you *where*. Logs explain *why*.

A **log record** captures a single event at a specific moment: an error message, a stack trace (the chain of function calls, with file names and line numbers, that led to the failure), the exact input value that triggered it. Each record carries a timestamp, a severity level (info, warn, error), and a message body. Where metrics summarize and traces narrate, logs preserve the raw detail.

```
14:32:01 ERROR  checkout-svc
  "Connection pool exhausted: max connections reached"
  exception.type  = PostgresError
  db.statement    = SELECT * FROM orders WHERE id = ?
  stacktrace:
    at db.getConnection(pool.py:142)
    at orders.query(orders.py:89)
    at checkout.handle(checkout.py:34)
```

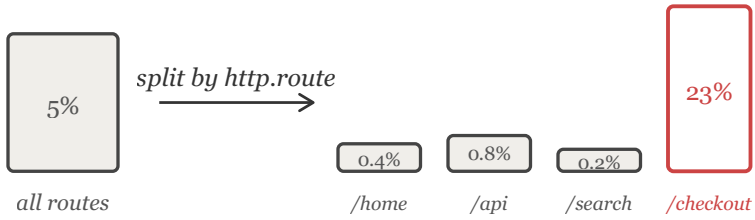
Adding Meaning

9. Attributes

A metric that says "error rate is 5%" tells you something is wrong but not where to look. **Attributes** are key-value pairs attached to any signal – spans, metrics, and logs alike. They turn a single number into something you can search, filter, and split apart.

Attach *http.route* to your error rate metric, and you can split by endpoint. Now you see that */checkout* is at 23% while every other route is below 1%. Add *customer.tier*, and it narrows further: only premium users are affected. (In the context of metrics, attributes are often called **dimensions** – the axes along which you can slice a measurement.)

```
http.route           = /checkout
http.request.method  = POST
http.response.status = 500
customer.tier        = premium
```

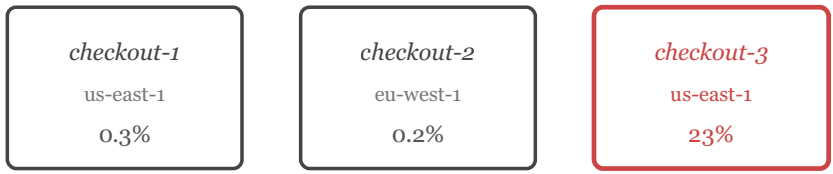


10. Resources

Attributes describe what happened. **Resources** describe who reported it.

A resource is a set of attributes that identify the source of telemetry: the service name, the host, the version, the deployment environment. You now know that */checkout* is failing for premium users. But checkout-svc runs as three instances. Resources let you narrow further – the errors are all coming from one instance in one region. The other two are healthy.

```
service.name     = checkout-svc
service.version  = 2.4.1
deployment.env   = production
host.name        = prod-checkout-3
cloud.region     = us-east-1
```



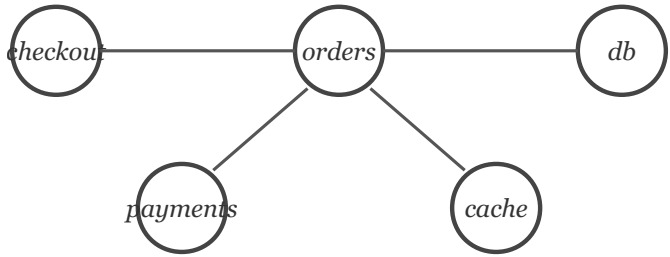
three instances of checkout-svc

Most resource attributes do not need to be set by hand. **Resource detectors** run at startup and automatically discover the environment – the cloud provider, the region, the host type, the container ID. The only attribute teams typically set manually is *service.name*.

```
service.name    = checkout-svc        ← set by you
cloud.provider  = aws                 ← detected
cloud.region    = us-east-1           ← detected
host.type       = m5.xlarge           ← detected
container.id    = a1b2c3d4e5f6        ← detected
```

Resources give the backend something else too. Because every span carries *service.name* and records which service it called, the backend can assemble a live **service map** – a topology of your

system drawn from real traffic, always current, never maintained by hand.



