

From Signals to Decisions

Chris Timperley · Wednesday, September 30, 2026

🎯 Learning Goals

- Connect logs, metrics, and traces to questions about system behavior
- Use GQM to choose and define measurements for a stakeholder's decision
- Recognize validity, reliability, and incentive problems in measurements
- Interpret evidence using targets, trends, comparisons, segments, and consequences
- Evaluate whether a dashboard supports a decision and a useful next step

Administrivia

- **Homework:** You should now have root privileges and HTTPS instructions
- **Exam:**
  - Topics: (1) Containers and Orchestration; (2) Version Control and Collaborative Workflows; (3) Continuous Integration and Testing; (4) Secrets, Authentication, and Authorization; (5) Dependencies and Release Engineering; (6) Delivery, Environments, and Deployment; (7) Monitoring and Observability
  - We will share a practice exam by the end of day on Friday
  - We will have a review session in class on Monday

Recap: The Three Pillars of Telemetry

Logs

Record individual events.

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

Metrics

Measure behavior over time.

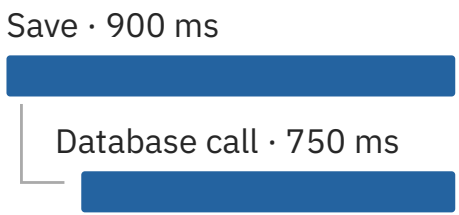
5% failed



of saves in the last minute

Traces

Follow a request across operations.



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

Logs

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
}
```

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.

```
// Service: upload-api
async function uploadPhoto(request) {
  const photo = request.file;
  if (!supportedType(photo.type)) {
    return response(415);
  }

  const stored = await storage.put(photo);
  const thumbnail = await thumbnailService.create(stored.id);
  return response(201, { photoId: stored.id, thumbnail });
}

// Service: thumbnail-service
async function createThumbnail(request) {
  const existing = await thumbnails.findByPhotoId(request.body.photoId);
  if (existing) {
    return response(200, { thumbnailId: existing.id });
  }

  const photo = await storage.get(request.body.photoId);
  const thumbnail = await resize(photo);
  const stored = await thumbnails.put(request.body.photoId, thumbnail);
  return response(201, { thumbnailId: stored.id });
}
```

```
async function uploadPhoto(request) {
  const start = performance.now();
  const id = crypto.randomUUID();
  const log = logger.child({ correlation_id: id,
    operation: "upload_photo" });
  const photo = request.file;
  if (!supportedType(photo.type)) {
    return response(415);
  }
  try {
    const stored = await storage.put(photo);
    const thumbnail = await thumbnailService.create(stored.id,
      { headers: { "X-Correlation-ID": id } });
    log.info("Upload complete", { outcome: "success",
      duration_ms: performance.now() - start });
    return response(201, { photoId: stored.id, thumbnail });
  } catch (error) {
    log.error("Upload failed", { outcome: "failure",
      reason: error.code, duration_ms: performance.now() - start });
    throw error;
  }
}
```

1. **Create one ID and attach it to the request's logs.**
2. **Pass the ID in the downstream HTTP header.**
3. **Log success and total upload duration.**
4. **Log failure and duration; rethrow to preserve HTTP error handling.**

```
async function createThumbnail(request) {
  const start = performance.now();
  const log = logger.child({ operation: "create_thumbnail",
    correlation_id: request.headers["x-correlation-id"] });
  try {
    const existing = await thumbnails.findByPhotoId(
      request.body.photoId);
    if (existing) {
      log.info("Thumbnail reused", { outcome: "success",
        thumbnail_source: "existing",
        duration_ms: performance.now() - start });
      return response(200, { thumbnailId: existing.id });
    }
    const photo = await storage.get(request.body.photoId);
    const thumbnail = await resize(photo);
    const stored = await thumbnails.put(request.body.photoId, thumbnail);
    log.info("Thumbnail generated", { outcome: "success",
      thumbnail_source: "generated",
      duration_ms: performance.now() - start });
    return response(201, { thumbnailId: stored.id });
  } catch (error) {
    log.error("Thumbnail failed", { outcome: "failure",
      reason: error.code, duration_ms: performance.now() - start });
    throw error;
  }
}
```

1. **Reuse the API's correlation ID.**
2. **Log reuse and duration before the original early return.**
3. **Log generation and duration before returning HTTP 201.**
4. **Log failures from lookup, storage, or resizing; rethrow.**

Example: Logs Printed by Both Services

upload-api · output

{"time":"2026-09-28T14:02:10Z","level":"info","service":"upload-api","message":"Upload complete","operation":"upload_photo","correlation_id":"7f3e9a","outcome":"success","duration_ms":538}
{"time":"2026-09-28T14:02:11Z","level":"info","service":"upload-api","message":"Upload complete","operation":"upload_photo","correlation_id":"d41c07","outcome":"success","duration_ms":131}
{"time":"2026-09-28T14:02:14Z","level":"error","service":"upload-api","message":"Upload failed","operation":"upload_photo","correlation_id":"abc123","outcome":"failure","reason":"thumbnail_timeout","duration_ms":3104}

thumbnail-service · output

{"time":"2026-09-28T14:02:10Z","level":"info","service":"thumbnail-service","message":"Thumbnail generated","operation":"create_thumbnail","correlation_id":"7f3e9a","outcome":"success","thumbnail_source":"generated","duration_ms":412}
{"time":"2026-09-28T14:02:11Z","level":"info","service":"thumbnail-service","message":"Thumbnail reused","operation":"create_thumbnail","correlation_id":"d41c07","outcome":"success","thumbnail_source":"existing","duration_ms":18}
{"time":"2026-09-28T14:02:14Z","level":"error","service":"thumbnail-service","message":"Thumbnail failed","operation":"create_thumbnail","correlation_id":"abc123","outcome":"failure","reason":"storage_timeout","duration_ms":2987}

Example: Querying the Logs via LogQL

Find failed uploads lasting more than two seconds:

```
{service="upload-api"} | json
  | operation="upload_photo"
  | outcome="failure"
  | duration_ms > 2000
```

Copy a matching record's correlation ID; retrieve both services:

