

Observability: Logs, Metrics, and Traces

Chris Timperley · Monday, September 28, 2026

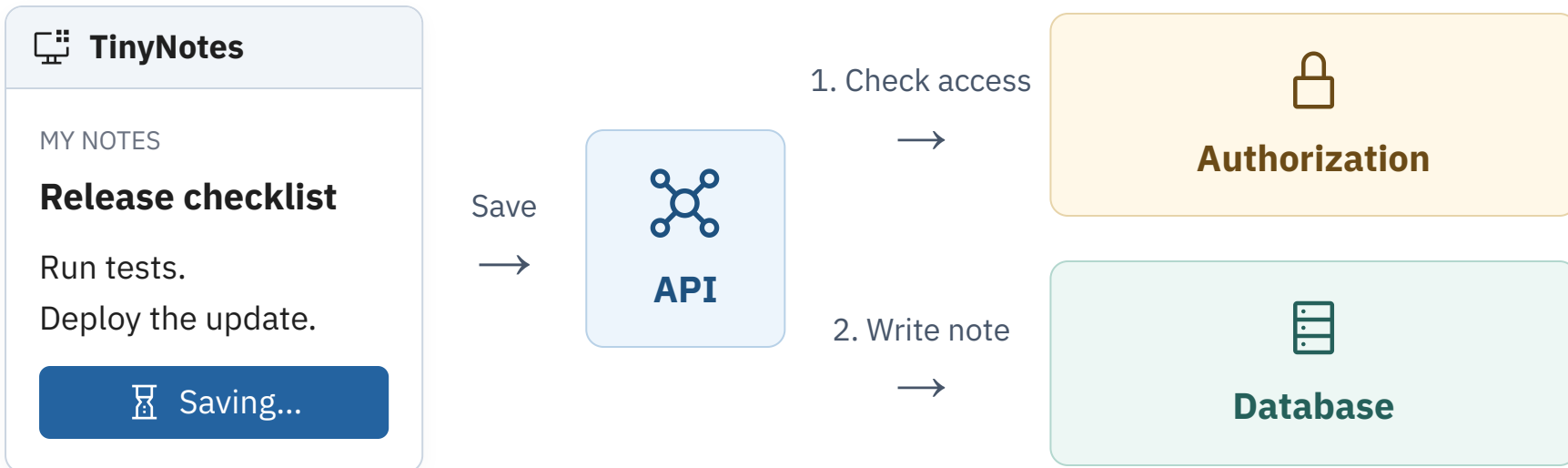
Administrivia

- **Homework 3 (Build and Operate a CD Pipeline)**
  - design your pipeline → implement → demonstrate
  - we will give **root privileges** to your accounts
- **Lab: OpenTelemetry, Prometheus, Grafana**
- **Midterm I**
  - covers the first half of the course
  - designed to be completed in 60 minutes
  - will share more details later this week

🎯 Learning Goals

- Explain the roles of logs, metrics, and traces in understanding a running system.
- Compare structured and unstructured logs and choose fields that make events easy to search and connect.
- Interpret counters, gauges, rates, and latency distributions, explaining how labels and aggregation affect what metrics reveal.
- Read a distributed trace and explain how spans and propagated context connect services.
- Choose a signal for a question about application behavior and explain its limitations.

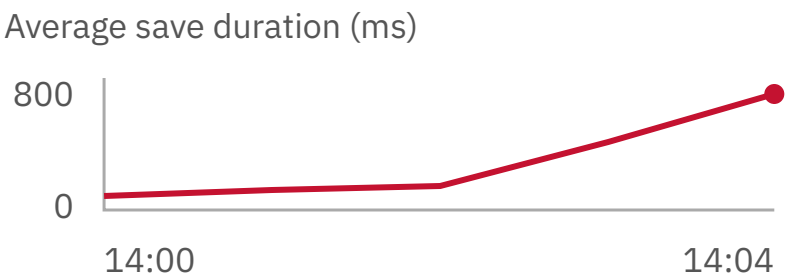
Today: Why is this save slow?



- A TinyNotes user clicks **Save** and waits. **Where is the delay?**

Monitoring is a practice

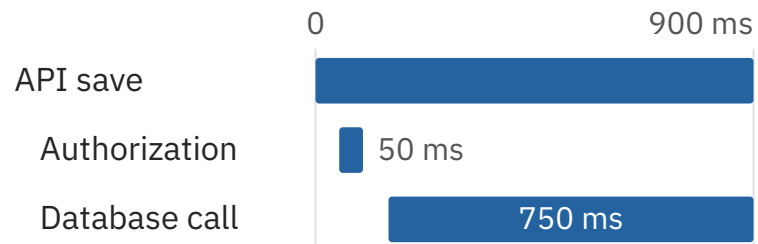
Are saves getting slower?



- **Practice:** Track selected measures over time to spot changes and problems.

Observability is a property

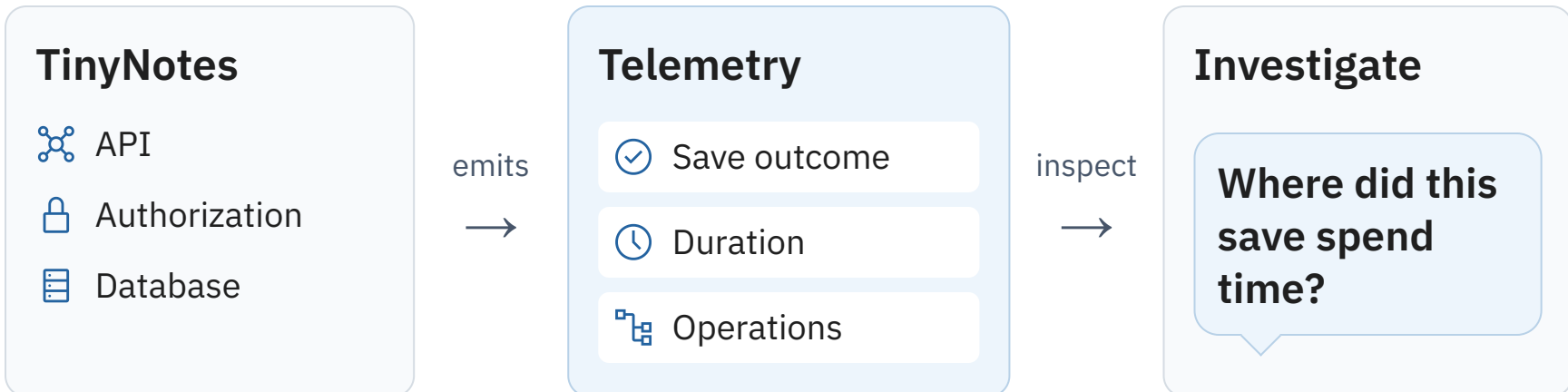
Why is this save slow?



- **Property:** How well we can explain what happens inside a system using the data it produces.

Observability depends on telemetry

Telemetry is data a running system produces about its behavior.