11. Semantic Conventions

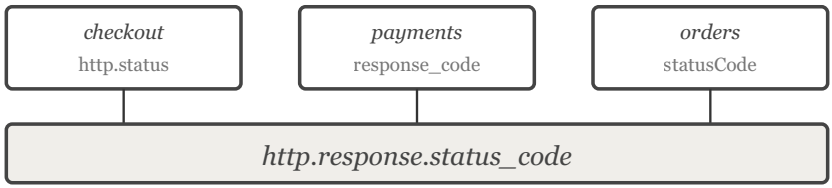
You have narrowed the problem to one instance of checkout-svc. Now you need to check the services it calls – payments, orders, inventory. But each team named their attributes differently. One query cannot find all three.

Semantic conventions are OpenTelemetry's shared naming rules. They cover attributes on spans, metrics, and logs – HTTP calls, database queries, messaging systems, cloud resources, and more. These are not generated by software. They are decided by people – contributors and maintainers from companies across the planet – who agree on what each name should mean. The full list is published as part of the OpenTelemetry specification.

```
checkout:   http.status       = 500
payments:   response_code     = 500
orders:     statusCode        = 500
```

With conventions, all three become:

```
http.response.status_code = 500
```



one query finds all three

Producing Telemetry

12. Instrumentation

An application does not produce telemetry on its own. **Instrumentation** is the code that creates spans, records metrics, and emits logs.

OpenTelemetry offers two approaches. **Automatic instrumentation** hooks into common libraries and frameworks – HTTP servers, database drivers, messaging clients – to produce telemetry without code changes. **Manual instrumentation** uses the OpenTelemetry API directly, for custom business logic that libraries cannot cover.

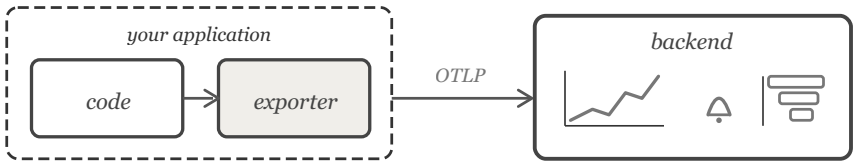
```
# automatic: wraps known libraries, no code changes
opentelemetry-instrument python app.py

# manual: you create spans in your own code
with tracer.start_as_current_span("process-payment"):
    charge(card, amount)
```

13. Exporters

Instrumentation creates telemetry inside a running process. An **exporter** sends it out.

The exporter serializes spans, metrics, and logs into a structured format and transmits them to a destination. That destination is

usually a **backend** – a platform like Datadog, Dynatrace, or Grafana that stores, queries, and visualizes telemetry. It can also be a collector (covered soon) that processes data before forwarding it to a backend.



the exporter sends telemetry out of your application

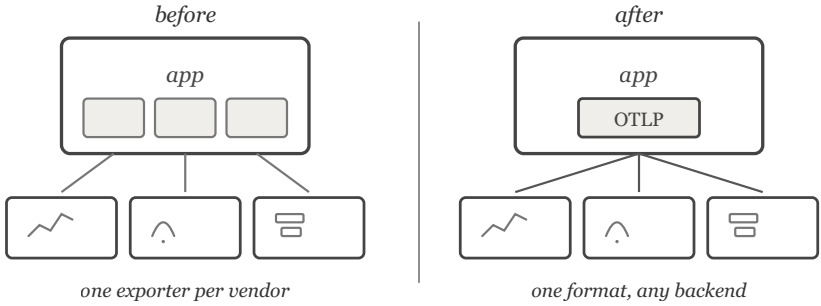
14. OTLP

Before OpenTelemetry, every backend had its own format. Sending telemetry to Datadog required a Datadog exporter and the Datadog format. Sending to Dynatrace required a different exporter and a different format. Supporting both meant maintaining both.

OTLP (OpenTelemetry Protocol) is a single wire format for all three signals. Combined with semantic conventions – which give everyone the same attribute names – it means an application can emit telemetry once, in one format, and any OTLP-compatible backend can receive it.