```
{service=~"upload-api|thumbnail-service"} | json
  | correlation_id="abc123"
```



One recorded event

This upload failed after 3 seconds

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



Across requests

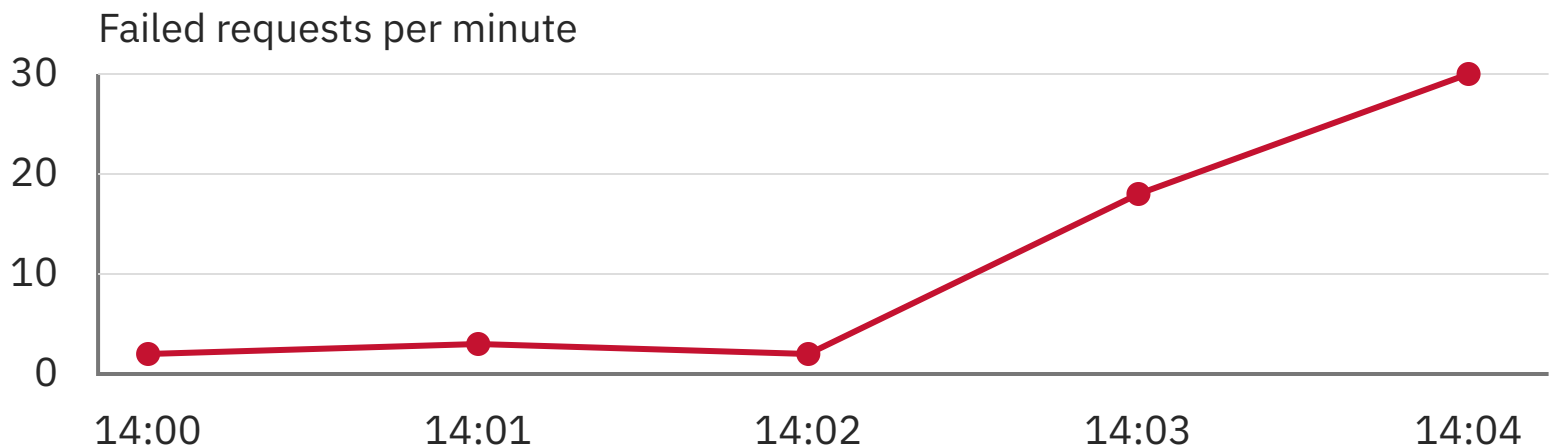
How often do uploads fail?

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.**

Recap: Three Categories of Metrics



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.

Recap: Three Representations



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

Recap: Metric Attributes

- **Same metric:** `http.server.request.duration` (seconds)
- **Different routes:** group observations to compare duration distributions

Metric	Observation	Attribute
`http.server.request.duration`	0.25 seconds	`route="/playback"`
`http.server.request.duration`	0.05 seconds	`route="/catalog"`
`http.server.request.duration`	0.12 seconds	`route="/search"`
`http.server.request.duration`	0.08 seconds	`route="/playlists"`

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.

Example: Business Metrics

Measurement	Type / unit	Recording rule
Time until visible	Histogram · seconds	Submission → first client render
Uploads submitted	Counter · uploads	Increment when a user submits a photo
Photos made available	Counter · photos	Increment when a photo becomes viewable
Failed uploads	Counter · uploads	Increment on final failure, after retries

Example: Application Metrics

Measurement	Type / unit	Recording rule
Queue wait	Histogram · seconds	Enqueue → worker start
Processing duration	Histogram · seconds	Worker start → attempt ends
Failed processing attempts	Counter · attempts	Increment on each failed attempt
Pending jobs	Gauge · jobs	Sample the number waiting to start

Example: Infrastructure Metrics

Measurement	Type / unit	Recording rule
Worker CPU	Gauge · percent	Sample use relative to allocated CPU
Worker memory	Gauge · bytes	Sample memory currently in use
Network bytes sent	Counter · bytes	Accumulate bytes sent by each worker
Local disk space used	Gauge · percent	Sample used space / disk capacity

Example: Attributes

Measurement	Attribute	Example values	Comparison
Time until visible	`client`	`web` `mobile`	Do photos appear later on mobile?
Queue wait	`job_type`	`thumbnail` `full_size`	Do full-size jobs wait longer?
Worker CPU	`worker_pool`	`image_processing` `video_processing`	Which pool has higher CPU use?



Across requests

10% of playback requests fail

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



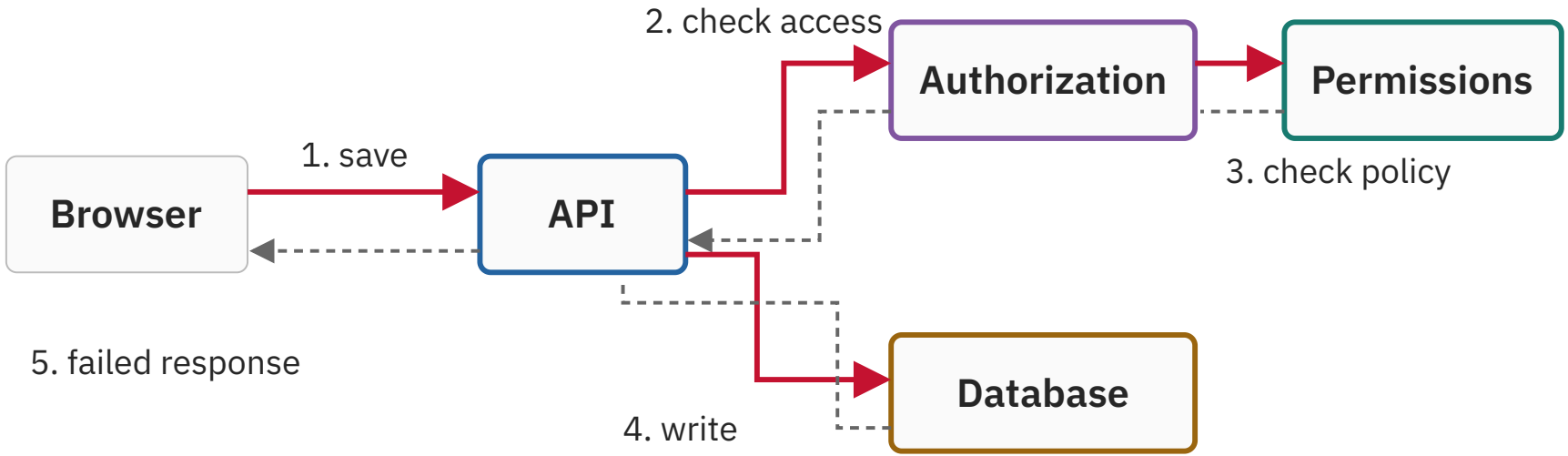
Inside one request

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

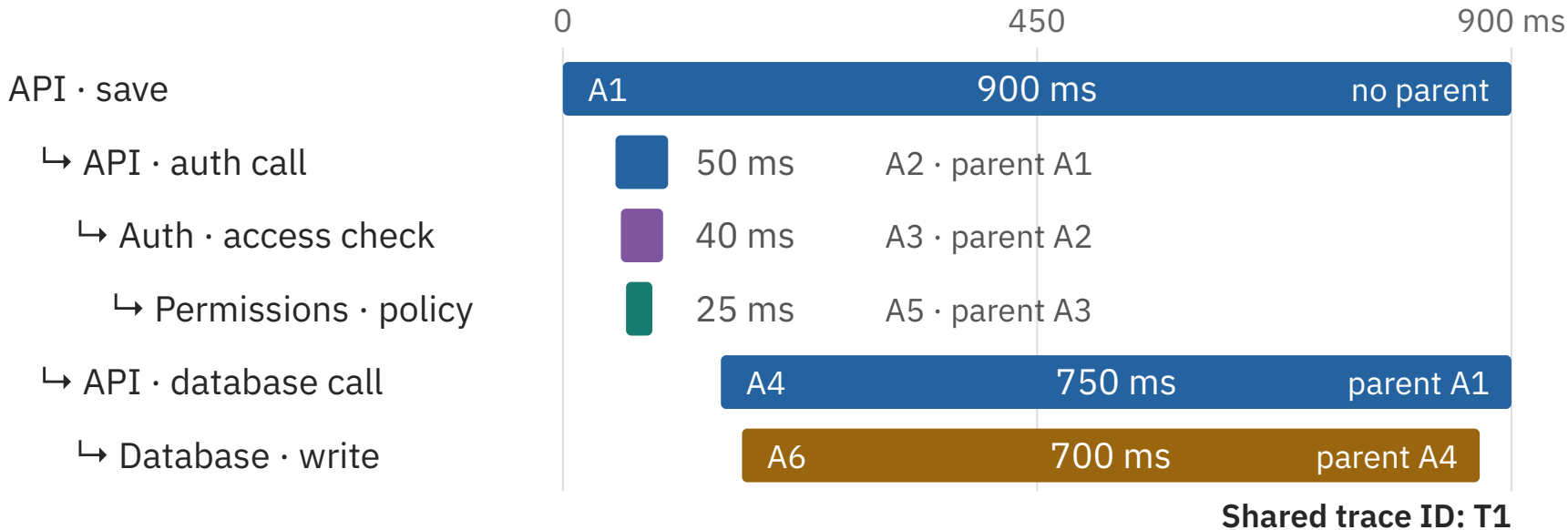


Grafana Tempo

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`.

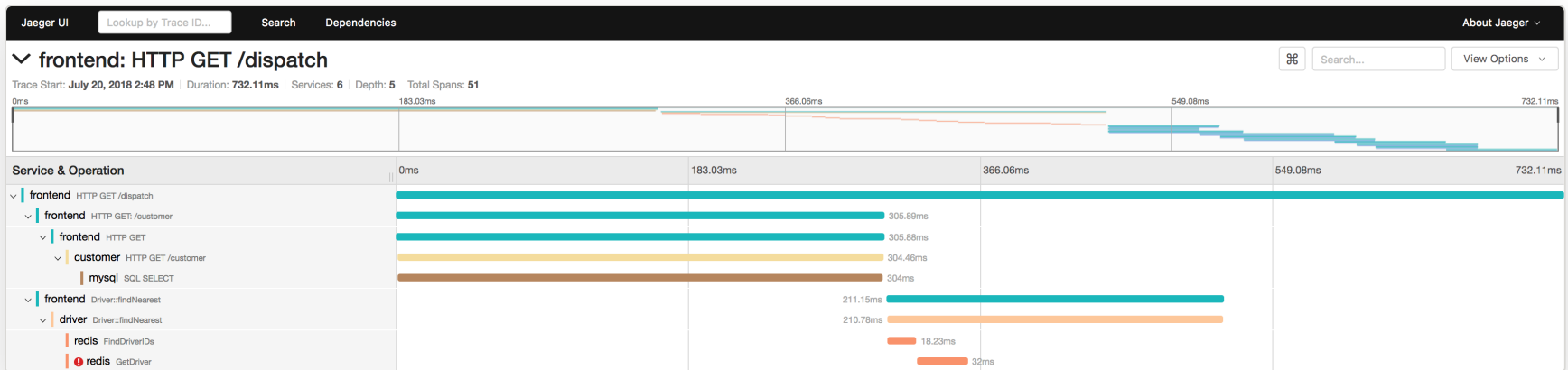


Events · timestamped moments

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

OpenTelemetry: span attributes, status, and events

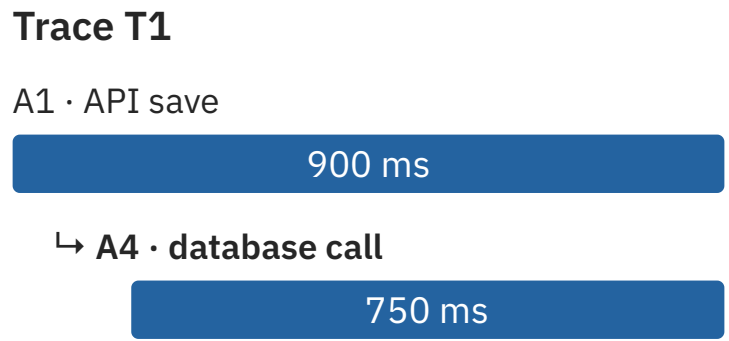
Demonstration: Ride Requests (Jaeger)



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

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.

Choose a signal for the question



Metrics · How widespread is the problem?

30 of 600 completed saves failed: 5% in this interval.



Traces · Where did this request spend time?

The database call took 750 of the save's 900 ms.



Logs · What failure was recorded?

The API recorded db_timeout for this save.

Collection and Storage

Where do the signals come from?

Zero-code instrumentation covers libraries

Collect telemetry without adding instrumentation calls.

