Observability

A Complete 11-Chapter Metrics, Logs & Traces Course

Topics covered:

The three pillars & the Prometheus data model · Scraping & PromQL

Grafana dashboards & Alertmanager · Structured logging & Loki

Distributed tracing & OpenTelemetry · Observability in Kubernetes

Capstone: a full stack, wired together end to end

Exercises: 33 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · builds directly on the site's own Cloud Platforms and Kubernetes courses

Table of Contents

01 What Observability Actually Means — The Three Pillars

02 Metrics & the Prometheus Data Model

03 Prometheus Architecture & Scraping

04 PromQL Deep Dive

05 Grafana — Dashboards & Visualization

06 Alerting with Alertmanager

07 Structured Logging & Log Aggregation

08 Distributed Tracing

09 The OpenTelemetry Standard

10 Observability in Kubernetes

11 Capstone: Building a Full Observability Stack

CHAPTER 1 OF 11

What Observability Actually Means — The Three Pillars

Observability

Chapter 1 · What Observability Actually Means — The Three Pillars

`cloud1-8` introduced the vocabulary — metrics, logs, traces, automatic vs. configured monitoring, alert fatigue. `cloud2-4` built real incident-response discipline on top of that vocabulary — the first-five-minutes triage habit, cross-service correlation IDs, "find the earliest alert, not the loudest." Both stayed conceptual and tool-agnostic, on purpose. This course goes deep specifically on the real tooling underneath — Prometheus, Grafana, Loki, OpenTelemetry, Jaeger — the systems that actually produce the metrics, logs, and traces those two chapters assumed were already there.

Monitoring vs. Observability — A Real Distinction

Monitoring watches for failure modes you already anticipated — a dashboard for CPU usage, an alert for disk space below 10%, a check for "is the process running." It answers questions decided in advance. **Observability** is different: the ability to ask a *new* question about a system's internal state, using only its external outputs, without shipping new code to answer it. `cloud2-4`'s own "first five minutes" already implied this — during a genuinely novel incident, you usually don't know what to look for yet. Monitoring gives you the dashboards someone thought to build in advance; observability gives you the raw material to explore a question nobody anticipated.

The Three Pillars

Three distinct categories of telemetry, each answering a different kind of question:

- **Metrics** — numeric measurements over time (request rate, CPU%, P99 latency). Cheap to store, efficient to aggregate, ideal for trends and alerting — but they summarize; a metric can't tell you what happened to one specific failed request.
- **Logs** — discrete, timestamped event records, as rich and arbitrary as the code that emits them. Detailed, but expensive to store and query at real scale, and hard to correlate across services without deliberate effort — exactly the correlation-ID discipline `cloud2-4` already named.
- **Traces** — the path one specific request takes across every service it touches, with timing for each hop. The pillar built specifically to answer "where, in this whole chain, did the time actually go?"

A Concrete Walkthrough — One Request, Three Views

A checkout request is slow. Each pillar answers a different piece of the same incident:

- **Metrics say:** P99 checkout latency has climbed from 200ms to 4s over the last ten minutes — a trend, aggregated across every request, telling you something is wrong and roughly when it started.
- **Logs say:** the payment service logged three timeout errors and a retry in the last minute — detail, but only for the services that happened to log something, and only if you already know which service to look at.
- **Traces say:** for this specific slow request, 3.8 of the 4 seconds were spent waiting on a single downstream call to the inventory service — the one view that actually pinpoints where in the chain the time went, rather than just confirming that some slowdown exists.

No single pillar answers the whole question alone. Metrics tell you *something* is wrong; logs tell you *what* a given service saw; traces tell you *where*, across the whole request's path, the problem actually lives.

PILLAR	BEST ANSWERS	COST / GRANULARITY
Metrics	Is something wrong, and roughly when did it start?	Cheap, highly aggregated — no single-request detail
Logs	What did this one service observe?	Rich detail, expensive at scale, hard to correlate alone
Traces	Where, across every service in the request's path, did the time go?	Detailed per-request, but only for instrumented paths

THE REAL GOAL IS CORRELATION, NOT JUST COLLECTION

Having all three pillars isn't the point by itself — the real payoff is being able to move *between* them: a metric spike sends you to the right time window, a log line's trace ID sends you straight to the matching trace. Chapter 9's own OpenTelemetry material is specifically about wiring that correlation together, rather than treating metrics, logs, and traces as three unconnected tools.

"WE HAVE DASHBOARDS" ISN'T THE SAME CLAIM AS "WE HAVE OBSERVABILITY"

A dashboard only answers the questions someone thought to build a panel for in advance — that's monitoring, and it's genuinely useful, but it's not the same capability. Real observability means the underlying data actually supports arbitrary, unanticipated exploration — a genuinely new question, asked for the first time during an incident, needs an answer to exist in the data already, not just in whichever charts happened to get built ahead of time.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, the real difference between monitoring and observability, using cloud2-4's own "first five minutes" incident-triage habit as part of your explanation.

View solution

EXERCISE 2

For a checkout request that returns a 500 error only once every few hundred attempts, explain which of the three pillars would most directly help you find the exact failing request, and why the other two pillars alone wouldn't be enough.

View solution

EXERCISE 3

A team says "we have observability" because they have twelve Grafana dashboards covering their known failure modes. Explain, using this chapter's own distinction, why that claim is only partly true.

View solution

CHAPTER 1 QUICK REFERENCE

- **Monitoring** — watches for anticipated failure modes; **observability** — supports answering unanticipated questions from the same underlying data
- **Metrics** — cheap, aggregated numeric trends; tells you something is wrong, not which specific request
- **Logs** — rich, detailed, per-event; expensive at scale, hard to correlate without deliberate effort
- **Traces** — the path and timing of one request across every service it touches
- The real payoff is correlation between pillars, not just collecting all three independently
- This course builds the real tools (Prometheus, Grafana, Loki, OpenTelemetry, Jaeger) that cloud1-8/cloud2-4 assumed were already in place

CHAPTER 2 OF 11

Metrics & the Prometheus Data Model

Observability

Chapter 2 · Metrics & the Prometheus Data Model

`obs1-1` named metrics as the pillar that tells you *something* is wrong, cheaply, in aggregate. This chapter goes underneath that claim: exactly how Prometheus — the tool this course builds around for the metrics pillar — actually represents a metric. Everything in Ch.3 (scraping) and Ch.4 (PromQL) builds directly on the data model established here.

A Metric Is a Time Series

A Prometheus metric isn't one number — it's a named stream of `(timestamp, value)` pairs collected over time, identified by a metric name plus a set of **labels**, key-value pairs attached to that specific series.

```
http_requests_total{method="GET", status="200", handler="/checkout"} 1027
```

`http_requests_total` is the metric name; the three labels together identify exactly *which* time series this particular value belongs to. This isn't one running total for the whole application — it's one time series among potentially many under the same metric name.

Labels — The Dimension That Makes Aggregation Possible

Every unique combination of label values creates a genuinely separate time series:

```
http_requests_total{method="GET", status="200"} 8934
http_requests_total{method="POST", status="500"} 12
http_requests_total{method="GET", status="404"} 201
```

Three distinct time series, one metric name. This is exactly what makes Chapter 4's PromQL genuinely useful — labels are the dimension you filter and aggregate along, letting a single instrumented metric answer

"what's my GET rate," "what's my 500 rate," and "what's my total request rate across every method and status combined" from the same underlying data.

The Four Metric Types

Prometheus defines exactly four kinds of metric, each with a different shape of value and a different intended use:

```
# HELP http_requests_total Total HTTP requests served
# TYPE http_requests_total counter
http_requests_total{method="GET"} 8934

# HELP memory_usage_bytes Current process memory usage
# TYPE memory_usage_bytes gauge
memory_usage_bytes 52428800

# HELP http_request_duration_seconds Request duration
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_bucket{le="0.1"} 8010
http_request_duration_seconds_bucket{le="0.5"} 8900
http_request_duration_seconds_bucket{le="1.0"} 8930
http_request_duration_seconds_bucket{le="+Inf"} 8934
http_request_duration_seconds_sum 452.3
http_request_duration_seconds_count 8934
```

- **Counter** — only ever increases (or resets to zero on restart). Right for "total requests served," "total errors," anything that's a running cumulative count. A raw counter value is rarely useful by itself; Chapter 4's `rate()` turns it into a meaningful per-second figure.
- **Gauge** — a value that can go up or down freely. Right for "current memory usage," "active connections," "queue depth right now."
- **Histogram** — sorts observations (like request durations) into configurable buckets, alongside a running sum and count. Quantiles are computed later, at query time, from the bucket counts.
- **Summary** — similar intent to a histogram, but quantiles are calculated client-side, inside the instrumented application itself, before the metric is ever exposed.