We can only explain what our telemetry lets us observe.

The Three Pillars of Telemetry

Logs

Record individual events.

```
14:02:11 ERROR
Save failed
db_timeout
```

Metrics

Measure behavior over time.

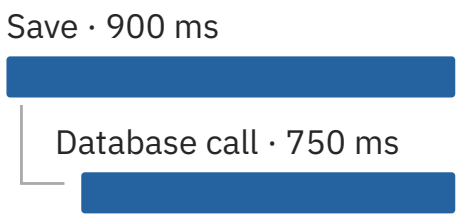
5% failed



of saves in the last minute

Traces

Follow a request across operations.



- Each **signal** gives us a different view of system behavior.

Logs

Why not just use console.log()?

A printed message is a starting point

A **log entry** records an event that happened while a program ran.

```
console.log("Save failed");
```

TinyNotes handles several saves at once. Its output might be:

TinyNotes · API output

```
Save failed
Save failed
```

Which request failed? When? Why?

A logging library centralizes output policy

Application code records **what happened** and assigns a severity:

```
logger.error("Save failed");
```

TinyNotes · API output

```
2026-09-28T14:02:11Z error [notes.save]: Save failed
```

Time narrows the search. **Module** locates the code. **Severity** marks the failure.

The library handles **level filtering**, consistent **formatting**, and output **destinations**.


Winston


Loguru


Log4j

Control log verbosity without changing code

```
// Diagnostic state
logger.debug("Pool: 2 waiting");

// Routine event
logger.info("Save completed");

// Recoverable problem
logger.warn("Retrying connection");

// Failed operation
logger.error("Save failed");
```

- **Change verbosity at runtime** without changing logging calls.
- Set `logger.level` to `debug` for detail or `warn` for warnings and errors.
- Only events at the threshold or higher are emitted.

How do we find saves that took too long?

⚠ Prose makes field extraction fragile

Find **saves that failed after more than 500 ms**:

TinyNotes · API output

```
error: Save failed after 900 ms: db_timeout
error: Database timed out; saving note took 1.1 seconds
```

Both messages describe a slow failed save, but use different wording and units.

- A text search can find known words.
- Comparing durations requires extracting and interpreting values.
- A wording change can break the extraction.

Structured logs make events queryable

```
logger.error("Save failed", {
  operation: "save_note",
  outcome: "failure",
  reason: "db_timeout",
  duration_ms: 900,
});
```

```
{
  "level": "error",
  "message": "Save failed",
  "service": "api",
  "timestamp": "2026-09-28T14:02:11Z",
  "operation": "save_note",
  "outcome": "failure",
  "reason": "db_timeout",
  "duration_ms": 900
}
```

Which events belong to the same request?

A correlation ID groups one request's events

```
7f3a9c2e  api            Save started
b6d104af  api            Save started
7f3a9c2e  authorization  Access allowed
7f3a9c2e  api            Save failed: db_timeout
```

At API entry, once per request:

```
const requestId =
  crypto.randomUUID();
```

Keep the ID in the request's logs and pass it to downstream services.

A **correlation ID** groups one request's events across services.

A child logger carries request context

```
const requestLogger = logger.child({
  correlation_id: requestId,
});

requestLogger.error("Save failed", {
  operation: "save_note",
  outcome: "failure",
  reason: "db_timeout",
  duration_ms: 900,
});
```

```
{
  "level": "error",
  "message": "Save failed",
  "service": "api",
  "timestamp": "2026-09-28T14:02:11Z",
  "correlation_id": "7f3a9c2e",
  "operation": "save_note",
  "outcome": "failure",
  "reason": "db_timeout",
  "duration_ms": 900
}
```

- A **child logger** attaches shared fields to each record it creates

Services must share the same correlation ID

API

Generate an ID once

`7f3a9c2e`

Request header

`X-Correlation-ID`

→

`7f3a9c2e`

Authorization

Reuse the received ID

`7f3a9c2e`

API · logs

```
7f3a9c2e  Save started
7f3a9c2e  Save failed
```

Authorization · logs

```
7f3a9c2e  Access allowed
```

Same ID in the header and every related log record.

Should we log every function call?

Choose events that answer questions

Question	Event and useful context
Did the save finish?	Completion: outcome, duration, correlation ID
Why did it fail?	Failure: operation, reason, relevant error details
Did a retry recover?	Retry: attempt number, reason; eventual outcome

- Record outcomes and decisions that help **explain system behavior**
- Prefer meaningful **outcomes** and **decisions** over a message at every line of code

Keep context; exclude sensitive content

Choose safe fields explicitly; **redact sensitive values before emitting the record**.

Keep operational context

```
{
  "correlation_id": "7f3a9c2e",
  "operation": "save_note",
  "outcome": "failure",
  "reason": "db_timeout"
}
```

Identifies the operation and result.

Exclude private data

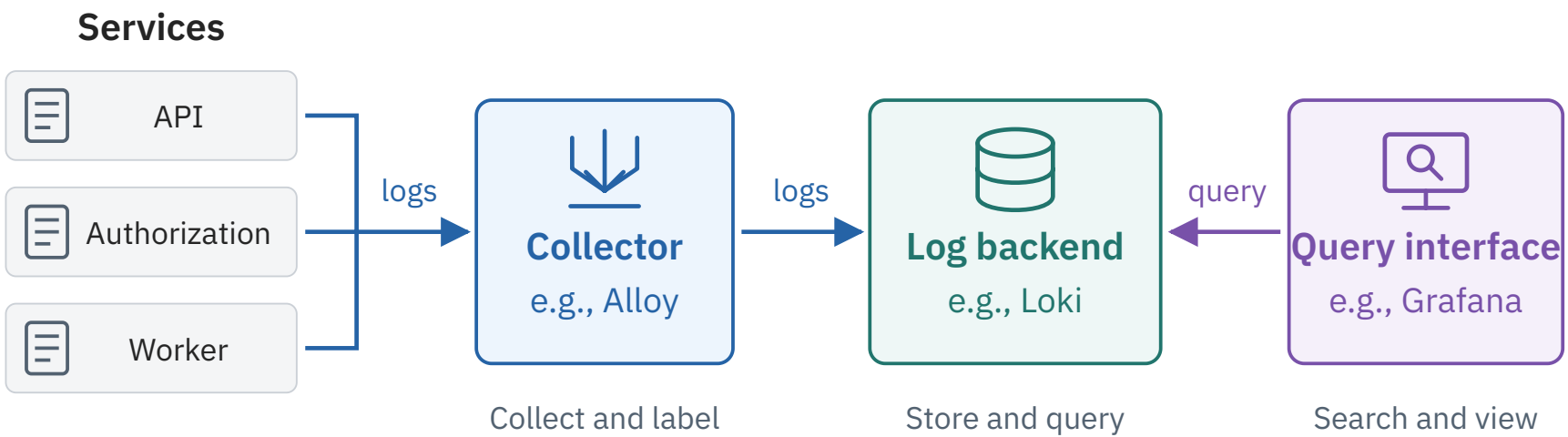
```
note_body: [private text]
authorization: [token]
password: [secret]
```

Private content and credentials can spread to files, collectors, storage, and backups.