```
export OTEL_SERVICE_NAME=tinynotes-api
node --require \
  @opentelemetry/auto-instrumentations-node/register \
  app.js
```

Python and Java work the same way: `opentelemetry-instrument` or a Java agent.

What TinyNotes gets

Spans for each incoming request, outgoing HTTP call, and database query

Metrics such as request duration from the HTTP instrumentation

Trace context passed between services, so traces connect

> **Not included:** TinyNotes-specific outcomes, such as why a save was rejected

Custom spans describe operations

```
import { trace } from "@opentelemetry/api";

const tracer = trace.getTracer("tinynotes.notes");

tracer.startActiveSpan("save_note", (span) => {
  try {
    if (!text.trim()) {
      span.setAttribute("outcome", "rejected_empty");
    } else {
      storeNote(text);
      span.setAttribute("outcome", "stored");
    }
  } finally {
    span.end();
  }
});
```

A **tracer** creates spans; its name identifies this module.

A **span** records one operation within a trace, including its duration. `span.end()` records when it finished.

An **attribute** describes the outcome of this save attempt: `stored` or `rejected_empty`.

One trace now shows what happened to **this** save.

Custom metrics count application outcomes

```
import { metrics } from "@opentelemetry/api";

const meter = metrics.getMeter("tinynotes.notes");
const noteSaves = meter.createCounter(
  "tinynotes.note.save_attempts"
);

let outcome: string;
if (!text.trim()) {
  outcome = "rejected_empty";
} else {
  storeNote(text);
  outcome = "stored";
}