Histogram vs. Summary — A Real Operational Tradeoff

These two look similar but behave very differently once there's more than one instance of a service running. A histogram's raw bucket counts from many different pods can simply be **summed together** — Chapter 4's `histogram_quantile()` then computes one fleet-wide p99 from the combined buckets. A summary's quantile,

by contrast, is already computed inside one specific instance before it's ever exposed — there is no meaningful way to average two different instances' own p99 values together and get a real fleet-wide p99. For anything that might ever need aggregating across replicas — which, in practice, is nearly everything in a real production system — histograms are the safer default.

TYPE	BEHAVIOR	AGGREGATABLE ACROSS INSTANCES?
Counter	Only increases, resets on restart	Yes — sum, or rate() over time
Gauge	Freely goes up or down	Yes — sum, avg, min, max
Histogram	Bucketed observations + sum + count	Yes — buckets sum cleanly across instances
Summary	Client-side quantiles + sum + count	No — per-instance quantiles can't be meaningfully combined

A REAL NAMING CONVENTION WORTH FOLLOWING

Prometheus doesn't enforce metric names, but the community convention is strong and worth adopting: a unit suffix (`_seconds`, `_bytes`) and a `_total` suffix specifically for counters. `http_request_duration_seconds` and `http_requests_total` both follow this pattern — it makes a metric's own type and unit legible from its name alone, without needing to check the `# TYPE` line.

LABEL CARDINALITY — A REAL, COMMON PRODUCTION INCIDENT

Never put an unbounded value — a raw user ID, a full URL with query parameters, a timestamp — into a label. Every distinct label-value combination creates a genuinely separate time series; a label that can take millions of distinct values multiplies Prometheus's own storage and memory usage by that same factor, and has caused real outages in real production Prometheus deployments. Labels should have a small, bounded set of possible values — `method`, `status`, `handler` — not anything that grows without limit.

Hands-On Exercises

EXERCISE 1

Write the raw exposition-format lines for a counter metric tracking total failed login attempts, with labels for reason ("bad_password" or "account_locked"), and explain why a counter — not a gauge — is the right choice here.

View solution

EXERCISE 2

A service exposes a gauge called active_websocket_connections. Explain why a gauge is the correct type here rather than a counter, referencing what would go wrong if it were implemented as a counter instead.

View solution

EXERCISE 3

A colleague proposes adding a raw user_id label to a request-duration histogram, planning to aggregate p99 latency across all instances afterward. Explain the two separate problems with this plan — one about cardinality, one about histogram vs. summary aggregation.

View solution

CHAPTER 2 QUICK REFERENCE

- A metric is a named time series, identified by its name plus a set of label key-value pairs
- Every unique label combination is a genuinely separate time series under the same metric name
- **Counter** — only increases; **Gauge** — goes up or down freely
- **Histogram** — bucketed observations, quantiles computed at query time, aggregatable across instances
- **Summary** — client-side quantiles, NOT meaningfully aggregatable across instances
- Naming convention: unit suffix (_seconds, _bytes), _total suffix for counters
- Never label with unbounded values — a real, common cause of production Prometheus incidents (cardinality explosion)

CHAPTER 3 OF 11

Prometheus Architecture & Scraping

Observability

Chapter 3 · Prometheus Architecture & Scraping

`obs1-2` established what a metric looks like once it exists. This chapter covers how it actually gets from an application into Prometheus in the first place — the architectural choice that shapes nearly everything else about how the system is operated.

The Pull Model — Prometheus Comes to You

Prometheus's core design decision: it **scrapes** (pulls) metrics by periodically sending an HTTP GET to a `/metrics` endpoint each target exposes, in the exposition format from `obs1-2`. Applications don't send their metrics anywhere — they just expose them on an endpoint and wait to be asked, on whatever schedule Prometheus's own configuration decides.

Why Pull? — Real Advantages

- **Centralized control** — scrape frequency and target lists live in one place (Prometheus's own config), not scattered across every application's own settings.
- **A failed scrape is itself a signal** — if Prometheus can't reach a target, that's meaningful information ("this target is unreachable") distinct from "this target reported zero" — something a push-based system can't easily distinguish from silence.
- **Simpler service discovery** — Prometheus decides who to scrape and finds them; individual services never need to know Prometheus's own address.
- **Trivially testable locally** — `curl`ing a `/metrics` endpoint shows exactly what Prometheus would see, with no special tooling required.

When Pull Doesn't Fit — Pushgateway

A short-lived batch job can finish and exit before any scheduled scrape would ever reach it — the pull model has nothing to pull from once the process is gone. The **Pushgateway** is the sanctioned exception: a batch job pushes its final metrics to the Pushgateway once, right before exiting, and Prometheus then scrapes the Pushgateway itself, like any other ordinary target.

PUSHGATEWAY IS AN EXCEPTION, NOT A GENERAL PUSH MECHANISM

Using the Pushgateway for regular, long-running services defeats the entire point of the pull model's own advantages — a failed scrape stops meaning "the target is down" once every value is being relayed through an intermediary. It exists specifically for short-lived jobs that genuinely can't be scraped directly, nothing broader.

Exporters — Getting Metrics Out of Things That Don't Speak Prometheus

Most systems — MySQL, Redis, the Linux kernel itself, older applications — don't natively expose a Prometheus-format `/metrics` endpoint. An **exporter** is a small adapter process that queries the underlying system in whatever native way it supports, then re-exposes that data as an ordinary Prometheus-format endpoint for scraping. This is exactly the missing piece `cloud1-8`'s own terminology-level coverage never got into.

- **node_exporter** — host-level metrics: CPU, memory, disk, network, straight from the OS
- **mysqld_exporter** / **redis_exporter** — connect to the database using its own native protocol, re-expose the results as Prometheus metrics
- **blackbox_exporter** — actively probes external endpoints (HTTP, TCP, ICMP) and reports reachability/latency as metrics

Configuring Scrape Targets — prometheus.yml

```
scrape_configs:
  - job_name: 'node'
    scrape_interval: 15s
    static_configs:
      - targets: ['localhost:9100']

  - job_name: 'my-app'
    static_configs:
      - targets: ['app1:8080', 'app2:8080']
```

Each `job_name` groups a set of targets sharing the same scrape configuration. Every scraped sample is automatically tagged with a `job` label (and an `instance` label identifying which specific target), giving every metric a built-in "which service, which instance" dimension for free.

Service Discovery — Beyond Hardcoded Targets

A static, hand-maintained target list breaks the moment instances scale up, scale down, or get replaced — exactly what happens continuously in Kubernetes or a cloud autoscaling group. Prometheus supports **service discovery** mechanisms — `kubernetes_sd_config`, `ec2_sd_config`, `consul_sd_config`, among others — that automatically discover the current set of scrape targets directly from the underlying platform, rather than requiring anyone to keep a list in sync by hand. `obs1-10` covers Kubernetes' own version of this in depth, via ServiceMonitors and the Prometheus Operator.

Retention — Local Storage Isn't Forever

Prometheus stores scraped samples on local disk in its own time-series database, with a configurable retention window — 15 days is a common default. This is a deliberate scoping decision, not an oversight: Prometheus is built for reasonably recent operational data, not as a long-term archive. For genuinely long-term retention or a unified view across multiple Prometheus instances, remote-writing to a dedicated long-term storage backend (Thanos, Cortex, Mimir) is the standard real-world answer — worth naming honestly here as beyond this course's own scope, a deliberate line drawn rather than an oversight.