How do we find what we need in the logs?

Log aggregation brings service logs together

- Collect logs centrally so you can search across services in one place



Structured logs let you query recorded events

“Which note saves failed?”

```
{service="api"} | json
  | operation="save_note"
  | outcome="failure"
```

Query results

```
14:02:11  api  save_note  failure
14:03:24  api  save_note  failure
...
```

Query by correlation ID to find related events

```
{service=~"api|authorization"}
  | json | correlation_id="7f3a9c2e"
```

Query results · 7f3a9c2e

```
api            Save started
authorization  Access allowed
api            Save failed: db_timeout
```

Breakout: What would you log?

A photo-upload API stores a photo, then calls a thumbnail service.

Using the handout, add **up to four logging calls** across both services.

- Choose each event's **location, level, and fields**.
- Show how a **correlation ID** travels between services.



Mark the code and complete the design sheet, including one example record.

Can your logs answer this?

Users report that **some uploads take several seconds and then fail**.

Write two LogQL queries using your fields:

1. Find **failed uploads lasting more than two seconds**.
2. For one match, find **related events from both services**.

Missing evidence? Revise one logging call.



One recorded event

This upload failed after 3 seconds

Logs preserve event details. A correlation ID links related events.



Across requests

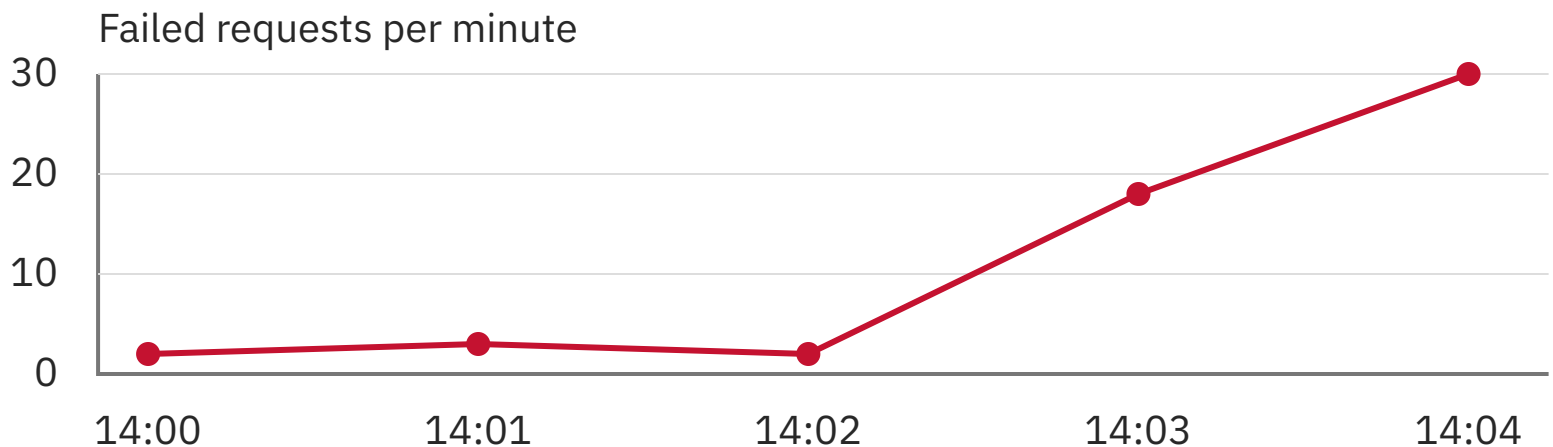
How often do uploads fail?

Metrics summarize failures and traffic over time so we can see whether the failure rate is rising.

Metrics

Metrics summarize system behavior over time

A **metric** is a numerical measurement of events or system state. Here we count failed playback API requests each minute:



Each point summarizes many requests. **The pattern shows when failures increased.**

Three categories answer different questions



Business

Are users succeeding?

Streaming: playback starts per second.



Application

Is the software working?

Streaming: playback API errors and request durations.

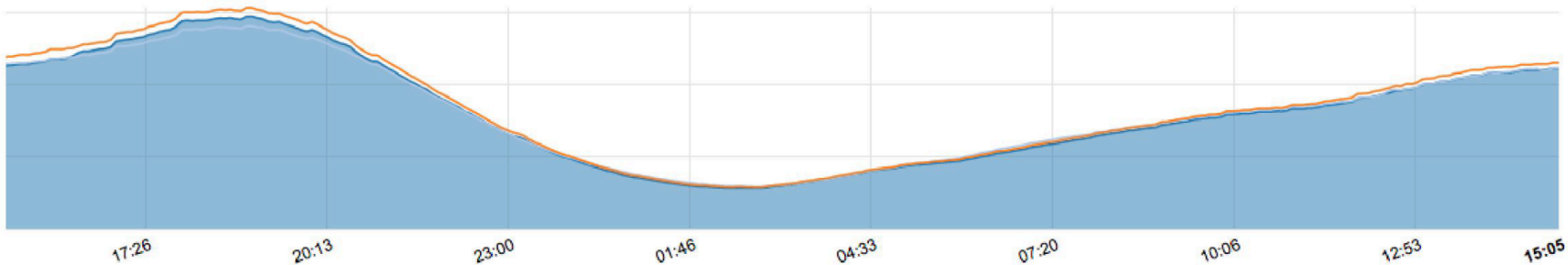


Infrastructure

Are resources healthy?

Streaming: server CPU, memory, and network utilization.

Business metrics track product outcomes



- Netflix's **stream starts per second (SPS)** tracks an event close to its core product goal: **people watching video**.
- A drop below the **expected pattern** can reveal trouble across devices, interfaces, or servers. The normal overnight decline is expected.

Netflix: SPS, the Pulse of Netflix Streaming (2015)

Start with what users came to accomplish

Product	What users want to do	Example metric
Streaming	Watch a video	Playback starts per minute
Shopping	Buy a product	Completed purchases per hour
Collaboration	Share work with others	Documents shared per day

- Choose a **successful user action**, then measure how often it happens
- Name the action, define what counts as success, and choose a time window. Compare with the expected pattern for that time of day or week

Application metrics describe behavior

Playback API

Handles requests
to begin playback

Request volume
Failed requests
Response times

Video worker

Processes video
in the background

Jobs completed
Failed jobs
Time jobs spend waiting

Video player

Loads and plays
video for the viewer

Time until playback starts
Buffering events
Playback errors

- SPS counts starts; application metrics reveal **slow or failing software**

RED measures request-driven services

Rate

How much traffic?

10 requests/s

600 requests ÷ 60 seconds

Errors

How often does it fail?

5% failed

30 failures ÷ 600 requests
0.5 failures/second

Duration

How long does it take?

50–900 ms

Record the distribution of request durations.