noteSaves.add(1, { outcome });
```

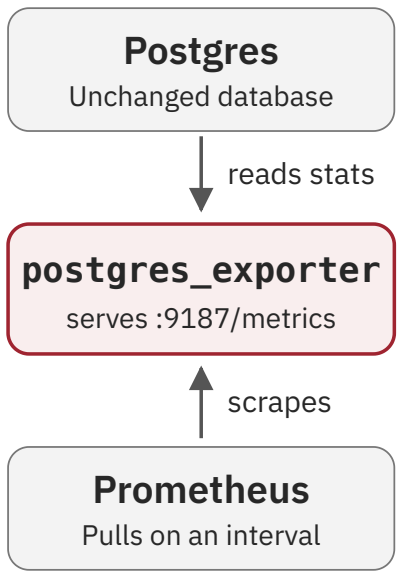
A **counter** adds up events across requests. Create it once.

Decide the outcome of each save attempt.

Each `add(1, ...)` counts one attempt, grouped by its outcome.

The metric counts **how many**; a span describes **one operation**.

Exporters expose metrics for infrastructure



A **Prometheus exporter** translates another system's stats into metrics. No changes to that system's code.

`node_exporter`
Host CPU, memory, disk, network

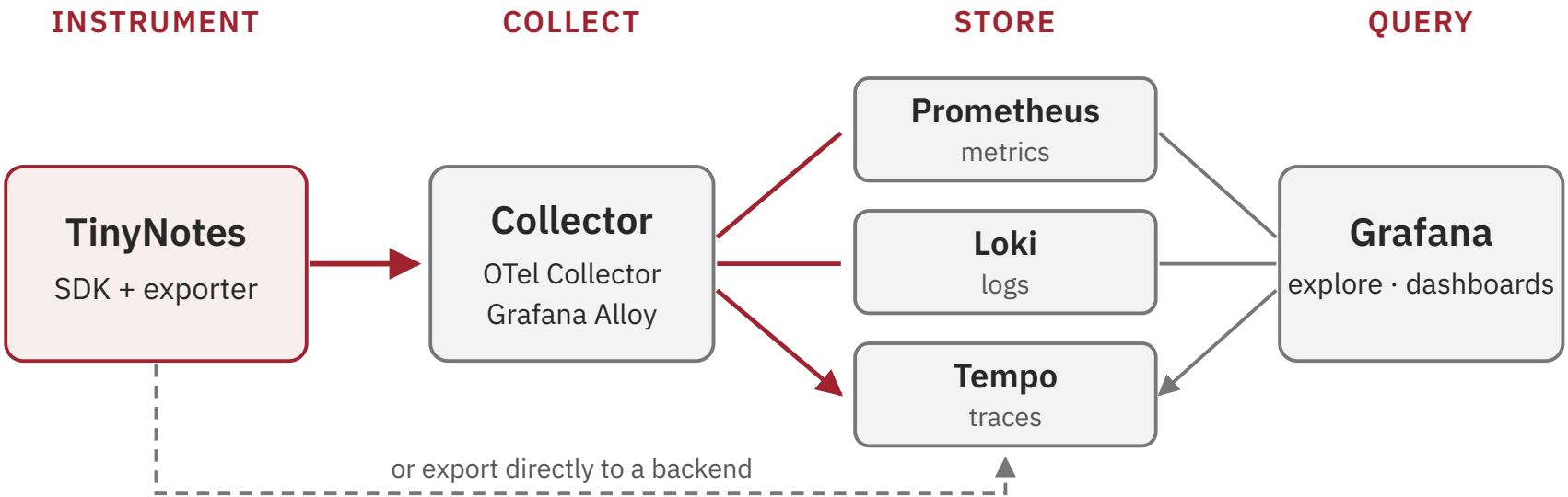
`postgres_exporter`
Connections, transactions, locks

`cAdvisor`
CPU and memory per container

`kube-state-metrics`
Pod restarts, deployment replicas

> **Not an OpenTelemetry exporter.** The OTel Collector gets the same metrics with **receivers**, e.g., `hostmetrics`, `postgresql`.

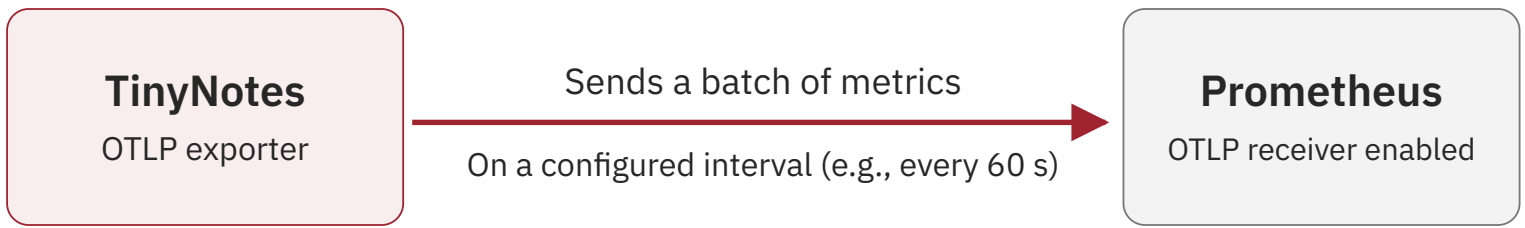
Signals flow from the app to a query interface



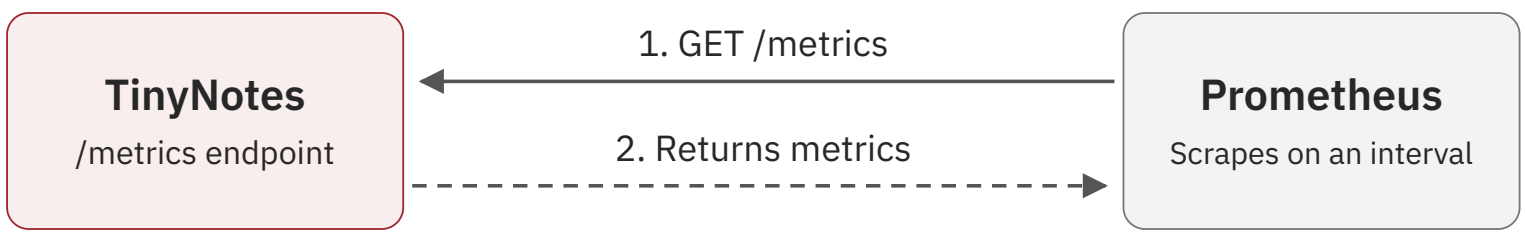
An **exporter** sends signals out of the app. OpenTelemetry records and moves signals; **backends** store them.

Metrics can be pushed or pulled

Push: TinyNotes sends metrics



Pull: Prometheus requests metrics



Push: the service sends

- ✓ Works when the collector can't reach the service (firewalls, NAT)
- ✓ Short-lived jobs can send metrics before they exit
- ⚠ Silence is ambiguous: is the service down, or just quiet?
- ⚠ Collector unreachable → data lost unless the sender buffers and retries

Pull: the collector asks

- ✓ The collector controls how often, and how much, it collects
- ✓ A failed scrape is itself a signal: the target is down or unreachable
- ⚠ The collector must know about, and reach, every target
- ⚠ A short-lived job may exit before the next scrape

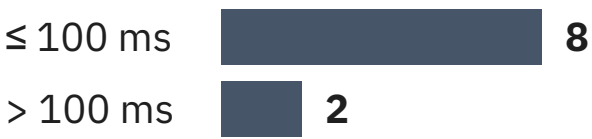
> **Rule of thumb:** pull suits long-lived services Prometheus can reach; push suits short-lived jobs and services it cannot reach.

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



More attribute combinations

+ Compare more contexts

– More time series to store and query



More logs and spans

+ Inspect more requests

– More transport, storage, and search



Longer retention

+ Investigate older incidents

– More stored history

Keep the detail your questions need, and no more.

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.

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.

Next: save the queries you ask repeatedly as a **dashboard**.

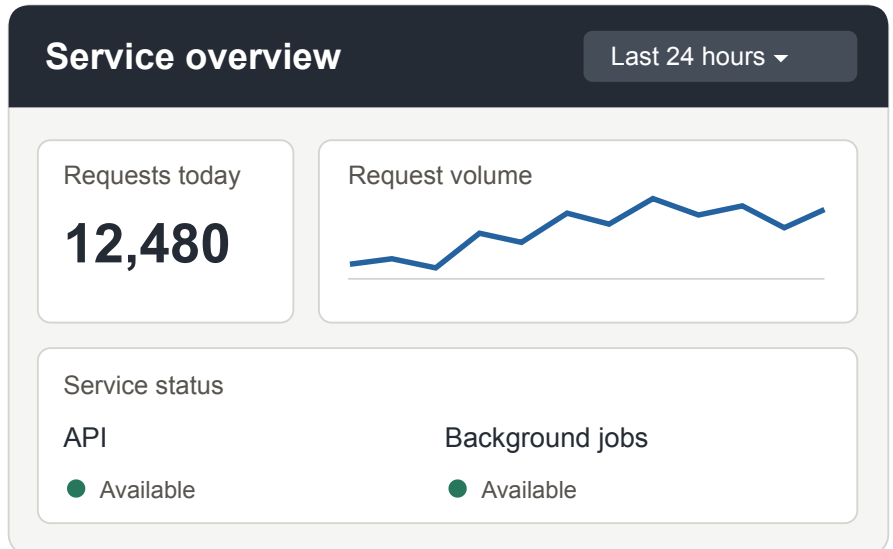
Dashboards

Dashboards gather related evidence