This is what makes OpenTelemetry *open*. The instrumentation in your code is no longer tied to a specific vendor. Here is a span exported as JSON over OTLP:

```
POST /v1/traces HTTP/1.1
Content-Type: application/json

{ "resourceSpans": [{
    "resource": { "attributes": [
      { "key": "service.name", "value": "checkout-svc" }
    ]},
    "scopeSpans": [{ "spans": [{ ... }] }]
}]}
```



15. Is It All Open?

OTLP and semantic conventions mean your instrumentation is portable. You can switch backends without changing your application code. But *instrumentation* is only one layer.

There is no open standard for dashboards, alert rules, or SLO definitions. An enterprise with 4,000 dashboards in one platform cannot export them to another. The telemetry data is portable. The experience built on top of it is not.

Portable

Instrumentation, data format, attribute names

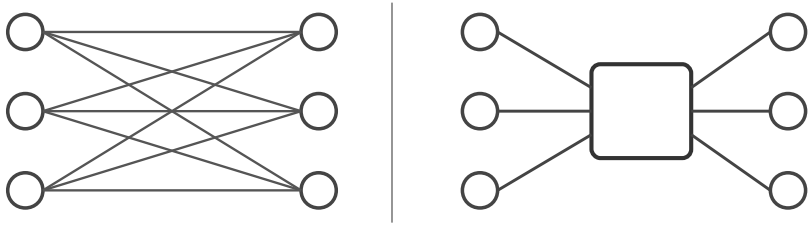
Not portable

Dashboards, alerts, SLOs, queries, integrations

16. The Collector

The **OpenTelemetry Collector** is a standalone program that receives telemetry, processes it, and forwards it to one or more backends.

Without it, every service must send telemetry directly to every backend it needs to reach. Ten services and three backends means thirty connections to manage. The Collector sits between them: services send to the Collector, and the Collector forwards to the right destinations. The number of connections drops from *services* × *backends* to *services* + *backends*.



without collector *with collector*

The Collector is configured with a YAML file:

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

processors:
  batch:

exporters:
  otlphttp:
    endpoint: https://backend.example.com

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

17. **Pipelines**

Inside that YAML configuration, the Collector is organized into **pipelines**. Each pipeline has three stages:

Receivers
: Accept incoming telemetry. A receiver might listen for OTLP, scrape Prometheus endpoints, or accept Jaeger spans.

Processors
: Transform telemetry in flight – batch it, filter it, enrich it, sample it.

Exporters
: Send the processed data to one or more backends.

Because receivers can accept many formats – OTLP, Prometheus, Jaeger, Zipkin, StatsD, and others – the Collector is also a translator. Legacy services that emit Jaeger traces or Prometheus metrics can feed into the same pipeline and export as OTLP, converting an organization's telemetry to one standard without touching application code.

Each signal type gets its own pipeline, so traces, metrics, and logs can be handled differently:

```
service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch, tail_sampling]
      exporters: [otlphttp]
    metrics:
      receivers: [otlp, prometheus]
      processors: [batch]
      exporters: [otlphttp]
    logs:
      receivers: [otlp]
      processors: [batch, filter]
      exporters: [otlphttp]
```

18. **Processors**

Receivers are straightforward – they accept data. Exporters are straightforward – they send it. The interesting work happens in between. **Processors** reshape, filter, enrich, and protect telemetry as it moves through a pipeline, without touching application code.

batch

Groups telemetry into batches before sending. One network call for a thousand spans instead of a thousand calls for one span each.

filter

Drops telemetry you do not want to store. Health-check endpoints, noisy internal routes, debug-level logs in production.

attributes

Adds, removes, or renames attributes. Tag everything with a region name, strip email addresses before data leaves your

infrastructure, normalize attribute names to match semantic conventions.

tail_sampling
: Decides which traces to keep after they complete. Keep errors and slow requests, drop routine successes. Requires gateway mode because it needs the full trace.

aggregate
: Combines many measurements into fewer, summarized ones. A thousand individual latency readings from across services become a single histogram, reducing data volume before it reaches the backend.

memory_limiter
: Prevents the Collector from running out of memory. When usage crosses a threshold, it drops data rather than crashing.