	PULL (PROMETHEUS'S DEFAULT)	PUSH
Who initiates	The monitoring system, on its own schedule	The application, whenever it decides to
A silent target	Detected directly — the scrape itself fails	Ambiguous — is it down, or just not pushing right now?
Config location	Centralized, in the monitoring system	Scattered across every application

THE UP METRIC IS ONE OF THE MOST USEFUL THINGS PROMETHEUS GIVES YOU FOR FREE

Every scrape target automatically gets an `up` metric — `1` if the last scrape succeeded, `0` if it didn't. This single built-in metric, direct proof of the pull model's own "a failed scrape is itself a signal" advantage, is often the very first thing worth checking during an incident.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own reasoning, why a failed scrape is more informative than a push-based system simply not receiving any data for a while.

View solution

EXERCISE 2

A nightly batch job runs for 45 seconds and then exits. Explain why scraping it directly wouldn't reliably work, and what the correct Prometheus-native solution is.

View solution

EXERCISE 3

Write a scrape_configs entry for a job named "redis" scraping targets at redis1:9121 and redis2:9121 every 30 seconds, and explain why redis_exporter is needed here rather than scraping Redis directly.

View solution

CHAPTER 3 QUICK REFERENCE

- Prometheus pulls (scrapes) metrics from a /metrics HTTP endpoint on a schedule, rather than applications pushing to it
- A failed scrape is itself meaningful — the built-in **up** metric reports 1 (success) or 0 (failure) per target
- **Pushgateway** — the sanctioned exception, for short-lived batch jobs only, not general-purpose push
- An **exporter** adapts a system that doesn't natively speak Prometheus into a scrapeable /metrics endpoint (node_exporter, mysqld_exporter, blackbox_exporter)
- scrape_configs in prometheus.yml defines jobs, targets, and scrape intervals; job/instance labels are added automatically
- Service discovery (kubernetes_sd_config, etc.) replaces hand-maintained static target lists in dynamic environments
- Local retention is intentionally limited (often ~15 days); Thanos/Cortex/Mimir are the standard long-term-storage answer, out of this course's own scope

CHAPTER 4 OF 11

PromQL Deep Dive

Observability

Chapter 4 · PromQL Deep Dive

`obs1-2` left a promise on the table: a raw counter value is rarely useful by itself. This chapter delivers on it — PromQL, the query language that turns Ch.2's raw time series into the numbers a dashboard or an alert actually cares about.

Instant Vectors vs. Range Vectors

`http_requests_total` alone is an **instant vector** — the current value of every matching time series, at one point in time. `http_requests_total[5m]` is a **range vector** — every value each matching series took over the trailing five minutes. Range vectors can't be graphed directly; they exist specifically as input to functions like `rate()` that need a window of history to compute something meaningful.

rate() and irate() — Turning a Counter Into Something Useful

```
rate(http_requests_total{status="500"}[5m])
```

`rate()` computes the per-second average rate of increase over the given window — exactly the transformation Ch.2 promised a raw counter needed. It also automatically detects and compensates for **counter resets** (a process restart dropping the counter back to zero), a genuinely important detail: without that handling, a restart would otherwise show up as a nonsensical negative rate. `irate()` computes an **instantaneous** rate using only the last two data points in the range, more responsive to sudden spikes but noisier — `rate()` is the safer default for dashboards and alerting; `irate()` suits fast-moving, high-resolution graphs where responsiveness matters more than smoothness.

Aggregation Operators — Collapsing Across Labels

```
# Total request rate, broken down by status code, collapsed across every instance/method/handler
sum(rate(http_requests_total[5m])) by (status)

# One single fleet-wide number, collapsing every label including status
sum(rate(http_requests_total[5m]))
```

`sum()`, `avg()`, `min()`, `max()`, and `count()` combine values across every matching series. The `by (...)` clause keeps specific label dimensions in the result, collapsing everything else; `without (...)` does the reverse — drop these labels, keep the rest. Choosing the right dimension to keep is exactly what turns `obs1-2`'s own per-instance, per-status, per-method time series into the one useful number a dashboard panel actually wants.

histogram_quantile() — Delivering on Chapter 2's Own Promise

```
histogram_quantile(0.99, sum(rate(http_request_duration_seconds_bucket[5m])) by (le))
```

`obs1-2` named histograms as aggregatable specifically because their bucket counts can be summed across instances. This is that promise made concrete: `rate()` turns each bucket's cumulative count into a per-second rate, `sum(...) by (le)` combines those rates across every instance while explicitly preserving the bucket-boundary label, and `histogram_quantile()` then computes an approximate p99 from the combined buckets.

Common Query Patterns

```
# Error rate as a percentage of total requests
sum(rate(http_requests_total{status=~"5.."}[5m]))
/
sum(rate(http_requests_total[5m]))
* 100

# How many errors happened in the last hour
increase(errors_total[1h])
```

`=~` matches a label against a regular expression — `"5.."` catches every 5xx status code in one pattern. `increase()` is `rate()`'s own close relative: instead of a per-second rate, it reports the total increase across the whole window, correctly handling counter resets the same way `rate()` does — the right choice for "how many of X happened," rather than "how fast is X happening."

FUNCTION	COMPUTES	BEST FOR
rate()	Average per-second rate over the window	Dashboards, alerting — smoother, safer default
irate()	Instantaneous rate from the last two points	Fast-moving graphs where responsiveness matters most
increase()	Total increase over the window	"How many happened" questions, not "how fast"

NEVER GRAPH A RAW COUNTER DIRECTLY

A raw counter, plotted as-is, is nearly always a misleading, ever-climbing line that tells you almost nothing useful on its own. Wrapping it in `rate()`, `irate()`, or `increase()` first — turning "a running total" into "how fast is this changing" or "how many happened in this window" — is what actually makes a counter worth looking at.

AGGREGATING AWAY THE LE LABEL BREAKS HISTOGRAM_QUANTILE() SILENTLY

`sum(rate(...)) by (status)` instead of `by (le)` on a histogram metric doesn't produce an error — it produces a query that runs, returns a number, and is **meaningless**, since `histogram_quantile()` has no bucket boundaries left to compute a quantile from once `le` is aggregated away. This is a genuinely common, easy-to-miss mistake precisely because nothing fails loudly when it happens.

Hands-On Exercises

EXERCISE 1

Write a PromQL query computing the per-second rate of 4xx responses over the last 10 minutes for the metric http_requests_total, and explain why rate() rather than the raw counter is the right choice for a dashboard panel.

View solution

EXERCISE 2

Write a query computing the p95 latency across all instances for the metric http_request_duration_seconds_bucket, and explain why the by (le) clause is required for the result to be meaningful.

View solution

EXERCISE 3

A service restarts mid-window, causing its request counter to drop back to zero. Explain what rate() does in this situation, and why a naive "current value minus value five minutes ago" calculation would produce a wrong (negative) result instead.

View solution

CHAPTER 4 QUICK REFERENCE

- **Instant vector** — one value per series right now; **range vector** — a window of values, e.g. [5m]
- **rate()** — smooth per-second average, counter-reset-aware, the safe default for dashboards/alerts
- **irate()** — instantaneous, from the last two points; noisier, more responsive
- **increase()** — total change over the window, for "how many" rather than "how fast"
- **sum()/avg()/min()/max()/count() ... by (...)** — aggregate across labels, keeping the dimensions listed in by
- **histogram_quantile(q, sum(rate(..._bucket[5m])) by (le))** — the le label must survive aggregation, or the result is silently meaningless
- Never graph a raw counter directly — always wrap it in rate/irate/increase first

CHAPTER 5 OF 11

Grafana — Dashboards & Visualization

Observability

Chapter 5 · Grafana — Dashboards & Visualization

`obs1-4` gave you PromQL. This chapter gives you somewhere to put it — Grafana, the visualization layer this course pairs with Prometheus. And in keeping with `obs1-1`'s own opening warn-box, this chapter is honest about what a dashboard, no matter how well-built, actually is and isn't.

What Grafana Actually Is — A Visualization Layer, Not a Data Store

Grafana itself stores no metrics. It's a query-and-render frontend that connects to one or more **data sources** — Prometheus here, but also Loki for logs (`obs1-7`) and Jaeger/Tempo for traces (`obs1-8`) — and turns their query results into panels. This separation matters in practice: losing Grafana loses no actual data, since everything it displays lives in the underlying data source, not in Grafana itself. It also means one Grafana dashboard can show metrics, logs, and traces side by side, sourced from entirely different systems.

Data Sources — Connecting Grafana to Prometheus

A data source is configured once — a name, a type (Prometheus), and the URL Grafana should query. Once that connection exists, **every** panel in **every** dashboard can query it directly using ordinary PromQL, exactly as written in `obs1-4`.