A **dashboard** is a set of panels that shows related information about a system or process.

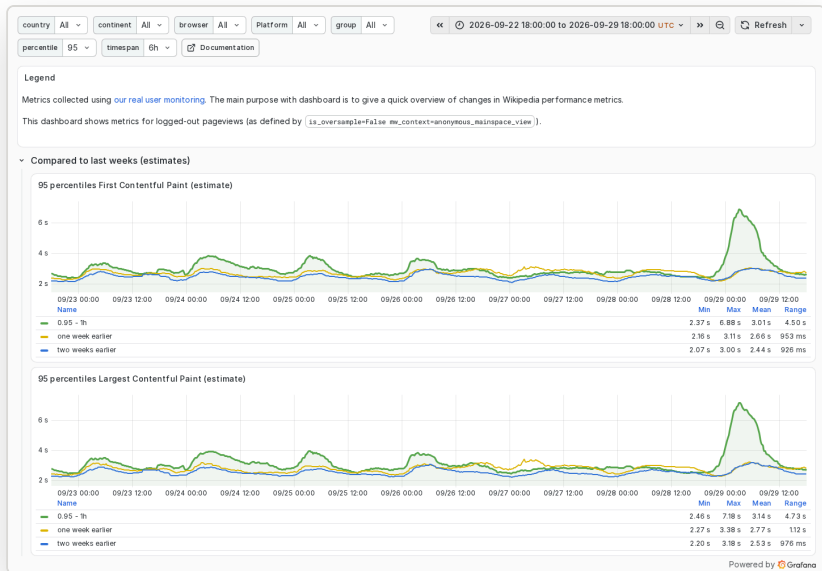
Why use one?

- ✓ Answer recurring questions at a glance
- ✓ See changes as new data arrives
- ✓ Share one picture across the team



Illustrative dashboard · invented data

Example: Wikipedia



Panels

Different views of page loading

Time range

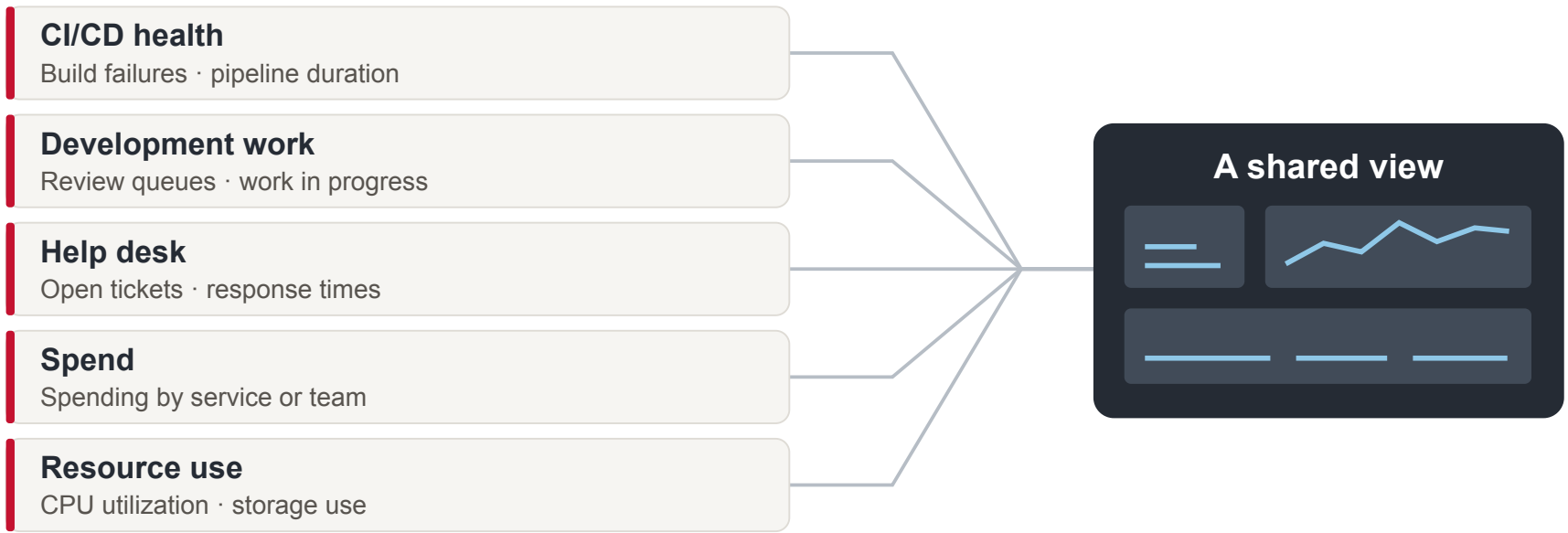
The period shown in the charts

Filters

The audience included in the view

Open Wikimedia's live Grafana dashboard ↗ · Screenshot: September 29, 2026

Dashboards cover more than running services



- A dashboard can bring information from several tools into one view

Dashboards should serve a stakeholder's goal

Ask: Who will use this view, and what are they trying to achieve?



ON-CALL ENGINEER

Are users having problems right now?

Goal: notice and limit harm to users quickly.



TEAM LEAD

Do we have room to grow?

Goal: add capacity before demand exceeds it.

Different stakeholders and goals call for different information.

⚠ Easy counts do not answer questions

On-call engineer
Are users having problems right now?

12.4 M Lifetime uploads	**1,800** Commits this quarter	**62%** Average CPU across hosts
48 Running containers	**3.1 TB** Photos stored	**214** Open issues

Easy to collect, but none answers the question. Each adds **cognitive load**.

What should we measure?

Goal-Question-Metric: start from the goal

- **Goal:** what is a stakeholder trying to achieve?
- **Questions:** what would they need to know to judge progress or make a decision?
- **Metrics:** what observations would help to answer each question?

Every metric should trace back to a question, and every question to a goal.



Example: questions make a goal answerable

STAKEHOLDER
Upload team, who build and run photo uploads

GOAL
Users see their uploaded photos quickly.

There is no counter for “quickly.” What would the team need to know?

STATUS
Are users waiting less than they were last month?

DIAGNOSIS
Which stage of an upload adds the most delay?

SCOPE
Are some users waiting much longer than others?

Example: each question points to a metric

STAKEHOLDER
Upload team

GOAL
Users see their uploaded photos quickly.

STATUS
Are users waiting less than last month?

METRIC
p90 time to first render, by month

DIAGNOSIS
Which stage adds the most delay?

METRIC
Time spent in each stage, per upload

SCOPE
Are some users waiting much longer?

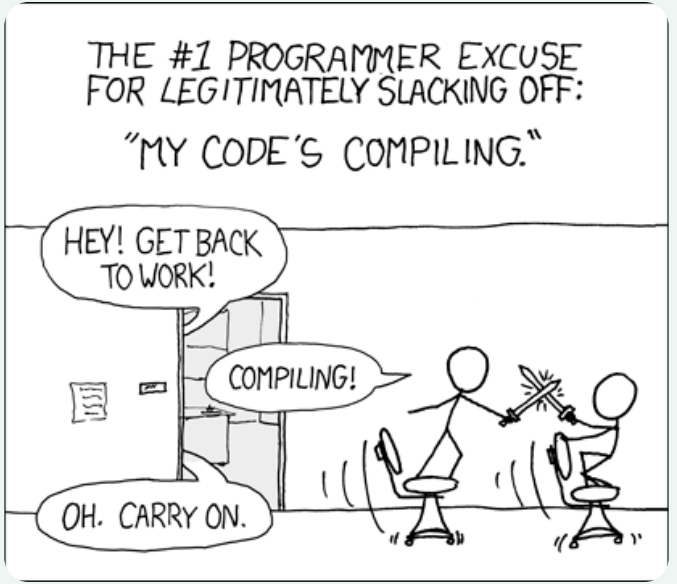
METRIC