Source: Tom Wilkie: The RED Method

Infrastructure metrics describe resources



CPU

Executes application code

Busy time; threads waiting to run



Memory

Holds data and video buffers

Bytes used; available capacity



Storage

Stores and reads video data

Free space; queued I/O; read errors



Network

Transfers video data

Bandwidth used; dropped packets

Exporters expose measurements for collection. Node Exporter reports host metrics.

USE organizes metrics for each resource

Utilization

How much is in use?

CPU busy time
Memory used / capacity
Network bandwidth used

Saturation

What is waiting?

Threads waiting for CPU
Queued disk operations
Full-queue packet drops

Errors

What is failing?

Failed disk reads
Network interface errors
Memory hardware errors

Source: Brendan Gregg: The USE Method

Connect user impact to a possible cause

	Earlier	Now
Business Stream starts per second	10	8
Application · RED Playback API	10 requests/s 1% failed 95% finish ≤ 100 ms	10 requests/s 10% failed 95% finish ≤ 800 ms
Infrastructure · USE Video storage disk	40% busy · 0 errors No queued work	90% busy · 0 errors 12 queued on average

Work can queue even when the disk reports no errors.
Investigate the disk queue.

How do we record these measurements?

Counts, current values, and distributions



Counter

Events accumulated so far

Playback starts
Failed API requests



Gauge

A quantity right now

Available memory
Queue length



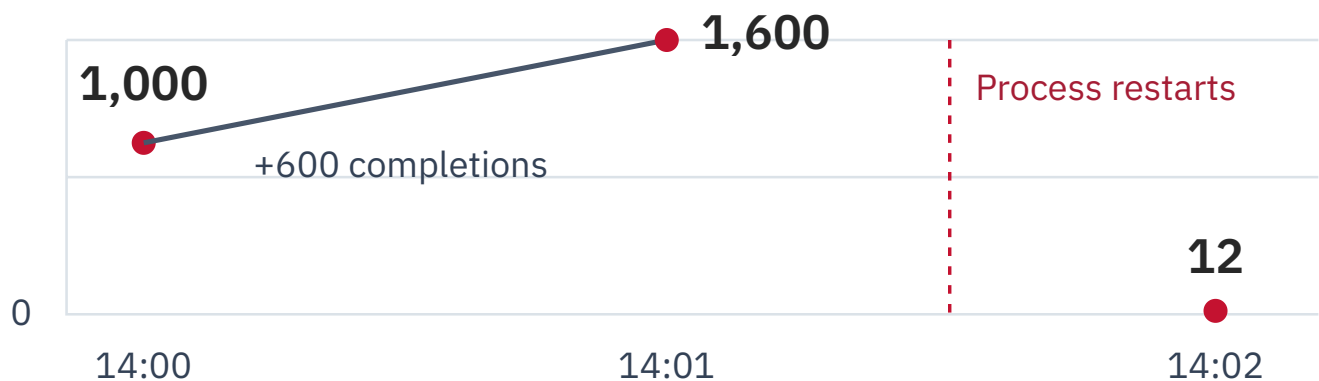
Histogram

How observations are distributed

Request durations
Response sizes

Counters accumulate events until they reset

Completed playback requests since this API process started:



600 completions ÷ 60 seconds = **10 requests/second**

Another counter: **login attempts since process start**.

Gauges measure quantities that rise and fall

Active playback API requests at each sample:



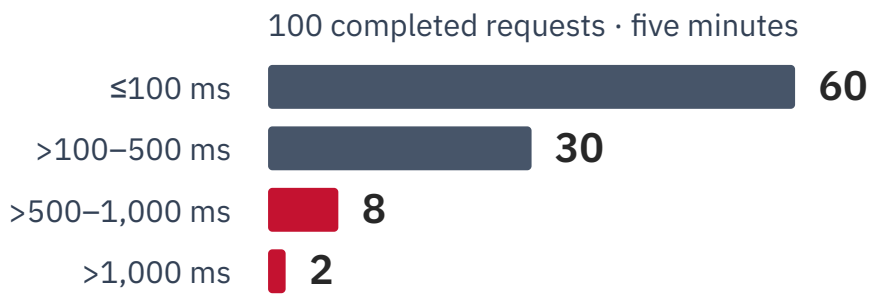
8 active requests means eight are still in progress.

Another gauge: **open database connections right now**.

Histograms count values in ranges

A **histogram** groups values into ranges called **buckets**.
In this example, boundaries are chosen before recording.

240 ms → the **>100–500 ms** range gains one request



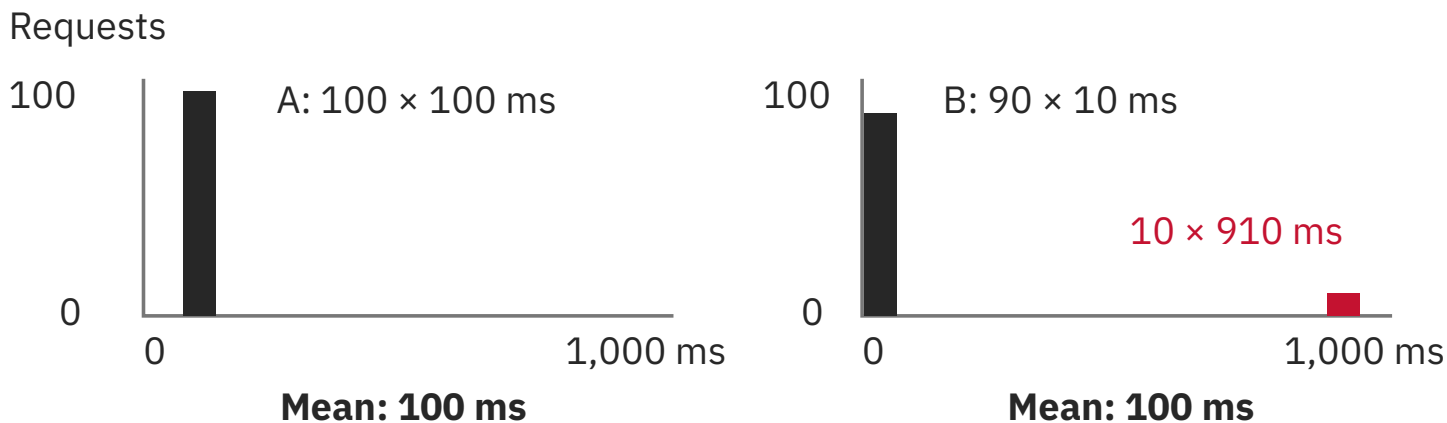
90 requests took at most 500 ms; 10 took longer.

Another histogram: **response sizes in bytes**.

Native histograms can create buckets dynamically at a configured resolution.