Panels — The Basic Building Block

A panel is one visualization, backed by one or more queries. Common types: a time series graph, a single stat/number, a gauge, a table, and a heatmap — genuinely useful specifically for visualizing a histogram's own bucket distribution over time.

```
# A panel's query is just PromQL — everything from Chapter 4 transfers directly
sum(rate(http_requests_total{status=~"5.."}[5m]))
/
sum(rate(http_requests_total[5m]))
* 100
```

That's the exact error-rate query from `obs1-4`, dropped straight into a panel — a panel is nothing more than a chosen visualization wrapped around a PromQL query you already know how to write.

Template Variables — One Dashboard, Many Contexts

```
sum(rate(http_requests_total{job="$service"}[5m]))
```

`$service` is a **template variable** — a dropdown at the top of the dashboard, typically populated by a `label_values()` query against Prometheus itself, that gets substituted into every panel's query at render time. One dashboard, built once, can then serve every service sharing that shape of metric — rather than hand-building a near-identical dashboard per service.

Dashboards as Code — Provisioning

Building a dashboard by clicking through Grafana's own UI works, but doesn't scale and isn't reviewable the way real code is. Grafana dashboards can instead be defined as **JSON**, checked into version control alongside application code — the same review-and-history discipline `git1-3` already established for everything else. **Provisioning** is what makes this practical: Grafana can automatically load dashboard JSON and data source configuration from files at startup, rather than requiring anyone to manually recreate them through the UI — a light echo of `k8s2-9`'s own GitOps pattern, applied here to dashboards instead of Kubernetes manifests.

An Honest Note — Dashboards Are Still Monitoring, Not Observability

`obs1-1`'s own warn-box applies here directly: a dashboard, however well-designed, only ever answers the questions its panels were built to answer in advance. Grafana's **Explore** mode is its own genuine answer to that limit — an interface for running ad-hoc PromQL queries against a data source without needing a saved panel first, letting you follow a brand-new question on the spot during an incident rather than being limited to whatever panels already exist.

	GOOD FOR	LIMITATION
A fixed dashboard	Recurring, anticipated questions — daily health checks, known failure modes	Only answers what its panels were built to answer
Explore mode	An unanticipated question, mid-incident, right now	Not saved or shared by default — built for exploration, not a recurring view

VERSION-CONTROL DASHBOARD JSON THE SAME WAY YOU VERSION-CONTROL CODE

A dashboard that only exists as manual clicks inside Grafana's own UI has no history, no review process, and no way to recover if someone accidentally breaks it. Provisioning dashboards from JSON files kept in the same repository as the application they monitor gives dashboards the exact same discipline this site's own `git1-3` and `tf1` already established for everything else.

A DASHBOARD CRAMMED WITH PANELS BECOMES NOISE, NOT SIGNAL

Cramming every metric that exists onto one screen "just in case" is a real, common anti-pattern — the resulting dashboard is technically comprehensive and practically useless, since nothing on it stands out during an actual incident. A small number of well-chosen panels, built to answer one specific operational question, beats an everything-at-once dashboard every time.

Hands-On Exercises

EXERCISE 1

Explain why deleting a Grafana dashboard doesn't delete any actual metrics data, and what this reveals about the relationship between Grafana and a data source like Prometheus.

View solution

EXERCISE 2

A team has 15 near-identical dashboards, one per microservice, each hand-built with the same panels. Explain how a template variable would let this become a single dashboard instead, and write the PromQL a request-rate panel would use with that variable.

View solution

EXERCISE 3

During an incident, an engineer needs to answer a question no existing dashboard panel covers. Explain which Grafana feature is built for exactly this situation, and why a fixed dashboard alone wouldn't have been enough — tying your answer back to obs1-1's own monitoring-vs-observability distinction.

View solution

CHAPTER 5 QUICK REFERENCE

- Grafana stores no data itself — it queries data sources (Prometheus, Loki, Jaeger/Tempo) and renders the results
- A panel is one visualization backed by a query — a PromQL query from Chapter 4 dropped straight in

- Template variables (e.g. $service) let one dashboard serve many contexts instead of duplicating dashboards
- Provisioning loads dashboard JSON and data source config from files at startup — dashboards as version-controlled code
- A dashboard only answers questions its panels were built to answer; Explore mode supports genuinely new, unanticipated questions
- Too many panels on one dashboard is a real anti-pattern — fewer, well-chosen panels beat comprehensive-but-noisy ones

CHAPTER 6 OF 11

Alerting with Alertmanager

Observability

Chapter 6 · Alerting with Alertmanager

`cloud1-9` named alert fatigue as a real problem back in the Cloud Platforms course, without the tooling to actually fix it. This chapter is where that gets addressed directly — Prometheus's own alert rules, and Alertmanager, the separate component that decides who actually gets told, and how.

Two Separate Jobs — Prometheus Rules vs. Alertmanager

Prometheus itself evaluates **alert rules** — PromQL expressions that, when true, mark something as firing. That's a deliberately separate concern from **Alertmanager**, which receives fired alerts and decides who gets notified, through which channel, and on what schedule. Prometheus decides *what* is wrong; Alertmanager decides *who* hears about it and *how*.

Alert Rules — Defining "This Is a Problem"

```
groups:
  - name: example
    rules:
      - alert: HighErrorRate
        expr: sum(rate(http_requests_total{status=~"5.."}[5m])) / sum(rate(http_requests_total[5m]))
        for: 10m
        labels:
          severity: critical
        annotations:
          summary: "Error rate above 5% for {{ $labels.job }}"
```

`expr` reuses `obs1-4`'s own error-rate query directly. `labels` attach values used for **routing** — `severity: critical` is what a routing rule matches against, not something a human reads. `annotations` are the reverse — human-readable context, with templating access to the alert's own labels and the value that triggered it, meant to be read by whoever gets paged.

The for: Duration and Alert States

`for: 10m` means the condition has to stay true continuously for ten minutes before the alert actually fires — a deliberate guard against a single brief blip triggering a page. An alert moves through real, named states: **pending** (the condition just became true, but `for` hasn't elapsed yet), **firing** (the full duration has passed, and it's now been sent to Alertmanager), and **resolved** (the condition is no longer true — Alertmanager can notify on this too, so a resolved incident doesn't require someone to manually confirm it's over).

Alertmanager — Routing, Grouping, and Silencing

- **Routing** — a tree of label matchers deciding which receiver (Slack, email, PagerDuty, ...) a given alert goes to, based on its own labels.
- **Grouping** — multiple alerts sharing similar labels get bundled into **one** notification, rather than flooding a channel with dozens of separate messages during one widespread incident — directly answering `cloud1-9`'s own alert-fatigue warning.
- **Silencing** — temporarily mutes alerts matching a given label set, for planned maintenance or a known ongoing issue, without touching the underlying rule itself.

A Routing Tree Example

```
route:
  receiver: 'default-slack'
  group_by: ['alertname', 'job']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 4h
  routes:
    - match:
        severity: critical
      receiver: 'pagerduty'
    - match:
        severity: warning
      receiver: 'slack-warnings'
```

`group_wait` is how long Alertmanager waits after the first alert in a new group before sending the initial notification, giving related alerts a chance to arrive and be bundled together. `group_interval` controls how long to wait before sending an update about new alerts joining an already-notified group. `repeat_interval` is how long to wait before re-sending a notification for a group that's still firing, unresolved — these three timers together are what actually implement grouping's own noise reduction, not just a conceptual idea.

Avoiding Alert Fatigue — Directly Building On cloud1-9

`cloud1-9` named the problem without the tooling; this chapter's routing, grouping, and the `for` duration together are the concrete fix. Route by real urgency — not everything to the same pager. Group related alerts so one incident produces one notification, not fifty. And before ever setting `severity: critical`, ask the honest question: if this fires at 3am, does someone genuinely need to get out of bed for it? An alert that doesn't clear that bar belongs at a lower severity, or shouldn't page a human at all.