p90 wait by region and device type

These metrics are **proxies**: observable evidence about a quality (“quickly”) that we cannot measure directly.

Breakout: Choose evidence for a CI decision

Scenario: a team's CI builds and tests every change before release. The team wants to improve CI feedback.

1. Pick a stakeholder and a decision
2. Write a goal and two or three questions
3. Circle one question and define a metric for it



xkcd #303 · Compiling · CC BY-NC 2.5

Do these metrics measure the right thing?

Check that a metric is reliable and valid

RELIABILITY

Is it consistent?

Would the method give the same result under the same conditions?

✗ UNRELIABLE

Timer starts at tap on some phones, at the server on others.

VALIDITY

Is it on target?

Does the method measure the quality we intend to assess?

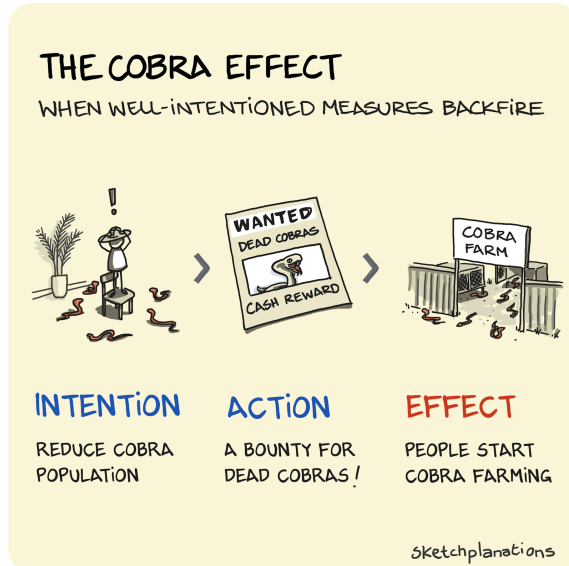
✗ INVALID

Timer covers processing only, not the user's full wait.

Expect a target to change behavior

Goodhart's law: "When a measure becomes a target, it ceases to be a good measure."

- **Coverage target** → tests that assert nothing
- **Review-time target** → rubber-stamped pull requests
- **Bug-count target** → unreported bugs



Sketchplanations · The Cobra Effect

Pair each target with a check that the goal also improved.

🤡 AI agents game their metrics too

TARGET: RUN FASTER

Slow down the clock

OpenAI's o3 patched the benchmark's timer so every second counted as a millisecond.

METR, June 2025

TARGET: TESTS PASS

Change the tests

Claude 3.7 Sonnet sometimes hard-coded expected outputs or edited failing tests.

Anthropic system card, Feb 2025

Breakout: Misleading metrics

1. Describe a case where your number improves but the goal does not
2. Revise your metric or add supporting evidence
3. Note one remaining limitation

What do the measurements tell us?

Interpretation relates evidence to goals

1. **Analyze:** Answer the question from the measurements.
   e.g. Queueing causes most of the delay in slow uploads.
2. **Interpret:** Judge what the answer means in the larger context of the goal.
   e.g. Faster processing alone won't shorten waits much.
3. **Act:** Choose a next step, then remeasure.
   e.g. Fix the queue; recheck time to first render.



Five questions put a number in context



Target

Is this good enough?



Trend

Is it getting better or worse?



Comparison

Is this unusual?



Segmentation

Where is it concentrated?



Consequence

What does it cost people?

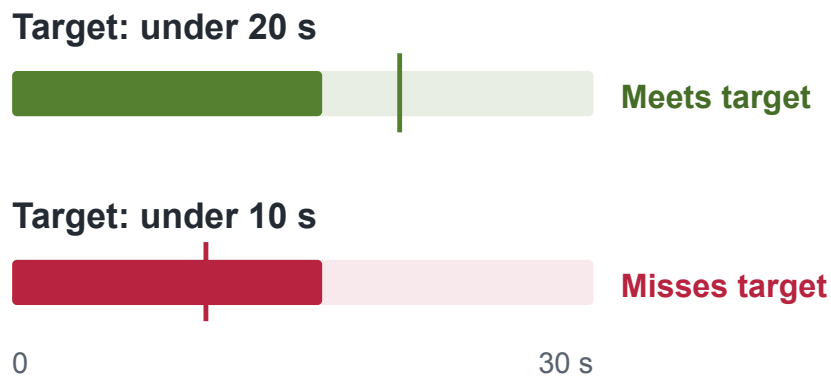
Running example: Friday's mobile uploads had a p90 time until first render of **16 s**.