The same mean can hide very different waits

Two groups of 100 completed playback API requests:

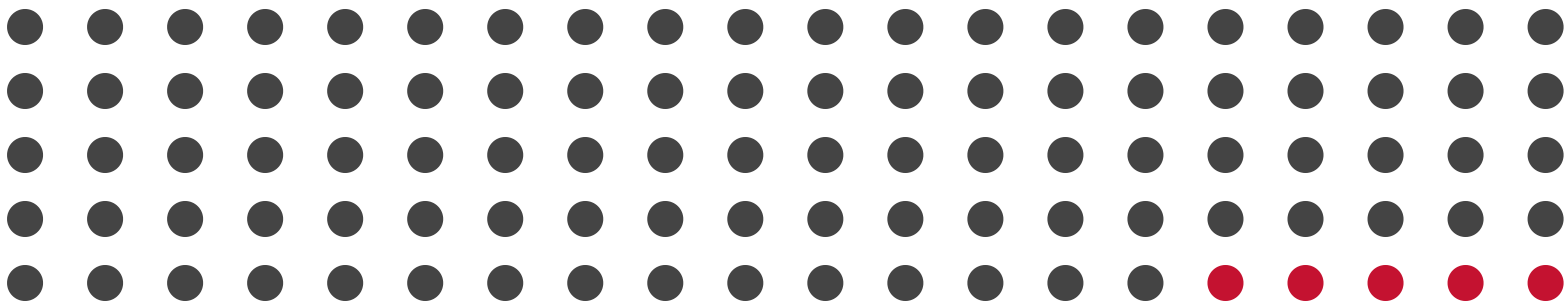


Both means are **100 ms**, but one in ten requests in B takes **910 ms**.

An average alone hides the experience of the slower requests.

A percentile describes the slower requests

p95 = 800 ms: at least 95% of requests took at most 800 ms.



95 requests at or below 800 ms

5 slower requests

p50 describes the middle; **p99** reaches farther into the slow tail. Histogram buckets let us estimate percentiles without keeping every duration.

How do we compare the same measurement across different contexts?

Attributes describe a measurement's context

One completed playback request

0.25 seconds

Request duration

HTTP method

`http.request.method = "POST"`

Route

`http.route = "/playback"`

- **Attributes** are name–value pairs that describe the request.
- Prometheus calls these grouping dimensions **labels**.

The metric stays the same across requests

Request duration · seconds

`http.server.request.duration`

One playback request

0.25 seconds

`http.route = "/playback"`

One catalog request

0.05 seconds

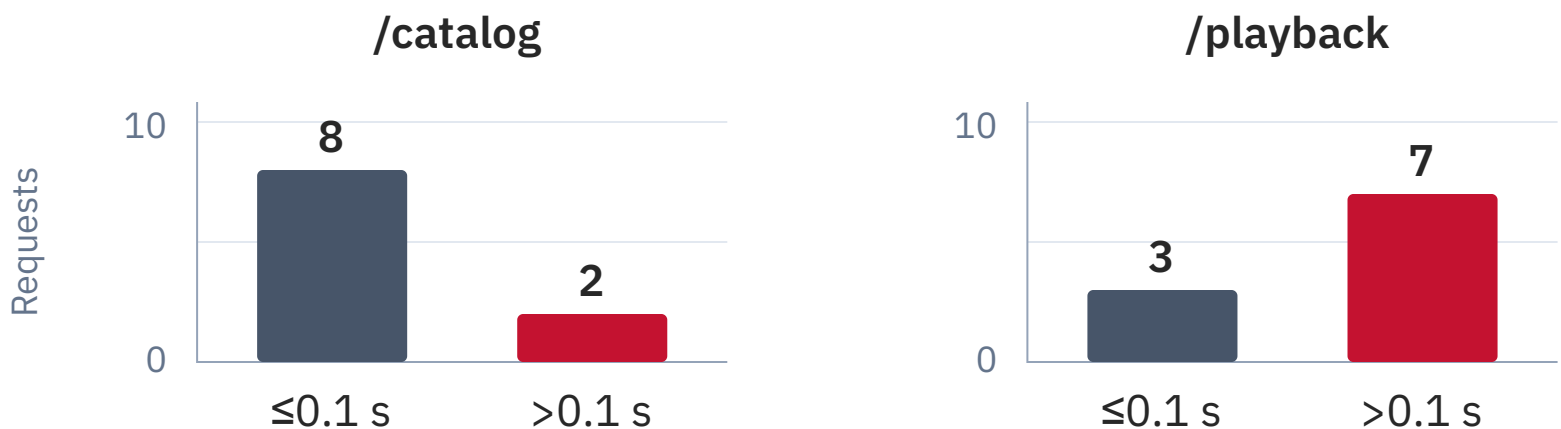
`http.route = "/catalog"`

Same metric and unit. **Different values and attributes.**

One metric can keep separate distributions

`http.server.request.duration`

Grouped by `http.route` · 10 requests per route in the same time window



Compare routes separately, or **combine their bucket counts** for all requests.

Breakout: What would you measure?

A photo-sharing service accepts uploads, then processes photos in the background before making them available to view.

Web and mobile users report that **photos sometimes take a long time to appear**.



- In pairs, propose **one or two metrics per category**: business, application, infrastructure.
- Define each on the worksheet, then exchange with another pair, challenge, and revise.



Across requests

10% of playback requests fail

Counts show the extent of the problem. Attributes isolate the affected operation.



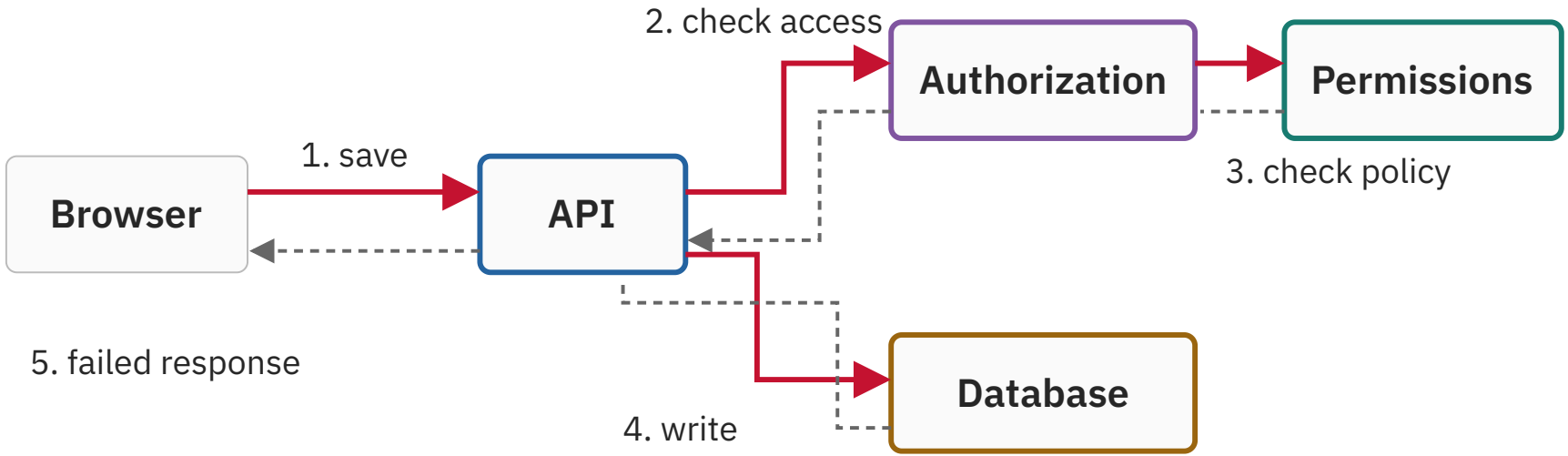
Inside one request

Where did it spend time?