SEVERITY	TYPICAL ROUTING	THE REAL QUESTION
critical	Pages a human immediately (PagerDuty, phone)	Does someone need to act right now, even at 3am?
warning	A chat channel, seen during working hours	Worth knowing about soon, not worth waking anyone for
info	Dashboard only, no active notification	Useful context if you're already looking, not urgent on its own

EVERY ALERT SHOULD POINT TO A NEXT ACTION

An annotation linking to a runbook, or containing enough context to act on immediately, is what turns an alert into something actionable rather than just another notification to dismiss. An alert with no clear next step is itself a fatigue source, whatever its severity label says.

A FOR: DURATION THAT'S TOO SHORT TRAINS PEOPLE TO IGNORE ALERTS

Firing on a single slow request or a brief network hiccup — anything that resolves itself before a human could realistically act — produces noise, not signal. Once people learn that a given alert "usually clears itself," they stop reacting to it at all, which defeats the entire purpose of alerting in the first place. A deliberately chosen `for` duration, long enough to filter out transient blips, is a real defense against exactly this.

Hands-On Exercises

EXERCISE 1

Write a Prometheus alert rule named "HighMemoryUsage" that fires when a gauge metric memory_usage_percent exceeds 90 for at least 15 minutes, with an annotation including the current value.

View solution

EXERCISE 2

Explain, using this chapter's own group_by/group_wait/group_interval settings, what would happen if 30 instances of the same service all fired the same alert within a few seconds of each other, and why this matters for avoiding alert fatigue.

View solution

EXERCISE 3

A team sets for: 30s on every single alert rule "to be safe." Explain what's likely to go wrong with this choice, and what a better default might look like for most rules.

View solution

CHAPTER 6 QUICK REFERENCE

- Prometheus evaluates alert rules (what's wrong); Alertmanager routes fired alerts (who's told, how)
- **for: duration** — the condition must hold continuously before firing, filtering out transient blips
- Alert states: **pending** → **firing** → **resolved**
- **labels** drive routing; **annotations** give humans context, with $labels/$value templating
- **Routing** sends by severity/team; **grouping** bundles related alerts into one notification; **silencing** mutes without disabling the rule
- group_wait/group_interval/repeat_interval are the concrete timers implementing grouping's own noise reduction
- Before marking anything critical, ask: does this genuinely need someone up at 3am? — the real answer to cloud1-9's own alert-fatigue warning

CHAPTER 7 OF 11

Structured Logging & Log Aggregation

Observability

Chapter 7 · Structured Logging & Log Aggregation

Metrics are done. This chapter moves to the second pillar `obs1-1` named — logs — and delivers the real tooling `cloud2-4` 's own correlation-ID material assumed was already in place.

Structured vs. Unstructured Logs

```
# Unstructured
2026-07-13 10:32:01 ERROR Payment failed for user 4821 - timeout after 5000ms

# Structured
{"timestamp": "2026-07-13T10:32:01Z", "level": "error", "msg": "payment failed", "user_id": 4821, "du
```

Unstructured logs are free text, meant for a human reading them one line at a time — extracting `user_id` or `duration_ms` afterward means writing a regex against the message, fragile and quick to break the moment someone tweaks the wording. Structured logs emit their fields directly, as data — every field is queryable and filterable from the moment it's written, with no parsing guesswork required downstream.

Log Aggregation — Why You Need a Central System

In a system with dozens of services across dozens of instances, logs scattered across that many individual machines are practically unsearchable during a real incident — nobody is going to SSH into thirty containers one at a time while something is on fire. A log aggregation system centralizes logs from every source into one searchable place, which is exactly the capability `cloud2-4` 's own "first five minutes" incident-response technique assumed was already sitting there, ready to use.

Loki vs. the ELK/EFK Stack — Two Different Philosophies

ELK (Elasticsearch, Logstash, Kibana) — or **EFK**, swapping in Fluentd/Fluent Bit — indexes the *full text* of every log line in Elasticsearch, enabling powerful free-text search across any word, anywhere, at real storage and

compute cost. **Loki** takes a deliberately different approach: it doesn't index log content at all — only **labels**, the exact same concept as `obs1-2`'s own Prometheus labels, applied here to logs instead of metrics. The actual log lines are stored compressed, cheaply, and only scanned — not separately indexed — within whichever label-selected stream a query has already narrowed down. This is a deliberate design choice, not a coincidence: Loki is built by the same team behind Grafana specifically to pair with Prometheus's own label-based philosophy, at a fraction of the resource cost full-text indexing requires.

LogQL — Loki's Own Query Language

```
{job="checkout-service"} |= "timeout"

# Extracting and filtering on a structured field
{job="checkout-service"} | json | duration_ms > 5000
```

`{job="checkout-service"}` is the label selector — cheap, indexed, exactly like a PromQL selector from `obs1-4`. `|= "timeout"` is a line filter, applied only within that already-narrowed stream — not indexed, but only scanning a small, pre-selected subset rather than every log line ever written. `| json` parses each structured line and exposes its fields for further filtering, as in the `duration_ms > 5000` example — turning a structured field written once at log time into something directly queryable later.

Correlation IDs — Delivering On cloud2-4's Own Material

`cloud2-4` named correlation IDs as the technique for tracing one request across multiple services' own logs, conceptually. Here's the concrete mechanism: a unique ID is generated once, at the edge — typically an API gateway — and passed along on every downstream service call, usually as an HTTP header, then included as a structured field in every log line each service emits while handling that request.

```
{job=~".+"} | json | request_id = "abc-123"
```

That single query, matching every job, filters down to every log line — across every service that touched this one request — sharing that exact `request_id`. This is the concrete technique that turns "find every log line related to this one failing request, across the whole system" from a conceptual goal into something you can actually run. `obs1-8`'s traces are the even more powerful version of this same underlying idea — correlating not just log lines, but full per-service timing across the whole request.

	INDEXING APPROACH	BEST FOR
ELK / EFK	Full-text index of every log line's content	Powerful free-text search across unknown content
Loki	Labels indexed only; content scanned within a narrowed stream	Cheap at scale, when you already know roughly which labels to filter by

EMIT STRUCTURED LOGS FROM DAY ONE, EVEN ON A SMALL PROJECT

Retrofitting structured logging onto an already-large unstructured system is real, tedious work — every existing log statement has to be found and rewritten. Starting structured from the very first log line costs almost nothing extra and pays off the moment you need to query anything.

A CORRELATION ID ONLY WORKS IF EVERY SERVICE ACTUALLY PROPAGATES IT

One service in the request's path that forgets to forward the incoming request ID header to the next service silently breaks the entire chain — with no error, no warning, just a gap in the trail where that ID stops appearing. This is a real, common integration gap, and it's usually only discovered during an actual incident, exactly when it's least convenient to find.

Hands-On Exercises

EXERCISE 1

Rewrite the unstructured log line "2026-07-13 09:15:44 WARN Rate limit exceeded for IP 203.0.113.7 on endpoint /api/checkout" as a structured JSON log line, and explain what becomes easier once it's structured.

View solution

EXERCISE 2

Write a LogQL query that selects logs from the job "inventory-service" and filters to only lines containing the text "connection refused."

View solution

EXERCISE 3

A request passes through an API gateway, an auth service, and a payments service, each logging with a request_id field — except the payments service, which was recently rewritten and doesn't include it. Explain exactly what breaks during an incident investigation, and why this failure is easy to miss until it matters.

View solution

CHAPTER 7 QUICK REFERENCE

- **Structured logs** (JSON key-value fields) are directly queryable; **unstructured logs** need fragile after-the-fact regex parsing
- Log aggregation centralizes logs from every instance/service into one searchable place — required for cloud2-4's own incident-response technique to actually work
- **ELK/EFK** — full-text indexes every log line's content; **Loki** — indexes labels only, scans content within the narrowed stream (Prometheus's own label philosophy, applied to logs)
- LogQL: `{label="value"}` selects a stream (cheap, indexed); `|= "text"` filters lines; `| json` exposes structured fields for further filtering
- A correlation/request ID, generated at the edge and propagated through every downstream call, is the concrete mechanism behind cloud2-4's own cross-service log correlation
- One service dropping the ID silently breaks the whole trail — a real, common, easy-to-miss integration gap

CHAPTER 8 OF 11

Distributed Tracing

Observability

Chapter 8 · Distributed Tracing

`obs1-7` closed with a promise: traces are the more powerful version of the same correlation idea logs only partially deliver. This chapter is where that becomes concrete — the third pillar, and genuinely new material nothing else on this site has covered before now.