Processors run in the order they appear in the pipeline configuration. A typical chain might batch first, filter out noise, then enrich what remains:

```
processors:
  memory_limiter:
    limit_mib: 512
  filter:
    traces:
      exclude:
        match_type: strict
        span_names: ["health-check"]
  attributes:
    actions:
      - key: environment
        value: production
        action: upsert
  batch:
    timeout: 5s
```

19. Deployment Modes

The Collector is a program – it needs to run somewhere. In most environments, that means Kubernetes. A few terms from that world are needed here.

Pod

One or more containers running together. Each service instance runs in a pod.

Node

A machine that runs pods. A cluster has many nodes.

DaemonSet

A Kubernetes resource that automatically runs one pod on every node.

Sidecar

An extra container that runs alongside your application inside the same pod.

With these terms, there are three ways to deploy the Collector.

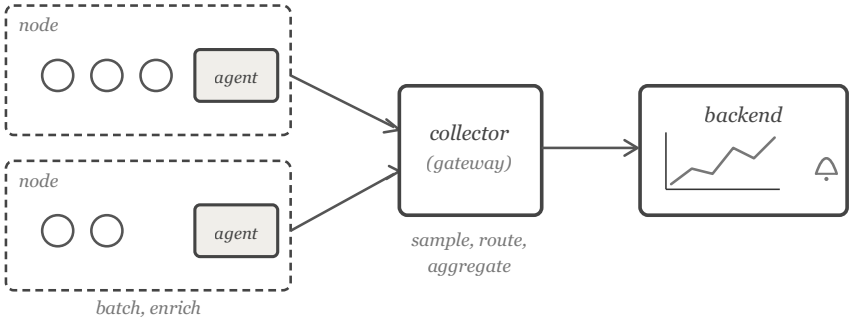
Agent mode

One Collector per node (deployed as a DaemonSet) or per pod (as a sidecar). Applications send to localhost. This is not the same as the auto-instrumentation agent from entry 12 – here "agent" means a Collector instance running close to the application, doing lightweight work.

Gateway mode

A centralized Collector deployment – multiple replicas behind a load balancer. Receives telemetry from agents. Handles heavier processing that requires seeing data from many services at once.

In practice, these modes are layered. Agents handle lightweight work and forward to a gateway for the expensive processing. The gateway exports to the backend.



Crossing Boundaries

20. Trace Context

A trace follows a request across services. But when a request leaves one service and arrives at another, how does the second service know they belong to the same trace?

Through a header. The W3C **traceparent** header carries four fields. The version (*00*) identifies the format. The trace ID and span ID link this request to its trace. The flags field controls sampling – *01* means "record this trace," *00* means "do not record." Sampling decisions are covered soon. Every service reads this header, continues the trace, and passes it to the next service it calls.

```
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
             ||                  ||                       ||       ||
          version            trace-id                 span-id   flags
                                                             (01 =
sampled)
```

21. Baggage

Trace context carries telemetry identity – the trace ID and span ID. **Baggage** uses the same propagation mechanism to carry application data.

A frontend service might set *user.tier=premium* in baggage. Every downstream service receives it automatically, without extra database lookups or API calls. Baggage values can then be added as attributes to spans, making business context available throughout the request.

```
baggage: user.tier=premium,region=us-east-1

frontend  →  api  →  orders  →  database
  (set)     (read)   (read)     (read)
```

Baggage travels with every request. Keep values small and remember that all downstream services can read them.

22. Sampling

At low traffic, you can keep every trace. At high traffic, storing everything becomes too expensive. **Sampling** is the practice of choosing which traces to keep.

Head sampling

Decides at the start of a trace, before anything has happened. Keep 10% at random. Simple, cheap, works anywhere. The tradeoff: it is blind – it will drop errors and keep routine successes because it cannot see the future.

Tail sampling

Decides after a trace completes, when the outcome is known. Keep all errors and slow requests, drop the rest. Guarantees you never lose an interesting trace. The tradeoff: it requires the full trace at a single location, so it only works at the gateway tier and uses more memory.

Many teams combine both: head sampling at the agent to reduce volume by a rough percentage, then tail sampling at the gateway to keep the interesting traces from what remains.