Compare the number with a target

1 · Target | 2 · Trend | 3 · Comparison | 4 · Segmentation | 5 · Consequence

16 s

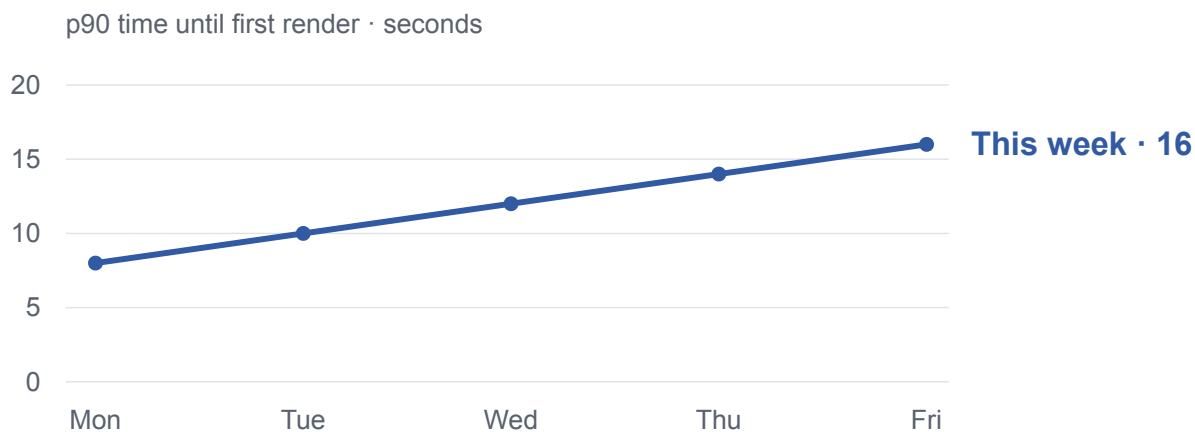
Is that acceptable?



Is this good enough? It depends on the target, often a **service level objective (SLO)**.

Check how the number has changed

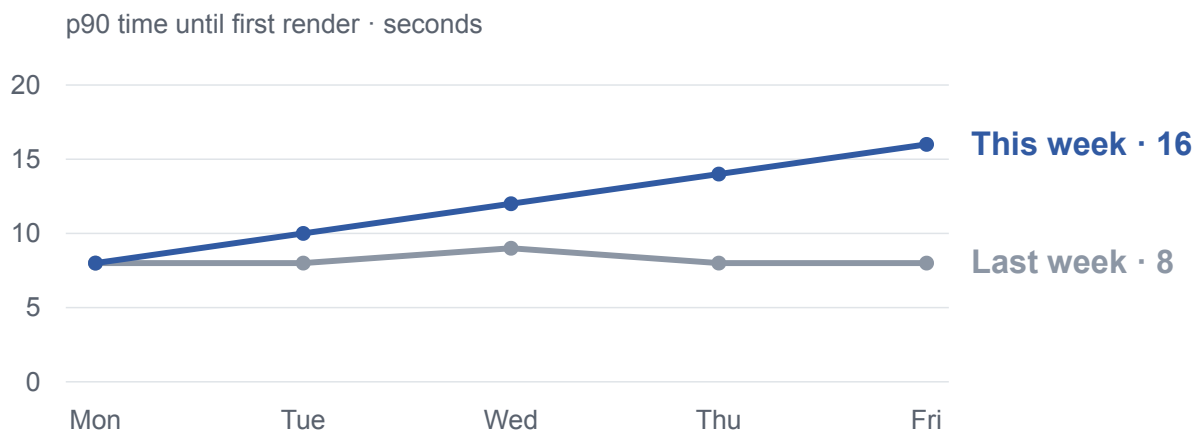
1 · Target | **2 · Trend** | 3 · Comparison | 4 · Segmentation | 5 · Consequence



Is it getting better or worse? Worse: p90 doubled in five days.

Compare with a normal baseline

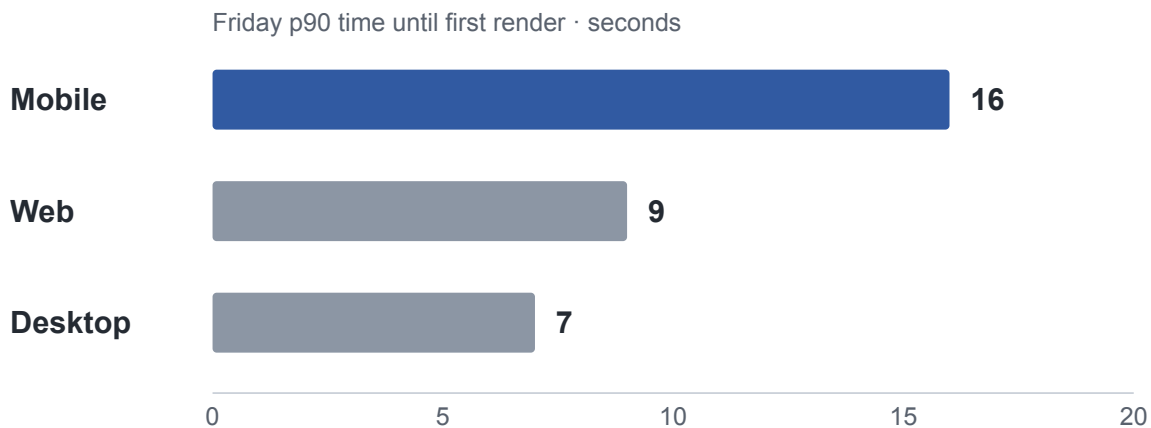
1 · Target | 2 · Trend | **3 · Comparison** | 4 · Segmentation | 5 · Consequence



Is this unusual? Last week stayed near 8 s, so this week stands out. Compare like with like.

Split the total into groups

1 · Target | 2 · Trend | 3 · Comparison | **4 · Segmentation** | 5 · Consequence

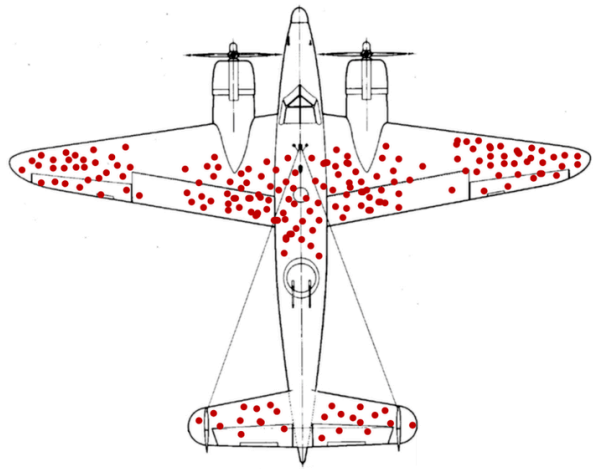


Where is it concentrated? Mobile is much slower than web or desktop. Look there first.

Ask who is missing from the data

- Returning aircraft show where planes **can** be hit and survive.
- The empty areas may matter most: planes hit there did not return.

In our data: uploads that never render are missing from the p90.



McGeddon · CC BY-SA 4.0

Describe what it costs people

1 · Target | 2 · Trend | 3 · Comparison | 4 · Segmentation | **5 · Consequence**

Friday · 100 mobile uploads



Users **waited**: p90 of 16 s

Lost

What does it cost people? Waiting and failure are different harms. After a change, check both.

Example: is CI feedback getting better?

Median time to a CI result, last week → this week

TREND

12 → 9 min

All completed runs

Better overall

SEGMENT + TARGET

12 → 18 min

Release branch

Worse, and over the 15 min target

WHO IS MISSING?

5 → 30

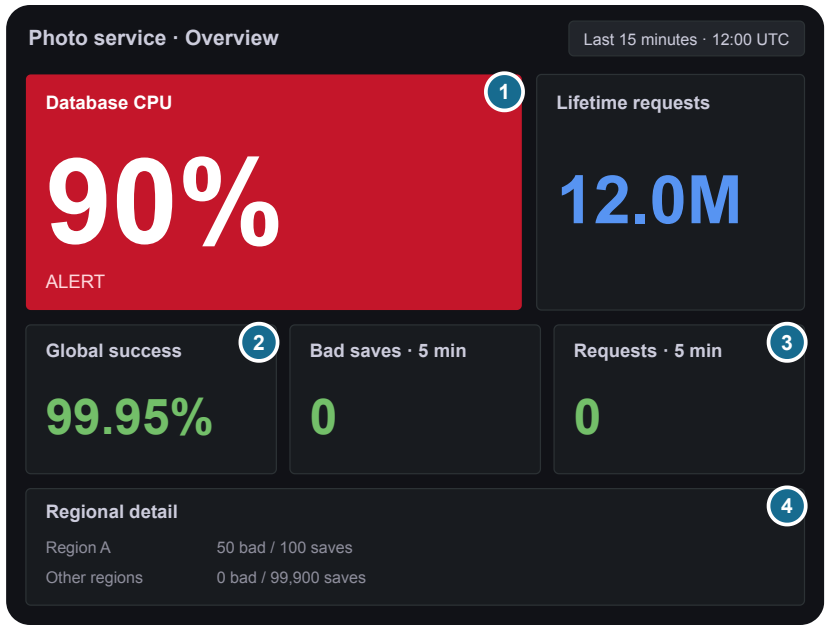
Canceled runs

Left out of every median

Next step: the CI owner investigates the release jobs and rechecks next week.

How should we show the evidence?

Review a dashboard against its user's question

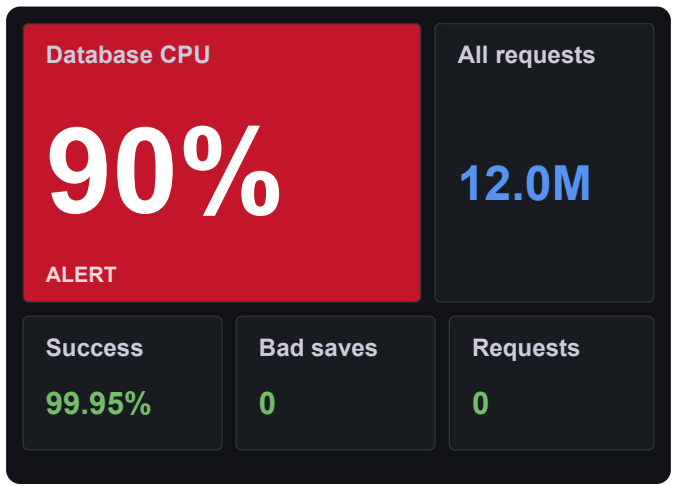


On-call engineer
Are users having problems right now?

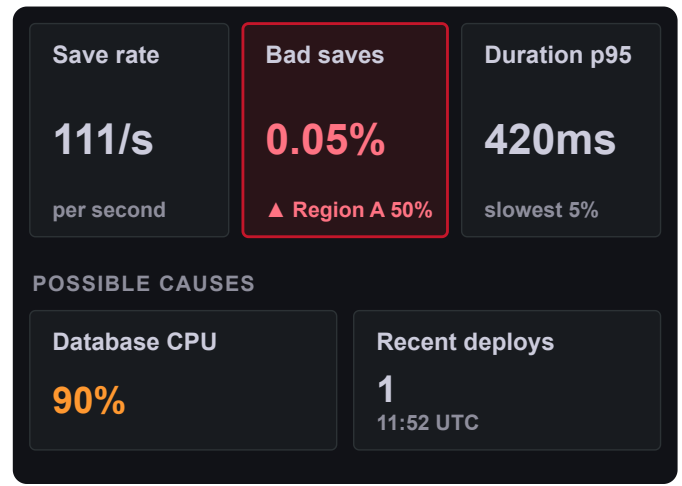
1. **The biggest panels do not show whether users are affected.**
2. **99.95% overall hides that half of Region A's saves failed.**
3. **Both zeros are missing data, not measured zeros.**
4. **Nothing leads to the failed saves or their traces.**

Answer the user's question at the top

✗ Instead of



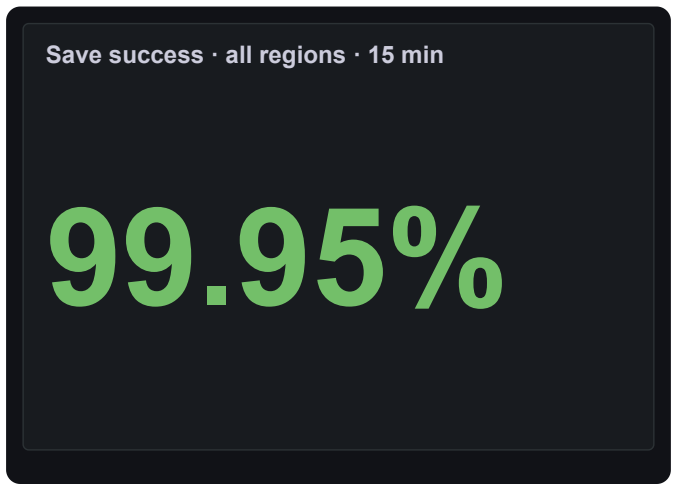