What a Trace Actually Is

A **trace** represents one request's full journey across every service it touches. It's built from **spans** — each span represents one unit of work: one service handling the request, or one specific operation inside it, like a database query. A span has a name, a start time, a duration, and a set of key-value attributes, similar in spirit to a metric's own labels.

Spans — The Building Block

Every span has a unique span ID, and — except the very first, **root** span — a parent span ID, forming a tree that reconstructs the exact call graph a request actually took. Every span belonging to the same request shares one **trace ID**, the thread tying the whole tree together.

```
Trace ID: abc-123

gateway (50ms)                    [root span]
  auth-service (10ms)
  checkout-service (35ms)
    inventory-service (28ms)
      db-query (25ms)
```

This is `obs1-1`'s own checkout example, made literal. "3.8 of the 4 seconds spent waiting on inventory" isn't a conclusion someone had to piece together from separate logs anymore — it's a visible span in this exact tree, with its own duration sitting right there.

Trace Context Propagation

For spans created in separate services to link into one trace, the trace ID — and the current span ID, to establish the parent-child relationship — has to travel along with every downstream call, usually as an HTTP header. This is mechanically the same idea as `obs1-7`'s own correlation ID, formalized into a real, standardized format: the **W3C Trace Context** `traceparent` header, rather than every team inventing its own ad-hoc header.

```
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
```

Four dash-separated parts: version, trace ID, the parent span's own ID, and trace flags. Every hop that receives this header, creates its own span as a child of that parent ID, and forwards an updated version downstream — the exact mechanism that turns separate services' separate spans into one connected tree.

OpenTelemetry — A First Look at Instrumenting Your Code

OpenTelemetry (OTel) is the vendor-neutral standard for generating this telemetry — an SDK application code calls directly to create spans, which are then exported to a tracing backend. `obs1-9` covers it properly; here's the shape of it:

```
with tracer.start_as_current_span("process_payment") as span:
    # do the actual work
    span.set_attribute("payment.amount", 49.99)
```

Many frameworks and libraries support **auto-instrumentation** — OTel can automatically create spans for common operations (incoming HTTP requests, outgoing calls, database queries) with no manual code changes at all, while manual spans like the one above cover anything genuinely business-specific a generic instrumentation library couldn't know to track on its own.

Jaeger — Storing and Visualizing Traces

Jaeger receives the spans instrumented services export, stores them, and provides a UI to search traces — by service, operation, duration, tags — and to visualize any single trace as a waterfall/timeline view, exactly like the tree above but interactive. It's the tracing pillar's own equivalent of what Grafana is for metrics: a query-and-visualization layer sitting on top of the actual data. And as `obs1-5` already noted, Grafana itself can visualize Jaeger (or Tempo, its close relative) traces directly as a data source, right alongside metrics and logs in the same dashboard.

PILLAR	UNIQUE PROPERTY	REAL COST CONSIDERATION
Metrics	Cheap, aggregated trends	Storage grows with cardinality (obs1-2)
Logs	Rich per-event detail	Storage grows with log volume
Traces	The full call graph and timing for one specific request	Instrumentation + export overhead per traced request — sampling is the standard mitigation

SAMPLING — A REAL, HONEST OPERATIONAL TRADEOFF

Tracing every single request at real production scale can be genuinely expensive, in both instrumentation overhead and storage. Most real systems **sample** — tracing only a percentage of requests, or deliberately always tracing 100% of errors and slow requests specifically — rather than tracing everything all the time. This is a deliberate tradeoff, not a compromise to feel bad about: the requests most worth having a full trace for are disproportionately the ones already flagged as errors or outliers.

A DROPPED TRACEPARENT HEADER DOESN'T ERROR — IT SILENTLY STARTS A NEW TRACE

A service that fails to forward the incoming `traceparent` header to its own downstream calls doesn't produce any visible failure — it just begins a brand-new, disconnected trace instead of continuing the existing one. The result looks, from Jaeger's own UI, like the request's journey simply stopped at that service, with no indication that work continued elsewhere under a different trace ID entirely. This is the tracing pillar's own exact version of `obs1-7`'s dropped-correlation-ID gotcha.

Hands-On Exercises

EXERCISE 1

Draw (as text, matching the chapter's own tree format) a trace for a request that hits an api-gateway (root span), which calls both an auth-service and, only after auth succeeds, a search-service that itself queries a cache and then a database.

View solution

EXERCISE 2

Explain, using the traceparent header's own four parts, what information a downstream service needs from an incoming request in order to correctly create its own span as a child of the correct parent.

View solution

EXERCISE 3

A team traces 100% of requests and notices real latency overhead from tracing itself during peak traffic. Propose a sampling strategy that still guarantees every error and every slow request gets a full trace, and explain why this is a reasonable tradeoff rather than a loss of visibility.

View solution

CHAPTER 8 QUICK REFERENCE

- A **trace** is one request's full journey; a **span** is one unit of work within it, with a name, duration, and attributes
- Spans share a trace ID and form a parent-child tree via span IDs, reconstructing the real call graph
- The **traceparent** header (W3C Trace Context) propagates trace/span IDs across services — the formalized version of obs1-7's correlation ID
- **OpenTelemetry** is the vendor-neutral SDK/standard for generating spans, via auto-instrumentation and manual spans (covered fully in obs1-9)
- **Jaeger** stores and visualizes traces — the tracing pillar's own equivalent of Grafana
- **Sampling** (not tracing every request) is a real, standard cost tradeoff, often biased toward always tracing errors/slow requests
- A dropped traceparent header silently starts a disconnected new trace rather than erroring — a real, easy-to-miss gotcha

CHAPTER 9 OF 11

The OpenTelemetry Standard

Observability

Chapter 9 · The OpenTelemetry Standard

`obs1-1`'s own tip-box promised this chapter would be about correlation, not just collection. `obs1-8` previewed the SDK's own shape. This is where both promises land — OpenTelemetry as the vendor-neutral standard unifying all three pillars under one instrumentation approach.

The Problem OpenTelemetry Solves — Vendor Lock-In and Fragmented Instrumentation

Before OTel, each observability vendor — Datadog, New Relic, and others — shipped its own proprietary instrumentation SDK. Switching vendors meant re-instrumenting an entire codebase from scratch. OpenTelemetry, a CNCF project, is the industry's answer: one vendor-neutral standard covering metrics, logs, and traces together, with a shared wire protocol — **OTLP** — for exporting telemetry to any compatible backend. Instrument once, send anywhere; swapping Jaeger for a different tracing backend, or Prometheus for a different metrics store, becomes an exporter configuration change, not an application code change.

The Three Signals, One SDK

OTel's own vocabulary calls metrics, logs, and traces **signals** — mirroring `obs1-1`'s own "three pillars" framing, just OTel's specific term for it. The genuine practical payoff: one SDK exposes a tracer, a meter, and a logger side by side, sharing one consistent API — rather than three separate libraries, each with its own conventions, that happen to be used together.

The OpenTelemetry Collector — The Piece That Ties Everything Together

```
receivers:   [OTLP]                          # applications send telemetry here
    ↓
processors:  [batch, filter, ...]            # transform/reduce before export
    ↓
exporters:   [prometheus, loki, jaeger, ...] # fan out to real backends
```

The **Collector** is a separate, standalone process that receives telemetry over OTLP from instrumented applications, can process or filter it, and exports it to one or more backends. This is the concrete mechanism behind `obs1-1`'s own correlation promise: one pipeline, configured once, feeding Ch.2-4's Prometheus, Ch.7's Loki, and Ch.8's Jaeger from the exact same instrumented application code, with no separate integration work per backend.

Auto-Instrumentation Revisited

`obs1-8` previewed this: many OTel language SDKs can auto-instrument common frameworks — HTTP servers and clients, database drivers — with zero code changes, automatically producing spans (and, for the fuller picture this chapter adds, metrics and logs too) for the "boring 80%" of a system's own telemetry. Manual instrumentation, like `obs1-8`'s own `process_payment` span example, is then reserved specifically for business-specific logic auto-instrumentation could never know to track on its own.