A trace follows the calls and waits behind one playback request.

Traces

One save crosses several services



The save took 900 ms. What slowed down the request?

Distributed tracing follows requests across services



OpenTelemetry

Instruments the code: records operations and carries trace context between services.



Jaeger

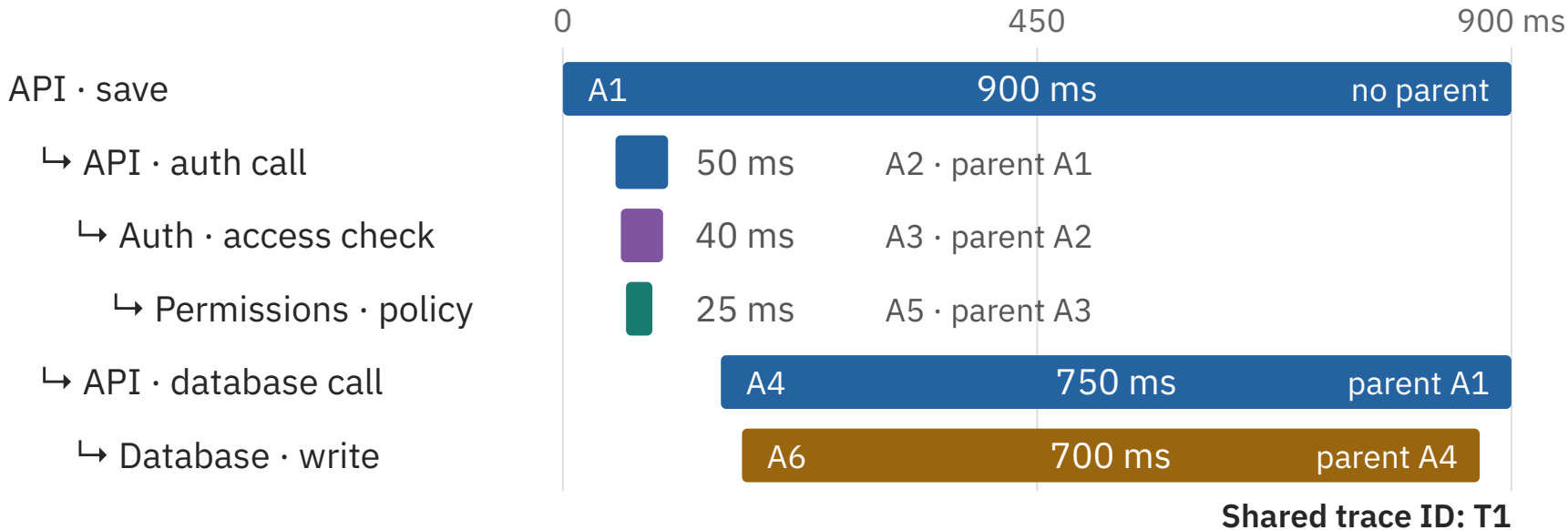


Grafana Tempo

Store and query traces. Inspect them in Jaeger or, for Tempo, in Grafana.

Record operations → collect traces → inspect one request.

A trace follows one request



The trace ID groups these operations into one trace.

Span IDs identify operations; parent IDs connect them.

Spans record more than duration



Attributes · details about the operation

Key–value pairs: `db.operation.name=INSERT`, `note.id=42`.



Status · the operation's outcome

The failed database call: **Error** · `database timeout`.

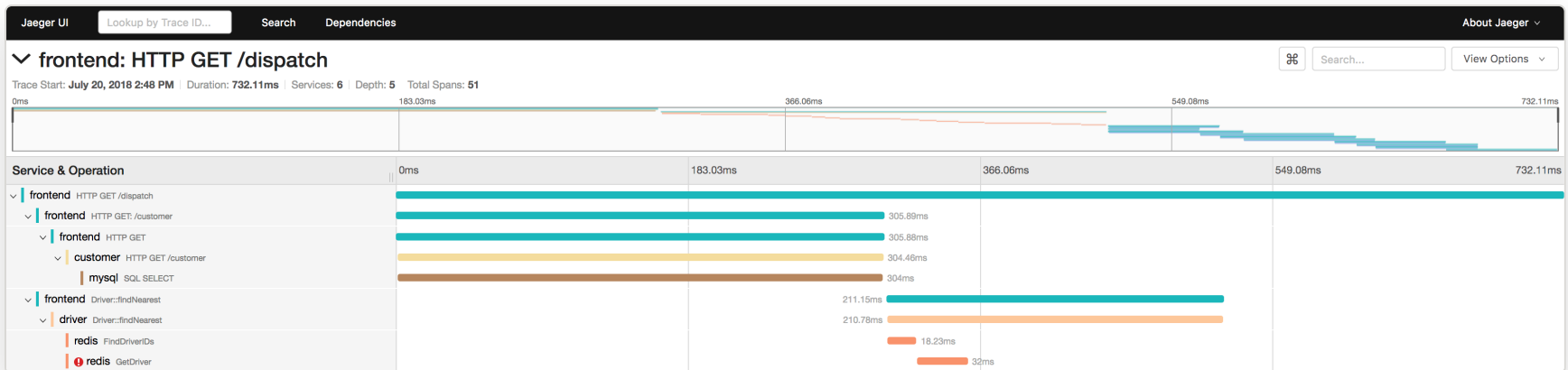


Events · timestamped moments

`connection acquired` at **+20 ms** into the database call.