23. Trace-Aware Routing

Tail sampling requires the complete trace – every span, from every service – at a single gateway replica. But when a trace has twelve spans arriving from five different DaemonSets, a normal load balancer scatters them across replicas. No replica sees the whole trace, and sampling breaks.

The solution is **trace-aware routing**. Instead of round-robin load balancing, each DaemonSet hashes the trace ID to choose which gateway replica receives the span. All spans sharing a trace ID are routed to the same replica, giving it the complete picture.

```
exporters:
  loadbalancing:
    protocol:
      otlp:
        endpoint: gateway-headless:4317
    resolver:
      dns:
        hostname: gateway-headless
```

The **loadbalancingexporter** uses a headless Kubernetes Service, which returns individual pod IPs rather than a single cluster IP. The exporter discovers all gateway pods and applies consistent hashing – so when a pod is added or removed, only a small fraction of traces shift to a new replica.

24. Events

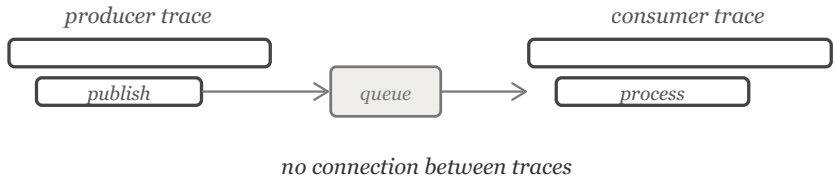
Sometimes you want to record that something happened inside a span without creating a child span for it. An **event** is a lightweight, timestamped annotation attached to a span.

Each event has a name, a timestamp, and optional attributes. Events mark moments like a cache miss, a retry attempt, or a validation failure – things worth noting that do not need their own duration measurement.

```
span: process-order  (340ms)
  events:
    t=28ms   cache.miss     key=inventory:42
    t=155ms  retry          attempt=2
    t=312ms  payment.ok     provider=stripe
```

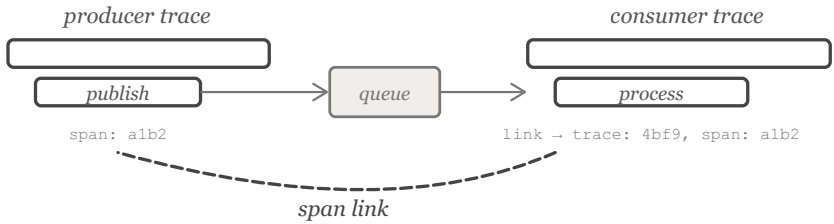
25. Span Links

Spans within a trace are connected by parent-child relationships. But some connections cross traces entirely. Consider a queue: one service publishes a message, another consumes it later. Each side has its own trace. Without a way to connect them, the consumer's trace has no record of where the message came from:

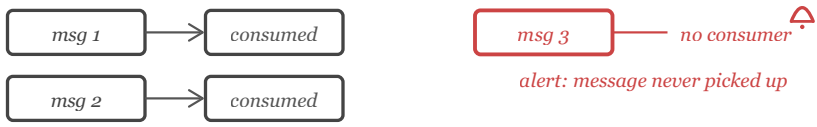


Span links solve this. The consumer's "process" span records a link containing the trace ID and span ID of the producer's "publish"

span. The two traces remain independent, but the causal relationship is preserved:



This matters for monitoring. Links let you measure how long messages wait in the queue, trace failures back to what the producer sent, and detect messages that were never consumed at all:



Connecting Signals

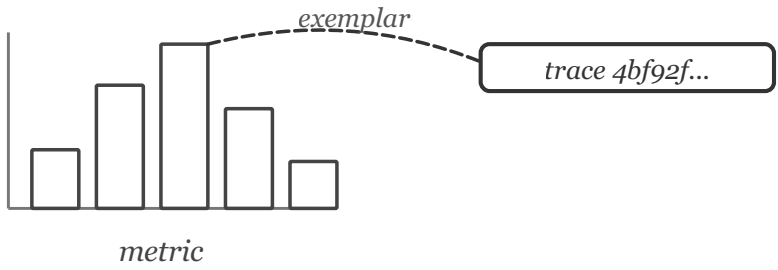
26. Correlation

So far, each signal lives in its own world. Metrics go to one place, traces to another, logs to a third. **Correlation** is the practice of connecting them so you can move between signals in a single investigation.