Correlating Signals — The Real Payoff

Because metrics, logs, and traces all flow through the same SDK and the same Collector, they can share correlation context **automatically**. A log line emitted while a span is active is automatically enriched with that span's own trace ID — the exact same idea `obs1-7`'s own `request_id` field manually engineered, now produced for free by the underlying mechanism itself:

```
{"timestamp": "2026-07-13T10:32:01Z", "level": "error", "msg": "payment failed", "trace_id": "4bf92f3
```

Metrics get their own version of this same idea via **exemplars** — a real Prometheus/Grafana feature that attaches an example trace ID directly to a specific data point on a histogram, letting you jump straight from "this latency spike, right here on the graph" to "here's an actual trace from exactly that moment." This is what `obs1-1`'s own tip-box meant by correlation being the real goal, not just having all three pillars collected somewhere.

	WITHOUT OPENTELEMETRY	WITH OPENTELEMETRY
Instrumentation	Separate SDK per vendor/tool, often per signal	One SDK, one API, all three signals
Switching backends	Re-instrument the application	Reconfigure the Collector's exporters
Cross-signal correlation	Manually engineered per team (obs1-7's own request_id)	Automatic — shared trace context across signals, plus exemplars

OTEL DOESN'T REPLACE THE BACKENDS FROM EARLIER CHAPTERS

Adopting OpenTelemetry doesn't mean abandoning Prometheus, Loki, or Jaeger — the Collector exports directly to all three. OTel unifies the *instrumentation and pipeline* layer; Ch.2-8's tools remain the actual storage and query backends underneath it.

THE COLLECTOR IS REAL INFRASTRUCTURE, NOT A FREE ABSTRACTION

Running a Collector means operating one more piece of infrastructure — its own scaling needs, its own failure mode, one more thing that can go down. For a small system, exporting directly from applications to each backend without a Collector in between is a genuinely reasonable choice, not a lesser one. Adding this unification layer is a real tradeoff, worth making deliberately rather than by default.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own OTLP/Collector/exporter pipeline, what changes (and what doesn't) in an application's own code when a team switches from exporting traces to Jaeger to exporting them to a different tracing backend instead.

View solution

EXERCISE 2

Explain how automatic trace_id enrichment in a log line differs from obs1-7's own manually-engineered request_id field, and why both ultimately answer the same underlying correlation question.

View solution

EXERCISE 3

A small three-service side project is deciding whether to run an OpenTelemetry Collector or have each service export directly to Prometheus/Loki/Jaeger on its own. Argue for the simpler direct-export approach here, referencing this chapter's own warn-box.

View solution

CHAPTER 9 QUICK REFERENCE

- OpenTelemetry (OTel) — a vendor-neutral, CNCF standard covering metrics, logs, and traces ("signals") with one SDK and the OTLP wire protocol
- Instrument once via OTel; switching backends is an exporter/Collector config change, not a code rewrite

- The **Collector** receives OTLP telemetry, processes it, and fans it out to Prometheus/Loki/Jaeger or any compatible backend
- Auto-instrumentation covers common frameworks automatically; manual spans cover business-specific logic
- Automatic trace_id enrichment in logs and metric **exemplars** deliver obs1-1's own "correlation is the real goal" promise concretely
- OTel unifies instrumentation and pipeline, not the backends themselves — Ch.2-8's tools remain the actual storage/query layer
- The Collector is real operational infrastructure — a small system may reasonably skip it and export directly instead

CHAPTER 10 OF 11

Observability in Kubernetes

Observability

Chapter 10 · Observability in Kubernetes

`k8s2-7`'s own Observability chapter covered Metrics Server, Prometheus scraping, and Grafana at a deliberately light touch, closing with an honest note that a fuller course existed for anyone who needed to go deeper. This is that chapter — applying everything from Ch.2 through Ch.9 specifically to Kubernetes' own genuinely different operating conditions.

Why Kubernetes Needs Its Own Observability Chapter

`obs1-3` already named the core problem: a hand-maintained, static target list breaks the moment instances scale up, scale down, or get rescheduled — exactly what happens continuously in Kubernetes, where pods are genuinely ephemeral and routinely get new IPs. And this is distinct from `k8s1-11`'s own liveness/readiness probes — probes are Kubernetes' own built-in mechanism for restarting or routing around an unhealthy pod; this chapter is about actually observing metrics, logs, and traces from those same pods, a separate and complementary concern.

kube-state-metrics — Metrics About Kubernetes Objects Themselves

`obs1-3`'s `node_exporter` reports host-level metrics — CPU, memory, disk of the underlying machine. **kube-state-metrics** reports something genuinely different: the *state of Kubernetes API objects themselves* — how many pods a Deployment currently has running versus desired, how many replicas are unavailable, pod restart counts, whether a PersistentVolumeClaim is bound. This is directly the same "desired vs. actual state" theme `k8s1-2`'s own reconciliation loop and `k8s1-5`'s ReplicaSet material already established, now exposed as real, queryable metrics.

```
kube_deployment_status_replicas_unavailable{deployment="checkout-api"} 3
```

A genuinely useful alert source on its own — a Deployment with unavailable replicas is a direct, concrete instance of `obs1-6`'s own alert-rule material, applied here to Kubernetes' own object state rather than an application's business metrics.

The Prometheus Operator — Managing Prometheus the Kubernetes Way

Running Prometheus by hand inside Kubernetes — a raw Deployment plus a ConfigMap holding `prometheus.yml` — works, but every new scrape target means manually editing that config and reloading it. The **Prometheus Operator** introduces Kubernetes-native Custom Resources that let Prometheus's own configuration be managed declaratively, the same way any other Kubernetes object is defined per `k8s1-4`'s own YAML-manifest material — rather than as one hand-edited file.

ServiceMonitors — Declarative Scrape Configuration

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: checkout-api
spec:
  selector:
    matchLabels:
      app: checkout-api
  endpoints:
    - port: metrics
      interval: 30s
```

A **ServiceMonitor** declares "scrape any Service matching these labels, on this port, at this interval." The Prometheus Operator watches for ServiceMonitor objects and automatically reconfigures the underlying Prometheus to match, live — no manual reload required. This is Kubernetes' own real fix for `obs1-3`'s static-target problem: instead of a generic `kubernetes_sd_config` still requiring manual relabeling rules, each application team declares its own scrape intent right alongside its own Service definition, reviewed and version-controlled the exact same way `obs1-5` already argued dashboards should be, and echoing `k8s2-9`'s own GitOps pattern.

Logs and Traces in Kubernetes — A Brief Practical Note

Containers write logs to stdout/stderr by convention; a node-level agent — Promtail for Loki, or an OTel Collector deployed as a **DaemonSet** (`k8s2-2`'s own one-pod-per-node pattern) — tails those logs and ships them centrally, automatically attaching pod, namespace, and container labels along the way. Traces work similarly: an OTel Collector, again commonly run as a DaemonSet or a sidecar, receives OTLP from instrumented pods and forwards it to Jaeger, with the same Kubernetes metadata attached automatically as span attributes. This is deliberately a light touch — the bulk of this chapter's genuinely new material is the metrics side (kube-state-metrics, the Operator, ServiceMonitors); logs and traces in Kubernetes are mostly Ch.7/8's own tools, deployed in a Kubernetes-shaped way.

	REPORTS ON	EXAMPLE QUESTION IT ANSWERS
node_exporter	The underlying machine	Is this node running out of memory?
kube-state-metrics	Kubernetes API object state	Does this Deployment actually have as many ready replicas as it should?

KUBE-STATE-METRICS + ALERTMANAGER CATCHES A REAL, COMMON INCIDENT

A Deployment silently stuck at 2 of 5 replicas ready for an hour is invisible without an alert built specifically on `kube_deployment_status_replicas_unavailable` — the pods that are running might look perfectly healthy on their own, while the cluster quietly runs at a fraction of its intended capacity. This is exactly the kind of gap `obs1-6`'s own alerting material, aimed at this specific metric, is built to close.

THE OPERATOR/SERVICEMONITOR PATTERN IS GENUINELY MORE SETUP THAN ONE STATIC CONFIG FILE

A single static `prometheus.yml` is simpler to reason about for a small cluster with few teams. The Prometheus Operator and ServiceMonitors earn their own added complexity specifically once there are enough services and teams that self-service, declarative scrape configuration actually starts paying for itself — not automatically, and not for every cluster, regardless of size.

Hands-On Exercises

EXERCISE 1

Explain, using a concrete example, the difference between what node_exporter would report and what kube-state-metrics would report for the same underlying incident — a pod repeatedly crashing and restarting.

View solution

EXERCISE 2

Write a ServiceMonitor for a service named "inventory-api" with label app=inventory-api, scraping port "metrics" every 15 seconds, and explain what happens automatically once this object is applied to the cluster.

View solution

EXERCISE 3

A small cluster with two services and one team is deciding between a single static prometheus.yml and the Prometheus Operator with ServiceMonitors. Recommend one, using this chapter's own warn-box as justification.

View solution

CHAPTER 10 QUICK REFERENCE

- Kubernetes' ephemeral pods break obs1-3's static target lists — this chapter is the real fix
- **node_exporter** reports on the machine; **kube-state-metrics** reports on Kubernetes object state (desired vs. actual, the k8s1-2 reconciliation-loop theme made queryable)
- The **Prometheus Operator** manages Prometheus declaratively via Kubernetes Custom Resources
- A **ServiceMonitor** declares scrape intent per-service; the Operator reconfigures Prometheus live, automatically, with no manual reload
- Logs/traces in Kubernetes: node-level DaemonSet agents (Promtail, OTel Collector) attach pod/namespace metadata automatically
- kube-state-metrics + Alertmanager catches "stuck at N/M replicas ready" — invisible without a metric built specifically for it
- The Operator/ServiceMonitor pattern is real added complexity — worth it at real scale, not automatically for every cluster

CHAPTER 11 OF 11

Capstone: Building a Full Observability Stack

Observability

Chapter 11 · Capstone — Building a Full Observability Stack

The final chapter. This capstone wires Prometheus, Grafana, Loki, Jaeger, Alertmanager, and the OpenTelemetry Collector together into one real stack, running the exact checkout-service scenario `obs1-1` opened this course with — and closes by walking one real incident through it end to end, using all three pillars the way an actual investigation would.

The Architecture — One Pipeline, Three Backends

Four OTel-instrumented services — `gateway`, `auth-service`, `checkout-service`, `inventory-service` — running in Kubernetes. Each exports telemetry via OTLP to an OpenTelemetry Collector running as a DaemonSet (`obs1-10`). The Collector fans out to Prometheus (metrics), Loki (logs), and Jaeger (traces). Grafana sits on top, querying all three as data sources (`obs1-5`). Alertmanager receives fired alerts from Prometheus and routes them (`obs1-6`). One pipeline, configured once, feeding every backend this course built.

Step 1 — Discovery via ServiceMonitors

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: checkout-stack
spec:
  selector:
    matchLabels:
      part-of: checkout-stack
  endpoints:
    - port: metrics
      interval: 15s
```