OpenTelemetry: span attributes, status, and events

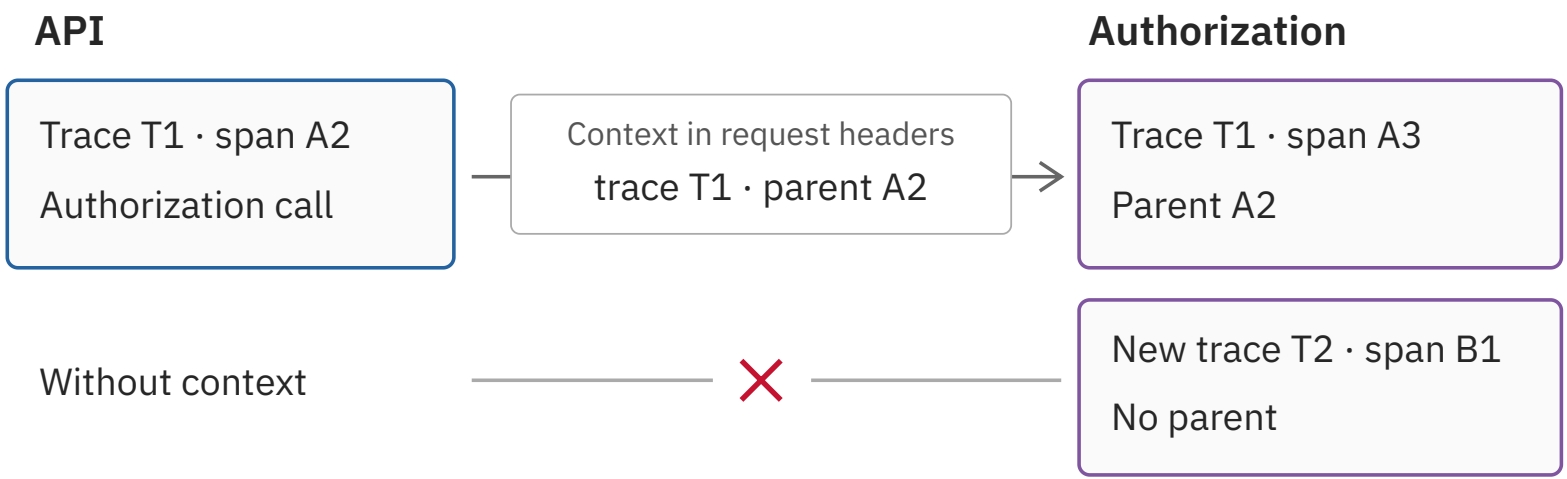
Demonstration: Ride Requests (Jaeger)



- Demo: https://www.jaegertracing.io/demo
- HotROD App: https://hotrod.demo.jaegertracing.io

Requests carry IDs across service boundaries

Trace context tells the next service which trace and parent to use.

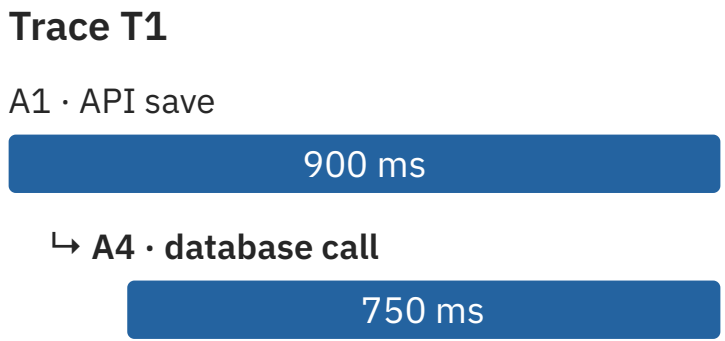


Both services can record spans and still produce disconnected traces.

OpenTelemetry: context propagation

Follow a log to the matching span

```
{
  "service": "api",
  "reason": "db_timeout",
  "trace_id": "T1",
  "span_id": "A4"
}
```



- Search for trace **T1**, then select span **A4**.
- The log records the timeout; the trace shows where it occurred in the request.

Collection and Storage

Bringing signals together across the system



Collectors

Gather signals and forward them.

- OpenTelemetry Collector
- Grafana Alloy



Backends

Store signals and answer queries.

- **Prometheus:** metrics
- **Loki:** logs
- **Tempo:** traces



Query interface

Search and display results from backends.

- Grafana

Where do the signals come from?

Instrumentation records evidence as a system runs

Code-based

Add calls that record application meaning.

```
from opentelemetry import trace

tracer = trace.get_tracer("tinynotes.notes")

with tracer.start_as_current_span(
    "save_note"
) as span:
    if not text.strip():
        span.set_attribute(
            "outcome", "rejected_empty"
        )
    else:
        store_note(text)
        span.set_attribute("outcome", "stored")
```

Zero-code

Capture library operations without adding instrumentation calls.

```
opentelemetry-instrument \
  python app.py
```

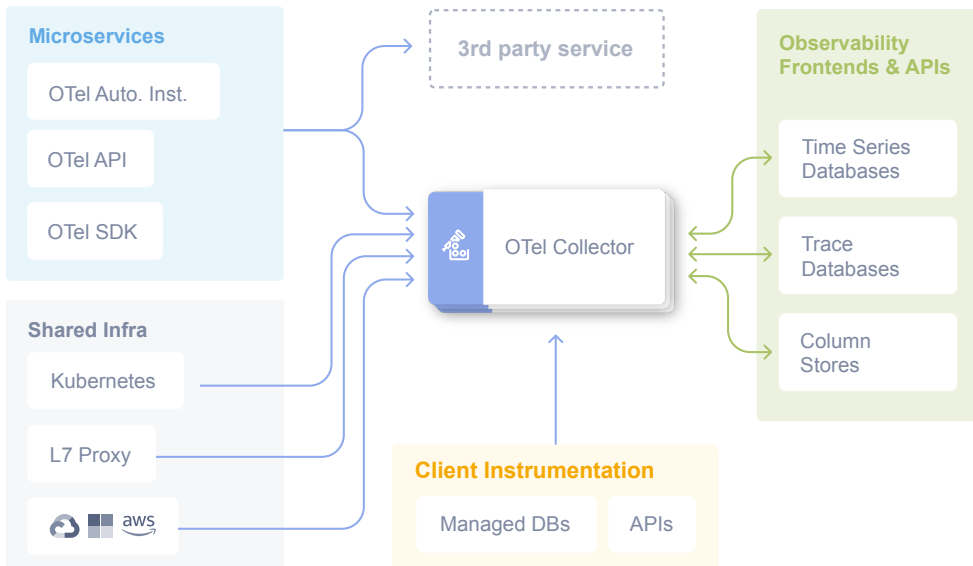
OpenTelemetry tools

Python auto-instrumentation: Flask, Django, and other supported libraries.

Java agent: similar coverage for Java.

Installation and export configuration are still required.

OpenTelemetry collects and exports signals



OpenTelemetry Authors · CC BY 4.0

Exporter
Sends signals from an SDK or Collector to a destination.

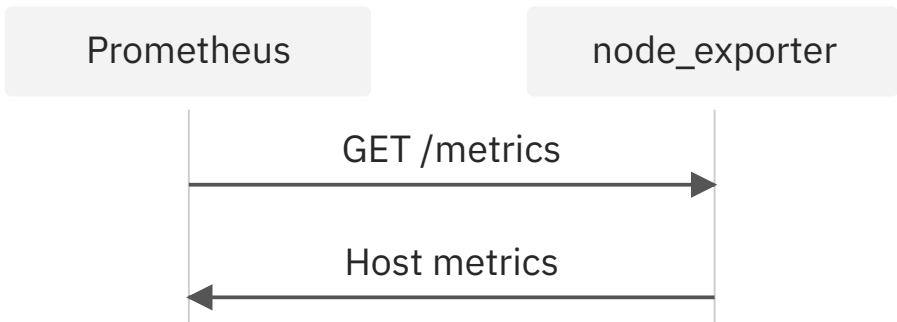
Collector
Receives, processes, and forwards signals.