Some correlation is wired into the data itself. An OpenTelemetry **log appender** injects the active trace ID and span ID into every log record, linking logs to traces automatically:

```
14:32:01 ERROR "Connection pool exhausted"
  trace_id = 4bf92f3577b34da6
  span_id  = 00f067aa0ba902b7
```

An **exemplar** does the same for metrics – it attaches a trace ID to a specific data point, so you can jump from "something was slow" to the exact trace that was slow:



These are examples of *explicit* correlation – identifiers placed in the data by the instrumentation. But not all correlation lives in the data. Backends can also *infer* connections on their own – using the service map topology, the timing of events, and in some cases AI – to detect that a database slowdown and a checkout error rate spike are related, even without an explicit trace ID linking them. The richer the telemetry, the more a backend can connect.

Explicit
: Trace IDs in logs, exemplars on metrics, span links across traces. Wired in by the instrumentation or Collector. Precise but requires setup.

Inferred
: Topology, timing, resource matching, AI. Discovered by the backend from the shape of the data. Broader but approximate.

Both kinds feed into the same goal: an investigation where you can move from one signal to another without starting over.

27. The Investigation

Correlation – explicit and inferred – gives the backend the raw connections. But connections alone are not an investigation. The backend detects a problem, fires an alert, and assembles the evidence into a single view. The investigation moves through three stages:

Triage

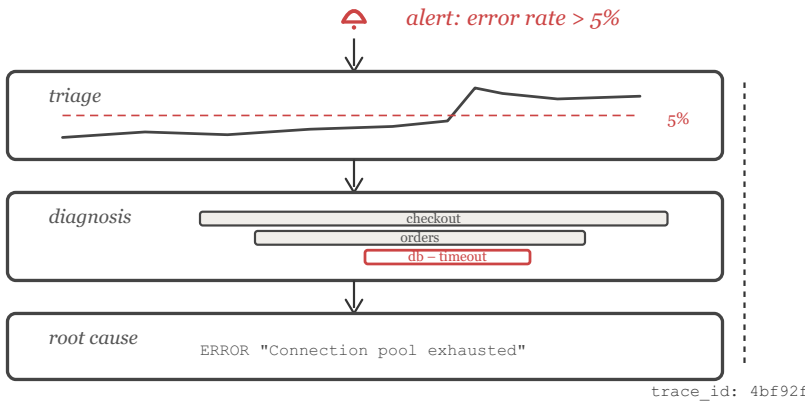
A metric crosses a threshold and an alert fires. How bad is it? Split by attributes – it is */checkout*, premium users, *us-east-1*.

Diagnosis

Follow a trace from the alert. The waterfall shows the orders service is timing out on calls to the database. Now you know where.

Root cause

Read the logs attached to that span. "Connection pool exhausted: max connections reached." Now you know why.



28. **Profiling**

Traces show where time is spent across services. **Profiling** shows where time is spent inside a single service – at the function level – by capturing the actual stack frames executing on the CPU.

OpenTelemetry's profiling signal links profiles to traces through span context. A slow span does not just tell you the operation took 700ms; it connects to a flamegraph showing which functions consumed that time.

```
processRequest ──────────────────────── 700ms ──────
├─ validateInput ── 20ms ──┤
├─ queryDatabase ──────────────── 580ms ──────────┤
│  ├─ getConnection ─── 490ms ──────────┤  ← slow
│  └─ executeSQL ── 90ms ──┤
└─ formatResponse ── 15ms ──┤
```

Profiling is the fourth signal. The specification was finalized in 2024, but SDK implementations are still emerging as of this writing. It is not yet as widely supported as metrics, traces, and logs.

About the Author

Matthew Reider has spent more than twenty years in software, much of it at the intersection of open source communities and infrastructure. He found his footing in the Ruby on Rails community at Engine Yard, helped grow Cloud Foundry into one of the first enterprise platform-as-a-service projects, and now works at Dynatrace alongside the Kubernetes and OpenTelemetry communities. He lives in Vienna, Austria with his wife, kids, cats, and dog.