One ServiceMonitor, matching a shared `part-of: checkout-stack` label across all four services — `obs1-10`'s own real fix for hand-maintaining a static target list as these pods scale and get rescheduled.

Step 2 — Instrumentation via OpenTelemetry

Auto-instrumentation covers incoming/outgoing HTTP calls and database queries across all four services with no code changes. `checkout-service` adds one manual span for its own business logic — the exact `obs1-8` / `obs1-9` pattern:

```
with tracer.start_as_current_span("reserve_inventory") as span:
    span.set_attribute("order.item_count", 3)
```

Step 3 — Metrics & Dashboards

Every service exposes `http_requests_total` (counter) and `http_request_duration_seconds` (histogram) via auto-instrumentation, plus `kube_deployment_status_replicas_unavailable` from kube-state-metrics (`obs1-10`). One Grafana dashboard, built with a `$service` template variable (`obs1-5`), covers all four services with three panels: request rate, error rate (`obs1-4`'s own percentage-of-5xx pattern), and p99 latency via `histogram_quantile()` — one dashboard, not four.

Step 4 — Alerting

```
groups:
  - name: checkout-stack
    rules:
      - alert: HighErrorRate
        expr: sum(rate(http_requests_total{status=~"5.."}[5m])) by (job) / sum(rate(http_requests_tot
        for: 10m
        labels:
          severity: critical
```

`obs1-6`'s own rule, reused directly, now grouped `by (job)` so each service's error rate is tracked independently. Routed critical → PagerDuty, warning → Slack, grouped by `alertname` / `job` so a widespread incident produces one notification, not one per affected instance.

Step 5 — Logs

```
{"timestamp": "2026-07-13T14:02:11Z", "level": "error", "msg": "inventory reservation timeout", "trac
```

Structured JSON (`obs1-7`), with `trace_id` attached automatically by the OTel SDK (`obs1-9`) rather than a hand-engineered correlation field. Any log line from any of the four services can be found via one LogQL query filtered on that exact `trace_id`.

Step 6 — Traces

```
Trace ID: 4bf92f3577b34da6a3ce929d0e0e4736

gateway (4.1s)                    [root span]
  auth-service (0.1s)
  checkout-service (3.9s)
    reserve_inventory (3.85s)
      inventory-service (3.8s)
        db-query (3.75s)
```

`obs1-1`'s own opening example, now a real trace flowing through the actual pipeline built across this entire course — 3.8 of 4.1 seconds spent in `inventory-service`, visible directly in Jaeger's own waterfall view.

A Real Incident, Walked End to End

1. **Alertmanager** fires `HighErrorRate` for `checkout-service`, routed to Slack (severity: warning at first, since error rate hasn't yet crossed into timeouts becoming outright 5xx failures).
2. **Grafana**'s dashboard, filtered to `$service=checkout-service`, shows the p99 latency spike and pinpoints the time window.
3. **Explore mode** against Loki, filtered to that time window and service, surfaces the `"inventory reservation timeout"` log line and its `trace_id`.
4. **Jaeger**, queried by that exact trace ID, shows the full tree above — `inventory-service`'s own `db-query` span is where the time actually went, not `checkout-service` itself.

Metrics said something was wrong; logs said what; traces said exactly where — `obs1-1`'s own three-question framing, now answered by a real, working pipeline rather than a conceptual walkthrough.

PIECE	CHAPTER
ServiceMonitor discovery	**obs1-10**
OTel instrumentation & Collector pipeline	**obs1-8, obs1-9**
PromQL error-rate/latency queries	**obs1-4**
Grafana dashboard with template variable	**obs1-5**
Alertmanager routing & grouping	**obs1-6**

PIECE	CHAPTER
Structured logs, LogQL, correlation	obs1-7
Trace tree, span attributes	obs1-8
Metrics vs. logs vs. traces framing	obs1-1, obs1-2

STILL OUT OF SCOPE, HONESTLY

This capstone doesn't cover long-term, cross-cluster storage (Thanos/Cortex/Mimir, explicitly named out of scope back in `obs1-3`), production-grade high availability for the observability stack itself (a single Prometheus/Loki/Jaeger instance each, not a replicated deployment), securing the stack's own endpoints (dashboards and metrics endpoints need their own access control, a separate concern from anything this course covered), or a real tail-based sampling implementation (`obs1-8` discussed the strategy, not the deployment). None of these are oversights — they're genuine next steps beyond what an 11-chapter course can reasonably claim to have finished.

Hands-On Exercises

EXERCISE 1

Write the LogQL query that would find the exact log line shown in Step 5, filtered to the checkout-stack's logs and the specific trace_id shown in Step 6.

View solution

EXERCISE 2

Explain why the incident walkthrough moves from Alertmanager to Grafana to Loki to Jaeger in that specific order, rather than starting directly with Jaeger.

View solution

EXERCISE 3

A colleague asks why this capstone doesn't include a production-HA setup for Prometheus/Loki/Jaeger themselves. Write a short, honest answer using this chapter's own scope note.

View solution

CHAPTER 11 QUICK REFERENCE — OBSERVABILITY COMPLETE

- One OTel Collector pipeline feeds Prometheus, Loki, and Jaeger from the same instrumented services — obs1-9's own unification, made real
- ServiceMonitors handle discovery; Grafana with template variables handles visualization; Alertmanager handles routing/grouping
- trace_id ties logs and traces together automatically; metrics point at the right time window to search within
- The incident walkthrough is obs1-1's own three-pillar framing (something's wrong / what happened / where) answered by a real, working pipeline
- Honest scope note: no long-term storage, no HA for the stack itself, no stack-endpoint security, no tail-based sampling deployment — genuine next steps, not oversights
- **The full Observability course — 11 chapters — is now complete.**