Backend
Stores signals and answers queries; an interface displays results.

Applications can also export directly to a backend.

Prometheus pulls metrics by scraping an endpoint

Pull: Prometheus starts the exchange



Prometheus requests and stores metrics at each scrape interval.

Push: the application sends a batch

Application → OTel Collector

An **OTLP exporter** sends batches of telemetry to the Collector.

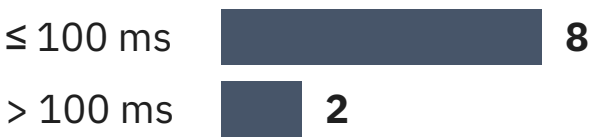
A **Prometheus exporter** exposes an endpoint for scraping, as node_exporter does for host metrics.

Do we need to keep every trace?

Aggregation

Summarize **all recorded observations**; lose individual detail.

Duration histogram · all 10 recorded



Distribution without request detail

Sampling

Keep **a subset of individual records**; discard the rest.

Retained traces · 3 of 10 requests

R2 ✓ R9 × R10 ×

Detail for selected requests only

> ⚠ **Sampling can bias the evidence.**
> Failures: 2/3 of retained traces, but 2/10 overall.

Sampling chooses which traces survive

Head — **Decide at start** → Record selected requests → Export

Example policy: keep 10% · final outcome is not yet known

Lower recording and transport cost; may miss rare failures.

Tail — Collect spans → **Buffer and decide** → Retain selected

Example policy: keep errors and slow requests · use available outcomes

Select by outcome; collect and buffer before discarding.

⚠ **Tail sampling cannot recover traces discarded upstream.**

⚠ Emitted telemetry may never reach storage

At the source

Log filters exclude events. Gauges can miss brief peaks between collection times.

In transit

A backend outage can fill a collector queue and cause drops.

In storage

Queries can only return the evidence that arrived and remains available.

Application emits → **Queue fills → drops** ✕ **Backend unavailable**

More detail costs more to retain

Choice	Benefit	Cost
More metric attribute combinations	Compare more contexts	More time series
More logs and spans	Inspect more requests	More transport, storage, and search
Longer retention	Investigate older incidents	More stored history

Attribute combinations multiply time series

Cardinality counts distinct attribute combinations for a metric.
Each combination identifies a separate series of measurements over time.

Example: request count, labeled by `method` and `route`.

2 methods × 3 routes → up to 6 series

	`/cart`	`/pay`	`/users`
GET	1 series	1 series	1 series
POST	1 series	1 series	1 series

Add 10 instances → up to 60 series



The same six combinations per instance.

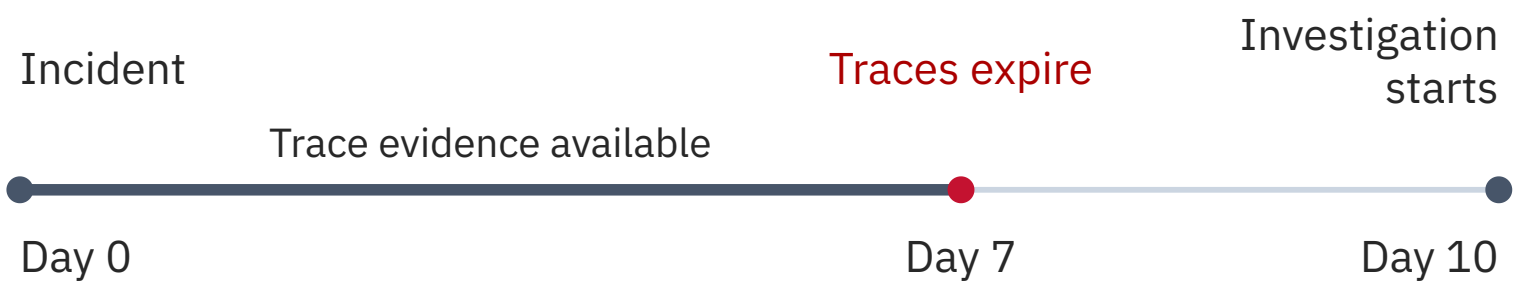
> **With a unique request_id, combinations grow with traffic.**
> More series means more data to maintain, store, and query.

Use bounded attribute values; keep request IDs in logs and traces.

Retention sets the investigation window

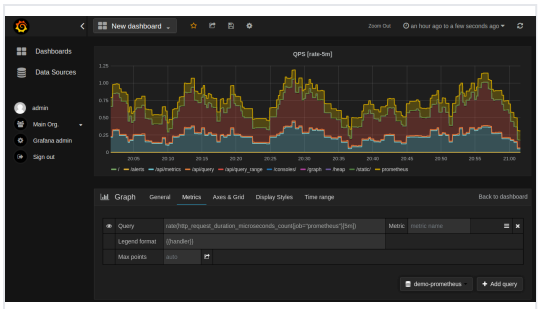
Each signal has its own retention window. Once records expire, that evidence is gone.

Example policy: traces retained for 7 days; metrics retained longer.



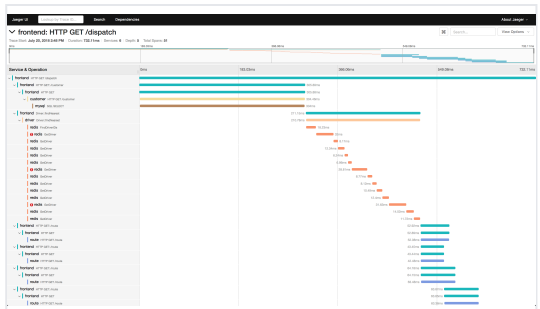
Metrics may remain; the expired traces are gone.

Exploring collected signals



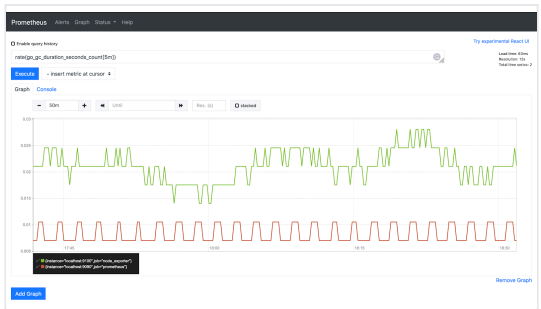
Grafana

Explore metrics, logs, and traces from connected backends.



Jaeger

Find traces and inspect the operations within a request.



Prometheus

Query metrics and graph how they change over time.

🚀 Today, we learned how to ...

- Explain the roles of logs, metrics, and traces in understanding a running system.
- Compare structured and unstructured logs and choose fields that make events easy to search and connect.
- Interpret counters, gauges, rates, and latency distributions, explaining how labels and aggregation affect what metrics reveal.
- Read a distributed trace and explain how spans and propagated context connect services.
- Choose a signal for a question about application behavior and explain its limitations.