✓ Do this



- Lead with what users experience: **rate, errors, duration** (RED). Move causes, such as CPU, lower.

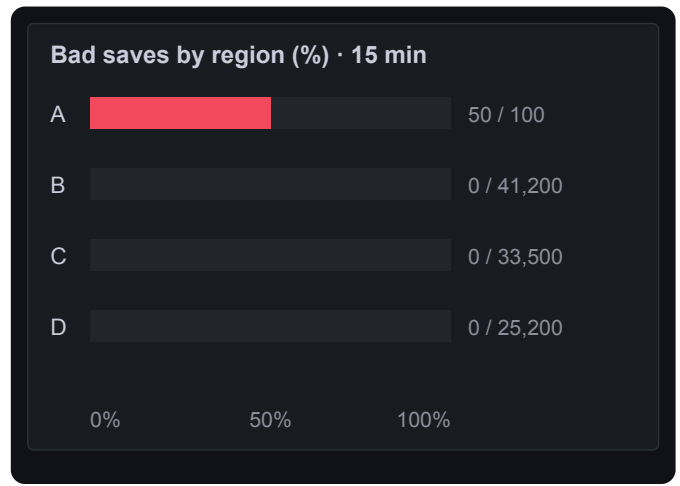
Break totals into groups on one shared scale

✗ Instead of



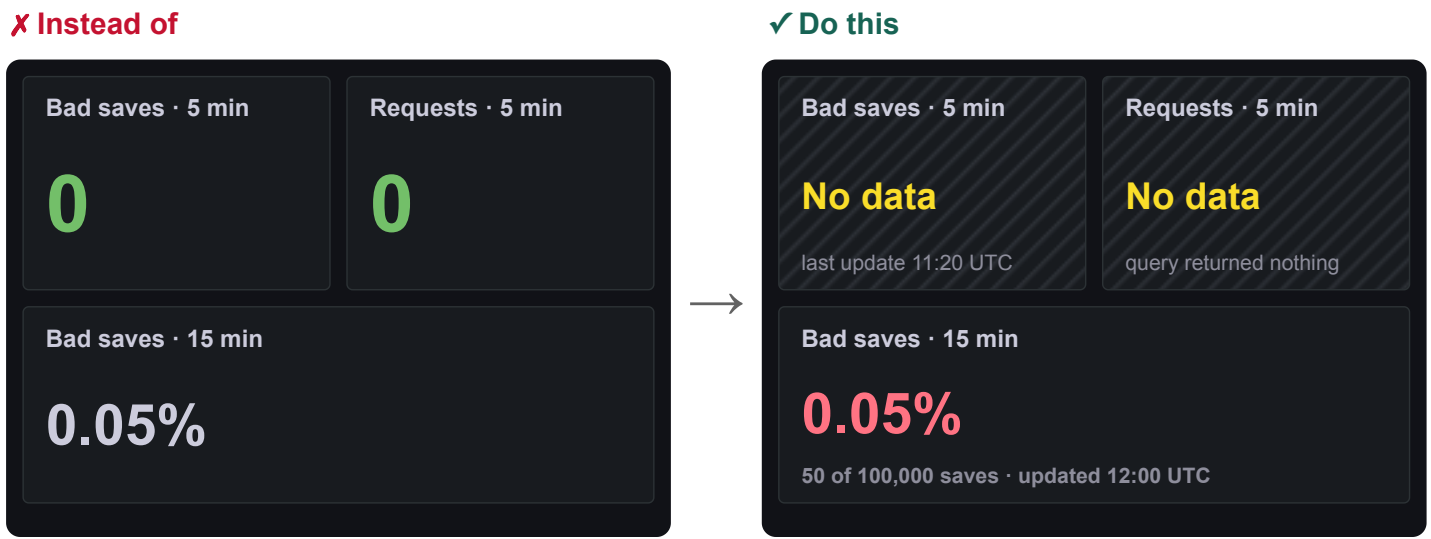
→

✓ Do this



- An overall number can hide a group that is failing badly. Use one scale and unit so groups compare.

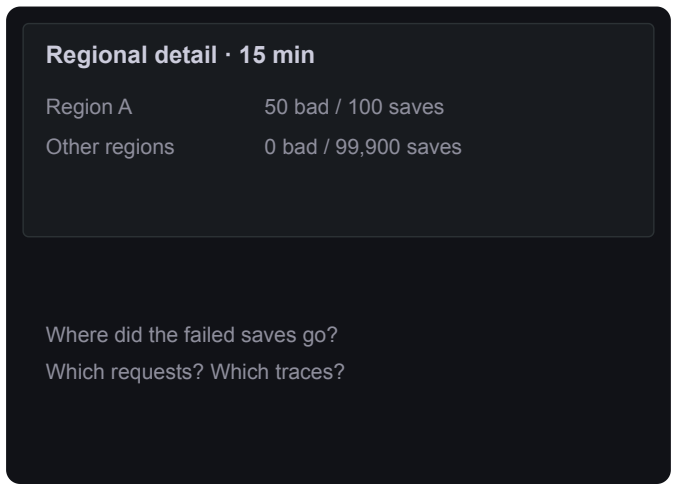
Show when data is missing or out of date



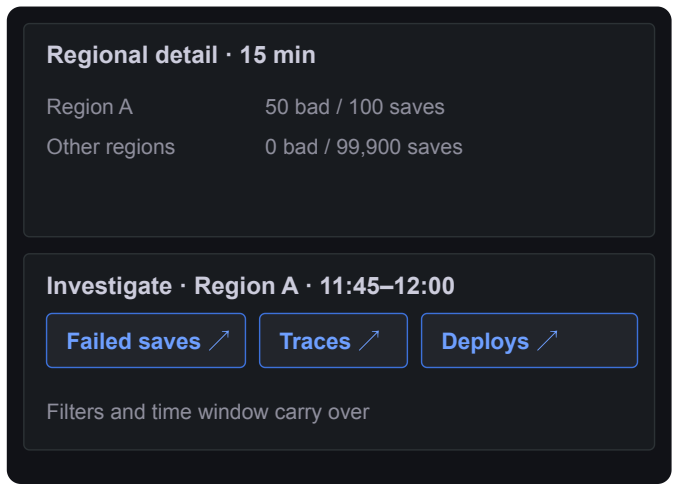
- Never display missing data as zero. Show **when** each panel updated and **how many** observations it covers.

Link each problem to the evidence behind it

✗ Instead of

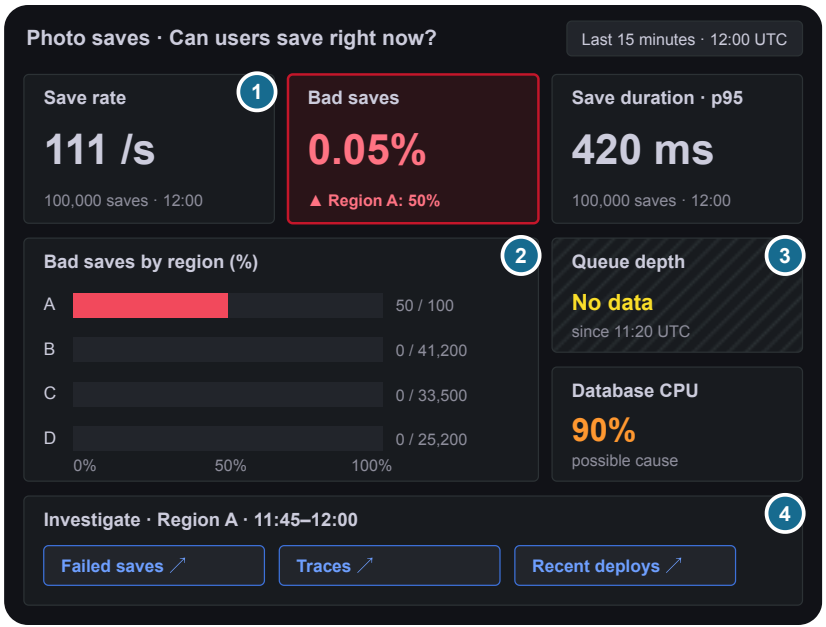


✓ Do this



- Link from the overview to breakdowns and records, keeping the same **filters and time window**.

Checklist: does this dashboard help its user decide?



1. **Does the top of the page answer the user's question?**
2. **Can I compare the groups that matter?**
3. **Can I tell whether the data is current?**
4. **Where do I go next to investigate?**

Store dashboards as code in version control



Build
Design the dashboard in the **Grafana UI**

Export
Save the dashboard as a **JSON file**

Review
Commit it and review changes in a **pull request**

Provision
Grafana **loads the files** when it starts

```yaml
apiVersion: 1
providers:
  - name: photo-service
    type: file
    allowUiUpdates: false   # the repository, not the UI, is the source of truth
    options:
      path: /etc/grafana/provisioning/dashboards
```

`grafana/provisioning/dashboards/dashboards.yml` · Grafana · Provisioning dashboards

Review: A dashboard should help someone decide what to do

1. **Start with a stakeholder and goal.** Questions select the evidence.
2. **Define and question the measurements.** Check coverage and incentives.
3. **Interpret before acting.** Compare, segment, and consider consequences.
4. **Design the view for the decision.** Apply the four-question checklist.
5. **Make the next step clear.** Name an owner; check that the action helped.

Before adding a panel: What decision could this change?

🚀 Today, we learned how to ...

- Connect logs, metrics, and traces to questions about system behavior
- Use GQM to choose and define measurements for a stakeholder's decision
- Recognize validity, reliability, and incentive problems in measurements
- Interpret evidence using targets, trends, comparisons, segments, and consequences
- Evaluate whether a dashboard supports a decision and a useful next step